B-CONVERT: PROJEÇÃO DE FACES TRIANGULARES BASEADA NO ALGORITMO DE TRAÇADO DE RETAS DE *BRESENHAM*

Luciano Lauand Viana de Paula

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Ricardo Cordeiro de Farias

Rio de Janeiro
Setembro de 2012

B-CONVERT: PROJEÇÃO DE FACES TRIANGULARES BASEADA NO ALGORITMO DE TRAÇADO DE RETAS DE *BRESENHAM*

Luciano Lauand Viana de Paula

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

Prof. Ricardo Cordeiro de Farias, Ph.D.

Prof. Ricardo Guerra Marroquim, D.Sc.

Prof. Esteban Walter Gonzalez Clua, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
SETEMBRO DE 2012

de Paula, Luciano Lauand Viana

B-Convert: Projeção de faces triangulares baseada no algoritmo de traçado de retas de *Bresenham*/Luciano Lauand Viana de Paula. – Rio de Janeiro: UFRJ/COPPE, 2012.

XII, 68 p.: il.; 29, 7cm.

Orientador: Ricardo Cordeiro de Farias

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2012.

Referências Bibliográficas: p. 52 – 56.

1. scan convert. 2. face projection. 3. volume rendering. I. de Farias, Ricardo Cordeiro. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

A Maria Clícia de Castro (D.Sc.) e minha família, em especial, José Luiz Fernandes Braga (in memorian).

Agradecimentos

Gostaria de agradecer a minha mãe Vania Aida Viana de Paula (*M.A.*), por todo seu apoio antes e durante esta pesquisa. À ela e minha tia Isis Fernandes Braga (*D.Sc.*) por me incentivarem a mergulhar no mestrado. Ao meu orientador, Ricardo Farias (*Phd*) por seus conselhos, explicações, dicas e ajuda. Mais que orientador, um amigo. Ao amigo e colega da linha de Computação Gráfica Guilherme Cox (*M.Sc.*) por suas valiosas contribuições e sugestões em otimização. Aos amigos e professores do LCG. À Cristiane Martins por me aguentar quando os resultados obtidos me forçavam a ”voltar à prancheta”, ao meu irmão Eduardo, meu pai e Júlia por serem quem são. À Rodolfo de Paula, pelo apoio imprescindível, e por fim à minha orientadora na graduação, Maria Clícia de Castro (*D.Sc.*), sem a qual eu não teria voltado minha atenção à pesquisa científica.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

B-CONVERT: PROJEÇÃO DE FACES TRIANGULARES BASEADA NO ALGORITMO DE TRAÇADO DE RETAS DE *BRESENHAM*

Luciano Lauand Viana de Paula

Setembro/2012

Orientador: Ricardo Cordeiro de Farias

Programa: Engenharia de Sistemas e Computação

Apresentamos uma proposta para o *scan convert* de faces triangulares usando apenas *pixels* dentre as arestas da face projetada. Nosso algoritmo calcula os pontos das arestas, pelo algoritmo de *Bresenham* para traçado de retas, posteriormente interpolando os pontos internos da face. Ao contrário do processo de *scan convert* tradicional, nossa abordagem não precisa testar a interseção de raios do *bounding box* de cada face contra o plano definido pelos vertices da face triangular. Uma representação mais próxima dos *pixels* da face é usada ao invés do *bounding box*. Pesquisa-se também as condições necessárias para haver ganho de desempenho bem como atuais limitações.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

B-CONVERT: TRIANGULAR FACE PROJECTION THROUGH BRESENHAM'S LINE DRAWING ALGORITHM

Luciano Lauand Viana de Paula

September/2012

Advisor: Ricardo Cordeiro de Farias

Department: Systems Engineering and Computer Science

We present a new approach to scan convert triangular faces using only pixels between the edges of the projected face itself. Our algorithm calculates edge points through *Bresenham*'s line drawing algorithm, interpolating points within the face. As opposed to traditional scan convert process, our approach does not need to test the intersection between rays from each face's bounding box against the plane defined by the vertices of the triangular face. A better match to the actual face's *pixels* is used instead of the *bounding box*. We also research conditions to achieve performance speed up as well as current limitations.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Introdução

Em visualização volumétrica pesquisa-se formas de extrair informações significantes de dados volumétricos estruturados ou não estruturados. Tais dados contêm informações adquiridas por amostragem em eventos reais através de diversos meios como ressonância magnética (*MRI*), tomografia computadorizada (*CT*) industrial ou médica e microscopia confocal, por exemplo. Os dados também podem ser gerados por simulação computacional como é o caso da área de dinâmica de fluidos (*CFD* ou *computational fluid dynamics*) ou gerados por modelo geométrico em aplicações mais tradicionais de computação gráfica como *CAD* (*computer aided design*).

Com aplicação em diversas áreas, onde destacam-se medicina, indústria, biologia e geologia, concerne a modelagem, manipulação, renderização e representação da informação presente nos já citados dados volumétricos. Estes são entidades tridimensionais com possível variação temporal contendo informações escalares e/ou vetoriais, podendo conter ou não superfícies internas. De fato, a quantidade e qualidade das informações que se deseja extrair dos dados volumétricos definirão a forma de renderização volumétrica mais adequada.

Os dados volumétricos podem ser entendidos como um sub espaço V contendo amostras (x, y, z, v) ou *voxels* representando o valor v (escalar ou vetorial) de uma propriedade do dado como cor, densidade, calor, pressão, velocidade, entre outras, em um local $3D$ discretizado nas coordenadas (x, y, z) [1]. Uma definição mais detalhada para dados volumétricos *escalares* (utilizados no presente caso de uso) é apresentada em [2], onde são descritos como um par (V, W), no qual V é um sub espaço finito de $\mathbb{R}^3$ tal que $V = \{v_i \in \mathbb{R}^3 \mid i = 1, ..., n\}$ e W, um sub espaço finito de $\mathbb{R}$ tal que $W = \{w_i \in \mathbb{R} \mid i = 1, ..., n\}$, contendo valores de um campo escalar $f(x, y, z)$ amostrados em V, através da função $w_i = f(v_i)$.

Existem diversas técnicas de renderização volumétrica, como veremos adiante, dentre

as quais destacamos a projeção de faces (*face projection*), por fazer uso constante do passo de *scan convert* de faces triangulares, configurando-se num interessante estudo de caso para aplicarmos o algoritmo proposto neste trabalho.

A renderização volumétrica é uma técnica essencial para a visualização de dados volumétricos. Simulando a absorção e dispersão da luz que atravessa o volume, tem por objetivo exibir o dado volumétrico como uma imagem bidimensional coerente e provida de significado, revelando informações interiores ao dado, importantes ao usuário.

Devido a já existirem técnicas para renderização de superfícies, muitas técnicas de renderização volumétrica se baseiam na renderização de uma superfície gerada por aproximação contida no dado volumétrico. Volumes visualizados desta forma indireta podem perder uma camada de informação pela própria natureza da aproximação da superfície. Um destes métodos de renderização envolve a geração de uma iso-superfície (através de algoritmos como *marching cubes* por exemplo) e é denominado de renderização indireta do volume (*IVR* ou *indirect volume rendering*).

Para evitar o problema da perda de informação na geração da iso-superfície, foram desenvolvidas técnicas de renderização direta que buscam transformar a informação contida no dado volumétrico em uma imagem bidimensional sem a necessidade de gerar representações intermediárias do volume. Este método é conhecido como renderização direta de volume (*DVR* ou *direct volume rendering*) e seus algoritmos são mais complexos e, consequentemente, mais custosos, porém podem representar de forma mais precisa, informação do volume original na imagem $2D$ resultante. Neste trabalho chamaremos a renderização direta de volumes apenas por renderização volumétrica, visto que a técnica utilizada no caso de uso refere-se apenas à renderização direta.

Para aprofundamendo do tema de visualização volumétrica ver [1], [3], [4], [5].

1.2 A Proposta do Trabalho

A técnica abordada neste trabalho, denominada *B-Convert*, surgiu da pesquisa em otimizar o tempo gasto no passo de *scan convert* durante a renderização de um volume. Pesquisamos a utilização do clássico algoritmo de *Bresenham*[6] para traçado de retas, *na projeção das faces de um dado volumétrico*.

Nossa proposta busca reduzir o tempo gasto com o *scan convert* de faces triangulares testando a interseção raio / plano apenas para os *pixels* que melhor representam a face triangular, e não todos os *pixels* presentes no *bounding box* da face como ilustrado na Figura 1.1. Obtemos esta representação triangular aplicando o algoritmo de traçado de retas de *Bresenham* sobre as arestas do triângulo da face no plano da imagem.

Como estudo de caso, pesquisamos seu desempenho quando implementado no *ZSweep*, um algoritmo de *rendering* volumétrico por projeção de faces (*face projection*).

Neste trabalho adotamos o sistema de coordenadas cartesiano $2D$ e $3D$ para ilustrar

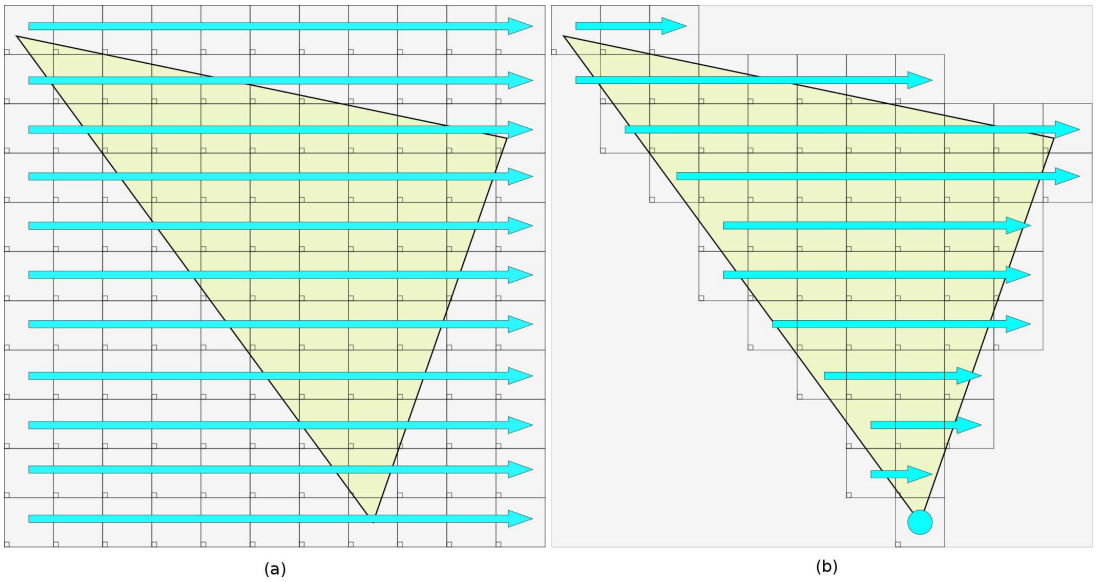

Figura 1.1: Em *(a)*, o *scan convert* é executado sobre todo o *bounding box* da face; no *B-Convert*, apenas *pixels* de uma representação mais próxima da face projetada são processados *(b)*.

o conteudo teórico. A implementação segue utilizando dados dos tipos inteiro e ponto flutuante de dupla precisão. As seguintes notações são utilizadas neste trabalho:

- (x, y): par de coordenadas num sistema bidimensional.
- $[x_i, x_j)$: intervalo fechado na abscissa x_i e aberto na abscissa x_j.
- $\lfloor y \rfloor$: arredondamento para o limite inferior da ordenada y; $y \in \mathbb{R}$. Ex: $\lfloor 2.85 \rfloor = 2$.
- $\lceil y \rceil$: arredondamento para o limite superior da ordenada y; $y \in \mathbb{R}$. Ex: $\lceil 2.85 \rceil = 3$.
- v_i ponto no plano bidimensional tal que $v_i = (x_{v_i}, y_{v_i})$, $v \in \mathbb{R}^2$; $i \in \mathbb{N}$.
- w_i ponto no espaço tridimensional tal que $w_i = (x_{w_i}, y_{w_i})$, $w \in \mathbb{R}^3$; $i \in \mathbb{N}$.

Capítulo 2

Revisão Bibliográfica

Procuramos, neste capítulo, abordar os trabalhos relacionados à rasterização de retas no espaço da imagem, situando o algoritmo de *Bresenham*[6] como um divisor de águas e um dos trabalhos seminais na rasterização de retas. Também relacionamos trabalhos relativos às técnicas de renderização volumétrica abordadas no algoritmo *ZSweep* no qual implementamos o *B-Convert*.

2.1 Rasterização (*scan convert*)

O processo de rasterização é uma operação fundamental em computação gráfica. Busca representar de forma matricial uma informação matemática anteriormente descrita na forma vetorial. Atualmente é a técnica mais utilizada para exibição de informação $3D$ em tempo real em dispositivos de telas bidimensionais. Consiste em mapear informações geométricas em *pixels*, que conterão informação de cor, definida por outra técnica como *shading*. Em outras palavras, uma cena contida em um espaço tridimensional, descrita através de polígonos (geralmente uma coleção de triângulos) tem as coordenadas $3D$ de todos seus vértices ou pontos visíveis (em relação a um modelo de câmera) transformadas para locais correspondentes em outro sistema de coordenadas. Este segundo sistema permite um suporte bidimensional para que os valores convertidos sejam empregados para formar uma imagem matricial a ser mapeada na tela de um dispositivo de saída como um monitor ou impressora.

Alguns autores consideram rasterização todo o processo de transformação da geometria em *pixels*. Abrangendo as transformações aplicadas sobre o dado, desde o sistema de coordenadas do objeto até a etapa de preenchimento dos polígonos, já mapeados em *pixels* no plano da imagem. Estas etapas são geralmente implementadas não apenas em bibliotecas gráficas como *OpenGL* mas também em *hardware* de placas gráficas.

Uma definição mais elaborada do processo pode ser obtida em AKENINE-MÖLLER *et al.* [7] e LENGYEL [8]. Tais autores decompõem o *pipeline* gráfico em duas etapas.

Uma geométrica e outra de rasterização. Na primeira, as primitivas geométricas da cena sofrem transformação do espaço do objeto (ou modelo) até o plano da imagem. Na segunda, também chamada *scan conversion* [7], em posse dos vértices já transformados e projetados, busca-se computar quais *pixels* são cobertos pelo objeto, além de definir suas cores. Gerando assim, uma representação do modelo no plano da imagem. O processo de *scan convert* é usado desde algoritmos de determinação de faces visíveis à aceleração em traçado de raios (*ray-tracing*) além de *rendering* em tempo real de gráficos $3D$. A Figura 2.1 apresenta as etapas de rasterização para uma face triangular.

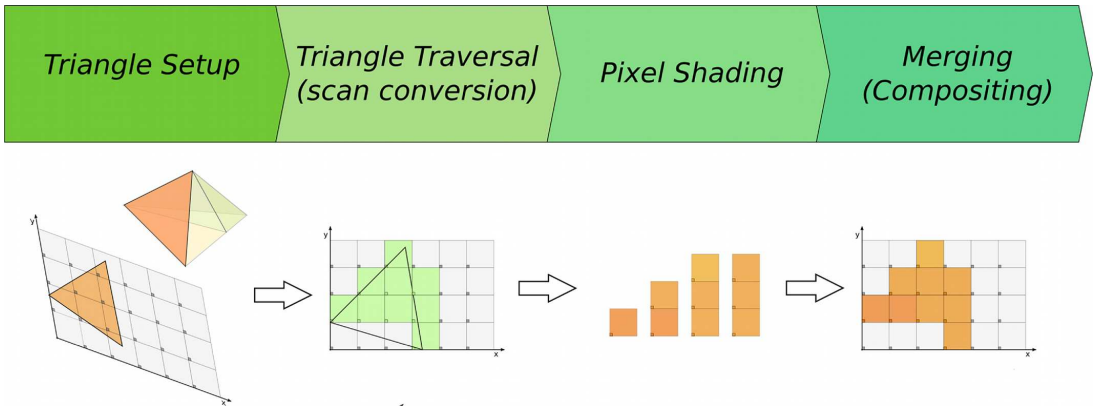

Figura 2.1: Na preparação do triângulo (*triangle setup*) ocorre a mudança de sistemas de coordenadas do espaço tridimensional (cena) para o sistema bidimensional (plano da imagem); na etapa de *scan conversion*, o *bounding box* da face triangular é atravessado e são identificados *pixels* que interceptam o semi-plano definido pelas arestas da face triangular; as cores resultantes da interação da luz com a face são acumuladas com os valores correspondentes para cada *pixel* (*pixel shading*); o passo final é a composição destes valores formando a imagem final (*merging, compositing*).

2.2 Rasterização de retas

A rasterização de uma reta pode ser feita através da equação da reta em sua forma reduzida, através de um algoritmo incremental como o *DDA* (*digtal differential analizer*) ou o algoritmo de traçado de retas de *Bresenham* [6] e suas inúmeras variantes. Devido à sua natureza eficiente e rápida, operando apenas sobre inteiros, o algoritmo de *Bresenham* tornou-se referência por sua simplicidade, velocidade e eficiência, sendo implementado em diversos tipos de *hardware*, de placas gráficas à sistemas de robótica. A partir dele, foram desenvolvidos outros algoritmos e técnicas procurando aumentar ainda mais a velocidade em rasterização de retas como as técnicas de multi-pontos, do qual destaca-se o algoritmo simétrico de Xiaolin Wu [9]. Para melhor entender o algoritmo de traçado de retas de *Bresenham* é inicialmente apresentado um algoritmo também incremental, porém mais simples, operando sobre o conjunto de números reais $\mathbb{R}$, o *DDA*.

2.2.1 *DDA* (*digtal differential analizer*)

Dados dois pontos em $\mathbb{R}^2$, $p_0 = (x_0, y_0)$ e $p_1 = (x_1, y_1)$ tal que:

$$\frac{y - y_0}{y_1 - y_0} = \frac{x - x_0}{x_1 - x_0}, \tag{2.1}$$

podemos chegar na equação da reta (2.2),

$$\begin{aligned} y &= [(\frac{y_1 - y_0}{x_1 - x_0})(x - x_0)] + y_0 \\ y &= \underbrace{\frac{\Delta_y}{\Delta_x}}_{m}(x - x_0) + y_0 \end{aligned} \tag{2.2}$$

que equivale à forma reduzida (2.3) da equação geral da reta $ax + by + c = 0$, no qual o termo q armazena o valor de y no caso de $x = 0$ ou $m = 0$.

$$y = mx + q,\ q = -\frac{c}{b} \tag{2.3}$$

Isso é tudo que precisamos para definir a estratégia mais simples para rasterização de retas de forma incremental. Comparando Δ_x com Δ_y encontramos o eixo maior, chamado de eixo diretor DA (*driving axis*) e o eixo menor PA (*passive axis*). Se, por exemplo, estivermos no primeiro octante do círculo trigonométrico, com $0 \leq m \leq 1$, o eixo x será o eixo dominante, e partindo do extremo inicial $p_0 = (x_0, y_0)$ até o ponto extremo final $p_1 = (x_1, y_1)$ incrementamos a coordenada do eixo diretor de uma unidade (um *pixel*), enquanto o coeficiente angular $m = \frac{\Delta_y}{\Delta_x}$ é usado para incrementar o eixo menor y, sendo que para cada iteração i iluminamos o *pixel* $p_i = (x_i, \lfloor 0.5 + y_i \rfloor)$. Apesar de eficaz, o Algoritmo 1, exige operações com números reais, incluindo ao menos uma operação de divisão, o que o torna não muito atraente do ponto de vista computacional. O algoritmo de *Bresenham* trata justamente destas limitações.

Algorithm 1 DDA: Desenha linha de x0, y0 até x1, y1.

```
Require: Round(r) = floor(r + 0.5f);
 1: dx := x1 − x0;
 2: dy := y1 − y0;
 3: m := dy/dx;
 4: dx := abs(dx)
 5: dy := abs(dy)
 6: if dx ≥ dy then
 7:     while x0 ≠ x1 do
 8:         WritePixel( x0, Round(y0) );
 9:         x0 := x0 + 1;
10:         y0 := y0 + m;
11: else
12:     inv_m := 1/m;
13:     while y0 ≠ y1 do
14:         WritePixel( Round(x0), y0 );
15:         x0 := x0 + inv_m;
16:         y0 := y0 + 1;
17: WritePixel( Round(x0), Round(y0) );
```

2.2.2 Algoritmo de traçado de retas de *Bresenham*

O algoritmo de *Bresenham* [6] é de fato o algoritmo clássico de traçado de retas em computação gráfica. Foi desenvolvido por *Jack Bresenham*, engenheiro da *IBM* em 1965, originalmente para comandar um *plotter* digital capaz de executar 8 tipos de movimentos lineares, permitindo o desenho aproximado de retas com diferentes coeficientes angulares como se pode ver na Figura 2.2 *(a)*.

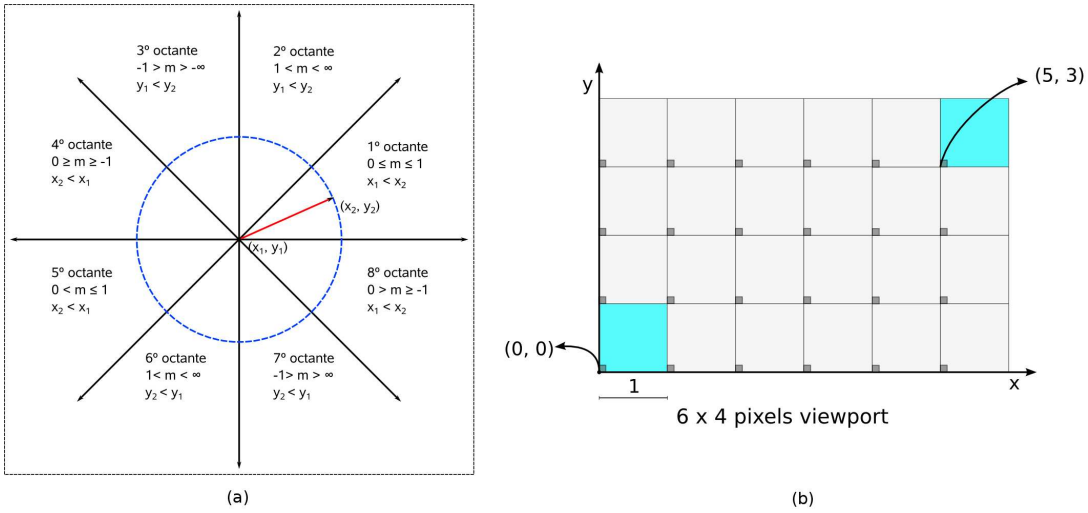

Figura 2.2: Em *(a)* são apresentados os movimentos possíveis do *plotter* bem como as variações no coeficiente angular m da reta e as relações entre as coordenadas dos pontos extremos nos octantes tratados no algoritmo; uma situação possível de ser tratada pelo algoritmo no primeiro octante é apresentada em *(b)*.

Desde então, este algoritmo vem sendo amplamente utilizado em computação gráfica sob diversas variações. O algoritmo é eficiente, podendo ser implementado em *hardware*, utilizando apenas matemática de inteiros (sua principal vantagem sobre o *DDA*) e uma variável de decisão ϵ responsável por prover uma aproximação da inclinação da reta.

O método funciona da seguinte forma: dados dois pontos extremos como na Figura 2.2 *(b)*, a rasterização de uma reta entre eles busca descrever os *pixels* de menor distância à reta real. De forma simples, o algoritmo de traçado de retas de *Bresenham escolhe* o *pixel* seguinte, incrementando em uma unidade a coordenada no eixo dominante, para então identificar qual coordenada no eixo passivo é mais próxima da interseção da reta real com a nova coordenada no eixo dominante, incrementândo-a ou não. Isso pode ser visto na Figura 2.3

Essa escolha é implementada através de uma variável de controle denominada ϵ cujo valor é alterado durante a execução do algoritmo incremental, baseando-se nos valores de Δ_x e Δ_y além do próprio valor de ϵ da iteração anterior. *Bresenham* usou as relações geométricas entre a matriz de pontos e a reta real para desenvolver seu algoritmo. Para ilustrar o método ao desenhar o terceiro *pixel* da representação da reta na matriz 6 x 4 utilizamos as seguintes notações:

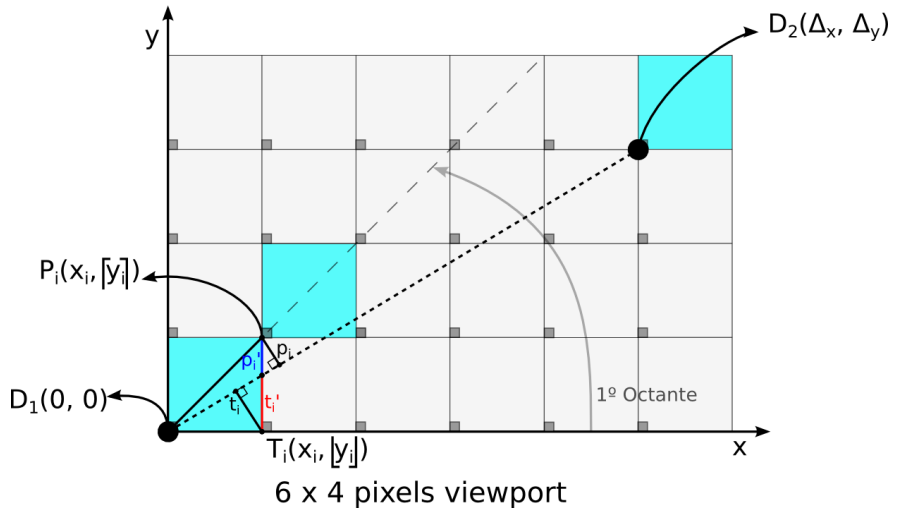

Figura 2.3: Escolha do primeiro *pixel* no traçado de uma reta do *pixel* $(0, 0)$ ao *pixel* $(5, 3)$. Pelo fato do *pixel* $(1, 1)$ ser mais próximo da interseção da reta com a grade matricial, ele é selecionado.

- $S_i = (x_i, y_i)$ o ponto pertencente à reta s que intercepta a grade matricial.
- *pixel* $R_i = (x_i, \lfloor y_i \rfloor)$ candidato pertencente à ordenada y atual.
- *pixel* $Q_i = (x_i, \lceil y_i \rceil)$ candidato pertencente à ordenada y seguinte em um eventual incremento no eixo passivo (y no caso).
- r_i a menor distância da reta s ao *pixel* R_i.
- q_i a menor distância do *pixel* Q_i à reta real s.
- $\Delta_x = x_2 - x_1,\ \Delta_y = y_2 - y_1$.

A partir destas notações podemos chegar em q_i' e r_i', respectivamente as distâncias de S_i à Q_i e de S_i à R_i.

Após escolher o *pixel* P_i, existem duas escolhas possiveis: R_i e Q_i. Por semelhança de triângulos, $r_i' - q_i'$ tem o mesmo sinal que $r_i - q_i$. Na iteração corrente, a decisão é regida pelo resultado de $\epsilon_i = (r_i' - q_i')\Delta_x$, como a reta se encontra no primeiro octante, temos que $\Delta_x > 0$. Se $\epsilon_i \geq 0$, o *pixel* mais perto será Q_i, imediatamente acima do ponto S_i. Do contrário o *pixel* R_i, imediatamente abaixo à S_i será escolhido.

Esta explicação do método ainda é feita usando aritmética com números reais. Deve-se notar na Figura 2.4 que a coordenada y_i do ponto S_i pode ser obtida por $y_i = (\Delta_y/\Delta_x)x_i$ e que as distâncias r_i', q_i' podem ser reescritas por $y_i - \lfloor y_i \rfloor$ e $\lceil y_i \rceil - y_i$

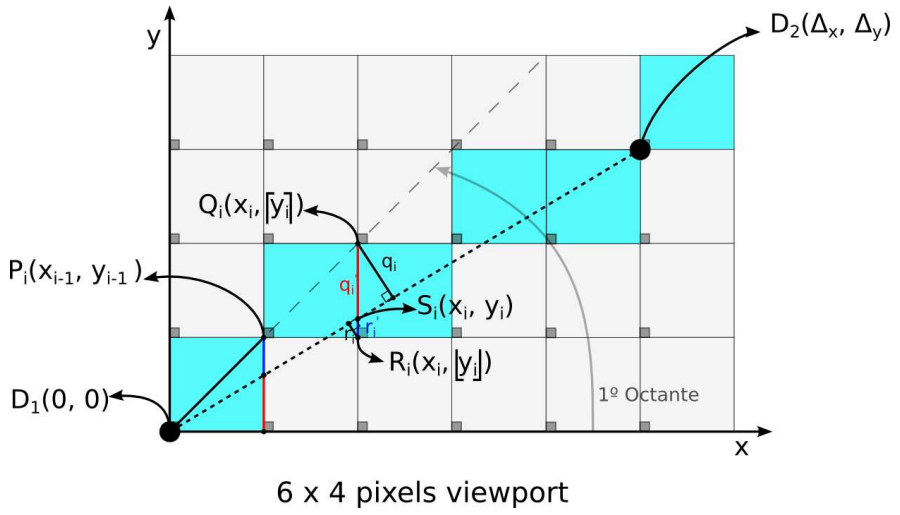

Figura 2.4: Critério de escolha entre possíveis *pixels* para representar uma reta real utilizando o algoritmo de *Bresenham*.

respectivamente. Permitindo reescrever ϵ_i como:

$$\epsilon_i = [\underbrace{(y_i - \lfloor y_i \rfloor)}_{r'_i} - \underbrace{(\lceil y_i \rceil - y_i)}_{q'_i}]\Delta_x$$

$$\epsilon_i = [2(\underbrace{y_i}_{(\frac{\Delta_y}{\Delta_x})x_i}) - (\lfloor y_i \rfloor + \lceil y_i \rceil]\Delta_x$$

$$\epsilon_i = 2(x_i\Delta_y) - (\lfloor y_i \rfloor + \lceil y_i \rceil)\Delta_x$$

Bresenham enxergou a relação dos limites inferior e superior de y_i com o vetor unitário para a coordenada y_{i-1}, proveniente da iteração imediatamente anterior, de forma que $\lfloor y_i \rfloor = \hat{y}_{i-1}$ e $\lceil y_i \rceil = \hat{y}_{i-1} + 1$, permitindo reescrever novamente a fórmula para ϵ_i, agora livre das coordenadas reais x_i e y_i:

$$\epsilon_i = 2x_{i-1}\Delta_y - 2\hat{y}_{i-1}\Delta_x + 2\Delta_y - \Delta_x \tag{2.4}$$

Pode-se notar que para ϵ_1, nossa condição inicial, usando as coordenadas do ponto $P_{i-1} = P_0$ usaríamos $x_0 = 0$ e $\hat{y_0} = 0$, chegando em:

$$\epsilon_1 = 2\Delta_y - \Delta_x \tag{2.5}$$

Logo, é possível considerar duas opções para $\hat{y}_i$ e ϵ_{i+1} a partir de (2.4) e (2.5):

$$\left.\begin{aligned} &\hat{y}_i = \hat{y}_{i-1} + 1 \\ &\epsilon_{i+1} = 2(x_{i-1} + 1)\Delta_y - 2(\hat{y}_{i-1} + 1)\Delta_x + 2\Delta_y - \Delta_x \end{aligned}\right\} se\ \epsilon_i \geq 0$$

$$\left.\begin{array}{l} \hat{y}_i = \hat{y}_{i-1} \\ \epsilon_{i+1} = 2(x_{i-1}+1)\Delta_y - 2(\hat{y}_{i-1})\Delta_x + 2\Delta_y - \Delta_x \end{array}\right\} se\ \epsilon_i < 0$$

Resultando em um sistema usando apenas aritmética de inteiros:

$$\epsilon_{i+1} = \begin{cases} \epsilon_i + 2\Delta_y - \Delta_x \ \ se\ \epsilon_i \geq 0, \\ \epsilon_i + 2\Delta_y \ \ se\ \epsilon_i < 0 \end{cases} \tag{2.6}$$

O algoritmo é sensível aos diferentes octantes bem como ao sinal de Δ_x, Δ_y e $|\Delta_x| - |\Delta_y|$. *Bresenham* construiu uma tabela [6] indicando a relação entre o sinal destas variáveis e os movimentos a serem executados, na prática reduzindo as outras situações possíveis ao exemplo do primeiro octante, invertendo a posição de Δ_x e Δ_y nas equações (2.5) e (2.6). O algoritmo de Bresenham para o primeiro octante pode ser visto no Algoritmo 2.

Algorithm 2 func bresenhamLine(x_0, y_0, x_1, y_1)

```
1:  inteiro Δy := y1 − y0;
2:  inteiro Δx := x1 − x0;
3:  inteiro ϵ := 2Δy − Δx;
4:  inteiro y := y0
5:  for x := x0 such that x ≤ x1 do
6:      WritePixel(x,y);
7:      x := x + 1;
8:      ϵ:=ϵ + 2Δy;
9:      if ϵ ≥ 0 then
10:         y := y + 1;
11:         ϵ := ϵ − Δx;
```

Um segundo trabalho foi apresentado por PITTEWAY [10] estendendo o trabalho de *Bresenham* para elípses e hipérboles através da técnica do ponto médio, no qual ϵ é associado a uma variável de deslocamento k e ao conceito de erro relativo entre os pontos médios dos *pixels*, tal que $-0.5 \leq k < 0.5$. Van Aken [11, 12] adapta este algoritmo e demonstra que para a rasterização de retas e círculos, tal técnica se reduz ao algoritmo de *Bresenham* [6].

Sob este ângulo, pode-se obter a representação em *pixels* de uma reta caminhando de um ponto extremo ao outro, incrementando, a cada iteração, um *pixel* no eixo diretor DA (*driving axis*). A cada passo, a variável de controle ϵ, a partir de sua condição inicial, é incrementada de um valor inteiro equivalente a incrementar de m (coeficiente angular da reta) a coordenada atual no eixo passivo. Quando ϵ ultrapassa um determinado limite, o eixo passivo PA (*passive axis*) é incrementado e ϵ é decrementada para representar o desvio do ponto de interseção entre a reta original e o limite inferior do *pixel* por ela atravessado. O processo se repete até que chega-se ao ponto extremo de destino, sendo que o *pixel* escolhido é iluminado ou *plotado* a cada passo do algoritmo como visto na Figura 2.5.

A versão otimizada do algoritmo de traçado de retas de *Bresenham* que desenvolvemos é generalizada para todos os octantes, como proposto em [13] e [14]. Nossa

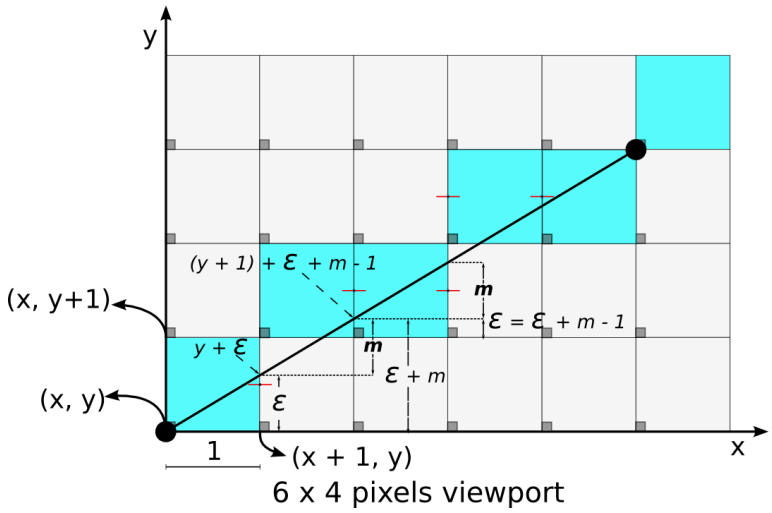

Figura 2.5: Representação em *pixels* de uma reta gerada com algoritmo de *Bresenham*, note a relação entre a variável de decisão ϵ e o coeficiente angular da reta real s.

implementação retorna apenas uma lista contendo as coordenadas (x, y) dos *pixels* iniciais e finais de cada ordenada y no intervalo que define a reta no plano da imagem. O codigo fonte para esta versão encontra-se na Listagem A.2.

2.3 Visualização Volumétrica

A visualização *direta* de volumes ou simplesmente visualização volumétrica [1],[4] neste trabalho está relacionada com as técnicas de projeção de faces e traçado de raios, mais especificamente com o algoritmo *ZSweep* proposto por FARIAS *et al.* [15]. Nosso interesse no *ZSweep* reside no fato de que por se tratar de um algoritmo de *rendering* volumétrico por projeção de faces, utiliza amplamente a conversão de faces triangulares, ou seja, o processo de *scan convert*, o qual buscamos substituir pelo *B-Convert* neste trabalho.

2.3.1 Projeção de Faces (*face projection*) e *ZSweep*

Algoritmos de projeção de face possuem a vantagem de utilizar a coerência dos raios de forma direta, trabalhando no espaço da imagem. Ao contrário da técnica de emissão de raios, onde cada *pixel* do plano da imagem emite raios sobre o espaço tridimensional (Figura 2.6 *(a)*), as faces das células são projetadas diretamente no plano da imagem e as interseções do raio com cada face são computadas durante a projeção (Figura 2.6 *(b)*). Embora essa abordagem privilegie interseções de *pixels* adjacentes, algoritmos de projeção de face necessitam do auxílio de um segundo algoritmo para ordernar as faces das células por visibilidade antes de projetá-las. Sem essa ordenação, é possivel ocorrer oclusão mútua entre células do dado volumétrico como visto na Figura 2.6 *(c)*.

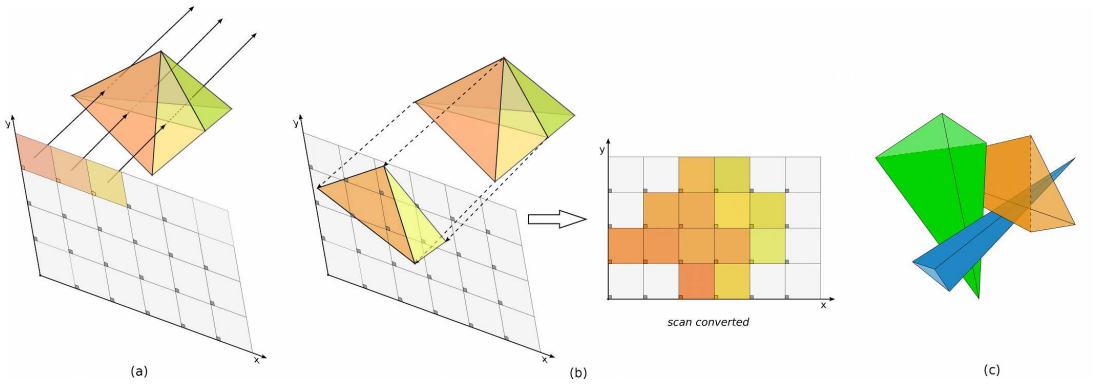

Figura 2.6: A técnica de emissão de raios é vista em *(a)*, na qual raios são disparados de cada *pixel* do plano da imagem sobre o espaço tridimensional. Os raios interceptam o volume de frente, atravessando-o; já em *(b)* a face é projetada do espaço tridimensional sobre o plano da imagem. A interseção raio / plano ocorre a partir dos *pixels* no *bounding box* desta projeção; em *(c)* três células tetrahedrais apresentando oclusão mútua (*visiblity cicle*). Para que a imagem resultante da projeção seja correta quanto a posição relativa das faces no espaço tridimensional é necessário que um segundo algoritmo seja responsável por ordená-las quanto à visibilidade.

O que o *ZSweep* faz é executar a tarefa de *rendering* volumétrico direto varrendo o dado volumétrico em ordem crescente de z com um plano paralelo ao plano da imagem, projetando as faces das células incidentes nos vértices do *dataset* a medida que estes são encontrados pelo plano de varredura.

Em uma primeira etapa, os vértices do *dataset* são ordenados em z de forma crescente em uma lista de eventos, implementada através de uma *heap*, o que abre a possibilidade para inserções na lista de forma dinâmica.

Na segunda etapa, o laço principal é executado, ou seja, caminha-se nesta lista de vértices, realizando a varredura propriamente dita, onde para cada vértice v_i, as faces de células incidentes no vértice são projetadas. A identificação dos *pixels* que interceptam a face bem como a coordenada z da interseção são obtidas no passo de *scan convert* durante a projeção. É neste ponto do *rendering* volumétrico por projeção de faces que focamos durante esta pesquisa, desenvolvendo o *B-Convert*, implementando-o nesta etapa do *ZSweep* e analisando seus resultados. As contribuições de cor e opacidade, bem como a profundidade (z) de cada interseção é salva em uma lista para cada *pixel* pertencente à projeção. Uma forma de prover uma projeção de faces de forma ordenada quanto à visibilidade no *ZSweep* ocorre justamente através do emprego eficiente do paradigma do plano de varredura durante esta etapa. Falaremos do plano de varredura mais adiante.

A terceira etapa do *ZSweep* consiste em compor de forma crescente em z, a cor e opacidade préviamente acumuladas gerando a imagem final. Isso é feito em dois momentos: sempre que o plano atinge o vértice de maior valor em z para o conjunto de faces que compartilham o vértice que iniciou o evento bem como após a varredura de todos os vértices na *heap*. Os valores de cor e opacidade acumulados são então compostos na

imagem final (*delayed compositing*). A Figura 2.7 ilustra um resumo deste processo.

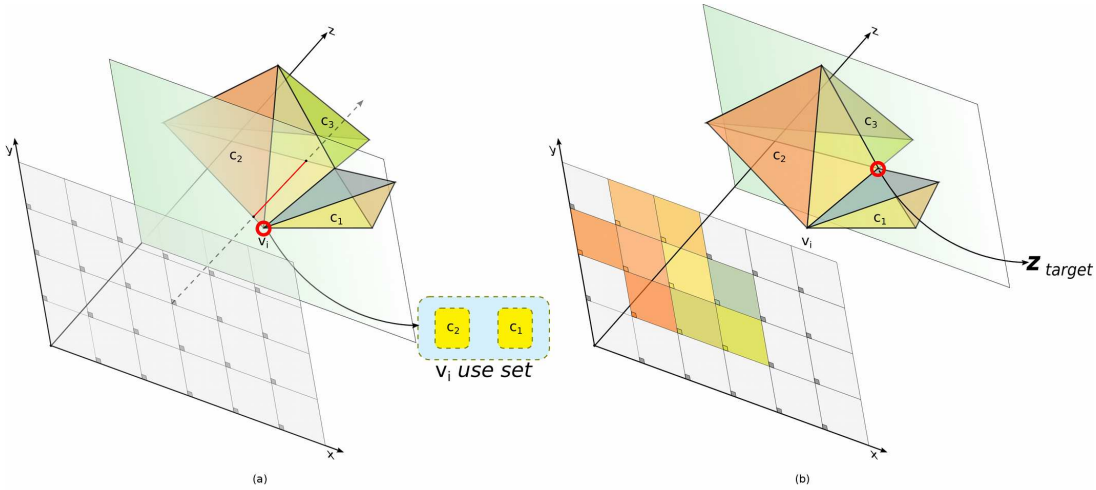

Figura 2.7: Em *(a)* tem inicio a projeção das faces incidentes no vértice varrido v_i. Isso é feito consultando uma lista de células cujas faces pertencem ao *use set* do vértice. São varridas as faces do vértice de menor coordenada z até o vértice de maior coordenada z (identificada aqui como z_{target}) pertencentes às faces do *use set* de v_i. Ao completar a varredura de todas estas faces, os valores acumulados durante a projeção são compostos no plano da imagem e o processo se repete para o próximo vértice não varrido na lista de eventos, até que todo o *dataset* tenha sido varrido.

Guardando similaridades com a projeção de faces, porém operando no espaço do objeto, existem algoritmos de projeção de células dentre os quais destaca-se o trabalho de SHIRLEY e TUCHMAN [16], eficiente algoritmo de projeção de células tetrahedrais. Com o advento de técnicas de *shaders* programáveis (e suas respectivas linguagens como *cg*, *glsl*) em unidades de processamento gráfico (*gpu*) bem como linguagens de programação genérica em *gpu* (*cuda* e *openCl*), alguns algoritmos foram estendidos para fazer uso das mesmas, aproveitando a capacidade computacional das placas gráficas para processar dados em ponto flutuante de forma rápida e eficiente. Em MAXIMO *et al.* [5], o algoritmo de projeção de tetrahedros [16] é inteiramente implementado em um único passo em *gpu*, inclusive realizando a ordenação por visibilidade também em *gpu*, resultando em *HAPT* (*Hardware Assisted Projected Tetrahedra*), o qual apresenta significativa vantagem computacional.

2.3.2 Equação de transferência

Como dito anteriormente, a visualização volumétrica procura demonstrar a interação entre a luz e o volume que se quer visualizar. Para isso, é necessário calcular a equação de transferência, responsável no caso, por resolver a integral de iluminação através do volume (*volume rendering integral*). Esta é a equação fundamental que modela o comportamento da luz em um meio capaz de absorver, emitir e dissipar radiação [17]. Define um

valor de cor, luz e opacidade para uma coordenada tridimensional qualquer pertencente ao volume. Existem vários modelos de iluminação como definido em MAX [18] que procuram implementar a equação de transferência ou integral de *rendering* [19], sendo que *ZSweep* utiliza o modelo de absorção mais emissão (*absorption plus emission*), que procura simular um volume como uma nuvem, onde e preciso levar em consideração tanto a luz que entra no volume quanto aquela gerada pelo brilho que as particulas do volume podem vir a emitir em contato com a luz passante ou por outra fonte luminosa. A equação (2.7) para absorção mais emissão apresentada em MAX [18] é definida por:

$$I(D) = I_0 \underbrace{T(D)}_{exp\left(-\int_0^D \tau(t)dt\right)} + \int_0^D g(s) \underbrace{T'(s)}_{exp\left(-\int_s^D \tau(t)dt\right)} ds \tag{2.7}$$

onde o primeiro termo representa a luz vinda do fundo (I_0) multiplicada pela transparência do meio volumétrico ($T(D)$) e o segundo termo representa a integral para a contribuição de $g(s)$ (*source term*), responsável no caso pela emissão luminosa ($\int_0^D g(s)ds$), multiplicada pela transparência existente entre a distância s e o plano da imagem ($T'(s)$). Maior aprofundamento sobre o assunto pode ser obtido em [4], [18], [19] e [20].

O cálculo da equação é implementado no *ZSweep* por uma integral que interpola linearmente os valores de cor e opacidade a medida que são encontrados, ao longo de cada raio. Estes valores, no intervalo de $[0, 255]$, oriundos de uma tabela definida pelo usuário, são mapeados nos pontos do *dataset* através de uma função de transferência. A interpolação linear é feita seguindo a Equação (2.8), onde o é a opacidade, c é a cor do *pixel*; z_c, c_c, o_c são respectivamente a profundidade, a cor e a opacidade do triângulo corrente na varredura; e z_n, c_n, o_n, os valores de profundidade, cor e opacidade do triângulo seguinte na interpolação linear.

$$\begin{aligned} o(z) &= \frac{o_c(z_n - z) + o_n(z - z_c)}{\Delta_z} \\ c(z) &= \frac{c_c(z_n - z) + c_n(z - z_c)}{\Delta_z} \end{aligned} \tag{2.8}$$

Uma análise mais detalhada do modelo de iluminação utilizado em *ZSweep* é encontrada em [21] e [15].

2.3.3 Emissão de raios (*ray casting*)

Apesar do *ZSweep* ser um algoritmo de projeção de faces, e não ser considerado um algoritmo de emissão de raios, é interessante neste trabalho descrever também a segunda técnica e suas características, já que consiste em uma das formas mais simples de se implementar o *rendering* volumétrico.

A técnica de emissão de raios (*ray casting*) em visualização volumétrica consiste em disparar raios a partir da câmera passando pelo plano da imagem (ao menos um por *pixel*)

sobre o dado volumétrico, calculando a integral de iluminação a medida que o raio passa pelo volume, interpolando os valores de opacidade e brilho para cada ponto de interseção entre o raio e as células que compõem o volume. Dentre os algoritmos para visualização volumétrica, *ray casting* é o método númerico mais objetivo para avaliar a integral de *rendering*, que geralmente é implementada como uma soma de *Riemann* devido à natureza discreta dos dados. A cada intervalo (seja equidistante ou definido por um evento) ao longo do raio, o dado volumétrico é re-amostrado e interpolado com a amostragem anterior a partir do primeiro ponto de interseção do raio com o volume. A acumulação termina após o raio sair do volume, sendo geralmente implementado partindo do plano da imagem (*front to back order*). Ao contrário do algoritmo de traçado de raios (*ray tracing*), o algoritmo de *ray casting* não emite raios secundários a partir das interseções entre o raio e o dado.

Um dos trabalhos seminais ligando a técnica de traçado de raios à renderização volumétrica é apresentado em BLINN [22]. Procurando simular uma nuvem de poeira, mais precisamente os anéis de saturno, o autor expõe os conceitos físicos necessários para modelar a geração de imagem através da interação entre um volume e raios luminosos.

Uma das primeiras técnicas de traçado de raios para *rendering* volumétrico, foi apresentada em GARRITY [23], nela, procura-se representar volumes definidos por grades irregulares estruturadas em células convexas, expondo a abordagem de se computar as interseções do raio com as faces das células que compõem o volume. A partir da interpolação dos valores presentes nestas faces, extraidos nos pontos de interseção, é calculada a contribuição final da célula interceptada à imagem final. No trabalho de Garrity [23], também são discutidas formas de otimizar o traçado de raios, testando no início, apenas a interseção com faces expostas, em células de fronteira, isto é, faces que pertencem a apenas uma célula, garantindo que esta será realmente a primeira contribuição para um raio emitido. Após visitar as células de fronteira, o raio visita as células internas do volume até sair do mesmo em outra face exposta.

Outra abordagem para emissão de raios pode ser vista em BUNYK *et al.* [21], onde as células são decompostas em faces e para cada *pixel* é criada uma lista de faces de fronteira, projetadas no espaço da tela (plano da imagem). Estas listas de faces ordenadas por visibilidade são usadas para o *ray casting* propriamente dito. O processo de escolha da próxima face a ser testada quanto à interseção com o raio em cada *pixel* se dá através da informação de conectividade presente nas células. A proposta de RIBEIRO *et al.* [24] reduz consideravelmente o custo de memória e melhora o tempo de execução em relação à BUNYK *et al.* [21]. Apesar de não ser mais rápido que este, apresenta maior corretude na imagem final, tratando casos degenerados. Denominada *VF-Ray (Visible Face Ray-Casting)*, procura manter na memória apenas informação das faces recentemente atravessadas, explorando coerência dos raios para isso.

Devido à natureza paralela tanto da técnica de emissão de raios quanto da técnica de

traçado de raios, foram propostos algoritmos para aceleração por *hardware* como o *HARC* (*Hardware-Based Ray Casting*) [25],[26]. O algoritmo *VF-Ray*[24] também foi estendido para diferentes arquiteturas de *hardware*. RIBEIRO *et al.* [27] produz uma versão fortemente auxiliada por *gpu* enquanto COX *et al.* [28] apresenta uma implementação para *cell broadband engine*. É interessante notar que nestas pesquisas, a necessidade de se trabalhar com arquiteturas distintas da *x86* resultou em novas abordagens no que tange estruturas de dados e utilização de memória, contribuindo para algoritmos mais eficientes.

Mais recentemente, SCHUBERT e SCHOLL [29] apresenta uma comparação de técnicas de *ray casting* acelerado por *GPU* aplicadas sobre múltiplos volumes ao mesmo tempo.

2.3.4 Plano de Varredura (*sweep plane*) em *Rendering* Volumétrico

Procurando mitigar uma limitação das técnicas de emissão / traçado de raios, mais precisamente o *não aproveitamento da coerência entre raios*, que implica em repetir a computação de todas as interseções entre raios provenientes de *pixels* adjacentes com praticamente as mesmas células, GIERTSEN [30] adaptou o algoritmo de varredura visto em geometria computacional [31], para visualização volumétrica [1],[4]. Nele, um plano perpendicular ao plano da imagem varre o modelo, sendo que o disparo de novos raios é executado apenas mediante ocorrência de certos eventos, como por exemplo novas interseções entre vértices do modelo e o plano de varredura. Isso resulta em uma atualização incremental das interseções. O uso eficaz de eventos no algoritmo é possível justamente pela coerência dos raios. Posteriormente SILVA e MITCHELL [32] implementam uma segunda varredura no algoritmo *Lazy Sweep*, desta vez sobre o plano, utilizando uma linha de varredura paralela ao eixo z e fazendo ainda melhor uso de memória e tempo computacional.

O trabalho de REED *et al.* [33] recorre também a um plano de varredura, mas desta vez paralelo ao plano da imagem, e assistido por *hardware*, integrando os dados de cor e opacidade de células tetrahedrais de uma grade irregular entre varios planos paralelos ao longo do volume. A utilização de um plano paralelo ao plano da imagem é parte fundamental do algoritmo *ZSweep*[15], no qual o paradigma de plano de varredura é empregado para auxiliar na ordenação de faces *durante a projeção*, sendo que a cada evento (interseção de um vértice com o plano), varrem-se as faces às quais o vértice pertence, projetando-as na tela. Ao caminhar na lista de eventos, estamos efetivamente definindo diferentes posições do plano no eixo z. Essa é a função do plano de varredura no *ZSweep*, sua contribuição ao algoritmo reside, não na coerência dos raios, algo natural à projeção de faces, mas na ordenação de visibilidade, de modo a evitar oclusão mútua entre as células do volume. A Figura 2.8 demonstra a diferença na utilização do plano de varre-

dura em dois algoritmos de *rendering* volumétrico.

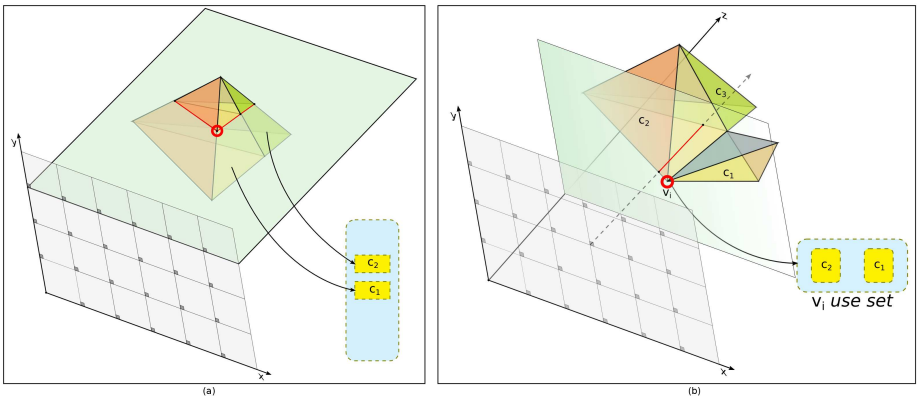

Figura 2.8: O modelo de varredura de Giertsen é visto em *(a)*, a lista *ACL* contem as células varridas em uma determinada linha. Ao passar para outra linha, esta lista é processada e é possível explorar a coerência de seu conteudo para a varredura seguinte. Já em *(b)*, é possível ver um modelo simplificado do plano de varredura empregado no algoritmo *ZSweep*, onde um evento é determinado pelo encontro de um vértice durante a varredura, quando as faces das células contidas no *use set* do mesmo são projetadas. A composição é feita posteriormente.

O conceito de varredura aplicado à visualização volumétrica também é empregado por SUNDEN *et al.* [34], onde os pesquisadores exploram o paradigma do plano de varredura empregando-o na iluminação de dados volumétricos acoplada a um *raycaster*.

Capítulo 3

B-Convert: projeção de faces por lista compacta

O enfoque deste trabalho é pesquisar uma nova aplicação do algoritmo de traçado de retas desenvolvido por *Bresenham*, inserindo-o no processo de rasterização (*scan convert*) em um algoritmo de visualização volumétrica por projeção de faces, o *ZSweep*, usado aqui como estudo de caso.

Durante o processo de *scan convert* de uma face é necessário executar um teste de interseção raio / plano sobre todos os *pixels* do seu *bounding box*. Este teste visa selecionar apenas os *pixels* cujas coordenadas transformadas para o espaço tridimensional interceptam a face descrita neste mesmo sistema de coordenadas. A hipótese que nos levou à presente pesquisa é a de que ao prover uma representação mais precisa da face no plano da imagem como domínio para o *scan convert* estaríamos evitando testes desnecessários de interseção. Nosso objetivo é responder às seguintes perguntas:

- Gerar e rasterizar uma representação mais próxima da face triangular oferece vantagem computacional?
- A visualização é feita de forma correta?
- Em quais condições esta abordagem é melhor que a abordagem convencional?

Para respondê-las, implementamos nosso algorítmo no *ZSweep* e analisamos os resultados. Utilizamos *datasets* compostos por células tetrahedrais por permitirem uma manipulação mais simples do que células hexahedrais. Como estas últimas também podem ser decompostas em tetrahedros, nos concentramos no momento apenas nas primeiras. Cada uma das 4 faces de uma célula tetrahedral poderá resultar em um triângulo a ser projetado no plano da imagem. Identificamos 3 situações distintas para este triângulo quanto ao desempenho durante a etapa de *scan convert*. O caso ótimo no *scan convert* clássico, visto na Figura 3.1(a) indica que 50% dos *pixels* presentes na *bounding box* da face projetada no plano da imagem não serão utilizados na composição final; o caso

genérico, apresentado na Figura 3.1(b) tem um aproveitamento inferior à 50% dos *pixels*; já o caso péssimo, observado na Figura 3.1(c) pode apresentar perda de até 100% dos *pixels*, podendo resultar em nenhuma contribuição da face ao *rendering*.

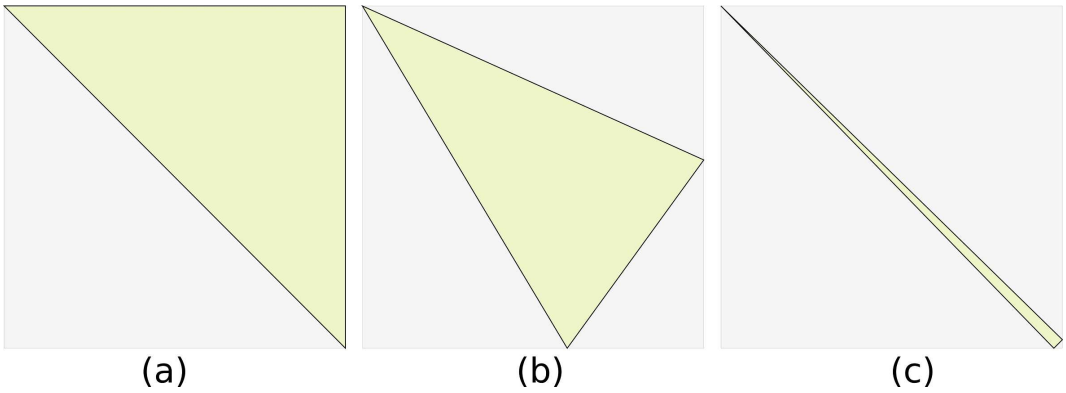

Figura 3.1: Tipos de triângulo quanto à projeção da face: em *(a)* temos o melhor caso, com 50% dos *pixels* perdidos. No caso geral, visto em *(b)*, mais de 50% dos *pixels* são perdidos. Já no caso péssimo, pode haver perda de até 100% dos *pixels (c)*.

O método proposto pode reduzir o tempo de *scan convert* das faces já que o custo computacional de testar uma linha inteira da *bounding box* pode ser usado para rasterizar quantidade equivalente de *pixels* provenientes da lista que contém a representação mais próxima da face no plano da imagem, denominada de *lista compacta*. Assim o tempo gasto para rasterizar uma linha (Figura 3.2) é empregado na rasterização de 4 linhas da lista compacta (Figura 3.3) trazendo mais resultados (interseções) no mesmo intervalo de tempo. Apesar de não eliminar o pior caso, nossa proposta busca reduzir o custo computacional gasto com testes que fatalmente resultarão em não interseção. Ao realizarmos o *scan convert* sobre uma representação mais próxima da projeção real da face, reduz-se a quantidade de falsos candidatos durante a rasterização, resultando em uma amostra com uma quantidade maior de interseções raio / plano para a face em questão.

Claramente há nessa proposta um custo computacional maior em gerar a lista compacta do que simplesmente obter os limites do *bounding box* da face. Nesta pesquisa, também buscamos identificar casos em que o custo computacional total de se gerar a lista compacta e testar cada *pixel* contido nela durante a etapa de *scan convert* é inferior ao custo de se testar todos os *pixels* do *bounding box*.

3.1 O algoritmo

O *B-Convert* é dividido em três partes. Na primeira parte geramos as listas das arestas através do algoritmo de traçado de retas. Na segunda parte, *compactamos* estas 3 listas em uma única lista representando a face através de intervalos mais precisos do que aqueles definidos pelo *bounding box*. Por fim, o processo de *scan convert* é então executado sobre os diferentes intervalos definidos nesta lista compacta.

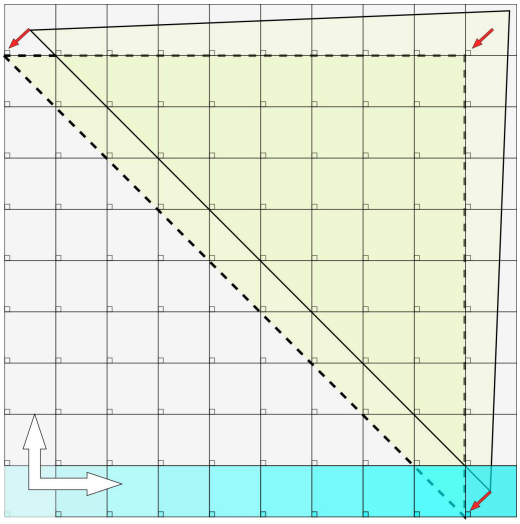

Figura 3.2: *Scan convert* sobre o *bounding box* da face no caso ótimo. O *scan* de uma linha pode não encontrar sequer um *pixel* que intercepta a face.

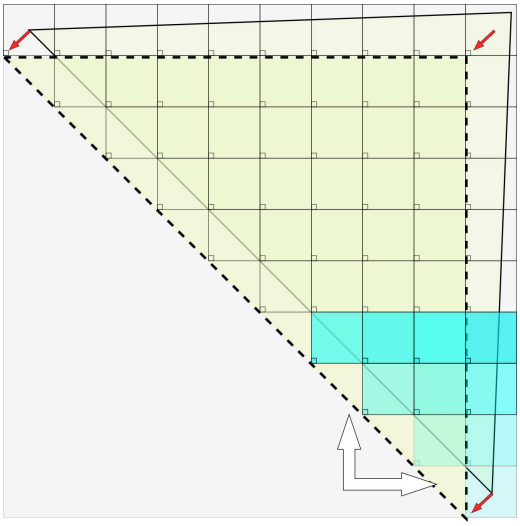

Figura 3.3: *Scan convert* sobre a lista compacta. No mesmo tempo computacional é possível encontrar ao menos 6 *pixels* que interceptam a face.

A idéia principal é:

- Gerar 3 listas, representando as arestas da face no plano da imagem: $l_0,\ l_1,\ l_2$.
- Compactar estas listas em uma lista l_c com um total de $l_2.\Delta_y$ elementos, cada um contendo intervalos $[x_{inicial}, x_{final}]$.
- Realizar o passo de *scan convert* sobre a lista compacta l_c.

Assim, supondo uma face triangular f, sua representação no espaço tridimensional pode ser definida por três vértices: $w_0,\ w_1,\ w_2\ \in W$, W sub-espaço de $\mathbb{R}^3$. Existe uma representação equivalente de f no plano da imagem definida pelos *pixels* $v_0,\ v_1,\ v_2\ \in S$, S sub-espaço de $\mathbb{Z}^2$, onde, *inicialmente*, v_0 é a coordenada correspondente à w_0 e assim sucessivamente.

O primeiro passo é ordenar estes *pixels* para obtermos $v_0.y \leq v_1.y \leq v2.y$. A partir destas novas coordenadas, geramos 3 listas de *pixels* representando as arestas $l_0 : [v_0, v_1]$, $l_1 : [v_1, v_2]$ e $l_2 : [v_0, v_2]$. Compactamos então as listas das arestas em uma única lista l_c de intervalos em y e em x. Esta lista conterá $l_2.\Delta_y + 1$ elementos, onde o primeiro elemento definirá o intervalo $[y_{inicial}, y_{final}]$ que abrange a face no plano da imagem. Este intervalo também define a quantidade de elementos restantes na lista. Cada um destes contendo um par $[x_{inicial}, x_{final}]$, respectivamente a menor e a maior abscissa x para cada ordenada y do intervalo estabelecido no primeiro elemento. A Figura 3.4 indica os passos para a construção da lista compacta e posterior rasterização.

3.1.1 Geração das listas das arestas

O algoritmo de *Bresenham* para traçado de retas é parte essencial desta etapa. Ele foi adaptado para armazenar as coordenadas do primeiro e último *pixel* escolhido de cada ordenada y no intervalo entre os pontos extremos de uma aresta. Ao contrário do propósito original do algoritmo de traçado de retas que executa uma operação sobre todos os *pixels* escolhidos, armazenar as coordenadas de *pixels* dentro de um segmento de mesma ordenada y é desnecessário para nosso objetivo além do que acarretaria em maior custo computacional.

Para cada face do dado volumétrico, seus vértices no espaço tridimensional são lidos e convertidos para coordenadas no plano da imagem, onde estes pontos são reordenados

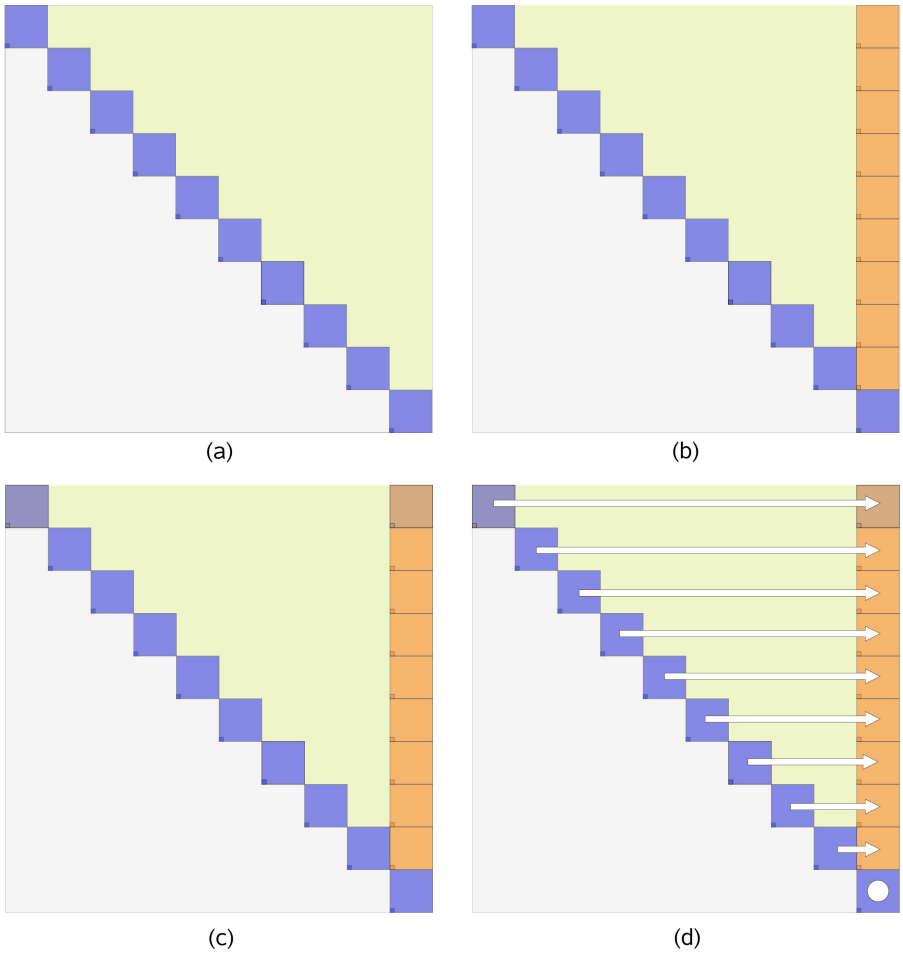

Figura 3.4: Geração da lista compacta de *pixels* que melhor representam a projeção da face triangular para o caso ótimo. Primeiro, obtem-se a representação da aresta l_2 *(a)*, depois a aresta l_0 *(b)*. Em *(c)* obtemos apenas os dois pontos extremos da aresta l_1, já que a mesma pode ser representada em apenas uma linha da matriz de *pixels* definida pela *bounding box* da face no caso ótimo. A rasterização (*scan convert*) é feita testando, para cada linha na lista compacta, a interseção entre a face e as coordenadas de cada *pixel* no intervalo $[x_{inicial}, x_{final}]$, transformadas para o sistema de coordenadas do espaço tridimensional (d).

de forma crescente em y. Dentre as 3 listas l_0, l_1 e l_2, elegemos l_2 para ser *sempre* a lista contendo *pixels* no intervalo $[v_0, v_2]$, ou seja contendo a aresta de maior intervalo Δ_y. No caso genérico, esta será a aresta dominante da face enquanto as outras duas arestas contidas nas listas l_0 e l_1 combinadas abrangem o mesmo intervalo Δ_y de l_2.

Visando otimizar a etapa de compactação, nossa implementação do algoritmo de traçado de retas (Listagem A.2) retorna sempre um par de coordenadas a cada linha. Isso garante que sempre haja comparação de exatamente um par de *pixels* por ordenada y na etapa de compactação. Esta modificação implica na necessidade de tratar de forma distinta dois casos de *pixels* durante a geração da lista da aresta. O primeiro caso refere-se ao primeiro *pixel* escolhido em uma dada ordenada y enquanto o segundo caso refere-se a qualquer *pixel* subsequente para o mesmo valor de y.

No caso de estarmos iniciando a análise de quais *pixels* melhor representam a reta real para uma ordenada qualquer y_k quando o eixo x (num sistema cartesiano) for o eixo dominante, uma variável de controle *first* é usada para inserir sempre o par de coordenadas (x_i, y_k) do primeiro *pixel* escolhido. Após inserir as coordenadas do primeiro *pixel*, esta *flag* é marcada como falsa (ver Listagem 3.1) e o algoritmo de *Bresenham* continua sua execução escolhendo os próximos *pixels* sem, no entanto, armazená-los na lista neste momento.

```
while (x1 != x2) {
  if (first) {
    buf.push_back(D2Pointi(x1, y1));
    first = false;
  }
  (...)
```

Listing 3.1: Primeiro *pixel* de uma linha

Somente ao entrar no laço que incrementa o eixo passivo, as coordenadas atuais x_{i+j}, y_k , onde $j > 0,\ j \in \mathbb{Z}$, são inseridas *antes da incrementação do eixo passivo propriamente dita* (y_{k+1}). A inserção é precedida de uma verificação contra as últimas coordenadas armazenadas (x_i, y_k) de modo a que a abscissa de maior valor, seja ela x_i ou x_{i+j} ocupe sempre a segunda posição para a mesma ordenada y_k (Listagem 3.2).

```
(...)
if (e >= 0) {
  if (!first) {
    D2Pointi tmp = buf.back();
    if (tmp.x > x1 && tmp.y == y1) {
      buf.pop_back();
      buf.push_back(D2Pointi(x1, y1));
      buf.push_back(tmp);
    } else {
      buf.push_back(D2Pointi(x1, y1));
    }
  }
  y1 += iny;
  e -= dx;
  first = true;
}
(...)
```

Listing 3.2: Último *pixel* de uma linha

Quando o eixo y for o eixo dominante, não há a necessidade de se ordenar o par de coordenadas por ordem crescente de abscissa, já que apenas um *pixel* é inserido na lista da aresta a cada incremento no eixo das ordenadas. É necessário apenas inserir novamente as mesmas coordenadas no *buffer* para obedecer à regra de compactação de um par de coordenadas para cada ordenada y no intervalo Δ_y da face. (3.3).

```
(...)
while (y1 != y2) {
  buf.push_back(D2Pointi(x1, y1));
  buf.push_back(D2Pointi(x1, y1));
  (...)
```

Listing 3.3: Inserção dupla de *pixel* para uma aresta onde $\Delta_y > \Delta_x$

Ao fim dos laços para $\Delta_x \geq \Delta_y$ ou $\Delta_y \geq \Delta_x$, dois casos distintos devem ser analisados: se houve uma reta desenhada ou apenas um ponto. No caso de algoritmo desenhar somente um ponto devido a um caso degenerado com dois vértices identificados pelo mesmo *pixel* no plano da imagem, nenhuma inserção terá sido feita pois as condições anteriores ($x_1 \neq x_2,\ y_1 \neq y_2$) não serão satisfeitas, sendo necessário outro tratamento. Neste caso, as coordenadas (x_1, y_1) são inseridas no buffer duas vezes (a *flag first* ainda será verdadeira). No caso de uma reta ter sido traçada, é preciso tratar a última inserção do ponto extremo final, verificando, *fora do laço principal*, o valor da abscissa do último elemento inserido. Isso é necessário para garantir a ordenação crescente por x para o último *pixel* da última ordenada y. O código fonte para estes tratamentos é listado em 3.4.

```
(...)
if (buf.size()) {
  D2Pointi tmp = buf.back();
  if (tmp.x > x1 && tmp.y == y1) {
    buf.pop_back();
    buf.push_back(D2Pointi(x1, y1));
    buf.push_back(tmp);
  } else {
    buf.push_back(D2Pointi(x1, y1));
  }
} else { // if(!buf.size())
  buf.push_back(D2Pointi(x1, y1));
}
if (first) // if (x1==x2 || y1==y2) && buf.size()==1
  buf.push_back(D2Pointi(x1, y1));
(...)
```

Listing 3.4: Última inserção e caso degenerado

Nesta fase, o algoritmo de traçado de retas é aplicado diretamente sobre os *pixels* correspondentes aos vértices da face, gerando uma representação *aproximada* das arestas na matriz de *pixels*. Deve-se notar que, ao caminhar de um ponto extremo ao outro, o algoritmo de traçado de retas *escolhe* qual *pixel* melhor representará a reta real, decidindo por incrementar (ou não) o eixo passivo além do eixo dominante. Esta característica natural do algoritmo traz consequências que serão discutidas no Capítulo 4.

3.1.2 Geração da lista compacta

Cada lista de aresta fornece duas abscissas x_i e x_f, tal que $x_i \leq x_f$ para cada ordenada y. Isso permite ao *B-Convert* montar uma lista com os diferentes intervalos que contém a projeção da face verificando apenas pares de valores a cada iteração do laço de compactação.

Nas primeiras versões do *B-Convert*, utilizavamos, uma única lista contendo os *pixels* das 3 arestas. Esta lista era então reordenada em uma tabela *hash*, com sua chave *hash* gerada a partir das duas coordenadas de cada *pixel*. Após a montagem da tabela, eram eliminadas as coordenadas repetidas e cada par de elementos da tabela tinha suas respectivas abscissas armazenadas em ordem crescente em um elemento de uma terceira lista.

Na versão atual, comparamos as 3 listas das arestas *durante a geração da lista compacta*, sem que haja necessidade de calcular uma função *hash* ou remover coordenadas repetidas.

O procedimento necessário para a compactação é simples.

- Carregar os vértices $w_0,\ w_1,\ w_2$ da face f.
- Converter suas coordenadas para o plano da imagem ($w_0 \rightarrow v_0,\ w_1 \rightarrow v_1,\ w_2 \rightarrow v_2$).

- Reordenar estes *pixels* em ordem crescente de ordenada ($v_0.y \leq v_1.y \leq v_2.y$).
- Executar o traçado de retas sobre $\underbrace{[v_0, v_1]}_{l_0}$, $\underbrace{[v_1, v_2]}_{l_1}$ e $\underbrace{[v_0, v_2]}_{l_2}$.
- compactar as 3 listas de arestas em uma única lista de intervalos.

A compactação em si é feita da seguinte forma: após reordenar os *pixels* contendo vértices da face de tal forma que $v_0.y \leq v_1.y \leq v_2.y$, o par $(v_0.y, v_2.y)$ é armazenado na lista. Ou seja, o primeiro elemento contém o intervalo Δ_y da face no plano da imagem. Os elementos subsequentes da lista compacta serão definidos a partir das listas das arestas l_0, l_1 e l_2. Da análise destas listas, serão formados os pares $(x_{inicial}, x_{final})$ para cada y contido no intervalo definido pelo primeiro elemento da lista compacta. Logo, para uma face f com vértices $v_0,\ v_1,\ v_2$ extraímos os valores mínimo e máximo da face no eixo y e para cada iteração dentro deste intervalo, comparamos os valores das abscissas presentes nas listas das arestas l_0 com l_2 e l_1 com l_2, armazenando na lista compacta, apenas as abscissas mínima e máxima resultantes da comparação. Um exemplo disto é visto na Figura 3.5, onde a ordenada y_k apresenta o exemplo mais simples, onde cada aresta contribui com um *pixel* apenas para a comparação. Na ordenada y_{f-1} é apresentado um exemplo onde a aresta l_1 contribui com duas coordenadas distintas, sendo que o elemento correspondente na lista compacta l_c é formado pela contribuição das abscissas mínima da lista l_2 e máxima de l_1 relativas à ordenada y_{f-1}.

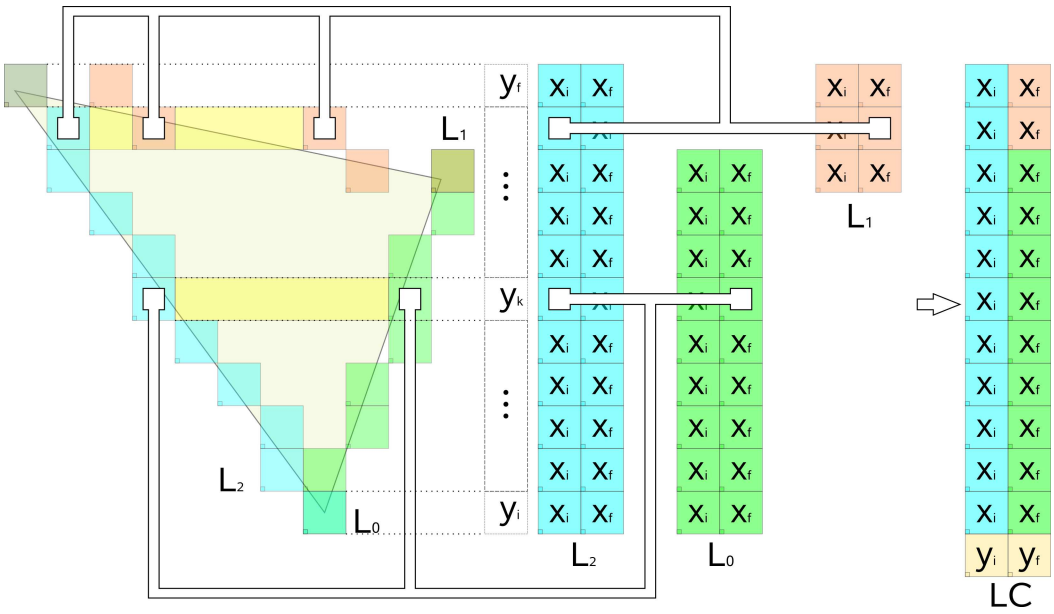

Figura 3.5: Esquema de geração da lista compacta de *pixels* da face a partir das listas de *pixels* das arestas. Note a repetição nas listas das arestas da mesma coordenada quando há apenas uma contribuição para uma determinada linha. Isso é feito de modo a otimizar a geração da lista compacta na segunda parte do algoritmo.

O intervalo Δ_y da face é necessariamente o intervalo Δ_y da lista l_2, já que esta encerra as coordenadas da aresta descrita entre os vértices v_0 e v_2.

Procuramos compactar as listas de arestas da forma mais genérica possível, isto é, casos particulares como $l_0.\Delta_y = l_2.\Delta_y$ (Figura 3.4) ou $l_1.\Delta_y = l_2.\Delta_y$ são compactados da mesma forma que o caso genérico $l_2.\Delta_y = (l_0.\Delta_y + l_1.\Delta_y) - 1$ onde $l_0.\Delta_y < l_2.\Delta_y$ e $l_1.\Delta_y < l_2.\Delta_y$. Para isso, o algoritmo utiliza dois laços, um comparando as coordenadas da lista l_2 e l_0 dentro do intervalo $l_0.\Delta_y$ e outro comparando as listas l_2 e l_1 dentro do intervalo $l_1.\Delta_y$. Neste último, iniciamos a lista l_2 a partir de sua ordenada y de índice $l_0.\Delta_y$. É importante notar a necessidade de tratar a linha que encerra o vértice v_1 que será repetido nas listas l_0 e l_1. Teoricamente poderíamos ignorar o elemento inicial de l_1, pois um *pixel* de mesmas coordenadas já foi analisado. Escolhemos no entanto verificar os valores das abscissas de l_2 e l_1 contra a última inserção na lista compacta, para assegurar a inclusão de $(x_{inicial}, x_{final})$ com $x_{inicial} \leq x_{final}$.

Para cada linha i em um determinado intervalo Δ_y, o valor da abscissa do elemento de índice par $(2i)$ de uma lista é comparado com o elemento de mesmo índice par na outra lista. A lista que possuir o elemento par de maior valor contribuirá com a abscissa do elemento impar imediatamente seguinte $(2i + 1)$. As abscissas são armazenadas em um único elemento na lista compacta em ordem crescente. Ao final do primeiro laço comparamos o último elemento compactado com o primeiro elemento da terceira lista. O segundo laço é percorrido de forma idêntica ao primeiro. Respeitando, é claro, os índices dos elementos dentro das respectivas listas quanto ao intervalo $l_1.\Delta_y = v_2.y - v_1.y$.

É necessário atentar que o algoritmo é sensível à forma do triângulo da face. Isso pode ser percebido no algoritmo de compactação, no qual, considerando um sistema de coordenadas cartesiano, tomamos a posição relativa entre as abscissas dos *pixels* correspondentes aos três vértices da face, bem como o valor em módulo dos coeficientes angulares das arestas l_0 e l_2 para delinearmos 3 casos distintos de compactação, resumidos na Figura 3.6.

O primeiro caso, o mais simples, trata a projeção de uma face representada em apenas uma linha. Dado que $v_0.y = v_1.y = v_2.y$, apenas um segundo elemento é adicionado à lista compacta contendo os valores mínimo e máximo entre $v_0.x,\ v_1.x,\ v_2.x$ (o primeiro elemento é o par $(v_0.y, v_2.y)$).

Tanto o segundo quanto o terceiro caso tratados na compactação referem-se a posição relativa entre as listas l_2 e l_0 (e consequentemente l_1) durante a rasterização de uma determinada linha. No segundo caso, na maior parte do tempo, l_2 contribuirá com $x_{inicial}$ enquanto x_{final} será obtido a partir das coordenadas nas listas l_0 ou l_1. A lista l_1 não precisa ser testada neste momento pois seu extremo inicial é o extremo final de l_0, estando à esquerda ou à direita de l_2 junto com l_1. Três condições distintas levam ao segundo caso:

1 O vértice v_1 possui a abscissa de maior valor ($v_1.x \geq v_0.x$ e $v_1.x \geq v_2.x$).

2 $m_{20} \geq m_{10}$ e $v_0.x \leq v_1.x \leq v_2.x$. Onde m_{20} é coeficiente angular de l_2 e m_{10} o coeficiente angular de l_0.

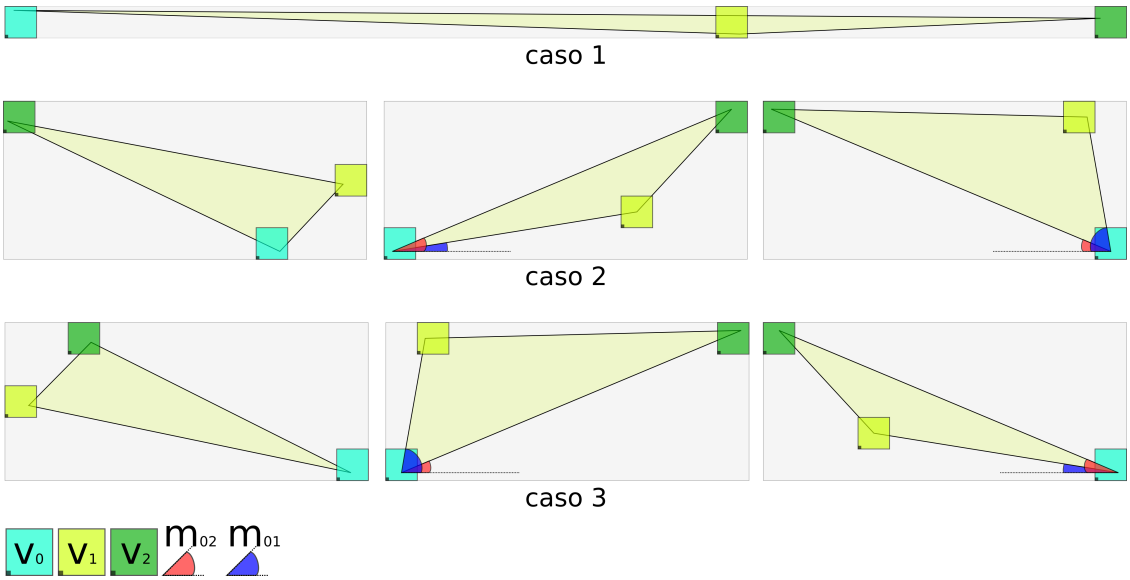

Figura 3.6: Os três casos distintos de compactação da face são obtidos a partir de 7 formas distintas do triângulo da face.

3 $m_{20} \leq m_{10}$ e $v_2.x \leq v_1.x \leq v_0.x$.

O terceiro caso é o espelhamento do segundo e analogamente:

1 O vértice v_1 possui a abscissa de menor valor ($v_1.x \leq v_0.x$ e $v_1.x \leq v_2.x$).

2 $m_{20} \leq m_{10}$ e $v_0.x \leq v_1.x \leq v_2.x$.

3 $m_{20} \geq m_{10}$ e $v_2.x \leq v_1.x \leq v_0.x$.

Em ambos os casos, armazenamos no primeiro elemento da lista compacta o par $(y_{inicial}, y_{final})$. Em seguida percorremos cada elemento de l_2 e l_0 no intervalo $\Delta_y = v_1.y - v_0.y$ (primeiro laço do caso genérico), salvando na lista compacta o par $(l_2.x_{inicial}, l_0.x_{final})$ ou $(l_0.x_{inicial}, l_2.x_{final})$ para cada ordenada do intervalo.

Vale lembrar que durante a implementação, o algoritmo de *Bresenham* foi modificado para *sempre armazenar duas coordenadas por linha, em ordem crescente de abscissa*. Dessa forma, garantimos que para cada elemento $2i$ par, nas listas das arestas, haverá um elemento $2i + 1$ impar, *de mesma ordenada*. Logo, cada elemento i, tal que $i \in [l_0.\Delta_y]$, na lista compacta l_c será composto de $(l_{2_{2i}}.x, l_{0_{2i+1}}.x)$ ou $(l_{0_{2i}}.x, l_{2_{2i+1}}.x)$ como é visto na Figura 3.7. Antes do segundo laço, o primeiro elemento da lista l_1 é testado contra o último par compactado de forma a garantir uma compactação correta. Após isso, os elementos de l_2 e l_1 são também comparados no intervalo restante $\Delta_y = v_2.y - (v_1.y + 1)$.

A modificação do algoritmo de *Bresenham* para retornar os *pixels* de menor e maior abscissas para cada ordenada garante, durante a compactação, uma ordenação correta evitando ao menos um *branch* para cada linha. Logo após o teste necessário para encontrar os *pixels* mínimo e máximo, são utilizados os elementos $2i$ e $2i + 1$ das listas, já que

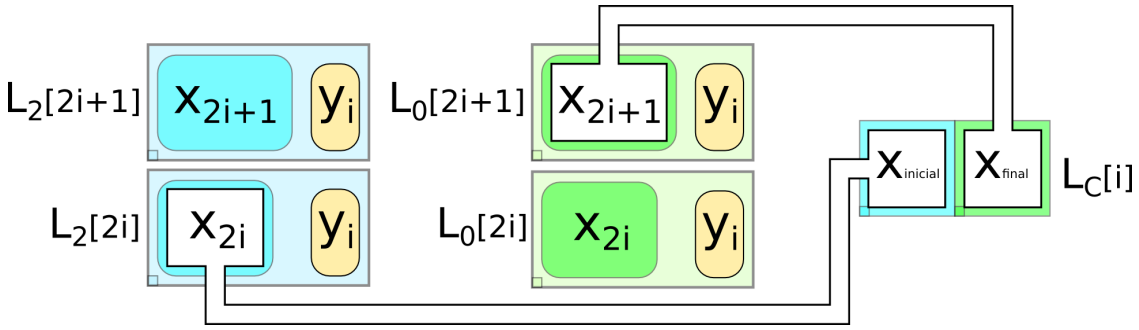

Figura 3.7: Critério para contribuição das listas das arestas no primeiro intervalo Δ_y do segundo caso de compactação. O elemento $l_{0_{2i+1}}.x$ (x_{final}) será sempre uma abscissa de maior valor que o elemento $l_{0_{2i}}.x$ ($x_{inicial}$) da mesma lista. No segundo caso, apesar de termos a lista l_2 percorrendo a amostra "sempre à esquerda"de l_0, é necessário confirmar que $l_{2_{2i}}.x \leq l_{0_{2i}}.x$ para então salvar na lista compacta o par $x_{inicial}$ vindo de $l_{2_{2i}}$ e x_{final} de $l_{0_{2i+1}}$.

durante a geração das mesmas, a inserção no *buffer* é feita sempre em ordem crescente de abscissa *para uma determinada ordenada*. Basicamente o conceito de um par de *pixels* para cada linha norteia todas as etapas da geração de lista compacta da face no *B-Convert*. A metodologia utilizada na compactação pode ser vista no Algoritmo implementado na Listagem A.1.

3.1.3 Utilizando os vértices originais no espaço tridimensional

Após os primeiros resultados verificamos a presença de artefatos nas imagens resultantes. Isto será melhor discutido na Seção 4.2. Para fornecer mais dados sobre as condições de formação destes artefatos decidimos gerar as listas das arestas a partir das coordenadas dos vértices no espaço tridimensional, o que exigiu alterações em ambas as fases do algoritmo, tanto no desenho das arestas quanto na geração da lista compacta.

Trabalhar diretamente nas coordenadas dos vértices das faces do modelo, já transformadas para o sistema de coordenadas *do mundo* implica em manipular números reais (ponto flutuante) ao invés de inteiros ao menos na fase de geração das listas de arestas. Isso acarreta um custo maior de programação que consideramos justificável para pesquisar formas de identificar as causas e tratar os artefatos na visualização. Como nossa proposta também consiste em pesquisar os problemas oriundos da utilização do algoritmo de traçado de retas de *Bresenham* para a geração da lista compacta, optamos por efetivamente implementar uma versão em números reais (ponto flutuante), ainda que isso vá de encontro à própria natureza do mesmo. Para fins de validação utilizamos a própria equação da reta na forma reduzida bem como o algoritmo *DDA*. Durante a geração das listas das arestas, o valor usado para incremento/decremento não mais consistirá de um *pixel*, nem do valor referente à escala do *pixel* no espaço tridimensional.

Falar em *pixel* como regiões de um espaço real é um modelo mental que pode induzir

à erros. *Pixels* podem ser melhor entendidos como amostragens discretas de uma função contínua, ou de uma função préviamente discretizada em dados de tipo ponto flutuante. No entanto, é mesmo a idéia de um determinado δ_s no espaço tridimensional, como equivalente à distância entre duas amostras no plano da imagem (*pixels*) que nos interessa. Não para substituir diretamente o valor unitário do incremento / decremento no algoritmo de traçado de retas, mas para lembrar que é necessário definir um determinado valor de passo tanto em x quanto em y para caminharmos na reta. Ao executarmos o algoritmo de traçado de retas sobre as coordenadas dos vértices da face no espaço tridimensional, estaremos inicializando nossa lista de amostras da aresta *em qualquer ponto do espaço* $3D$. Assim, é necessário encontrar um valor incremental, um $step_x$ e $step_y$ particular à cada aresta, calculado cada vez que o algoritmo é chamado. Se considerarmos dois pontos no espaço tridimensional, p_1 e p_2, o valor do incremento usado no desenho da reta entre estes dois pontos é dado pela equação (3.1) para um dado eixo $\overrightarrow{OI}$.

$$step_i = \frac{\Delta_{i_{mundo}}}{\left(|\Delta_{i_{imagem}}| + 1\right)} \tag{3.1}$$

Onde $\Delta_{i_{mundo}} = p_2.i - p_1.i$ no espaço tridimensional e $\Delta_{i_{imagem}} = p_2.i - p_1.i$ no plano da imagem. A partir disso podemos iniciar a amostragem da reta no espaço tridimensional. A forma reduzida da equação da reta (2.3) e o algoritmo *DDA* 1 já foram descritos no capítulo anterior, sendo que sua variante no espaço tridimensional é apresentada na Listagem A.6.

Implementamos também uma versão do algoritmo de *Bresenham* para o espaço tridimensional (Listagem A.7) como exercício de comparação com o algoritmo *dda* modificado (Listagem A.6), de modo a verificar diferenças entre o uso direto do coeficiente angular (*dda*) na amostragem da reta e escolha por uma variável de controle (*Bresenham*). Obtivemos resultados semelhantes utilizando tanto *dda* quanto *Bresenham* nos tempos e imagens para os *datasets* menores *spx.off* e *oceanU.off*. Porém o algoritmo A.7 não foi capaz de processar *datasets* maiores como *spx2.off*, *post.off* e *delta.off* deixando pontos abertos nas listas compactas de várias faces. No cálculo de $step_x$ e $step_y$ foi preciso arredondar para o inteiro mais próximo, tanto a abscissa do *pixel* mais à direita, bem como a ordenada do ponto extremo inicial, no cálculo do Δ_x e Δ_y no plano da imagem. Isso permitiu um valor de passo mais preciso, encontrando *pixels* ausentes no método original. Para isso criamos uma segunda função de conversão considerando a mantissa.

É necessário lembrar que devido ao algoritmo ser executado sobre um domínio teoricamente definido em números reais e implementado em ponto flutuante, erros de precisão podem afetar sua execução, ainda que utilizando dupla precisão. Logo, uma condição como `(x1 != x2)` precisa ser reescrita como na Listagem 3.5, onde o valor absoluto da diferença entre os termos que precisamos comparar é testada contra uma constante infinitesimal eps, no caso, da ordem de 10^{-6}.

```
if (fabs(x1 − x2) > eps∗fabs(x1))||(fabs(x1 − x2) > eps∗fabs(x2)) {
  (...)
}
```

Listing 3.5: Comparação de ponto flutuante considerando erro mínimo

A compactação das listas geradas a partir dos vértices originais requer outro tratamento. A cada iteração do algoritmo de traçado de retas, onde há incremento no eixo das abscissas, *não necessariamente estaremos mudando de linha na matriz de pixels*, o $step_y$ para uma determinada aresta pode ser menor que a metade do intervalo δ_s de amostragem do espaço tridimensional no plano da imagem. Sendo assim, é necessário ordenar os *pixels* escolhidos por *bucket sort*.

A ordenação é simples, para cada elemento em $l_2.\Delta_y$ associamos um balde (*bucket*), de modo que cada balde terá ao menos um *pixel*. Os *pixels* das arestas l_0 e l_1 são movidos para os baldes correspondentes às suas respectivas ordenadas y. O resultado é um número variável de elementos (pares (x, y)) por balde. Uma segunda função precisa ser chamada para *compactar os baldes*, isto é, gerar a lista compacta extraindo um par de elementos de cada balde contendo necessariamente a menor e a maior abscissa. Esta metodologia pode ser vista na Figura 3.8

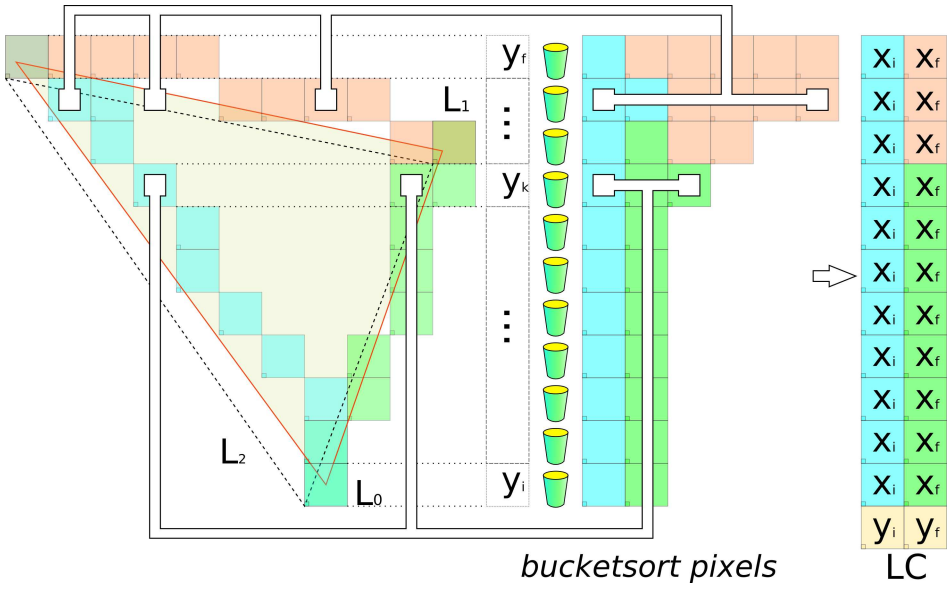

Figura 3.8: Esquema de geração da lista compacta de *pixels* a partir do cálculo das listas das arestas utilizando as coordenadas dos vértices da face no espaço tridimensional. A conversão para o plano da imagem é feita no momento de inserir nas listas das arestas, as coordenadas que o algoritmo escolhe como mais apropriadas. Ao contrário da versão totalmente no plano da imagem, é possível haver mais de duas coordenadas com a mesma ordenada, sendo necessário o uso de *bucket sort* antes de compactá-las.

A geração das listas das arestas usa apenas uma lista como estrutura de saída. Três formas distintas do triângulo da projeção da face nos levam à um caso geral, onde as três

listas $l_0,\ l_1,\ l_2$ são geradas, e dois casos particulares: um no qual a projeção de p_0 e p_1 no plano da imagem possui a mesma ordenada ($p_0.y = p_1.y$) sem gerar a lista l_0; e um terceiro caso, similar ao anterior, porém com $p_1.y = p_2.y$ no plano da imagem e sem gerar a lista l_1.

Utilizamos a implementação no espaço tridimensional para fins de pesquisa e *debug*. Passamos para o *buffer* não apenas as coordenadas dos *pixels* mas também dos pontos no espaço tridimensional antes da conversão, bem como os valores das coordenadas dos *pixels* sem a exclusão da mantissa, a fim de identificar falhas de precisão no algoritmo. Esse trabalho nos ajudou a identificar a forte relação do algoritmo com a precisão no desenho da reta e sua dependência à resolução da imagem. Mais detalhes são discutidos no Capítulo 4.

3.2 Implementação no *ZSweep*

A implementação do algoritmo dentro do *ZSweep* visa substituir a rasterização sobre a *bounding box* durante a projeção de face como ilustrado na Figura 3.9.

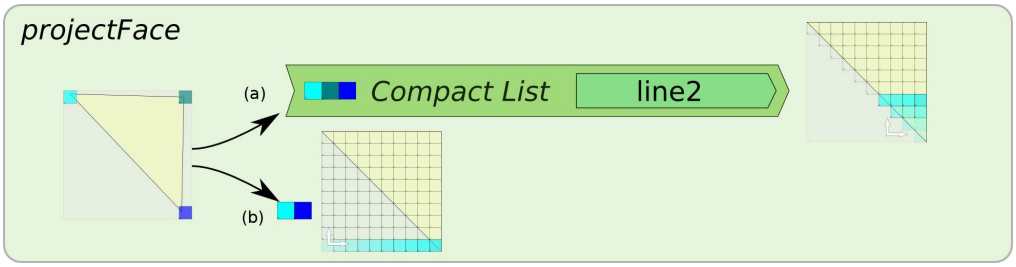

Figura 3.9: Visitando os índices dos vértices na lista de faces, extraimos seus valores e operamos o *B-Convert* sobre eles em *(a)* ao invés de extrairmos o *bounding box* da face a partir de seus vértices mais distantes para realizar o *scan convert* clássico visto em *(b)*

.

Criamos os arquivos `zs_bresenham.cc` e `zs_bresenham.hh` para definir e implementar as funções necessárias à amostragem das arestas e compactação da lista. Mantivemos a linguagem utilizada no *ZSweep*, C++. O algoritmo *ZSweep* originalmente utilizava o tipo de dados *float* (ponto flutuante de simples precisão) para os valores dos vértices das faces na cena e na conversão entre o espaço tridimensional e o plano da imagem. No *BZSweep* (*ZSweep* com *B-Convert*), adotamos o tipo *double* (dupla precisão) de modo a garantir uma conversão mais correta. Efetuamos essa mudança também no código original para fins de comparação de desempenho. Embora tenhamos inserido funções de *debug* condicionadas por diretivas em pré-processamento em vários arquivos, apenas os arquivos `zs_basic.hh, zs_scene.hh, zs_render.cc` realmente necessitaram de alterações para implementar o *B-Convert* no *ZSweep*. As listas utilizam a classe `D2Pointi` que implementa um par de inteiros bem como métodos para acessá-los. Uma

variante contendo as coordenadas em ponto flutuante de dupla precisão foi utilizada na versão que trabalha com os vértices originais no espaço tridimensional.

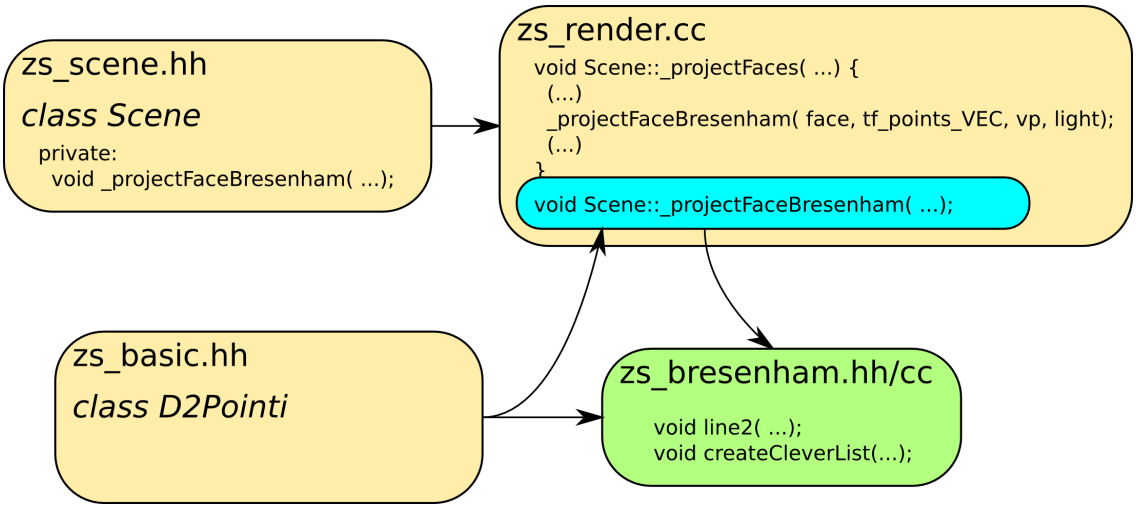

Figura 3.10: A implementação do algoritmo no *ZSweep* exigiu além de uma função de projeção que trate o *scan convert* da lista compacta, apenas o acréscimo de arquivos necessários às funções de geração e compactação das listas de arestas e uma classe para definir o elemento das listas.

A função `_projectFaceBresenham()` substitui a função `_projectFace()` e é responsável por gerar a lista compacta e rasterizá-la. Ela invoca a função `createCleverList()` (Listagem A.1), passando-lhe as coordenadas dos vértices projetados da face e esta, por sua vez, chama a função com o algoritmo de traçado de retas adaptado para retornar as listas das arestas (`line2()`). Após isso, `createCleverList()` retorna a lista compacta para `_projectFaceBresenham()`. O passo de *scan convert* é então, realizado. Enquanto `_projectFace()` trabalha sobre o o *bounding box* da face, `_projectFaceBresenham()` utiliza os vértices da face, como definido na descrição do algoritmo e pode ser vista na listagem 3.6.

A implementação desta função na versão que utiliza os vértices originais no espaço tridimensional é bastante similar, apenas inserindo os valores destes diretamente em p_0, p_1, p_2 e utilizando a função `createListCompact()` A.3 para gerar a lista compacta. Esta por sua vez, identifica 3 casos com relação à forma do triângulo para chamar a função de geração das listas das arestas de forma otimizada por caso, gerando uma única lista com o resultado de até 3 arestas. A função `pixelMapSort()` A.4 é chamada para agrupar elementos com a mesma ordenada no plano da imagem. Posteriormente, a função `compactPixelBinList()` A.5 executa a compactação destes elementos na lista final. A diferença na implementação nas duas formas do algoritmo ocorre pelo fato de que ao se incrementar o eixo das ordenadas no espaço tridimensional podemos continuar na mesma ordenada no plano da imagem, necessitando reescrever a função de compactação.

```
void Scene::_projectFaceBresenham(Face& face, vector<Point> &tf_points_VEC,
                                  class ViewPlane *vp, class Lighting *light)
{
  double coords[6]; // array of XY coords for a given face.
  vector<D2Pointi> pixel_buf; // list of pixels for bresenham output
  vector<D2Pointi>::const_iterator p;
  double wX, wY, wXi, zi, vi, wXf, zf, vf, stp_z, stp_v;
  unsigned yi, yf, xi, xf, i, j;
  __totalFaces_++;

  if (face.coplanar(tf_points_VEC)) return;
  if (!face.updateCoeffs(tf_points_VEC)) return;
  face.getVertXY(tf_points_VEC, coords); // Get x,y from tf_points_VEC into coords.
  // Convert world coordinates to screen.
  D2Pointi p0(vp->w2sX(coords[0]), vp->w2sY(coords[1]));
  D2Pointi p1(vp->w2sX(coords[2]), vp->w2sY(coords[3]));
  D2Pointi p2(vp->w2sX(coords[4]), vp->w2sY(coords[5]));

  createCleverList( p0, p1, p2, pixel_buf ); // Generate compact list

  p = pixel_buf.begin();
  yi = (*p).x < (*p).y ? (*p).x : (*p).y;
  yf = (*p).y > (*p).x ? (*p).y : (*p).x;

  p = pixel_buf.begin()+1; // Runs from min y (yi) to max y (yf).

  unsigned cId = face.getCellIdx();
  bool bId = face.getIsBoundary();
  // Scan convert compact list
  for ( j = yi; j <= yf; j++ ) {
    wY = vp->s2wY(j); // generate Y world coords from pixel coord.
    // Assumes x is already ordered ascendent. (*p).x is xi; (*p).y is xf;
    xi = (*p).x;
    xf = (*p).y;
    for (i = xi; i <= xf; i++) {
      wX = vp->s2wX(i); // generate X world coords based on pixel coord.
      if (face.IsWithin(wX, wY))  // Generate viewPort screenList for each coord within
          cell...
        vp->_scrList->insertUnit(i,j,face.GetZ(wX, wY),face.GetVal(wX, wY),cId,bId);
    }
    p++; // Increment vector counter to get next j's xi,xf values.
  }
}
```

Listing 3.6: Projeção de face por lista compacta no algoritmo *ZSweep*

O próximo capítulo apresenta os resultados dos testes efetuados com o algoritmo na sua versão usando apenas inteiros, *BZSweep* contra o algoritmo original *ZSweep*.

Capítulo 4

Resultados e Discussões

4.1 Desempenho do algoritmo

Os testes de desempenho foram realizados inicialmente em um intel core-i7 970 3.2GHz, com 16GB de RAM. Nosso interesse reside na comparação dos algoritmos sequenciais, logo o código não sofreu otimização visando *multithread*. Utilizamos a versão 1.04 do *software ZSweep* para plataformas de 64 bits executado sobre o sistema operacional *GNU Linux*. A implementação em C++ foi compilada utilizando *GNU C Compiler* com *flags* de otimização -02 e *fexpensive-optimizations*.

Utilizamos os seguintes dados volumétricos: *spx.off, spx2.off, post.off. oceanU.off, delta.off, f117.off, torso.off*. Todos dispostos em células tetrahedrais.

Dataset	Pontos	Faces	Tetraedros
spx.off	2.896	27.252	12.936
spx2.off	149.224	1.677.888	827.904
post.off	109.744	1.040.588	513.375
oceanU.off	595.434	93.158	44.595
delta.off	211.680	2.032.084	1.005.675
f117.off	48.518	485.186	240.122
torso.off	168.930	2.168.505	1.082.723

Tabela 4.1: Resolução dos dados volumétricos em número de elementos geométricos.

Inicialmente, obtivemos uma diferença de tempo maior, mas para fins de comparação e *debug* do algoritmo, alteramos o código de *scan convert* no *ZSweep* original para que o laço *for* mais externo seja aquele responsável pelo eixo das ordenadas, da forma como é no *BZSweep*. Somente essa alteração gerou um melhor tempo de processamento do algoritmo original para vários *datasets*, especialmente *oceanU.off* e *spx.off*, demonstrando haver possibilidade de explorar coerência espacial dos dados no próprio *ZSweep*. Essa modificação foi necessária para que a comparação dos algoritmos ocorresse de forma mais criteriosa.

Foram feitos vários testes de desempenho, considerando tanto o tempo total de *rendering* quanto apenas o processo de *scan convert*. Como nosso algoritmo é pertinente a um passo extremamente sensível às dimensões do plano da imagem, realizamos os testes em diferentes resoluções: 512^2, 1024^2, 2048^2, 4096^2, 8192^2, 16384^2 *pixels*. Testamos um *dataset* também na resolução 32768^2, para confirmar nossas avaliações do comportamento do algoritmo nas resoluções menores. As tabelas com os tempos em milisegundos separados por geração de lista e *scan convert* para cada resolução se encontram na Seção B.1. Um resumo dos mesmos tempos para o algoritmo original *ZSweep* se encontra na Tabela 4.2. Os tempos da versão utilizando projeção de faces pelo *B-Convert* apelidado de *BZSweep* se encontram na Tabela 4.3. Os *datasets* foram renderizados com ângulo de rotação $(0, 0, 0)$.

ZSweep	scan convert time (msec)						
Dataset	512^2	1024^2	2048^2	4096^2	8192^2	16384^2	32768^2
spx.off	12,39	29,86	97,32	358,63	1.359,29	5.286,64	20.938,82
spx2.off	217,6	269,3	442,7	1.094,14	3.601,23	13.254,74	-
post.off	94,03	163,07	413,83	1.377,02	5.106,3	-	-
oceanU.off	7,28	12,14	31,42	100,17	382,22	-	-
delta.off	221,13	282,36	499,01	1.301,68	4.368,24	-	-
f117.off	57,41	73,17	128,9	334,75	1.121,78	4.203,25	-
torso.off	272,27	331,38	511,36	1.170,82	3.713,23	13.471,97	-

Tabela 4.2: Tempos para gerar o *bounding box* e executar o passo de *scan convert* no algoritmo original *ZSweep*.

BZSweep	scan convert time (msec)						
Dataset	512^2	1024^2	2048^2	4096^2	8192^2	16384^2	32768^2
spx.off	88,45	151,51	264,94	493,78	1.103,61	3.048,55	10.010,64
spx2.off	1.698,43	2.286,18	3.189,63	5.178,98	8.897,1	16.912,09	-
post.off	543,55	773,86	1.195,16	2.113,38	4.448,65	-	-
oceanU.off	56,35	83,03	135,69	245,2	506,38	-	-
delta.off	1.316,06	1.732,55	2.463,65	3.933,67	7.017,96	-	-
f117.off	315,46	426,86	600,35	939,93	1.640,61	3.530,41	-
torso.off	1.937,97	2.726,52	3.776,31	5.772,56	9.769,87	18.441	-

Tabela 4.3: Tempos para execução do passo de *scan convert* no algoritmo de projeção de faces por lista compacta *BZSweep*. Incluindo o tempo gasto com a geração das listas.

Com base nestes tempos, foi possível gerar gráficos identificando o comportamento do algoritmo. Para os *datasets oceanU.off*, *delta.off* e *torso.off* o custo de se gerar a lista compacta e realizar o *scan convert* sobre a mesma excedeu o custo do *scan convert* sobre o *bounding box* da face para todas as resoluções nas quais estes *datasets* foram testados como pode-se ver nas Figuras 4.1, 4.2 e 4.3.

O mesmo não ocorre com os *datasets post.off* e *f117.off*, onde é possível perceber que a partir de uma dada resolução o custo de geração da lista compacta e posterior *scan*

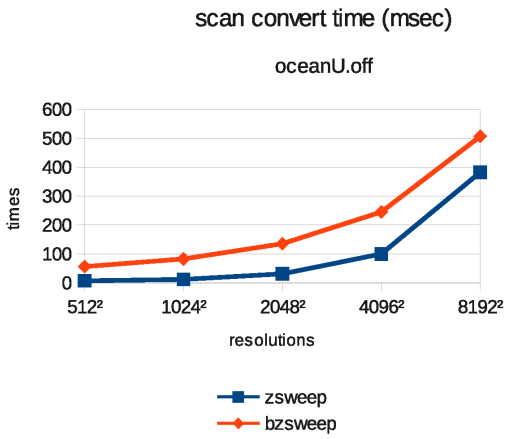

Figura 4.1: *ZSweep x BZSweep (scan convert)*, oceanU.off *dataset*.

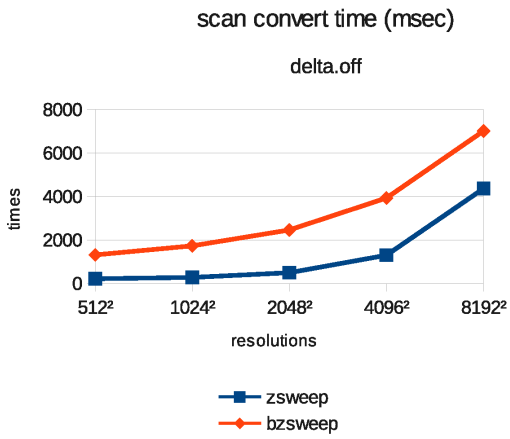

Figura 4.2: *ZSweep x BZSweep (scan convert)*, delta.off *dataset*.

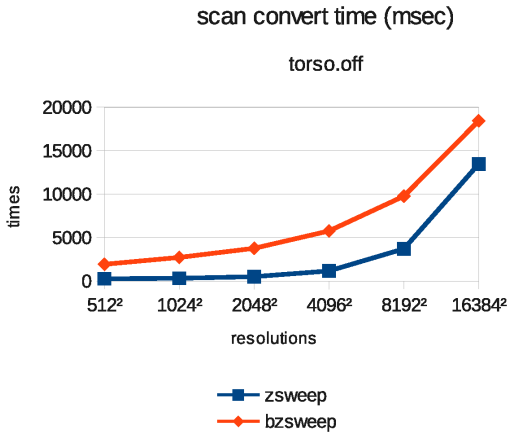

Figura 4.3: *ZSweep x BZSweep (scan convert)*, torso.off *dataset*.

convert sobre a mesma torna-se mais atrativo que o *scan convert* sobre o *bounding box* da face, como visto nas Figuras 4.4 e 4.5.

Isso nos levou a comparar tempos para o mesmo *dataset* (*spx*), em duas resoluções de grade distintas. *Spx2.off* possui cerca de 60 vezes o número de faces presente em *spx.off*, o que significa um maior *overhead* na criação das listas compactas e uma menor área em *pixels* por face. De fato, para uma resolução de tela de $512\ x\ 512$ *pixels*, onde o *rendering* cobre um retângulo de $233\ x\ 275$ *pixels* ou uma área de 64.075 *pixels*, enquanto a área média do total de faces de *spx.off* é de cerca de 60 *pixels*, para o *dataset spx2.off* esse valor é de apenas 4 *pixels*. Analogamente, a área média do total de *bounding boxes* fica em torno de 160 *pixels* para *spx.off* e de apenas 11 *pixels* para *spx2.off*. Os resultados sugerem que o *B-Convert* é vantajoso sobre *datasets* representados em grades irregulares nas quais as faces das células cobrem uma área em *pixels* de valor não inferior a cerca

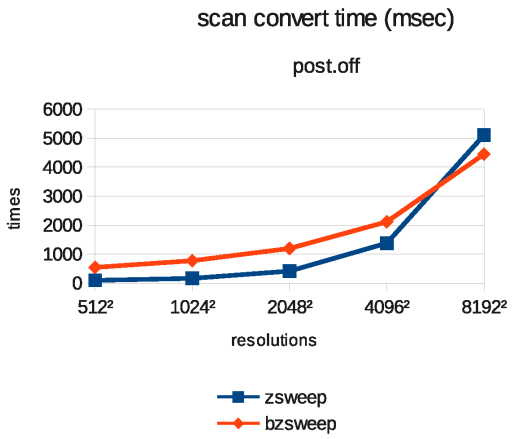

Figura 4.4: *ZSweep x BZSweep (scan convert)*, post.off *dataset*.

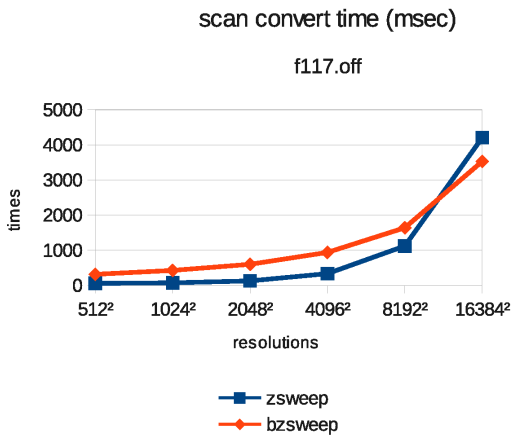

Figura 4.5: *ZSweep x BZSweep (scan convert)*, f117.off *dataset*.

de 0.094% da área renderizada para uma imagem com resolução acima de $4096\ x\ 4096$ *pixels*.

Deve-se notar que o *bounding box* é gerado no plano da imagem e não do objeto, da mesma forma como é feito no algoritmo original. Assim, uma face que gere como projeção, um triângulo equilátero ou isósceles cuja altura h não seja ortogonal a um dos eixos que definem o plano da imagem (considerando eixos cartesianos) poderá gerar uma *bounding box* com mais de 50% de *pixels* não pertencentes à projeção da face. A área A de cada face projetada foi obtida aplicando o teorema de Heron (4.3) sobre a magnitude (4.1) e o semiperímetro s (4.2) dos vetores e_i que definem as arestas das faces no plano da imagem.

$$||e_0|| = \sqrt{x_i^2 + y_i^2} \tag{4.1}$$

$$s = \frac{1}{2}(||e_0|| + ||e_1|| + ||e_2||) \tag{4.2}$$

$$A = \sqrt{s(s - ||e_0||)(s - ||e_1||)(s - ||e_2||)} \tag{4.3}$$

A tabela 4.4 apresenta a razão entre a média das áreas das faces e de suas respectivas *bounding boxes*.

Dataset	spx.off	spx2.off	post.off	oceanU.off	delta.off	f117.off	torso.off
$A_f/A_{f_{bb}}$	0.37	0.35	0.35	0.50	0.38	0.36	0.37

Tabela 4.4: Razão da média das áreas do total de faces pela média das áreas dos *bounding boxes* das mesmas (ângulo de rotação dos *datasets* de $(0, 0, 0)$).

Verificamos que a quantidade de *pixels* falsos não testados por face é baixa em

spx2.off. A área média (em *pixels*) do total de faces é muito pequena em relação à área dos *pixels* coberta pela imagem do dado volumétrico no plano da imagem (cerca de 0.00624%, no caso de *spx2.off* contra 0.09364% para *spx.off*), não justificando o cálculo extra com as listas. Ao mesmo tempo, a quantidade de listas compactas a serem geradas em *spx2.off* é 60 vezes maior que em *spx.off*. Um *dataset* dividido em muitas células pequenas acarreta um maior custo de geração de listas compactas das faces. Isso indica um ponto a ser trabalhado no algoritmo: otimizar a geração da lista compacta. Para que nossa abordagem apresente uma vantagem real é preciso olhar para um *dataset* com características similares às apresentadas por *spx.off*. Este volume possui menos pontos, sendo dividido em um menor número de células. As faces destas, por sua vez, cobrem mais *pixels*. Temos então menos listas compactas a serem geradas e maior número de falsas interseções que deixam de ser testadas, sendo este o cenário ideal para nossa abordagem como demonstrado nos gráficos das figuras 4.7 e 4.6.

Deve-se notar que não é apenas a quantidade de faces que torna o algoritmo mais interessante, mas a área média em *pixels* para as faces testadas contra a área média dos seus respectivos *bounding boxes em pixels*. O desempenho na conversão de faces em volumes irregulares (maior quantidade de faces do caso genérico) mostrou ganho em resoluções maiores como 8192^2 e 16384^2 porém não de forma uniforme entre os *datasets* utilizados. Testes empíricos demonstraram não ser este fator isolado suficiente para que haja ganho de desempenho, indicando que a relação entre quantidade de pontos no *dataset* em relação ao número de faces (e consequentemente células da grade irregular) exerce uma influência considerável.

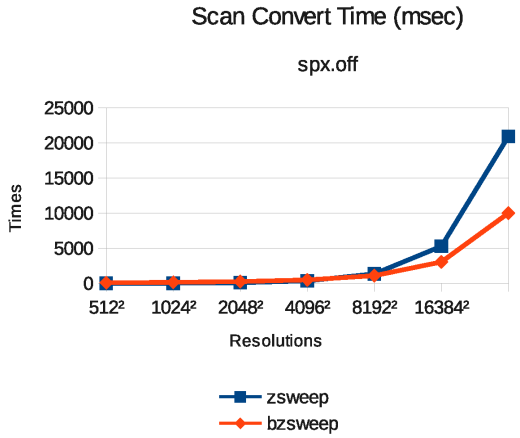

Figura 4.6: *ZSweep x BZSweep (scan convert)*, spx.off *dataset*.

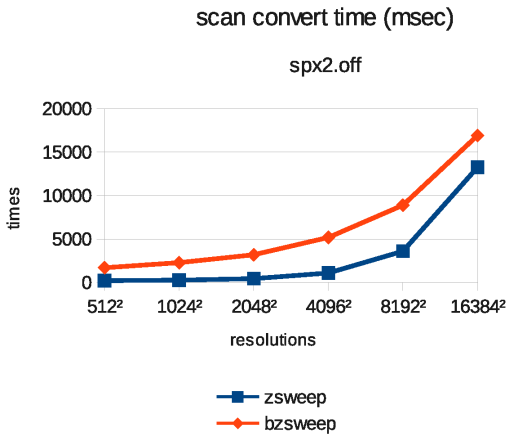

Figura 4.7: *ZSweep x BZSweep (scan convert)*, spx2.off *dataset*.

Se compararmos apenas o custo do *scan convert* própriamente dito, ou seja, sem computarmos o custo da geração das listas compactas, e observando apenas a rasterização das faces *a partir das listas geradas*, é possível notar que respondemos parcialmente nossa

hipótese inicial. O algoritmo proporciona uma diminuição no custo do processo de *scan convert* (Tabela 4.5) justamente por fornecer uma seleção mais criteriosa dos *pixels* sobre os quais ocorrerá o teste de interseção com a face no espaço tridimensional.

BZSweep x ZSweep: custo (%) do *scan convert*							
Dataset	512^2	1024^2	2048^2	4096^2	8192^2	16384^2	32768^2
spx.off	90%	68.85%	52.06%	43.48%	40.24%	38.72%	38.21%
spx2.off	93.08%	95.23%	84.43%	63.77%	48.41%	40.65%	-
post.off	86.35%	66.56%	50.45%	42.18%	38.44%	-	-
oceanU.off	74.3%	95.56%	74.13%	62.87%	54.33%	-	-
delta.off	97.68%	86.32%	67.67%	53.19%	45.11%	-	-
f117.off	95.66%	82.08%	63.07%	49.27%	42.27%	39%	-
torso.off	101.57%	94.37%	85.15%	66.82%	51.08%	42.95%	-

Tabela 4.5: Tempo gasto com *scan convert* proprimamente dito (sem o pré-processamento das listas) no algoritmo *BZSweep* contra *ZSweep* (100%).

Claro que para isso ocorrer é necessário gerar as listas, fator predominante no tempo gasto na execução da projeção de faces por lista compacta. A Tabela 4.6 demonsta a necessidade de se pesquisar formas de diminuir o custo de geração das listas. Cientes disso, passamos a direcionar a pesquisa relativa à etapa de geração das listas das arestas para algoritmos que procuram melhorar o desempenho do algoritmo original de *Bresenham*, dos quais destacamos o trabalho de LEE e HODGES [35], ROKNE *et al.* [36] e WYVILL [37], os quais discutiremos brevemente na Seção5.1. Iniciamos testes com o algoritmo de Wyvill, mas este, ao ser adaptado para nosso propósito necessita de uma estrutura de dados auxiliar para armazenar metade da lista, sendo necessário um laço para realocar estas coordenadas no final da estrutura principal utilizada no algoritmo, acarretando um custo computacional maior que a adaptação do algoritmo original de *Bresenham* para nosso propósito. Estamos pesquisando formas de trazer as vantagens destes algoritmos para a geração das listas compactas.

BZSweep	custo de geração das listas						
Dataset	512^2	1024^2	2048^2	4096^2	8192^2	16384^2	32768^2
spx.off	90.83%	88%	81.54%	68.72%	50.57%	32.89%	20.09%
spx2.off	94.36%	93.54%	91.3%	87.94%	81.03%	68.41%	-
post.off	92.11%	90.01%	84.41%	73.4%	56.26%	-	-
oceanU.off	94.94%	90.5%	84.98%	75.31%	59.4%	-	-
delta.off	93.03%	92.25%	89.79%	84.12%	72.74%	-	-
f117.off	92.75%	92.4%	89.96%	84.2%	71.95%	53.96%	-
torso.off	93.51%	93.69%	91.82%	88.18%	81.37%	68.97%	-

Tabela 4.6: Porcentagem do tempo do algoritmo *BZSweep* durante a projeção de faces gasto apenas com a geração das listas.

4.2 *Rendering* e Artefatos

Ao testar efetivamente o *rendering* dos dados volumétricos utilizando *BZSweep* obtivemos tempos que corroboram a crescente vantagem do algoritmo em resoluções maiores para determinados *datasets* como é visto na Tabela 4.7. No entanto, estes tempos demonstram, um maior peso dos processos de *rendering*. Especialmente o custo de inserção nas listas de *pixels* dos valores de cor e opacidade do *pixel* em ordem de profundidade (z) e a posterior composição dos mesmos. Tais processos possuem proporcionalmente maior peso no tempo total de *rendering* que o passo de *scan convert*.

rendering time (BZSweep over ZSweep)							
dataset	512^2	1024^2	2048^2	4096^2	8192^2	16384^2	32768^2
spx.off	103.14%	89.12%	82.18%	79.18%	78.06%	77.61%	81.45%
spx2.off	135.34%	95.47%	92.33%	95.91%	95.17%	81.19%	-
post.off	84.71%	88.99%	89.87%	94.06%	94.60%	-	-
oceanU.off	93.32%	91.71%	89.94%	96.16%	95.72%	-	-
delta.off	98.07%	94.19%	91.37%	95.95%	97.10%	-	-
f117.off	112.69%	88.13%	90.50%	95.45%	94.92%	95.30%	-
torso.off	132.44%	97.13%	92.62%	95.87%	95.31%	96.10%	-

Tabela 4.7: Porcentagem do tempo de *rendering* do algoritmo *BZSweep* comparado com *ZSweep*. Os tempos cronometrados levam em consideração o custo com o *scan convert*, inserção dos *pixels* selecionados nas listas de *pixels* e posterior composição (*delayed compositing*) dos valores de cor e opacidade destes.

Ao executar o *scan convert* sobre a lista compacta, foi constatada uma diferença no número de pontos interceptados pelo algoritmo original e o algoritmo de projeção de faces, podendo resultar em uma quantidade diferente de *pixels* renderizados. As Tabelas 4.8 e 4.9 apresentam a relação de *pixels* cujas coordenadas cobrem ou não uma determinada face, a taxa de sucesso x perda para o teste de interseção durante o *scan convert*, bem como o número total de *pixels* renderizados na resolução 2048^2 para os algoritmos *ZSweep* e *BZSweep* respectivamente.

2048^2	ZSweep			
dataset	intercepta	não intercepta	$\frac{\text{intercepta}}{\text{não intercepta}}$	nº *pixels*
spx.off	26.612.807	48.363.517	55.03%	638.040
spx2.off	112.303.532	258.242.983	43.49%	637.621
post.off	181.648.768	358.605.744	50.65%	1.621.864
oceanU.off	18.437.360	21.131.424	87.25%	1.152.335
delta.off	157.479.845	299.332.826	52.61%	1.139.291
f117.off	39.096.611	76.496.189	51.11%	624.988
torso.off	119.671.465	258.631.646	46.27%	599.943

Tabela 4.8: Quantidade de pontos interceptados e não interceptados durante o passo de *scan convert* na resolução 2048^2.

2048^2	BZSweep			
dataset	intercepta	não intercepta	$\frac{\text{intercepta}}{\text{não intercepta}}$	nº *pixels*
spx.off	26.334.127	2.272.744	1158.69%	637.882
spx2.off	107.986.626	38.679.609	279.18%	637.528
post.off	179.754.008	16.282.280	1103.99%	1.612.182
oceanU.off	18.128.992	2.344.560	773.24%	1.133.062
delta.off	154.191.534	40.749.523	378.39%	1.139.123
f117.off	38.546.451	5.718.911	674.02%	624.988
torso.off	115.219.422	40.870.860	281.91%	599.767

Tabela 4.9: Quantidade de pontos interceptados e não interceptados durante o passo de *scan convert* na resolução 2048^2 para o algoritmo *BZSweep*.

As diferenças na quantidade de *pixels* renderizados e na quantidade total de testes de interseção válidos entre os algoritmos *ZSweep* e *BZSweep* para um mesmo *dataset* são visíveis através de artefatos no *rendering* dos dados volumétricos. A causa destes artefatos reside na própria natureza do algoritmo de traçado de retas de *Bresenham*. Como explicado na Seção 2.2.2, ocorre uma escolha entre dois *pixels*, sendo que eventualmente, ao gerar a lista de uma aresta, um *pixel* mais interno à face, pode ser escolhido em detrimento de um mais externo. *Bresenham* procura aproximar a reta real da aresta sobre a matriz de *pixels*. Esta aproximação, somada à perda da mantissa na conversão de um ponto (real) no espaço tridimensional para um *pixel* (inteiro) no plano da imagem pode levar a uma amostragem inadequada. Ao verificarmos os resultados dos primeiros testes, classificamos os *pixels* em três casos distintos quanto ao teste de interseção com a face no espaço tridimensional: intercepta, não intercepta e ausente.

Intercepta O *pixel* pertence à projeção da face no plano da imagem.

Não intercepta O *pixel* selecionado não pertence à projeção da face.

Ausente Um *pixel* pertencente à projeção da face não foi escolhido.

Verificou-se na lista compacta, a presença predominante de *pixels* que efetivamente interceptam a face e consequentemente menor presença de *pixels* que não a interceptam. Isso corrobora a premissa de que estariamos realizando a rasterização sobre uma amostra mais próxima do resultado que se deseja obter. A Tabela 4.10 demonstra a quantidade de *pixels* falsos em uma linha antes e depois de *hits* para o *dataset spx.off*. Isto é, quantos *pixels* falsos ocorrem antes do primeiro *pixel* que intercepta a face e quantos após o último *pixel* de uma linha. É importante notar que *pixels* pertencentes à representação das arestas, não necessáriamente interceptam a face. Esta característica resulta não apenas em casos com 1 *pixel* não interceptador no início e/ou no final da linha, mas também casos com 2 e 3 ou mais falsos positivos no início e/ou final de uma linha. Dependendo do coeficiente angular das arestas.

hits diretos e *misses* para BZSweep					
dataset	*misses*	512^2	1024^2	2048^2	4096^2
spx.off	0 *pixel*	459.404	849.351	1.619.249	3.146.578
	1 *pixel*	327.960	607.107	1.151.707	2.245.308
	2 *pixels*	63.289	129.719	265.404	537.308
	3+ *pixels*	27.145	54.195	113.122	227.578

Tabela 4.10: Quando o primeiro *pixel* e / ou o último *pixel* de uma linha pertencem à face, constitui um caso de 0 *pixels* fora da face em uma determinada linha. É possivel ver que o número total de casos onde, para uma linha da lista, o primeiro e/ou o último *pixels* são falso-positivos é inferior à quantidade de casos sem falso positivos. Deve-se notar que a quantidade de casos com 2, 3 ou mais *pixels* iniciando e/ou terminando a linha da lista compacta é muito menor.

Durante os testes, foi constatada também a necessidade de se manter o teste de interseção no espaço tridimensional, já que, como mencionado, muitos dos *pixels* usados na representação das arestas, não *necessariamente* interceptam a face. Em um exercício, removemos o teste de interseção, salvando diretamente a amostra compacta nas listas dos *pixels*. O resultado foi uma imagem incorreta e maior tempo de processamento. Isso ocorreu porque um *pixel que não intercepta* uma determinada face passou a conter o valor de opacidade e cor da mesma. Estes foram acrescidos erroneamente em sua lista para posterior composição no algoritmo *ZSweep*. Tal exercício demonstrou também um maior custo de processamento na etapa de composição, devido a estes falsos positivos serem repetidos em diversas faces.

A situação *ausente* é a mais crítica no algoritmo. Ocorre quando o algoritmo de *Bresenham* escolhe outro *pixel* (interno à face) deixando de fora da lista um *pixel* que intercepta a face na projeção de uma de suas arestas. O resultado é um artefato na imagem. O *pixel ausente* é mais facilmente percebido ao rasterizar uma face externa. A imagem (b) da Figura 3.4 exibe o caso onde as listas contribuem com todos os *pixels* que interceptam a face. Há, no entanto, o caso inverso, onde durante a escolha de que *pixel* iluminar no caminho da reta, o algoritmo de *Bresenham* escolhe um *pixel* interno à face, deixando de fora um *pixel* de fronteira, gerando um artefato na imagem final. Para faces internas ao volume isso não é necessáriamente um problema pois o *pixel* de fronteira ignorado para uma determinada aresta em uma face pode vir a ser considerado como *pixel* da face imediatamente vizinha. Para ilustrar melhor o caso consideremos a situação em duas dimensões como exemplificado na Figura 4.9.

Note que pode haver perda de *pixels* ainda assim. Pois dadas duas faces f_k e f_{k+1} separadas pela mesma aresta l_i, consideremos um *pixel*, originalmente pertencente à f_k mas excluido de sua lista compacta e agora contido na lista compacta de f_{k+1}, este *será testado quanto à interseção contra esta face no espaço da cena*. Podendo retornar verdadeiro ou falso. O caso crítico, no entanto, ocorre em faces de fronteira (ou externas)

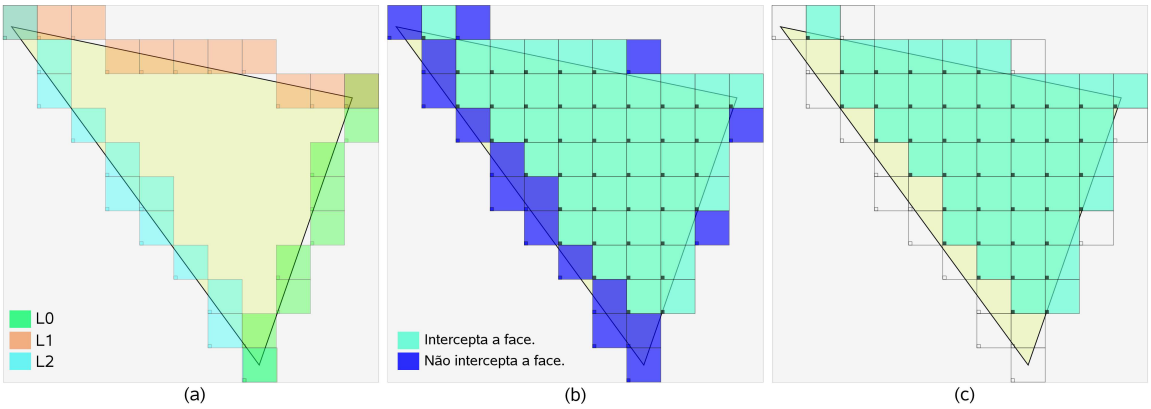

Figura 4.8: Em *(a)* encontram-se as três aproximações em *pixels* das arestas sobre o triângulo da face. Note que o triângulo também está representado no plano da imagem, porém seus vértices mantém a mantissa apenas para fins de comparação (ela é eliminada ao se converter do espaço da cena para a imagem). Em *(b)* estão representados em azul os *pixels* que foram considerados na abordagem por *Bresenham* mas que são descartados no teste de interseção com o triângulo no espaço tridimensional. Em *(c)* é exibida a representação final esperada tanto por rasterização sobre a lista compacta gerada com o algoritmo de traçado de retas de *Bresenham* quanto por rasterização sobre o *bounding box* da face projetada.

pois eventualmente, uma das projeções que *cobriria* a lacuna deixada pelo *pixel* ausente deixará de ocorrer e todos os *pixels* omitidos resultarão em artefatos. O resultado desta situação pode ser visto na Figura 4.10.

Comparando a função de teste de interseção e o algoritmo de traçado de retas é possível notar uma diferença na forma como ambos os algoritmos ”percebem”um *pixel* da aresta. Enquanto o teste da interseção utiliza a coordenada equivalente do *pixel* no espaço da cena $3D$ para verificar se este se encontra dentro do triângulo da face, *Bresenham* escolherá iluminar o *pixel* mais perto da coordenada real da reta **ao entrar no pixel**.

No plano da imagem, consideramos apenas uma matriz de inteiros e ao converter uma coordenada do espaço tridimensional para o plano da imagem, desprezamos sua mantissa, arredondando para seu limite inferior. No entanto, ao retornarmos a coordenada do *pixel* para a cena $3D$ a fim de realizar o teste de interseção, a coordenada resultante é próxima à coordenada original, mas não necessariamente a mesma. Isso ocorre para cada *pixel* da lista compacta.

Na rasterização sobre o *bounding box* isso não é problema pois já estamos fornecendo amostras de sobra. Ao rasterizar uma amostra irregular como a lista compacta, onde buscamos justamente limitar a quantidade de *pixels*, esta perda de precisão numérica torna-se mais evidente e influente sobre o algoritmo. A partir dessa informação, passamos a investigar a causa dos artefatos e possíveis soluções. Para isso, estudamos a forma como é feita a conversão do espaço tridimensional para o plano da imagem.

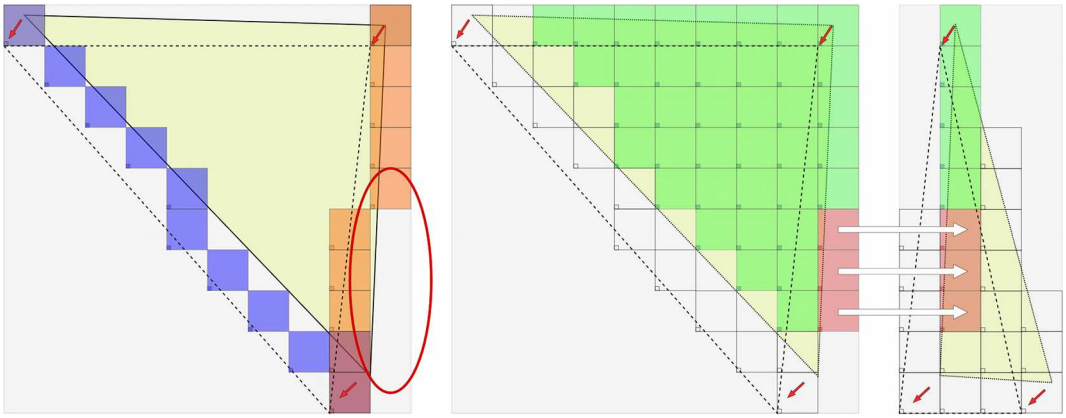

Figura 4.9: Em *(a)* temos dois triângulos, o primeiro, em amarelo, resulta dos valores não truncados da conversão do espaço da cena $3D$ para o plano da imagem enquanto o triângulo tracejado representa a face na matriz de *pixels*. Ainda em *(a)* nota-se à direita que ao menos três *pixels* mais externos a face não foram escolhidos pelo algoritmo de traçado de retas, podendo resultar em artefatos na imagem final *(b)*. Em *(c)* estes mesmos *pixels* ausentes da face f_k estarão presentes na lista compacta da face f_{k+1}, vizinha imediata de f_k à direita, podendo ser renderizados se passarem no teste de interseção com a mesma.

Dados dois pontos p_1 e p_2, definimos a escala s e a origem o dos *pixels* em duas dimensões no plano da imagem através das equações (4.4) e (4.5) respectivamente.

$$s_i = \frac{res_i - 1}{p_2.i - p_1.i} \tag{4.4}$$

$$o_i = \frac{p_1.i + p_2.i}{2.0} - \frac{d_i(res_i - 1)}{2.0} \tag{4.5}$$

Onde d_i é a distância no espaço tridimensional entre duas coordenadas que representam dois *pixels* consecutivos no plano da imagem. E é definida por:

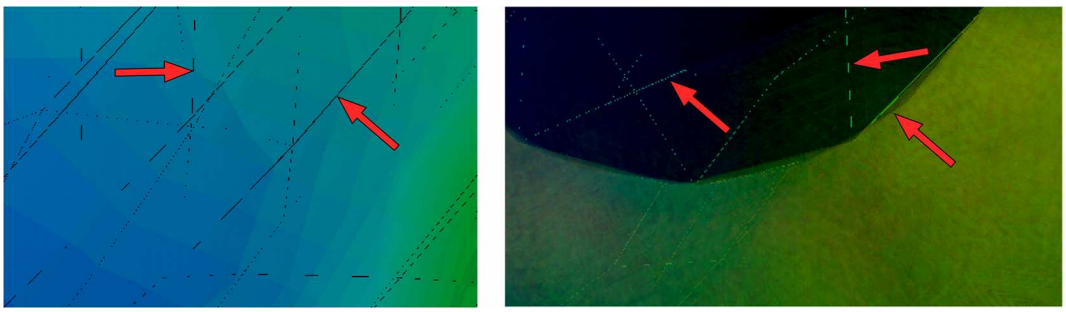

Figura 4.10: Artefatos causados pela ausência de *pixels* de fronteira na lista compacta de faces externas. Em *(a)* apenas faces externas foram renderizadas e interseções omitidas geram *pixels* com a cor de fundo (preto). Em *(b)*, faces externas e internas foram renderizadas e onde há intersecções omitidas, os *pixels* apresentam a coloração do interior do volume.

$$d_i = \frac{1}{s_i} \tag{4.6}$$

A resolução total da imagem sobre um determinado eixo de coordenadas i é armazenada em res_i. Este processo é denominado *scaling* e é responsável por *mapear* coordenadas do espaço tridimensional da cena no plano da imagem.

A convenção dita que o valor da coordenada convertida para a tela seja truncado para o menor inteiro mais próximo. Para a coordenada x por exemplo, isso é feito através da função 4.1:

```
inline unsigned w2sX(double X) {
  return (unsigned)((x-_xOrig)*_xScale);
}
```

Listing 4.1: Função de conversão para os valores de x *w2sX* (*world to screen*)

Esse procedimento é correto, porém não é suficiente para inicializar a variável de controle ϵ de forma a representar com precisão a reta descrita pelos vértices da face no espaço tridimensional. Se por exemplo, em uma determinada face f_k, dois de seus vértices no espaço tridimensional w_0, w_1 são convertidos para as coordenadas $w2s_0, w2s_1$ definidas por $(1.9, 2.8)$ e $(10.75, 8.9)$ respectivamente, os *pixels* de coordenadas $(1, 2)$ e $(10, 8)$ serão usados para representar os vértices v_0, v_1 no plano da imagem. A partir deles será gerada a lista l representando a aresta e_{01} no plano da imagem. Porém, o coeficiente angular $m_{e_{01}}$ e consequentemente sua inclinação (*slope*) serão diferentes daqueles da aresta $e_{w2s_{01}}$ definida por $w2s_0, w2s_1$. Utilizando uma função de conversão que arredonde para o *pixel* mais próximo, usariamos $(2, 3)$ e $(11, 9)$ para representar v_0, v_1. Ainda assim teríamos uma aresta com inclinação diferente, porém esta seria uma aproximação melhor da aresta $e_{w2s_{01}}$ do que e_{01}. De forma a investigar isso, renderizamos alguns *datasets* arredondando as coordenadas inteiras para o valor mais próximo do resultado da conversão do espaço tridimensional para o plano da imagem. A função 4.1 foi então substituida por 4.2.

```
inline unsigned w2sXr(double x) {
  double scaledX = (x-_xOrig)*_xScale;

  if ( (unsigned)( scaledX + 0.5 ) != _xres ) {
    return (unsigned)( scaledX + 0.5 );
  } else {
    return (unsigned)(scaledX);
  }
}
```

Listing 4.2: Função de conversão para os valores arredondados (*rounded*) de x *w2sXr* (*world to screen*)

Este experimento resultou em imagens com número muito menor de artefatos como pode ser visto na Figura 4.11 *(b)*, demonstrando que a precisão numérica possui um maior peso na projeção de faces por lista compacta. Enquanto a execução de *w2sX* e *w2sY* apenas despreza a mantissa, truncando para o limite inferior inteiro, a utilização de *w2sXr* e *w2sYr* na conversão dos vértices, antes de operar sobre eles o algoritmo de traçado de retas, demonstra a sensibilidade do algoritmo justamente ao *desenho da aresta* no plano da imagem.

A utilização da conversão com arredondamento traz em si um problema. É preciso testar se o *pixel* gerado após o arredondamento não ultrapassa o escopo da resolução da imagem (ex. *pixel* $(512, 0)$ em uma imagem de $512\ x\ 512$ *pixels*). Esse teste se traduz em mais um custo, sendo que o arredondamento na conversão para o inteiro mais próximo não resolve totalmente o problema dos artefatos. Também verificamos uma possível solução mais simples que consiste em apenas decrementar o valor de $x_{inicial}$ e incrementar o valor de x_{final} para cada elemento da lista compacta. Sem arrendondar o valor convertido para o espaço da tela e sem precisar testar o limite superior da resolução da imagem. Essa metodologia também ajudou a reduzir os artefatos mas não foi suficiente para eliminá-los em todas as resoluções testadas como visto na Figura 4.11 *(d)*. A Figura 4.12 demonstra tanto o método de arredondamento antes de se gerar a lista compacta como o método posterior de decremento / incremento das abscissas extremas de cada linha da lista compacta.

Ocorre que dependendo do coeficiente angular m de uma dada aresta, é possível perder, a representação de mais de um *pixel* adjacente. Por exemplo, supondo um exemplo similar ao da aresta com artefatos vista na Figura 4.9, porém com o eixo $\overrightarrow{OY}$ configurado como eixo passivo, o incremento deste pode vir 3 iterações depois do que o necessário para inserir na lista compacta *pixels* pertencentes à projeção da face.

A partir dos resultados obtidos com o incremento e decremento de uma abscissa nos extremos inicial e final de cada linha da lista compacta, pesquisamos também a dilatação da face com base no baricentro do triângulo da mesma. Essa técnica não provou-se eficiente em triângulos com presença de ângulos muito agudos, onde o vértice dilatado recaía no mesmo *pixel*. Testamos também a substituição das medianas menores pela maior das três na dilatação dos três vértices. Obtivemos bons resultados com relação aos artefatos mas essa situação não provou-se muito diferente de realizar o *scan convert* sobre o *bounding box* da face, visto que apenas aumentávamos indiscriminadamente o espaço amostral para o teste de interseção. Além de que, o custo do traçado de retas e compactação da lista foi acrescido do *overhead* de encontrar o baricentro, dilatar a face (com ou sem a escolha da maior mediana) e realizar o *scan convert* sobre um espaço amostral relativamente diferente da proposta inicial do algoritmo. Também é preciso salientar que esta técnica não provou-se operacional sobre todos os *datasets* pelo custo de processamento e memória, funcionando apenas nos menores (*spx.off* e *oceanU.off*), com resoluções relativamente pequenas (1024^2). A idéia é apresentada na Figura 4.13.

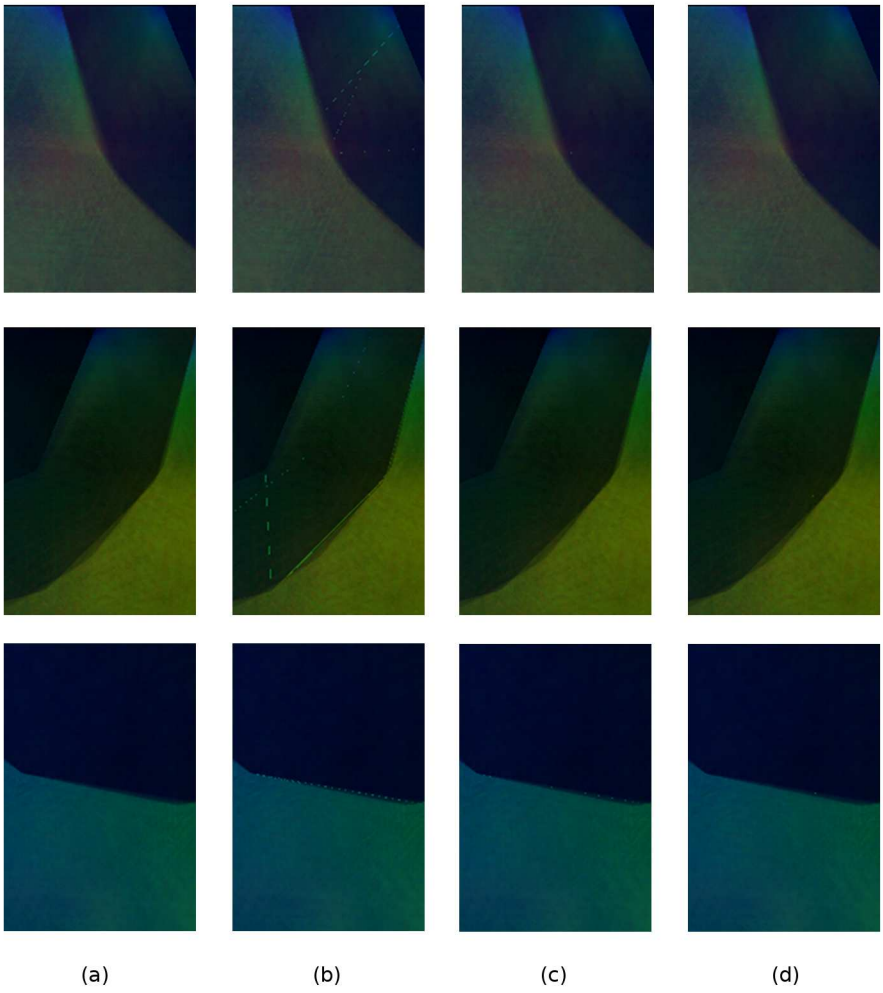

Figura 4.11: *Rendering* do *dataset spx.off* em 4096^2. Amostras restultates do *scan convert* sobre o *bounding box* em *(a)*; por *B-Convert* apenas truncamento as coordenadas durante a conversão para o plano da imagem em *(b)*; *B-Convert* extendendo em uma unidade o limite inferior e superior do intervalo de abscissas em cada linha da lista compacta *(c)*; por fim, *B-Convert* sobre coordenadas convertidas com arredondamento para o *pixel* mais próximo em *(d)*.

A completa corretude na imagem final é o tópico que estamos pesquisando atualmente. Prover uma forma de garantir a correção da imagem em todos os *pixels* é essencial para o *B-Convert* substituir por completo o *scan convert* sobre o *bounding box* na projeção de faces triangulares. Com os testes feitos na versão usando as coordenadas originais da face (Seção 3.1.3) verificamos uma maior precisão ao cortar o $step_i$ em 4, efetivamente recorrendo a *multisampling*. Estamos no momento pesquisando algoritmos de *antialiasing* para prover os *pixels* ausentes às listas das arestas ou ao menos minimizar sua presença na imagem final.

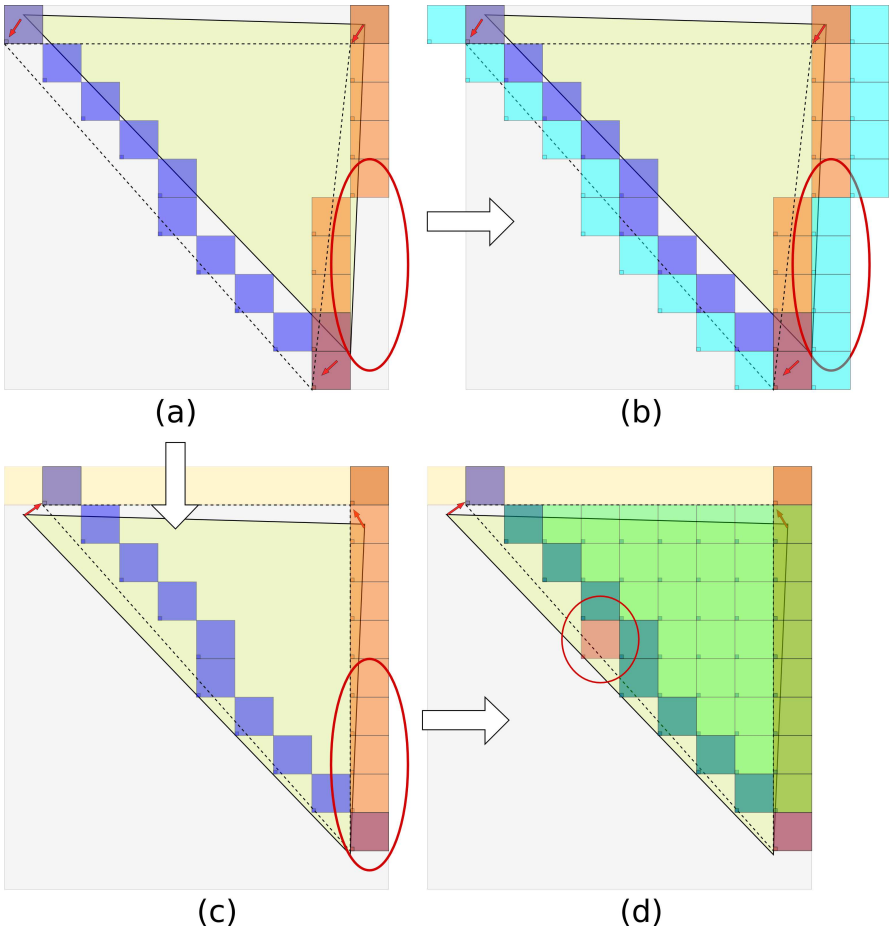

Figura 4.12: Procurando reduzir a presença de artefatos, a partir da lista compacta vista em *(a)*: acrescentamos uma unidade às abscissas dos extremos em cada linha da lista *(b)*; também foi testado o arrendondamento do valor das coordenadas dos *pixels* contendo os vértices da face *(c)*, considerando a mantissa, outrora desprezada na conversão para o plano da imagem. Artefatos ainda podem ocorrer para ambas as técnicas, como é ilustrado em *(d)* para o método do arredondamento.

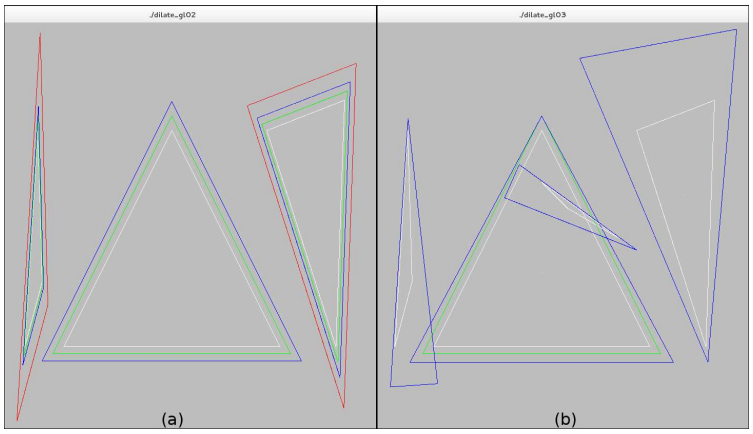

Figura 4.13: Em *(a)* vemos alguns testes de dilatação da face, note que dependendo da forma do triângulo, algumas arestas não se beneficiam da dilatação. Já em *(b)*, ao usarmos a maior mediana em todos os vértices, podemos obter uma amostragem exagerada e ainda irregular.

Capítulo 5

Conclusões

Os resultados obtidos corroboraram nossa hipótese de que o algoritmo de traçado de retas oferece uma vantagem considerável no tempo de *scan convert* propriamente dito, no qual obtivemos ganhos consideráveis como demonstrado na Tabela 4.5. Sendo que para que isso ocorra, é necessário gerar a lista compacta, um pré-processamento ao passo de *scan convert* custoso em boa parte dos casos analisados e visto na Tabela 4.6, com razoável impacto no tempo total de *rendering*(4.7).

Verificamos pelas imagens renderizadas que o algoritmo é capaz de uma aproximação razoável da face projetada no plano da imagem. Nossa abordagem ainda precisa ser refinada para poder substituir *integralmente* o *scan convert* sobre o *bounding box* da face. No entanto, quando aplicado à renderização volumétrica por projeção de faces, o *B-Convert* já oferece uma aproximação adequada para visualizações em alta resolução a partir de *datasets* representados por grades irregulares pequenas. Isso sugere sua aplicabilidade em situações nas quais o tempo de *rendering* para imagens de alta resolução é um fator mais significante do que a completa corretude da imagem final. É importante salientar que em resoluções acima de 4096^2 *pixels* a ocorrência de artefatos é baixa e, em muitos dos *datasets* testados, a percepção visual dos mesmos é mínima.

Atualmente o algoritmo depende de fatores como a resolução do *dataset* e a razão entre as áreas da face projetada e seu *bounding box* bem como a resolução total da imagem para apresentar vantagens significativas.

Datasets irregulares como *spx, post* e *f117*, cuja média da razão entre área de face/*bounding box* varia em torno de 0.36 conseguem um ganho de desempenho a partir de uma resolução de 8192^2 *pixels*. Estes são os *datasets* que possuem a menor quantidade de pontos dispersos no volume, respectivamente 2.896, 109.744 e 48.518 pontos. O número total de faces renderizadas nestes volumes para um ângulo de rotação $(0.0, 0.0, 0.0)$ é cerca de 9.5 vezes maior que o número de pontos. Outros *datasets* irregulares possuem a mesma relação de faces / pontos, mas sua quantidade de pontos é maior. A quantidade menor de pontos aliada à resolução mais alta significa menos testes e leituras dos vetores bem como mais *pixels* por faces, sugerindo ser este o melhor cenário para o algoritmo. Es-

peramos reduzir as limitações do algoritmo, específicamente diminuir o tempo de geração das listas compactas e minimizar ou eliminar a presença de artefatos nas imagem resultantes em trabalhos futuros.

5.1 Trabalhos Futuros

Constatamos dois pontos passíveis de melhoria no algoritmo: o custo computacional de geração da lista compacta e a presença de artefatos nas imagens resultantes.

Em parte, esperávamos por um custo razoável na geração das listas, visto que o algoritmo de traçado de retas de *Bresenham*, um método incremental rápido e eficiente, já sofreu inúmeras otimizações para torná-lo ainda mais veloz, especialmente em aplicações onde é implementado em *hardware* [36]. Este custo computacional pode ser minimizado por algoritmos utilizando métodos de multi-pontos. Nestes, busca-se reduzir o *overhead* necessário para traçar uma reta gerando mais de um *pixel* a cada escolha. Como exemplo, investigamos o algoritmo simétrico de passo duplo descrito em WYVILL [37] e ROKNE *et al.* [36], onde dois pares de *pixels* são escolhidos a cada iteração, ao invés de apenas um *pixel*. Chegamos a implementar uma versão utilizando este trabalho mas o uso de simetria em nosso algoritmo de projeção de faces requer uma segunda estrutura para armazenar metade da lista de cada aresta, além de um passo de cópia desta estrutura para a lista final, acarretando em maior custo computacional. Ainda assim, é necessário que tratemos mais de um *pixel* por iteração para tornar a geração da lista atraente para uma gama maior de *datasets* e resoluções. Passamos então a considerar a idéia de usar apenas o passo duplo, sem a simetria, podendo chegar perto do dobro da velocidade no traçado de retas[36] (excluindo o custo de escrita nas listas).

Além de multi-pontos, outra abordagem bastante popular para otimizar o traçado de retas é a utilização de métodos estruturais. Estes, visam analisar o comportamento de segmentos de retas discretas em busca de padrões para diminuir o uso de testes durante a tomada de decisão. Algoritmos de *run length* [35, 38] reconhecem certos intervalos de *pixels* como estruturas periódicas e as agrupam de modo a prover uma forma de compactação e otimização no processo de desenho da reta. De fato, o trabalho de LEE e HODGES [35] busca unificar métodos estruturais com métodos de multi-pontos, propondo um algoritmo híbrido de passo quádruplo que divide a linha traçada pelo algoritmo de traçado de retas em segmentos (*runs*) horizontais, conseguindo resolver até 4 *pixels* com um máximo de 2 testes sem necessáriamente fazer uso de simetria.

De fato, o autor constata que [37] faz uso de um método estrutural para um *lengh* fixo de 2 *pixels*. Uma característica interessante indicada no trabalho é a substituição da divisão necessária para a inicialização do algoritmo por uma comparação e operações de *shift* e subtração, indicando através de testes empíricos ser mais vantajoso não usar *lengths* maiores que 4 *pixels*.

Sendo este trabalho a segunda otimização sobre BRESENHAM [6] que estamos investigando.

Para o tratamento dos artefatos, estamos investigando métodos de *antialiasing*. Os artefatos ocorrem devido à ausência de alguns *pixels* na lista compacta. Marcados como pertencentes à projeção da face no teste de interseção sobre o *bounding box*, estes *pixels* nem mesmo são selecionados para a lista compacta, sendo na realidade excluídos durante a geração das listas das arestas. No entanto, a medida que a resolução da imagem aumenta, a quantidade de *pixels* ausentes diminui. Este comportamento sugere que a aproximação da aresta gerada por *Bresenham* se torna mais próxima da reta real quando sua amostragem cobre um número maior de *pixels*, justificando estender a pesquisa à algoritmos de *antialiasing* para compensar perdas oriundas do critério de escolha de *pixels* no algoritmo de traçado de retas. Estamos pesquisando dois algoritmos nessa linha. O algoritmo proposto em WU [9] procura gerar retas com *antialiasing* ao seu redor porém mantendo sua eficiência próxima à de [6]. Este algoritmo trabalha no plano da imagem e faz uso da análise estrutural de um segmento de reta discreto. Pretendemos investigá-lo na recuperação dos *pixels* ausentes *ainda na fase de geração das listas*, buscando compor a lista compacta com mais amostras, possivelmente evitando artefatos.

O segundo algoritmo, *morphological antialiasing (MLAA)* apresentado por RESHETOV [39] apenas minimiza o impacto visual das ausências na lista compacta. Estamos considerando-o pela robustez e desempenho sugeridos [40], [41] além de abordar a análise da forma assumida por um grupo de *pixels* de cor semelhante. Apesar de não ocorrer em tempo real, consegue ser proporcionalmente mais rápido que *multisampling (MSAA)* a medida que aumenta-se a resolução da imagem. Inclusive, para uma eventual implementação em *GPU*, já há trabalhos que apontam nova redução do custo computacional em *GPGPU* [41]. Este algoritmo funciona como um filtro, trabalhando também sobre o plano da imagem, porém após o passo de *rendering* (e consequentemente de projeção de faces). O *MLAA* verifica discrepâncias em padrões de cores (ou outro atributo), a partir de estruturas (U, Z e L) formadas pela disposição dos *pixels* sendo filtrados. É dividido em 3 passos principais: Identificação de *pixel* com anomalias nos eixos horizontal e/ou vertical, determinação das estruturas L (U e Z podem ser divididas em outras estruturas L) a partir dos *pixels* selecionados no primeiro passo e posterior recomposição da cor dos *pixels* pertencentes aos padrões identificados no segundo passo.

Referências Bibliográficas

[1] KAUFMAN, A., MUELLER, K. "Overview of Volume Rendering". In: Johnson, C. R., Hansen, C. D. (Eds.), *The Visualization Handbook*, Academic Press, pp. 127–174, Stony Brook University, 2005.

[2] CIGNONI, P., MONTANI, C., ET AL. "Optimal Isosurface Extraction". In: Johnson, C. R., Hansen, C. D. (Eds.), *The Visualization Handbook*, Academic Press, pp. 63–76, New York, USA, 2004. Disponível em: <http://vcg.isti.cnr.it/Publications/2004/CMPS04>.

[3] KAUFMAN, A. E. "Volume visualization", *ACM Comput. Surv.*, v. 28, n. 1, pp. 165–167, 1996. ISSN: 0360-0300. doi: http://doi.acm.org/10.1145/234313.234383.

[4] MAXIMO, A. *Projeção de Células baseada em GPU para Visualização Interativa de Volumes*. Tese de M.Sc., COPPE/UFRJ, Rio de Janeiro, RJ, Brasil, 2006.

[5] MAXIMO, A., MARROQUIM, R., FARIAS, R. "Hardware-Assisted Projected Tetrahedra", *Computer Graphics Forum*, v. 29, pp. 903–912, 2010. ISSN: 0167-7055. doi: http://dx.doi.org/10.1111/j.1467-8659.2009.01673.x.

[6] BRESENHAM, J. E. "Algorithm for Computer Control of a Digital Plotter", *IBM Systems Journal. Surv.*, v. 4, n. 1, pp. 25–30, 1965. ISSN: 0018-8670. doi: http://dx.doi.org/10.1147/sj.41.0025.

[7] AKENINE-MÖLLER, T., HAINES, E., HOFFMAN, N. *Real-Time Rendering 3rd Edition*. Natick, MA, USA, A. K. Peters, Ltd., 2008. ISBN: 987-1-56881-424-7.

[8] LENGYEL, E. *Mathematics for 3D Game Programming and Computer Graphics, Second Edition*. Rockland, MA, USA, Charles River Media, Inc., 2003. ISBN: 1584502770.

[9] WU, X. "An efficient antialiasing technique", *SIGGRAPH Comput. Graph.*, v. 25, n. 4, pp. 143–152, jul. 1991. ISSN:

0097-8930. doi: 10.1145/127719.122734. Disponível em: <http://doi.acm.org/10.1145/127719.122734>.

[10] PITTEWAY, M. L. V. "Algorithm for drawing ellipses or hyperbolae with a digital plotter", *The Computer Journal*, v. 10, n. 3, pp. 282–289, 1967. doi: 10.1093/comjnl/10.3.282.

[11] VAN AKEN, J. "An Efficient Ellipse-Drawing Algorithm", *IEEE Comput. Graph. Appl.*, v. 4, pp. 24–35, September 1984. ISSN: 0272-1716. doi: 10.1109/MCG.1984.275994. Disponível em: <http://dl.acm.org/citation.cfm?id=1299971.1300479>.

[12] VAN AKEN, J., NOVAK, M. "Curve-drawing algorithms for Raster displays", *ACM Trans. Graph.*, v. 4, pp. 147–169, April 1985. ISSN: 0730-0301. doi: http://doi.acm.org/10.1145/282918.282943. Disponível em: <http://doi.acm.org/10.1145/282918.282943>.

[13] ZHANG, K., AMMERAAL, L. *COMPUTAÇAO GRAFICA PARA PROGRAMADORES JAVA*. Rio de Janeiro, RJ, Brasil, LTC, 2007. ISBN: 9788521616290. Disponível em: <http://books.google.com.br/books?id=MzFiPgAACAAJ>.

[14] HECKBERT, P. S. "Graphics gems". Academic Press Professional, Inc., cap. Digital line drawing, pp. 99–100, San Diego, CA, USA, 1990. ISBN: 0-12-286169-5. Disponível em: <http://dl.acm.org/citation.cfm?id=90767.90793>.

[15] FARIAS, R., MITCHELL, J. S. B., SILVA, C. T. "ZSWEEP: an efficient and exact projection algorithm for unstructured volume rendering". In: *VVS '00: Proceedings of the 2000 IEEE Symposium on Volume visualization*, pp. 91–99, New York, NY, USA, 2000. ACM Press. ISBN: 1-58113-308-1. doi: http://doi.acm.org/10.1145/353888.353905.

[16] SHIRLEY, P., TUCHMAN, A. "A polygonal approximation to direct scalar volume rendering". In: *Proceedings of the 1990 workshop on Volume visualization*, VVS '90, pp. 63–70, New York, NY, USA, 1990. ACM. ISBN: 0-89791-417-1. doi: 10.1145/99307.99322. Disponível em: <http://doi.acm.org/10.1145/99307.99322>.

[17] CHANDRASEKHAR, S. *Radiative Transfer*. 2 ed. New York, NY, USA, Dover Publications, Inc, 1960.

[18] MAX, N. “Optical Models for Direct Volume Rendering”, *IEEE Transactions on Visualization and Computer Graphics*, v. 1, n. 2, pp. 99–108, 1995. ISSN: 1077-2626. doi: http://dx.doi.org/10.1109/2945.468400.

[19] HADWIGER, M., LJUNG, P., ET AL. “Advanced illumination techniques for GPU volume raycasting”. In: *ACM SIGGRAPH ASIA 2008 courses*, SIGGRAPH Asia ’08, pp. 1:1–1:166, New York, NY, USA, 2008. ACM. doi: http://doi.acm.org/10.1145/1508044.1508045. Disponível em: <http://doi.acm.org/10.1145/1508044.1508045>.

[20] PHARR, M., HUMPHREYS, G. *Physically Based Rendering: From Theory to Implementation*. San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 2004. ISBN: 012553180X.

[21] BUNYK, P., KAUFMAN, A. E., SILVA, C. T. “Simple, Fast, and Robust Ray Casting of Irregular Grids”. In: *Dagstuhl ’97, Scientific Visualization*, pp. 30–36, Washington, DC, USA, 1997. IEEE Computer Society. ISBN: 0-7695-0505-8.

[22] BLINN, J. F. “Light reflection functions for simulation of clouds and dusty surfaces”. In: *SIGGRAPH ’82: Proceedings of the 9th annual conference on Computer graphics and interactive techniques*, pp. 21–29, New York, NY, USA, 1982. ACM Press. ISBN: 0-89791-076-1. doi: http://doi.acm.org/10.1145/800064.801255.

[23] GARRITY, M. P. “Raytracing irregular volume data”. In: *VVS ’90: Proceedings of the 1990 workshop on Volume visualization*, pp. 35–40, New York, NY, USA, 1990. ACM Press. ISBN: 0-89791-417-1. doi: http://doi.acm.org/10.1145/99307.99316.

[24] RIBEIRO, S., MAXIMO, A., BENTES, C. E. A. “Memory-Aware and Efficient Ray-Casting Algorithm”. In: *Proceedings of the XX Brazilian Symposium on Computer Graphics and Image Processing*, SIBGRAPI ’07, pp. 147–154, Washington, DC, USA, 2007. IEEE Computer Society. ISBN: 0-7695-2996-8. doi: 10.1109/SIBGRAPI.2007.28. Disponível em: <http://dx.doi.org/10.1109/SIBGRAPI.2007.28>.

[25] WEILER, M., KRAUS, M., ERTL, T. “Hardware-Based View-Independent Cell Projection”. In: *VVS ’02: Proceedings of the 2002 IEEE Symposium on Volume visualization and graphics*, pp. 13–22, Piscataway, NJ, USA, 2002. IEEE Press. ISBN: 0-7803-7641-2.

[26] ESPINHA, R., CELES, W. “High-Quality Hardware-Based Ray-Casting Volume Rendering Using Partial Pre-Integration”. In: *SIBGRAPI ’05: Proceedings of the XVIII Brazilian Symposium on Computer Graphics and Image Processing*, p. 273. IEEE Computer Society, 2005. ISBN: 0-7695-2389-7. doi: http://dx.doi.org/10.1109/SIBGRAPI.2005.29.

[27] RIBEIRO, S., MAXIMO, A., BENTES, C. E. A. “Memory Efficient GPU-Based Ray Casting for Unstructured Volume Rendering”. pp. 155–162, Los Angeles, California, USA, 2008. Eurographics Association. ISBN: 978-3-905674-12-5. doi: http://doi.ieeecomputersociety.org/10.2312/VG/VG-PBG08/155-162.

[28] COX, G., MAXIMO, A., BENTES, C. E. A. “Irregular Grid Ray-casting Implementation on the Cell Broadband Engine”. In: *Proceedings of the 2009 21st International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD ’09, pp. 93–100, Washington, DC, USA, 2009. IEEE Computer Society. ISBN: 978-0-7695-3857-0. doi: 10.1109/SBAC-PAD.2009.15. Disponível em: <http://dx.doi.org/10.1109/SBAC-PAD.2009.15>.

[29] SCHUBERT, N., SCHOLL, I. “Comparing GPU-based multi-volume ray casting techniques”, *Comput. Sci.*, v. 26, n. 1-2, pp. 39–50, fev. 2011. ISSN: 1865-2034. doi: 10.1007/s00450-010-0141-1. Disponível em: <http://dx.doi.org/10.1007/s00450-010-0141-1>.

[30] GIERTSEN, C. “Volume Visualization of Sparse Irregular Meshes”, *IEEE Comput. Graph. Appl.*, v. 12, n. 2, pp. 40–48, 1992. ISSN: 0272-1716. doi: http://dx.doi.org/10.1109/38.124287.

[31] BERG, M. D., CHEONG, O., ET AL. *Computational Geometry: Algorithms and Applications*. 3rd ed. ed. Santa Clara, CA, USA, Springer-Verlag TELOS, 2008. ISBN: 3540779736, 9783540779735.

[32] SILVA, C. T., MITCHELL, J. S. B. “The Lazy Sweep Ray Casting Algorithm for Rendering Irregular Grids”, *IEEE Transactions on Visualization and Computer Graphics*, v. 3, n. 2, pp. 142–157, 1997. ISSN: 1077-2626. doi: http://dx.doi.org/10.1109/2945.597797.

[33] REED, D., YAGEL, R., ET AL. “Hardware assisted volume rendering of unstructured grids by incremental slicing”. In: *Proceedings of the 1996 symposium on Volume visualization*, VVS ’96, pp. 55–ff., Piscataway, NJ, USA, 1996. IEEE Press. ISBN: 0-89791-865-7. Disponível em: <http://dl.acm.org/citation.cfm?id=236226.236233>.

[34] SUNDEN, E., YNNERMAN, A., ROPINSKI, T. “Image Plane Sweep Volume Illumination”, *IEEE Transactions on Visualization and Computer Graphics*, v. 17, pp. 2125–2134, dez. 2011. ISSN: 1077-2626. doi: http://dx.doi.org/10.1109/TVCG.2011.211. Disponível em: <http://dx.doi.org/10.1109/TVCG.2011.211>.

[35] LEE, E. J., HODGES, L. F. “Run-Based Multi-Point Line Drawing”. 1993.

[36] ROKNE, J. G., WYVILL, B., WU, X. “Fast line scan-conversion”, *ACM Trans. Graph.*, v. 9, n. 4, pp. 376–388, out. 1990. ISSN: 0730-0301. doi: 10.1145/88560.88572. Disponível em: <http://doi.acm.org/10.1145/88560.88572>.

[37] WYVILL, B. “Graphics gems”. Academic Press Professional, Inc., cap. Symmetric double step line algorithm, pp. 101–104, San Diego, CA, USA, 1990. ISBN: 0-12-286169-5. Disponível em: <http://dl.acm.org/citation.cfm?id=90767.90794>.

[38] BRESENHAM, J. E. “Incremental Line Compaction”, *The Computer Journal*, v. 25, n. 1, pp. 116–120, 1982. doi: 10.1093/comjnl/25.1.116.

[39] RESHETOV, A. “Morphological antialiasing”. In: *Proceedings of the Conference on High Performance Graphics 2009*, HPG ’09, pp. 109–116, New York, NY, USA, 2009. ACM. ISBN: 978-1-60558-603-8. doi: http://doi.acm.org/10.1145/1572769.1572787. Disponível em: <http://doi.acm.org/10.1145/1572769.1572787>.

[40] JIMENEZ, J., GUTIERREZ, D., YANG, J. E. A. “Filtering approaches for real-time anti-aliasing”. In: *ACM SIGGRAPH 2011 Courses*, SIGGRAPH ’11, pp. 6:1–6:329, New York, NY, USA, 2011. ACM. ISBN: 978-1-4503-0967-7. doi: 10.1145/2037636.2037642. Disponível em: <http://doi.acm.org/10.1145/2037636.2037642>.

[41] BIRI, V., HERUBEL, A., DEVERLY, S. “Practical morphological antialiasing on the GPU”. In: *ACM SIGGRAPH 2010 Talks*, SIGGRAPH ’10, pp. 45:1–45:1, New York, NY, USA, 2010. ACM. ISBN: 978-1-4503-0394-1. doi: http://doi.acm.org/10.1145/1837026.1837085. Disponível em: <http://doi.acm.org/10.1145/1837026.1837085>.

Apêndice A

Código Fonte

A.1 *B-Convert*: versão utilizada no *BZSweep*.

```
void createCleverList( D2Pointi p0, D2Pointi p1, D2Pointi p2, vector<D2Pointi>& buf ) {
  buf.clear();
  if( p1.y < p0.y ) swapPoints( p0, p1 );
  if( p2.y < p0.y ) swapPoints( p0, p2 );
  if( p2.y < p1.y ) swapPoints( p1, p2 );

  unsigned x0, y0, x1, y1, x2, y2;
  x0 = p0.x; y0 = p0.y; x1 = p1.x; y1 = p1.y; x2 = p2.x; y2 = p2.y;

  vector<D2Pointi> Line0, Line1, Line2;
  vector<D2Pointi>::const_iterator it;
  unsigned i, nmin, nmax;
  float m10 = 0.0, m20 = 0.0;

  m20 = fabs((float) (y2 - y0) / (x2 - x0));
  m10 = fabs((float) (y1 - y0) / (x1 - x0));

  // Caso geral: y0 < y1 < y2
  getLine( x0, y0, x1, y1, Line0 ); // get line 0 (L0)
  getLine( x1, y1, x2, y2, Line1 ); // get line 1 (L1)
  getLine( x0, y0, x2, y2, Line2 ); // get line 2 (L2)

  buf.push_back(D2Pointi(y0, y2));  // first point holds min and max Y coords (y0-y2).

  if (y0 == y1 && y1 == y2) { // First case
    nmin = min(x0, x1);
    nmin = min(nmin, x2);
    nmax = max(x0, x1);
    nmax = max(nmax, x2);

    buf.push_back(D2Pointi( nmin, nmax ));

  } // Second case
  else if ( (m10 >= m20 && x2 <= x1 && x1 <= x0) || (m20 >= m10 && x0 <= x1 && x1 <= x2)
       || (x1 >= x0 && x1 >= x2) ) {
    for (i = 0; i <= y1 - y0; i++) { // During Line0's dy, push_back pixBuf values
      if (Line2[2 * i].x <= Line0[2 * i].x)
        buf.push_back(D2Pointi(Line2[2 * i].x, Line0[2 * i + 1].x)); // (xi, xf),
            starting from Line2.x to Line0.x
```

```
      else
        buf.push_back(D2Pointi(Line0[2 * i].x, Line2[2 * i + 1].x)); // (xi, xf),
            starting from Line0.x to Line2.x
    }

    if (Line2[2 * (y1 - y0)].x <= Line1[0].x) {  // Now, deal with Line1.  If Line2(dy).
        x <= Line1(0).x
      if (buf.back().x > Line2[2 * (y1 - y0)].x)
        buf.back().x = Line2[2 * (y1 - y0)].x; // (xi) Correct xi to Line2[ Line0' dy ].
            x

      if (buf.back().y < Line1[1].x)
        buf.back().y = Line1[1].x; // (xf) Correct xf to Line1' second index.x

    } else { // If Line2(dy).x > Line1(0).x

      if (buf.back().x > Line1[0].x)
        buf.back().x = Line1[0].x; //(xi) Correct xi to Line1' first index.x

      if (buf.back().y < Line2[2 * (y1 - y0) + 1].x)
        buf.back().y = Line2[2 * (y1 - y0) + 1].x; //(xf) Correct xf to Line2[ Line0' dy
             + 1 ].x
    }

    for (i = 1; i <= y2 - y1; i++) { // During Line1's dy, push_back pixBuf values
      if (Line2[2 * (i + y1 - y0)].x <= Line1[2 * i].x)
        buf.push_back(D2Pointi(Line2[2 * (i + y1 - y0)].x, Line1[2 * i + 1].x)); //(xi,
            xf): Line2[Line0'dy+i].x to Line1'next i.x
      else
        buf.push_back(D2Pointi(Line1[2 * i].x, Line2[2 * (i + y1 - y0) + 1].x)); //(xi,
            xf): Line1'i.x to Line2[ Line0'dy+i+1].x
    }
  }else if ( (m10 >= m20 && x0 <= x1 && x1 <= x2) || (m20 >= m10 && x2 <= x1 && x1 <= x0
      ) || (x1 <= x0 && x1 <= x2) ) {

    for (i = 0; i <= y1 - y0; i++) {
      if (Line0[2 * i].x <= Line2[2 * i].x)
        buf.push_back(D2Pointi(Line0[2 * i].x, Line2[2 * i + 1].x)); //(xi, xf): Line0'
            index.x to Line2' next index.x
      else
        buf.push_back(D2Pointi(Line2[2 * i].x, Line0[2 * i + 1].x)); //(xi, xf): Line2'
            index.x to Line2' next index.x
    }

    if (Line1[0].x <= Line2[2 * (y1 - y0)].x) {

      if (buf.back().x > Line1[0].x)
        buf.back().x = Line1[0].x; //(xi) Correct xi to Line1' first index.x

      if (buf.back().y < Line2[2 * (y1 - y0) + 1].x)
        buf.back().y = Line2[2 * (y1 - y0) + 1].x; //(xf) Correct xf to Line2[ Line0'dy
            + 1 ].x
    } else {

      if (buf.back().x > Line2[2 * (y1 - y0)].x)
        buf.back().x = Line2[2 * (y1 - y0)].x; //(xi) Correct xi to Line2[ Line0'dy ].x

      if (buf.back().y < Line1[1].x)
        buf.back().y = Line1[1].x; //(xf) Correct xf to Line1' second index.x
```

```
    }

    for (i = 1; i <= y2 - y1; i++) { // Now test the larger line against Line1' dy.
      if (Line1[2 * i].x <= Line2[2 * (i + y1 - y0)].x)
        buf.push_back(D2Pointi(Line1[2 * i].x, Line2[2 * (i + y1 - y0) + 1].x));  //(xi,
             xf): Line1'i.x to Line2[Line0'dy+i].x
      else
        buf.push_back(D2Pointi(Line2[2 * (i + y1 - y0)].x, Line1[2 * i + 1].x));  //(xi,
             xf): Line2[ Line0'dy+i].x to Line1'next i.x
    }
  }
}
```

Listing A.1: Compactação otimizada

```
void line2( unsigned x1, unsigned y1, unsigned x2, unsigned y2, vector<D2Pointi>& buf )
{

  int dx, dy, inx, iny, e;
  bool first = true;

  dx = x2 - x1;
  dy = y2 - y1;
  inx = dx > 0 ? 1 : -1;
  iny = dy > 0 ? 1 : -1;
  dx = abs(dx);
  dy = abs(dy);

  if (dx >= dy) {

    dy <<= 1;
    e = dy - dx;
    dx <<= 1;

    while (x1 != x2) {

      if (first) {
        buf.push_back(D2Pointi(x1, y1));
        first = false;
      }

      if (e >= 0) {

        if (!first) {
          D2Pointi tmp = buf.back();

          if (tmp.x > x1 && tmp.y == y1) {
            buf.pop_back();
            buf.push_back(D2Pointi(x1, y1));
            buf.push_back(tmp);

          } else { buf.push_back(D2Pointi(x1, y1));}
        }

        y1 += iny;
        e -= dx;
        first = true;
      }
```

```
      e += dy;
      x1 += inx;
    }

  } else {

    dx <<= 1;
    e = dx - dy;
    dy <<= 1;

    while (y1 != y2) {

      buf.push_back(D2Pointi(x1, y1));
      buf.push_back(D2Pointi(x1, y1));

      if (e >= 0) {

        x1 += inx;
        e -= dy;
      }
      e += dx;
      y1 += iny;
    }
  }

  if (buf.size()) {
    D2Pointi tmp = buf.back();

    if (tmp.x > x1 && tmp.y == y1) {
      buf.pop_back();
      buf.push_back(D2Pointi(x1, y1));
      buf.push_back(tmp);

    } else { buf.push_back(D2Pointi(x1, y1));}
  } else { buf.push_back(D2Pointi(x1, y1));}

  if (first) {buf.push_back(D2Pointi(x1, y1));}
}
```

Listing A.2: Traçado de retas de *Bresenham* modificado

A.2 Versão experimental, a partir dos vértices no espaço tridimensional

```
void createListCompact( D2Pointlf p0, D2Pointlf p1, D2Pointlf p2,
      class ViewPlane* vp,
      vector<D2Pointi>& pixel_buf) {
  unsigned int bucketCount;
  vector<D2Pointi> auxBuf;
  vector<D2Pointi>::const_iterator it;

  // Colocar em ordem os pontos em crescente de Y
  if( p1.y < p0.y ) swapPoints( p0, p1 );
  if( p2.y < p0.y ) swapPoints( p0, p2 );
  if( p2.y < p1.y ) swapPoints( p1, p2 );
```

```
  // Criar a lista de acordo com o caso do triangulo
  if( p0.y == p1.y ) {  // Caso 1:  p0.y == p1.y

    getLine( p0.x, p0.y, p2.x, p2.y, vp, auxBuf );
    bucketCount = auxBuf.size();
    getLine( p1.x, p1.y, p2.x, p2.y, vp, auxBuf );

  } else {
    if( p1.y == p2.y ) { // Caso 2:  p1.y == p2.y
      getLine( p0.x, p0.y, p2.x, p2.y, vp, auxBuf );
      bucketCount = auxBuf.size();
      getLine( p0.x, p0.y, p1.x, p1.y, vp, auxBuf );

    } else { // Caso 3:  p0.y < p1.y < p2.y
      getLine( p0.x, p0.y, p2.x, p2.y, vp, auxBuf );
      bucketCount = auxBuf.size();
      getLine( p0.x, p0.y, p1.x, p1.y, vp, auxBuf );
      getLine( p1.x, p1.y, p2.x, p2.y, vp, auxBuf, true );
    }
  }
  unsigned dyL02px = abs((int)(vp->w2sY(p2.y) - vp->w2sY(p0.y))) + 1;

  vector<PixelBin> pixelBins(1);
  pixelMapSort( auxBuf, pixelBins, pixel_buf, bucketCount, dyL02px );
}
```

Listing A.3: Compacta listas obtidas a partir dos vértices originais

```
// Remember that L02 may have consecutive elements within same pixel Y.
void pixelMapSort( vector<D2Point>& buf, vector<PixelBin>& pixelBins,
                   vector<D2Pointi>& pixel_buf, unsigned int l02Count,
                   unsigned binHeight ) {

  unsigned int i, j;
  PixelBin tmp;
  vector<PixelBin>::iterator b;
  vector<D2Point>::iterator p;
  //-------------------------------------------------------------------
  // Populate pixel buckets with L02 pixels each:
  //-------------------------------------------------------------------
  pixelBins[0].pixels.push_back( D2Pointi(buf[0].s.x, buf[0].s.y) );
  pixelBins[0].count = 1;
  tmp.count = 1;
  for ( i = 0, j = 1; j < l02Count; j++ ) { // next L02 points...

    if ( buf[j].s.y == pixelBins[i].pixels[0].y ) {

      pixelBins[i].pixels.push_back( D2Pointi(buf[j].s.x, buf[j].s.y) );
      pixelBins[i].count++;
    } else { // Condition to create new bucket in bucket list was met.

      // Dump pixel into new bucket.
      tmp.pixels.push_back(D2Pointi(buf[j].s.x, buf[j].s.y));
      pixelBins.push_back(tmp); // append new bucket at end of bucket list.
      i++; // move pointer to new bucket.
      tmp.pixels.clear(); // Destroy tmp bucket pixel array.
    }
  }
```

```
  //-------------------------------------------------------------------------
  // Populate pixel buckets with L01, L12 pixels in buffer (O(n)):
  //-------------------------------------------------------------------------
  for ( j = l02Count; j < buf.size(); j++ ) { // For each point in buf...
    for ( i = 0; i < pixelBins.size(); i ++ ) { // and each bucket:
      // Add point to corresponding bucket.
      if ( buf[j].s.y == pixelBins[i].pixels[0].y ) {
        pixelBins[i].pixels.push_back( D2Pointi(buf[j].s.x, buf[j].s.y) );
        pixelBins[i].count++;
      }
    }
  }
  compactPixelBinList( pixelBins, pixel_buf );
}
```

Listing A.4: Bucket Sort

```
void compactPixelBinList( vector<PixelBin>& pixelBins, vector<D2Pointi>& pixel_buf ) {

  unsigned int i, size, minX, maxX, y, yi, yf;
  vector<PixelBin>::iterator b;

  yi = pixelBins[0].pixels[0].y; // Get y value of L02 first pixel.
  yf = pixelBins[pixelBins.size()-1].pixels[0].y; // y value of L02 last pixel.

  pixel_buf.push_back(D2Pointi( yi, yf )); // Send to pixel_buf.

  for ( b = pixelBins.begin(); b != pixelBins.end(); b++ ) {
    size = (*b).count;
    y = (*b).pixels[0].y;

    if ( size > 2 ) { // If more then 2 entries find min and max, add to list.

      minX = (*b).pixels[0].x;
      maxX = (*b).pixels[0].x;

      for ( i = 0; i < size; i++ ) {
        minX = min( minX, (*b).pixels[i].x);
        maxX = max( maxX, (*b).pixels[i].x);
      }
      pixel_buf.push_back(D2Pointi( minX, maxX));
    }

    // If only two entries in bucket, check who has min x, add xi, xf to list.
    if ( size == 2 ) {
      if ( (*b).pixels[0].x > (*b).pixels[1].x )
        swapPoints( (*b).pixels[0], (*b).pixels[1] );

      pixel_buf.push_back(D2Pointi( (*b).pixels[0].x, (*b).pixels[1].x ));
    }
  }
}
```

Listing A.5: Compact Bucket Sorted

```
void simpleLine ( double x1, double y1, double x2, double y2, class ViewPlane* vp,
    vector<D2Point>& edge ) {

  double dx, dy, m, inv_m, x, y, stpx, stpy;
```

```
  dx = x2 - x1; dy = y2 - y1;
  m = dy*(1.0/dx); inv_m = 1.0/m;
  fabs(dx); fabs(dy);
  x = x1; y = y1;

  // These consider the deltas (dx and dy) of the current edge.
  stpx = dx*(1.0/(double)( abs((int)(vp->w2sX(x2) - vp->w2sX(x1))) + 1));
  stpy = dy*(1.0/(double)( abs((int)(vp->w2sY(y2) - vp->w2sY(y1))) + 1));

  double mstpx = m*stpx;
  double mstpy = inv_m*stpy;

  if ((dx > dy) || ( ( fabs(dx-dy)<=EPS))) { // if (dx >= dy)

    while ( ( fabs(x - x2) > EPS*fabs(x) ) || ( fabs(x - x2) > EPS*fabs(x2) ) ) { //
        while x1!=x2

      edge.push_back( D2Point( D2Pointlf(x, y), D2Pointi(vp->w2sX(x), vp->w2sY(y)) ) );
      x+=stpx; // x coord belongs to major axis
      y += mstpx; // DDA: y = m*(x-x1) + y1;
    }
  } else {

    while ( ( fabs(y - y2) > EPS*fabs(y) ) || ( fabs(y - y2) > EPS*fabs(y2) ) ) { //
        while y1!=y2

      edge.push_back( D2Point( D2Pointlf(x, y), D2Pointi(vp->w2sX(x), vp->w2sY(y)) ) );
      y+=stpy; // y coord belongs to major axis
      x += mstpy; // DDA: x = (y - y1)*inv_m + x1;
    }
  }
  edge.push_back( D2Point( D2Pointlf(x, y), D2Pointi(vp->w2sX(x), vp->w2sY(y)) ) );
}
```

Listing A.6: DDA sobre vértices originais

```
void line( double x1, double y1, double x2, double y2,
     class ViewPlane* vp, vector<D2Point>& edge,
     bool flOpenBegin, bool flOpenEnd )
{
  vector<D2Point> buf;
  double dx, dy, inx, iny, e, stpx, stpy, inv_m;
  double eps = 1e-6;
  dx = x2 - x1; dy = y2 - y1;

  inx = dx > 0 ? 1.0 : -1.0;
  iny = dy > 0 ? 1.0 : -1.0;

  dx = fabs(dx); dy = fabs(dy);

  if (inx > 0)
    stpx = dx/(double)( abs((int)(vp->w2sXr(x2) - vp->w2sX(x1))) + 1);
  else
    stpx = dx/(double)( abs((int)(vp->w2sX(x2) - vp->w2sXr(x1))) + 1);

  stpy = dy/(double)( abs((int)(vp->w2sY(y2) - vp->w2sYr(y1))) + 1);

  inx*=stpx;
  iny*=stpy;
```

```
  if (dx >= dy) {

    e = dy*stpx - dx*stpy*0.5; // initial e

    while ((fabs(x1 - x2) > eps*fabs(x1))||(fabs(x1 - x2) > eps*fabs(x2))){

      buf.push_back( D2Point( D2Pointlf(x1, y1), D2Pointi(vp->w2sXr(x1), vp->w2sYr(y1))
          ));

      if ( e >= -eps ) {
        y1 += iny;
        e-= dx*stpy;
      }

      e += dy*stpx;
      x1 += inx;
    }
  } else { // dy > dx

    e = dx*stpy - dy*stpx*0.5;

    while ((fabs(y1 - y2) > eps*fabs(y1))||(fabs(y1 - y2) > eps*fabs(y2))){

      buf.push_back( D2Point( D2Pointlf(x1, y1), D2Pointi(vp->w2sXr(x1), vp->w2sYr(y1))
          ));

      if ( e >= -eps ) {

        x1 += inx;
        e -= dy*stpx;
      }

      e += dx*stpy;
      y1 += iny;
    }
  }
  buf.push_back( D2Point( D2Pointlf(x1, y1), D2Pointi(vp->w2sXr(x1), vp->w2sYr(y1)) ));

  // send from buf to pixBuf.
  vector<D2Point>::iterator start = buf.begin();
  vector<D2Point>::iterator end   = buf.end();
  if( flOpenBegin ) start++;
  if( flOpenEnd   ) end--;
  for( ; start < end ; start++ )
    edge.push_back( *start );
}
```

Listing A.7: Bresenham sobre vértices originais

Apêndice B

Tempos

B.1 Tempos para *bzsweep* x *zsweep*

512^2	zsweep	bzsweep			$\frac{bzsweep}{zsweep}$
Dataset	*scan convert*	gerar listas	*scan convert*	total	
spx.off	12.39	80.35	8.11	88.45	7.14
spx2.off	217.6	1602.62	95.81	1698.43	7.81
post.off	94.03	500.68	42.86	543.55	5.78
oceanU.off	7.28	53.5	2.85	56.35	7.74
delta.off	221.13	1224.31	91.75	1316.06	5.95
f117.off	57.41	292.6	22.87	315.46	5.50
torso.off	272.27	1812.26	125.71	1937.97	7.12

Tabela B.1: Apenas *scan convert*, algoritmo original x projeção de faces por lista compacta (*bzsweep*). Resolução: 512^2 *pixels*. Tempo em milisegundos.

1024^2	zsweep	bzsweep			$\frac{bzsweep}{zsweep}$
Dataset	*scan convert*	gerar listas	*scan convert*	total	
spx.off	29.86	133.33	18.18	151.51	5.07
spx2.off	269.3	2138.5	147.68	2286.18	8.49
post.off	163.07	696.52	77.34	773.86	4.75
oceanU.off	12.14	75.14	7.89	83.03	6.84
delta.off	282.36	1598.21	134.34	1732.55	6.14
f117.off	73.17	394.44	32.42	426.86	5.83
torso.off	331.38	2554.47	172.05	2726.52	8.23

Tabela B.2: Apenas *scan convert*, algoritmo original x projeção de faces por lista compacta (*bzsweep*). Resolução: 1024^2 *pixels*. Tempo em milisegundos.

2048^2	zsweep	bzsweep			$\frac{bzsweep}{zsweep}$
Dataset	*scan convert*	gerar listas	*scan convert*	total	
spx.off	97.32	216.04	48.91	264.94	2.72
spx2.off	442.7	2912.17	277.46	3189.63	7.21
post.off	413.83	1008.82	186.34	1195.16	2.89
oceanU.off	31.42	115.3	20.38	135.69	4.32
delta.off	499.01	2212.12	251.53	2463.65	4.94
f117.off	128.9	540.09	60.26	600.35	4.66
torso.off	511.36	3467.35	308.96	3776.31	7.38

Tabela B.3: Apenas *scan convert*, algoritmo original x projeção de faces por lista compacta (*bzsweep*). Resolução: 2048^2 *pixels*. Tempo em milisegundos.

4096^2	zsweep	bzsweep			$\frac{bzsweep}{zsweep}$
Dataset	*scan convert*	gerar listas	*scan convert*	total	
spx.off	358.63	339.31	154.47	493.78	1.38
spx2.off	1094.14	4554.43	624.55	5178.98	4.73
post.off	1377.02	1551.26	562.12	2113.38	1.53
oceanU.off	100.17	184.67	60.53	245.2	2.45
delta.off	1301.68	3309.07	624.6	3933.67	3.02
f117.off	334.75	791.46	148.47	939.93	2.81
torso.off	1170.82	5090.12	682.44	5772.56	4.93

Tabela B.4: Apenas *scan convert*, algoritmo original x projeção de faces por lista compacta (*bzsweep*). Resolução: 4096^2 *pixels*. Tempo em milisegundos.

8192^2	zsweep	bzsweep			$\frac{bzsweep}{zsweep}$
Dataset	*scan convert*	gerar listas	*scan convert*	total	
spx.off	1359.29	558.07	545.53	1103.61	0.81
spx2.off	3601.23	7209.09	1688.01	8897.1	2.47
post.off	5106.3	2502.84	1945.81	4448.65	0.87
oceanU.off	382.22	300.8	205.57	506.38	1.32
delta.off	4368.24	5104.72	1913.24	7017.96	1.61
f117.off	1121.78	1180.41	460.2	1640.61	1.46
torso.off	3713.23	7949.56	1820.31	9769.87	2.63

Tabela B.5: Apenas *scan convert*, algoritmo original x projeção de faces por lista compacta (*bzsweep*). Resolução: 8192^2 *pixels*. Tempo em milisegundos.

16384^2	zsweep	bzsweep			$\frac{bzsweep}{zsweep}$
Dataset	*scan convert*	gerar listas	*scan convert*	total	
spx.off	5286.64	1002.82	2045.73	3048.55	0.58
spx2.off	13254.74	11570.4	5341.69	16912.09	1.28
f117.off	4203.25	1904.93	1625.48	3530.41	0.84
torso.off	13471.97	12719.3	5721.70	18441	1.37

Tabela B.6: Apenas *scan convert*, algoritmo original x projeção de faces por lista compacta (*bzsweep*). Resolução: 16384^2 *pixels*. Tempo em milisegundos.

32768^2	zsweep	bzsweep			$\frac{bzsweep}{zsweep}$
Dataset	*scan convert*	gerar listas	*scan convert*	total	
spx.off	20938.82	2010.8	7999.84	10010.64	0.48

Tabela B.7: Apenas *scan convert*, algoritmo original x projeção de faces por lista compacta (*bzsweep*). Resolução: 32768^2 *pixels*. Tempo em milisegundos.

Apêndice C

Renderings

C.1 Imagens dos *datasets* em 4096^2 *pixels*

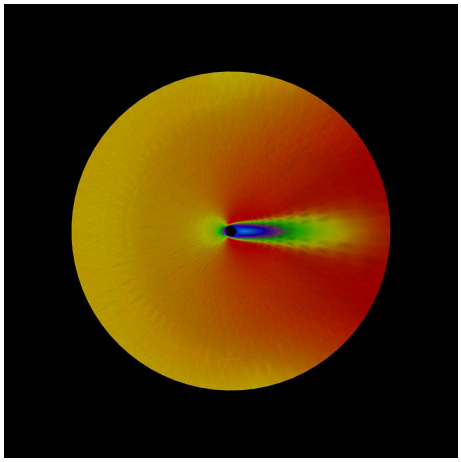

Figura C.1: *post.off*

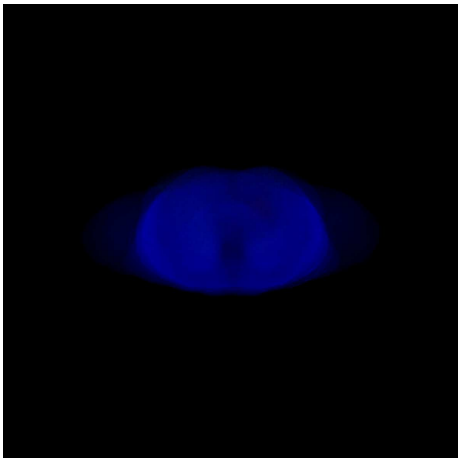

Figura C.2: *torso.off*

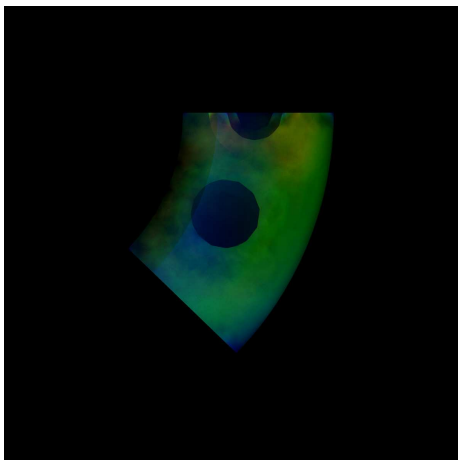

Figura C.3: *spx.off*

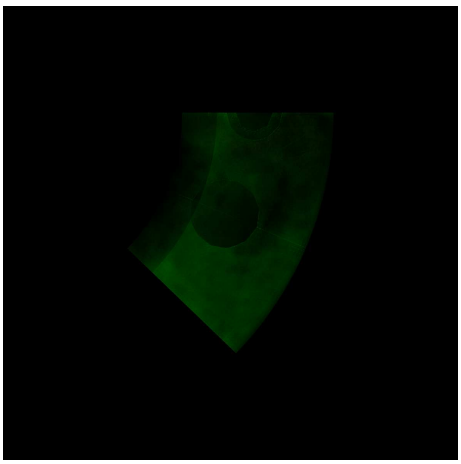

Figura C.4: *spx2.off*

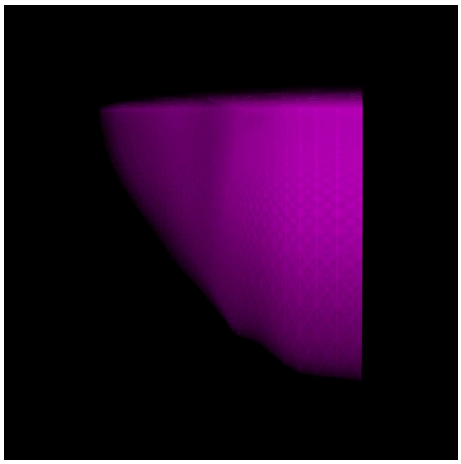

Figura C.5: *delta.off*

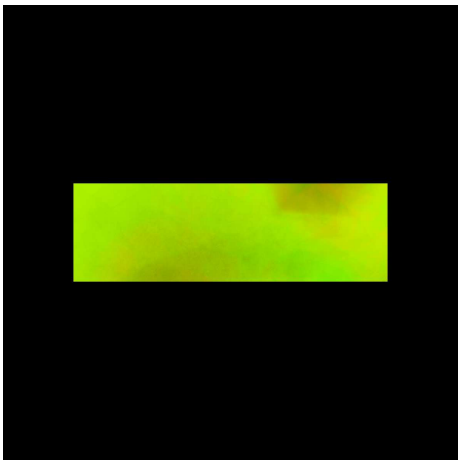

Figura C.6: *f117.off*

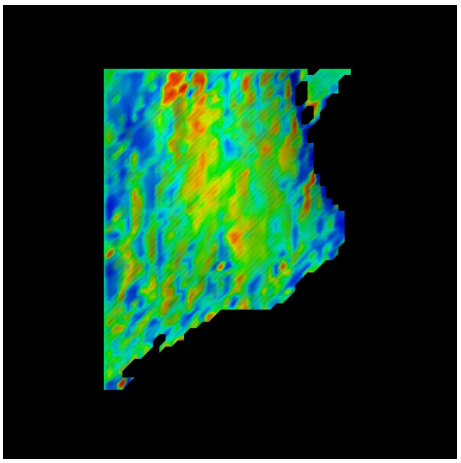

Figura C.7: *oceanU.off*