FLÁVIO SZENBERG

UM ALGORITMO PARA VISUALIZAÇÃO DE TERRENOS COM OBJETOS

DISSERTAÇÃO DE MESTRADO

DEPARTAMENTO DE INFORMÁTICA

PUC-Rio

Rio de Janeiro, 27 de agosto de 1997

FLÁVIO SZENBERG

UM ALGORITMO PARA VISUALIZAÇÃO DE TERRENOS COM OBJETOS

Dissertação apresentada ao Departamento de Informática da PUC-Rio como parte dos requisitos para a obtenção do título de Mestre em Ciências em Informática.

Orientador: Marcelo Gattass

Co-orientador: Paulo Cezar Pinto Carvalho

Departamento de Informática

Pontifícia Universidade Católica do Rio de Janeiro

Rio de Janeiro, 27 de agosto de 1997

Aos Meus Pais

AGRADECIMENTOS

Aos meus pais, irmã e cunhado pela compreensão, apoio e ajuda durante todo o tempo.

Aos meus sobrinhos, Rosa Malca Dorf e Samuel Moshé Dorf, pelas diversas vezes que não foi possível brincar com eles tendo que dedicar-me ao trabalho.

Ao orientador, Prof. Marcelo Gattass, pelo apoio, incentivo e orientação em todo o tempo e pela confiança depositada.

Ao co-orientador, Prof. Paulo Cezar Pinto Carvalho, pela ajuda, orientação e as valiosas contribuições ao longo da construção deste trabalho.

Aos professores do Departamento de Informática, ao Prof. Marcelo Dreux, ao Prof. Jonas de Miranda Gomes e outros pelas informações passadas em seus cursos.

Ao Prof. Luiz Fernando Martha e ao Prof. Luiz Carlos Guedes pelos conhecimentos passados para mim.

Aos amigos e companheiros do TeCGraf/PUC-Rio que contribuíram direta ou indiretamente para a realização deste trabalho.

À CAPES, CNPq e PROTEM/CC-GEOTEC pelo apoio financeiro.

RESUMO

Este trabalho descreve um método para a visualização de terrenos e objetos por meio da combinação de dois algoritmos, um para terrenos e um para objetos. Nosso propósito é gerar, eficaz e rapidamente, imagens aéreas de um terreno com objetos como casas, veículos e linhas de transmissão e assim permitir um vôo simulado. Para os objetos, descritos por linhas e polígonos, é usado o algoritmo de Z-Buffer; para o terreno, descrito através de mapas de alturas, é usado o algoritmo de lançamento de raios aperfeiçoado, chamado de algoritmo de Horizonte Flutuante.

ABSTRACT

This work describes a method for the visualization of terrains and objects by means of the combination of two algorithms, one for terrains and one for objects. Our purpose is to generate, effective and quickly, aerial images of a terrain with objects as houses, vehicles and transmission lines and like this to allow a simulate flight. For the objects, described by lines and polygons, the algorithm of Z-Buffer is used; for the terrain, described through maps of heights, the algorithm of improved release of rays is used, called of algorithm of Flotation Horizon.

SUMÁRIO

AGRADECIMENTOS .... **I**

RESUMO .... **II**

ABSTRACT .... **III**

SUMÁRIO .... **IV**

LISTA DE FIGURAS .... **VI**

LISTA DE TABELAS .... **VIII**

LISTA DE NOTAÇÕES .... **IX**

1. INTRODUÇÃO **1**

1.1. TRABALHOS RELACIONADOS 2

1.2. ORGANIZAÇÃO DO TRABALHO 4

2. DESCRIÇÃO DO PROBLEMA **5**

3. RENDERIZAÇÃO VETORIAL DO TERRENO **9**

4. ALGORITMOS EFICIENTES PARA TERRENOS **14**

4.1. ANÁLISE DA COMPLEXIDADE 18

5. ALGORITMO COMPOSTO **19**

5.1. MODELO DE CÂMARA 20

5.2. CÁLCULO DE PROFUNDIDADE 23

6. APERFEIÇOAMENTO DA IMAGEM DO ALGORITMO DE TERRENO **26**

6.1. AUMENTO DO NÚMERO DE FATIAS 27

6.2. DIVISÃO DOS VOXELS 29

6.3. COMBINAÇÃO DAS SOLUÇÕES 31

7. IMPLEMENTAÇÃO **33**

8. EXEMPLOS **42**

8.1. TERRENO COM OBJETOS 42

8.2. TABULEIRO DE XADREZ 47

8.3. São José dos Campos 50

9. CONCLUSÕES 54

9.1. Sugestões para Trabalhos Futuros 55

REFERÊNCIAS 57

LISTA DE FIGURAS

Figura 2.1 - Mapas de terreno. 6

Figura 3.1 - Triangulação dos topos, construída pelas diagonais. 9

Figura 3.2 - Visualização de Terreno com Z-Buffer. 13

Figura 4.1 - A posição especial da câmara. 15

Figura 4.2 - Projeção cônica simples. 15

Figura 5.1 - Estratégia para combinar ambos os algoritmos. 19

Figura 5.2 - Parâmetros da câmara. 20

Figura 5.3 - Frustum de visão. 20

Figura 5.4 - Definição do plano de cerceamento *near*. 21

Figura 5.5 - Distâncias na direção de $raio_k$. 24

Figura 6.1 - Problemas com as regiões entre fatias. 27

Figura 6.2 - Inclusão de mais fatias. 28

Figura 6.3 - Subdivisão de um *voxel*, com contribuição dos *voxels* vizinhos. 30

Figura 7.1 - Definições dos raios. 35

Figura 8.1 - Imagem gerada pelo Horizonte Flutuante. 42

Figura 8.2 - Objetos a serem adicionados ao terreno. 43

Figura 8.3 - Visão da planta do terreno com objetos. 43

Figura 8.4 - Imagem com terreno e objetos. 44

Figura 8.5 - Visualização do terreno com mais fatias. 46

Figura 8.6 - Visualização do terreno com interpolação. 47

Figura 8.7 - Visualização de um terreno xadrez. 47

Figura 8.8 - Visualização de um terreno xadrez com mais fatias. 48

Figura 8.9 - Visualização de um terreno xadrez com interpolação. 48

Figura 8.10 - Visualização de um terreno xadrez com mais fatias e interpolação. 48

Figura 8.11 – Relação entre os pixels (*a*) do plano de visão com os pontos do terreno (*b*). 49

Figura 8.12 - Gráfico de $g(x)$ gerada automaticamente pelo Maple V. 50

Figura 8.13 - Gráfico de $g(x)$ gerado pelo Maple V com intervalo de $\Delta x=1$. 50

Figura 8.14 - Visualização de São José dos Campos. 51

Figura 8.15 - Visualização de São José dos Campos com mais fatias. 51

Figura 8.16 - Visualização de São José dos Campos com interpolação. 52
Figura 8.17 - Visualização de São José dos Campos com mais fatias e interpolação. 52
Figura 8.18 - Mapas de um terreno de São José dos Campos - SP... 53

LISTA DE TABELAS

Tabela 3.1 - Tempo em segundos para gerar as imagens da Figura 3.2. 13

Tabela 8.1 - Tempo em segundos para cada passo da estratégia proposta (ponto fixo). 45

Tabela 8.2 - Tempo em segundos para cada passo da estratégia proposta (ponto flutuante). .. 45

Tabela 8.3 - Taxa em quadros por segundo da animação com Z-Buffer. 46

LISTA DE NOTAÇÕES

h ....Altura da janela da aplicação.

w ....Largura da janela da aplicação.

eye ....Posição do observador.

ref...................Ponto de referencia de visão.

vup ....Vetor de inclinação do observador.

n, near............Distância do plano frontal do frustum de visão.

f, far ....Distância do plano distante do frustum de visão.

l, left...............Limite esquerdo do frustum de visão, definido no plano frontal.

r, right............Limite direito do frustum de visão, definido no plano frontal.

t, top...............Limite superior do frustum de visão, definido no plano frontal.

b, bottom........Limite inferior do frustum de visão, definido no plano frontal.

θ.....................Ângulo do cone de visão (frustum).

d_{pk}...................Distância do plano de projeção, utilizado no algoritmo do horizonte flutuante, ao observador.

m_0...................Inclinação inicial do raio (horizonte).

m_j....................Inclinação do raio relativo ao *j*-ésimo *pixel* (de baixo para cima)de uma coluna da tela.

incl.................Deslocamento vertical do plano de projeção, em pixels da tela.

h_{pi}...................Tamanho do *i*-ésimo *voxel* da direção do raio, projetado na tela.

Capítulo 1

1. Introdução

Este trabalho descreve um método para visualizar terrenos e objetos por meio de dois algoritmos, um específico para terrenos e o outro para objetos descritos por polígonos e linhas. O propósito é gerar, com eficiência e rapidez, imagens aéreas de terreno com objetos como casas, veículos e linhas de transmissão, permitindo assim um vôo simulado.

Neste tipo de aplicação, estamos principalmente preocupados com o desempenho do algoritmo empregado, de modo a permitir a interatividade na navegação. Um obstáculo a ser enfrentado é que a representação detalhada da geometria e da textura de um terreno a ser sobrevoado exige uma grande quantidade de memória. Além disso, embora a porção do terreno envolvida em uma cena represente, de modo geral, uma pequena parcela dessa informação, sua visualização utilizando recursos genéricos de um sistema gráfico pode inviabilizar a interatividade desejada. Tais questões têm motivado pesquisas tanto em técnicas para sucintamente representar terrenos, seja de modo vetorial ou raster, quanto em algoritmos otimizados de visualização.

Neste trabalho, consideramos a situação em que a cena contém, além do terreno, objetos poliedrais representados por polígonos e linhas. Esses objetos podem ser visualizados com grande eficiência pelos sistemas gráficos padrões disponíveis em *workstations* modernas e PCs, como o OpenGL [Neider+93]. Comparamos os resultados obtidos com duas abordagens para a visualização de cenas contendo um terreno e objetos poliedrais. A primeira consiste em empregar o sistema gráfico OpenGL para visualizar terreno e objetos. A segunda abordagem consiste em utilizar um algoritmo otimizado para visualizar o terreno; a imagem e as informações de profundidade obtidas são, então, passadas ao OpenGL para serem integradas à cena contendo os objetos. São apresentados resultados comparativos dos dois métodos.

1.1. Trabalhos Relacionados

Uma primeira etapa para qualquer processo de visualização diz respeito à representação e modelagem do mundo. Muitas vezes o tipo de modelagem original não satisfaz as especificações de parâmetros de entrada para os algoritmos de visualização. [Kaneda+89] descreve um problema deste tipo. O terreno original está descrito por linhas de contorno e o algoritmo proposto se baseia em uma malha. Com isto, é necessário transformar esta modelagem em uma outra baseada em uma malha de dados. A partir deste novo modelo é apresentado um algoritmo de visualização do terreno a partir de uma transformação perspectiva e um algoritmo de remoção de superfícies escondidas. O mesmo trabalho descreve, ainda, um método para inserção de árvores no terreno a partir de uma modificação na malha, que recalcula um novo mapa de alturas e textura, e também uma técnica de sombriamento para um maior grau de realismo.

Em [Camara+96] são descritos estruturas de dados para modelos de terrenos e diversas áreas de aplicações onde sua visualização é importante. [Paglieroni+94] propôs um algoritmo baseado em traçados de raios chamado *Height Distributional Distance Transform* (HDDT), que utiliza resultados de um pré-processamento do mapa de alturas. Este pré-processamento tem a finalidade de diminuir o número de cálculos de interseção de raios com o terreno.

[Cohen+94] apresenta um algoritmo de visualização terreno foto-realista onde os terrenos são descritos a partir de dois conjuntos de dados, um contendo a textura e outro as alturas. A proposta se baseia em "voxelizar" toda a cena, inclusive objetos tais como casas e tanques e utilizar o algoritmo de *ray-casting*, onde os raios percorrem o terreno visualizando primeiro pontos mais próximos do observador. Implicitamente, aí está sendo aplicado um método de remoção de superfícies escondidas. Como deseja-se ter uma simulação de vôo, isto é, uma aplicação interativa, é proposto o uso de processamento paralelo. [tvcg+96] apresenta em mais detalhes as idéias propostas em [Cohen+94]. [Graf+94] propõe um algoritmo para visualização de informações geofráficas, também baseado em lançamento de raios, que combina três tipos de dados: terreno, céu e objetos. É apresentado um aplicativo chamado RAYSHADE que utiliza as técnicas propostas para a renderização da cena. Como a massa de dados usado é muito grande, da ordem de 1,5Gb, os dados são descritos em

múltiplos arquivos. Não é proposta uma navegação em tempo real mas é desejado um grau de realismo maior, com o uso de técnicas tais como efeitos atmosféricos.

Algoritmos eficientes com relação a velocidade de visualização são apresentados em [Freese+95] e [LaMothe95]. Os dois apresentam técnicas de visualização baseadas em lançamento de raios. A diferença dos algoritmos propostos em cada um é que no primeiro o terreno é desenhado de um ponto mais distante do observador para um ponto mais perto ao longo do raio, tendo a vantagem de não requerer nenhum cálculo de interseção, enquanto que na segunda referência os pontos do terreno são pintados na tela de um ponto mais perto do observador para um mais distante ao longo do raio, tendo a maior vantagem de não pintar na tela um ponto mais de uma vez. Estes algoritmos foram analisados em [Frederick+96], identificando as distorções geradas na visualização. Tais distorções são indesejáveis quando queremos inserir objetos, por causarem imperfeições no casamento do terreno com objetos. Nos algoritmos propostos, o observador tem total liberdade de translação mas apenas uma liberdade muito restrita na rotação (o observador não pode inclinar na vertical a cabeça mas para simular esta inclinação é sugerido um deslocamento do plano de projeção na vertical). Para solucionar este problema, [GuGaCa97] descreve um algoritmo de visualização de terrenos com uma liberdade a mais de câmara, isto é, o observador além de poder girar ao redor do eixo vertical pode inclinar o plano de projeção (inclinação da direção de visualização).

Neste trabalho o problema de visualizar terrenos com objetos é resolvido através de um método que não requer processamento paralelo visando uma interatividade na navegação. O algoritmo proposto para visualizar terrenos não causa distorções na cena e o algoritmo para combinar os objetos com os terrenos é o algoritmo de Z-Buffer, utilizado pela biblioteca OpenGL, que por ser uma biblioteca padrão permite que sua implementação seja a mesma para diversas plataformas. No algoritmo para terrenos existem parâmetros para permitir uma melhor visualização, como suavizações de topologia e textura. Porém estas melhorias em visualização reflete em uma degradação na interatividade. Um dos objetivos principais deste trabalho é propor um novo algoritmo para visualizar terrenos que tenha um resultado sem distorções e um cálculo de profundidade para cada ponto cena eficiente, de modo que não influa no desempenho. O motivo principal de não seguir a idéia de “voxelizar” os objetos

proposto em [Cohen+94] é que para realizar esta tarefa de transformar os objetos em uma porção de cubos necessita de um pré-processamento caro além de poder não representar corretamente o objeto original.

1.2. Organização do trabalho

O Capítulo 2 apresenta o problema e a descrição da representação de terrenos.

No Capítulo 3 são apresentadas diversas formas de se modelar terrenos e suas visualizações utilizando um sistema gráfico genérico (OpenGL).

O Capítulo 4 apresenta um algoritmo particular para visualizar terrenos a partir de uma técnica de lançamento de raios.

O Capítulo 5 descreve o modelo de câmara utilizado no algoritmo proposto no Capítulo 4 e um método para calcular as profundidades, em sistemas de coordenadas da tela, de cada ponto do terreno visualizado.

No Capítulo 6 são descritos os problemas decorrentes das aproximações feitas pelo algoritmo proposto no Capítulo 4, com algumas soluções.

O Capítulo 7 apresenta detalhes de implementação.

O Capítulo 8 contém imagens resultantes da visualização de terrenos e objetos, com os tempos necessários para gerá-las.

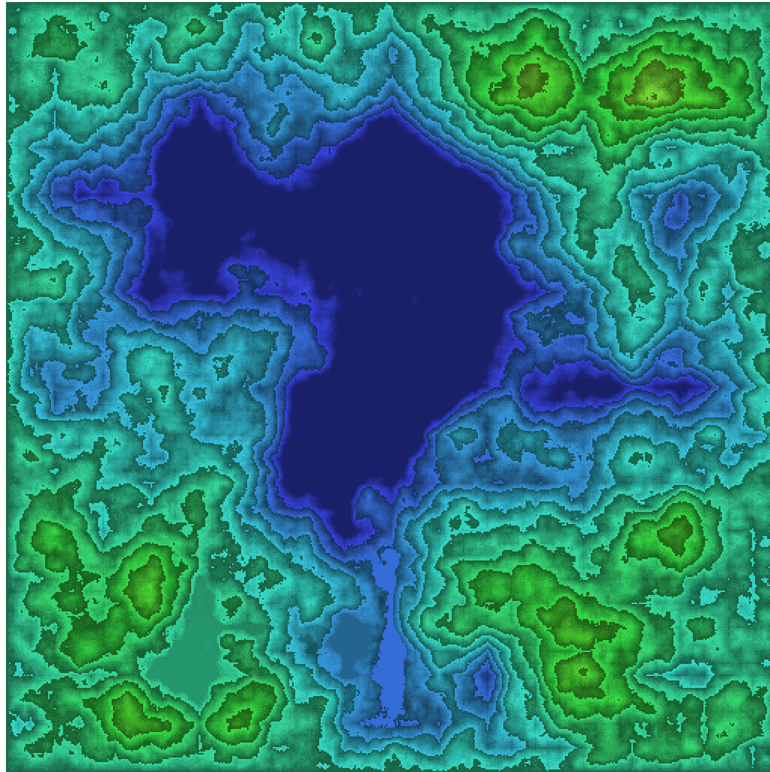
No Capítulo 9 são apresentadas as conclusões baseadas nos testes e sugestões para trabalhos futuros.

Capítulo 2

2. Descrição do Problema

Do ponto de vista conceitual, um terreno pode ser visto como um campo geográfico [Camara+96] caracterizado pelas alturas associadas a pontos de uma superfície de referência. No trabalho, consideraremos apenas terrenos de dimensões moderadas, para as quais a superfície de referência possa ser tomada como um plano.

A representação de um terreno em computador envolve, necessariamente, alguma forma de discretização, seja por meio de uma grade (normalmente regular) ou por uma Rede Irregular Triangular (TIN). Neste trabalho, os terrenos são representados por grades regulares e descritos por duas matrizes bidimensionais de mesmas dimensões, uma determinando a altura a cada ponto (mapa de alturas) e outra determinando a textura (mapa de cores). São ilustrados exemplos destes mapas na Figura 2.1. Estas imagens foram geradas pelo programa VistaPro [VistaPro] e alteradas para a inclusão da estrada. As imagens geradas pelo VistaPro têm uma iluminação implícita, o que contribui para aumentar o grau de realismo sem degradar o algoritmo de visualização com modelos de iluminação caros.


(a) Mapa de alturas.

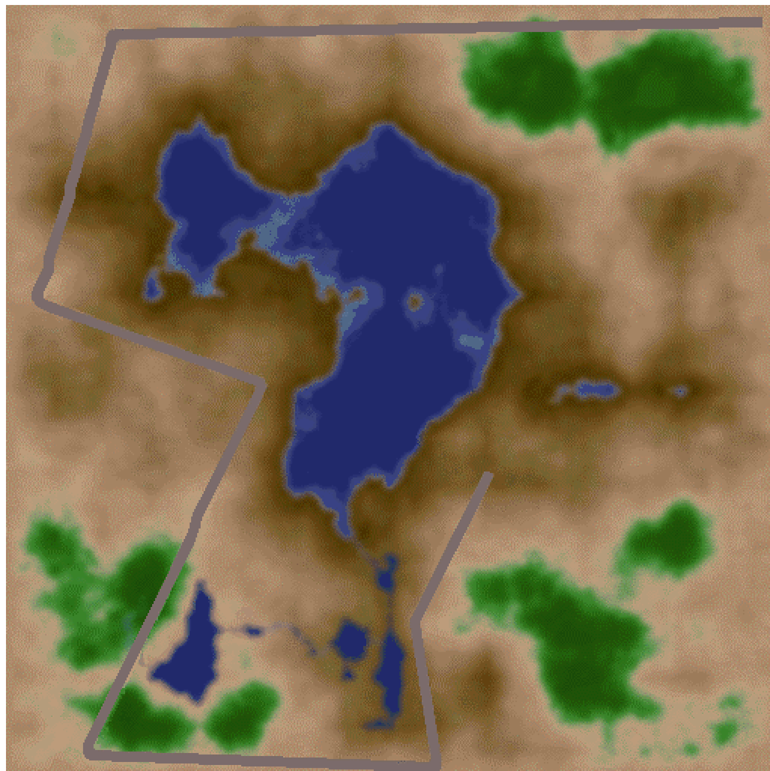

(b) Mapa de cores.

Figura 2.1 - Mapas de terreno.

A forma de representação descrita acima fornece um modelo geométrico para um terreno, que pode ser considerado como um conjunto de paralelepípedos retangulares alinhados pelos eixos, com largura e comprimento iguais à largura de cada elemento da grade regular e altura dada pelo valor correspondente no mapa de alturas. Se associarmos a cada face de cada um destes paralelepípedos a cor fornecida pelo elemento correspondente no mapa de textura, teremos um modelo vetorial do terreno, que pode ser visualizado, assim, por meio de um sistema genérico, como o OpenGL. A vantagem desta aproximação é a integração imediata de objetos vetoriais ao terreno, desde que a mesma descrição seja usada para o terreno e para os objetos. A grande desvantagem é o grande número de faces a serem visualizadas, o que pode impedir a visualização interativa (veja Capítulo 3).

Uma alternativa consiste em olhar o terreno sob o ponto de vista volumétrico. Neste caso, consideramos que os paralelepípedos descrevem uma ocupação espacial pelo terreno. Devido a esta interpretação, na literatura de jogos ([Freese+95] e [LaMothe95]) é comum chamar cada um destes paralelepípedos de *voxel* (*Volume Pixel*). Existem duas classes de algoritmos de visualização: uma classe onde os algoritmos se baseiam na imagem e outra classe onde os algoritmos se baseiam em objetos. Como será visto no Capítulo 4, os algoritmos de visualização pertencentes à classe dos algoritmos baseados em imagens para a visualização de terrenos podem ser aperfeiçoados e assim cumpridas as exigências de visualização interativa. O uso de tais algoritmos, porém, é dificultado pela presença de objetos vetoriais a serem adicionados à cena. Uma solução é obter uma representação volumétrica dos objetos para serem postos em cena, como é proposto em [tvcg+96]. Tal solução, não obstante, além de envolver um esforço de pré-processamento considerável, não funciona para objetos arbitrários: a estrutura do terreno deve ser preservada depois de serem colocados os objetos. Em outras palavras, cada objeto tem que permanecer fixo no terreno e cada linha reta vertical com pontos comuns ao terreno tem que cruzá-lo de acordo com um segmento que tem um fim no terreno.

No Capítulo 4 investigaremos outra alternativa para incorporar objetos vetoriais ao terreno, na qual são processados terreno e objetos separadamente e faz-se uso de algoritmos eficientes para cada tipo de dados. Então são combinadas as imagens resultantes de cada

processo em uma imagem, levando em consideração as informações de profundidade extraídas de cada algoritmo.

Capítulo 3

3. Renderização Vetorial do Terreno

Um terreno modelado como um conjunto de paralelepípedos, conforme descrito no capítulo anterior, pode ser representado visualmente por:

[a] pontos nas faces de topo;

[b] faces de topo;

[c] faces frontais;

[d] segmentos de retas verticais;

[e] os próprios paralelepípedos (constituídos de 6 faces);

[f] triângulos gerados a partir das faces de topo (triângulos *flats*); e

[g] triângulos gerados a partir das faces de topo, com diferença de suavização nas cores (triângulos *suaves*).

A velocidade de visualização e a qualidade da imagem gerada variam de acordo com a representação usada.

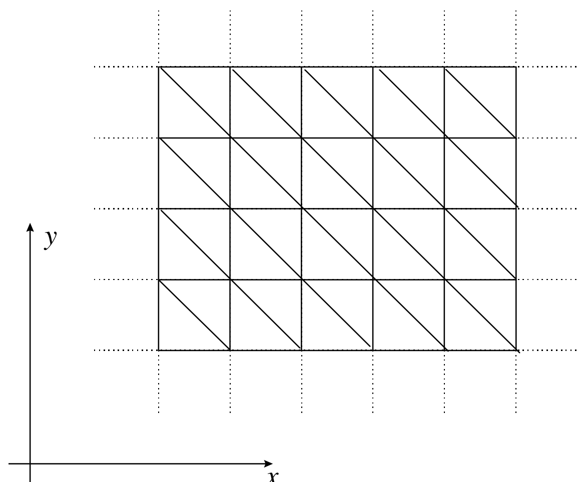
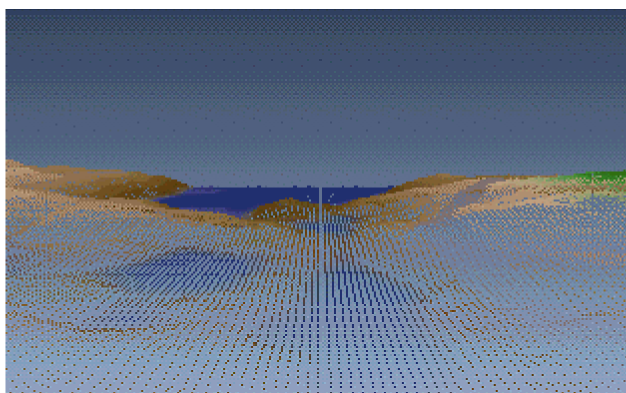


Figura 3.1 - Triangulação dos topos, construída pelas diagonais.

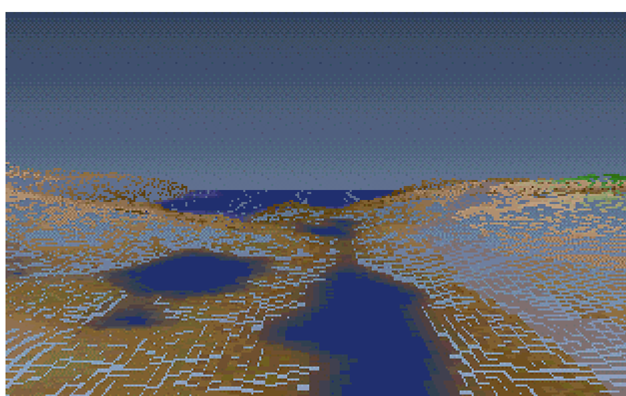
As triangulações geradas para as representações [f] e [g] se baseiam na Figura 3.1. Essas triangulações são boas opções de representação para o OpenGL pois permitem utilizar uma primitiva de desenho muito rápida, como veremos na Tabela 3.1. Com esta representação, temos uma visualização do terreno de forma contínua, pois aqui no mapa de alturas (e mapa de cores para o caso de visualização suave) os valores são dados pontualmente e com isto cada vértice de cada triângulo é dado por uma altura (e cor) no ponto relativo a esse vértice e cada altura (e cor) no interior do triângulo é calculada por uma interpolação dos vértices.

A Figura 3.2 ilustra as imagens obtidas pelo algoritmo de Z-Buffer para cada uma das representações acima. A Tabela 3.1 mostra o tempo, em segundos, para se gerar cada uma das imagens da Figura 3.2, utilizando um PC PENTIUM 166MHz e uma *workstation* Silicon Indigo 2 (SGI).

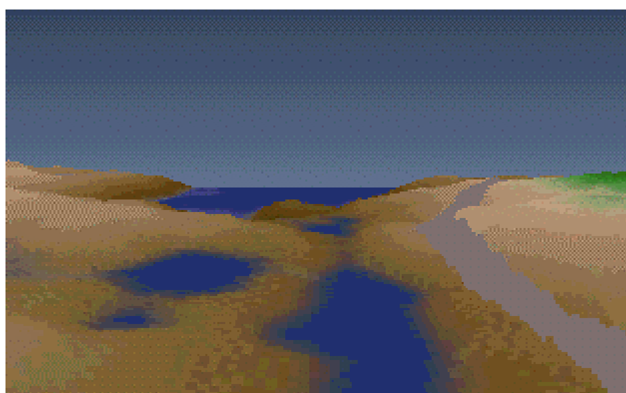
Os tempos mostrados na Tabela 3.1 são muito grandes para suportarem uma visualização interativa somente do terreno. Assim, as representações vetoriais do terreno apresentadas tiveram rendimentos de desempenho inaceitável com os computadores disponíveis hoje.



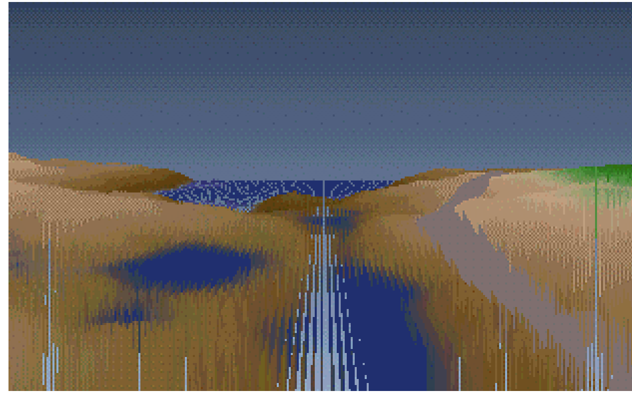
(a) Pontos nas faces de topo.

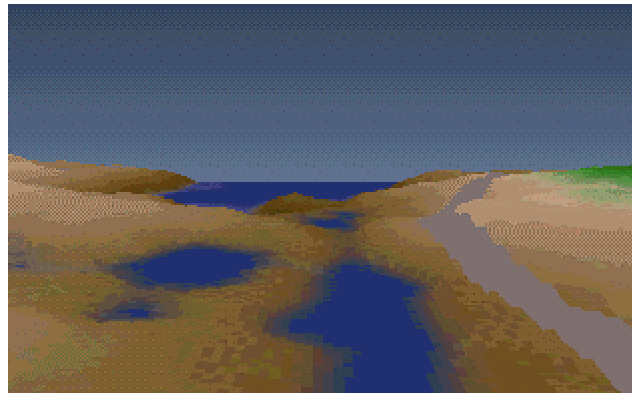


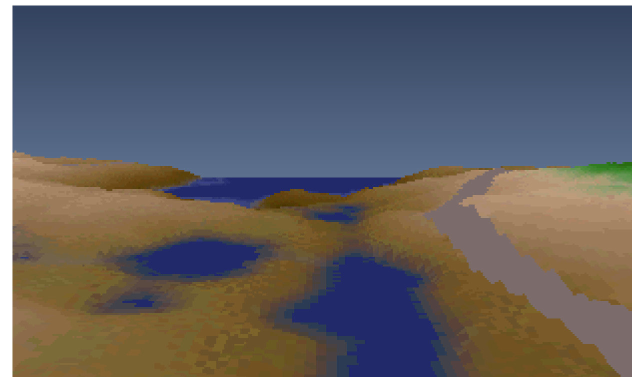
(b) Faces de topo.

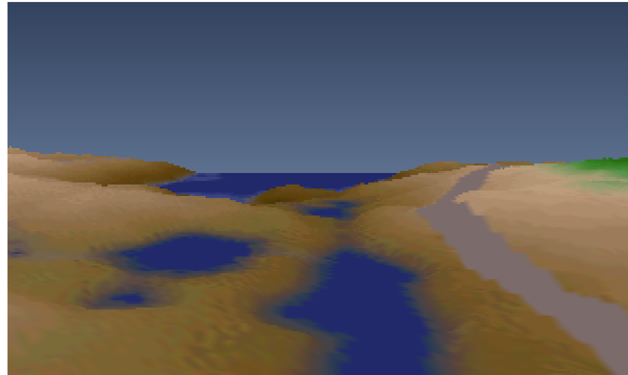


(c) Faces frontais.


(d) Segmentos de reta.


(e) Paralelepípedos.


(f) Triângulos *flats*.



(g) Triângulos suaves.

Figura 3.2 - Visualização de Terreno com Z-Buffer.

Terreno representado por	**PC**	**SGI**
[a] Pontos nas faces de topo	1,6	1,2
[b] Faces de topo	3,5	2,0
[c] Faces Frontais	4,2	2,1
[d] Segmentos de reta	2,3	2,2
[e] Paralelepípedos	14,2	6,2
[f] Triângulos *flats*	2,6	2,2
[g] Triângulos suaves	2,8	2,3

Tabela 3.1 - Tempo em segundos para gerar as imagens da Figura 3.2.

Capítulo 4

4. Algoritmos Eficientes para Terrenos

Outra alternativa clássica para terrenos é usar o algoritmo de Ray-Casting. Este algoritmo consiste em lançar raios do olho do observador para cada posição do plano de projeção que corresponda a um *pixel* na tela. A cor deste *pixel* é obtida no mapa de cores na posição de interseção entre o raio e o terreno. Se nenhuma consideração especial for feita, a eficiência deste algoritmo será muito baixa, pois o número de interseções a ser calculado é muito alto. Por exemplo, considere uma tela com 320x200 *pixels* e um terreno em uma grade de 512x512. O número de interseções neste caso é de mais de 16 bilhões por quadro de animação. Nenhum teste é preciso para indicar que esta estratégia não parece alcançar um tempo interativo provável. Portanto, faz-se necessária uma versão aperfeiçoada.

Assumiremos que a cabeça do observador está na vertical e que o plano de projeção é perpendicular à grade *xy* do terreno, como mostrado na Figura 4.1. Nesta posição particular, são lançados raios do observador para cada coluna de *pixels* no plano de visão, formando um plano que também é perpendicular ao plano de *xy*. Este, indicado na Figura 4.1 como plano de amostragem, simplifica o problema de visualização em grande parte. Só os *voxels* de terreno que são interceptados por este plano podem influenciar a coluna de *pixels* do plano de visão. Além disso, se os *voxels* são amostrados em intervalos uniformes ao longo da interseção, a projeção é reduzida a um problema 2D simples, conforme ilustrado na Figura 4.2.

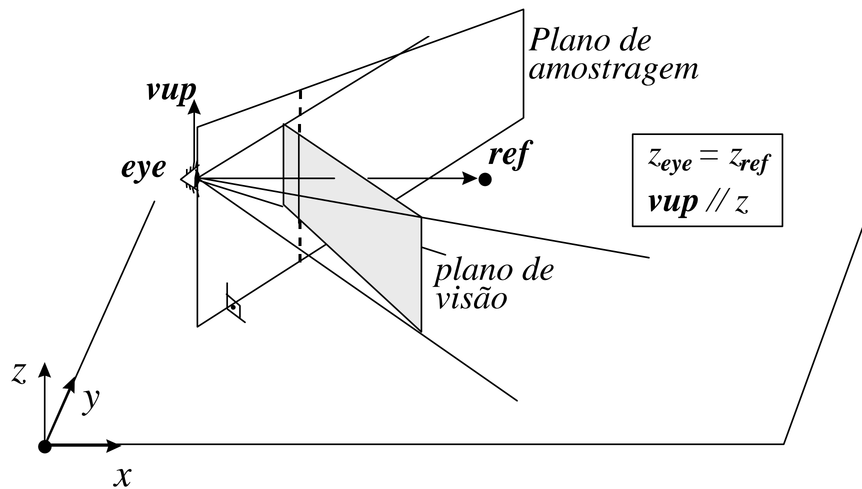


Figura 4.1 - A posição especial da câmara.

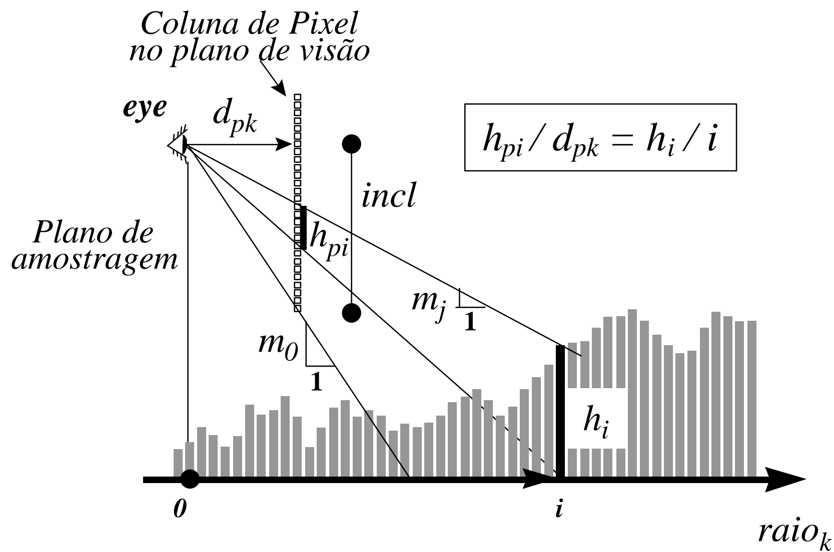


Figura 4.2 - Projeção cônica simples.

Na literatura de programação de jogos ([LaMothe95] e [Freese+95]), podem ser encontrados algoritmos que exploram este modelo particular do terreno. Estes tratam cada coluna da tela separadamente e pintam, para cada coluna, os *pixels* da tela de baixo para cima, seguindo a idéia de um horizonte flutuante. O algoritmo de Horizonte Flutuante inicia lançando o primeiro horizonte, mostrado como m_0 na Figura 4.2. Dado um horizonte, somente fração do terreno que está acima deste pode ser visualizado. Cada horizonte pode ser caracterizado pela sua inclinação (m_j). Para determinar a cor do *pixel* da tela, o algoritmo testa a altura de cada coluna do terreno que começa ao pé do observador, marcada como *O* na

Figura 4.2, e avançando na direção de $raio_k$. Para a primeira coluna que está sobre o horizonte, o *pixel* é pintado e o horizonte é movido para cima. O algoritmo usa o fato de que os *voxels* de terreno que estão fora da direção de $raio_k$ não podem obscurecer o *pixel* pintado.

Uma implementação do algoritmo de Horizonte Flutuante para uma coluna de *pixels* é ilustrada no Algoritmo 1, mostrada a seguir.

Note que, quando nos movemos de uma posição *i* (fatia *i*) para a próxima posição no terreno, a altura do horizonte diminui pelo valor do coeficiente angular atual, m_j, conforme mostrado na linha 6 do Algoritmo 1.

A Figura 4.2 também mostra que o coeficiente angular do primeiro horizonte é determinado por:

$$m_0 = incl / d_{pk} \tag{4.1}$$

e, com relação à mudança no coeficiente angular no *voxel i*, quando nos movemos para o *pixel* de cima na tela, isto é, de *j* para *j+1*, a nova inclinação do raio é dada por:

$$m_{j+1} = (incl-(j+1)) / d_{pk} = m_j - 1 / d_{pk} \tag{4.2}$$

A constante *incl* indica o deslocamento vertical do plano de projeção: quanto maior seu valor, mais o plano de projeção está abaixado; quanto menor seu valor, mais o plano de projeção está levantado. Este deslocamento fornece a impressão de liberdade do observador quanto a sua inclinação vertical. Esta técnica de deslocamento do plano não produz uma mesma visualização se ao invés disto usássemos um plano de projeção inclinado, sempre perpendicular à direção de visão. Isto levaria então inclinar a direção de visão. Com esta técnica de deslocamento do plano de projeção na vertical não é possível visualizar o terreno dando a impressão de que o observador está olhando na vertical.

As linhas 3 e 13 do Algoritmo 1 mostram, respectivamente, a iniciação e a atualização do coeficiente angular do horizonte.

A mudança na altura de horizonte, *z*, no *voxel* de posição *i*, pode ser calculada fixando-se h_{pi} igual a 1 na equação mostrada na Figura 4.2, resultando a equação mostrada na linha 14

do Algoritmo 1. As divisões surgidas neste algoritmo podem ser evitadas facilmente sendo calculadas de forma incremental. Por motivo de clareza, apresentamos o algoritmo sem otimizações de implementação. As constantes *dx* e *dy* passadas como parâmetros para o algoritmo são referentes a direção de visão, isto é, $raio_k = (dx, dy)$. A constante f da linha 5 define um limite de visão no terreno, indicando o número máximo de passos no terreno. O valor de f geralmente é 256.

Para aumentar a velocidade do algoritmo de Horizonte Flutuante aplicado a mapas de terreno, [Freese+95] e [LaMothe95] sugerem duas aproximações: [a] todas as colunas de *pixels* estão à mesma distância do observador, isto é, $d_{pk}=d_p$; e [b] o ângulo entre dois planos de amostragem sucessivos é constante.

```
   HorizFlut (col, incl, dx, dy){
1        x = eye_x; y = eye_y; z = eye_z;
2        j = 0; pixel de fundo na coluna
3        m = incl/dpk; primeiro horizonte
4        i = 0; voxel da posição do pé do observador
5        enquanto (i < f) { f é far e define um limite de visão no terreno
6              y += dy; x += dx; z -= m; Passo em direção de raio_k
7              h = MapaAlt(x,y); Obtém h do mapa de alturas
8              se (h > z) { altura do horizonte
9                    c = MapaCor(x,y);
10                   faça {
11                         BufferCor[col,j] = c; pinta o pixel
12                         j += 1; move um pixel para cima
13                         m -= 1/dpk; atualiza coeficiente do horizonte
14                         z += i/dpk; corrige a altura do horizonte
15                   } enquanto (h > z);
16             } fim do se
17             i += 1; próximo voxel
18       } fim do enquanto
19 } fim de HorizFlut
```

Algoritmo 1 - Horizonte Flutuante para visualizar um terreno.

[Frederick+96] mostrou que estas aproximações distorcem as imagens resultantes. Para combinar dois algoritmos diferentes, um para terreno e outro para objetos, não se deve aceitar nenhuma distorção em um deles que não esteja presente no outro. Se não for assim,

um edifício localizado no terreno, por exemplo, seria movido. Por isto, nenhuma aproximação na projeção cônica é permitida para a finalidade deste trabalho.

4.1. Análise da complexidade

Um algoritmo genérico do tipo lançamento de raios para a visualização de um volume baseado em uma grade composta de n por n elementos produzindo uma imagem de m por m *pixels* tem complexidade $O(m^2n^2)$. Estes algoritmos genéricos tem esta complexidade alta pelo motivo da necessidade de calcular todas as interseções entre cada raio lançado com o terreno.

Para o algoritmo proposto, não é necessário este calculo de interseção. Para cada coluna de *pixels* da tela, um raio é lançado e percorre o terreno até seus limites ou então quando o raio chegou no topo da coluna da tela. Com isto, a complexidade do algoritmo para cada coluna da tela é dada por $O(m+n)$. Portanto a complexidade do algoritmo é $O((m+n)m)$.

Capítulo 5

5. Algoritmo Composto

A estratégia proposta neste trabalho para combinar o algoritmo de Z-Buffer com o algoritmo de Horizonte Flutuante. A proposta é gerar com o algoritmo de Horizonte Flutuante a partir do terreno, descrito pelos mapas de alturas e cores, dois *buffers*: um *buffer* de cor (que é a imagem propriamente dita) e um *buffer* de profundidade. Estes dois *buffers* junto com os objetos são então passados para a biblioteca OpenGL, que utilizando o algoritmo de Z-Buffer produz o resultado desejado: terreno com objetos. Para uma animação basta repetir esta estratégia. O esquema desta proposta é ilustrada na Figura 5.1.

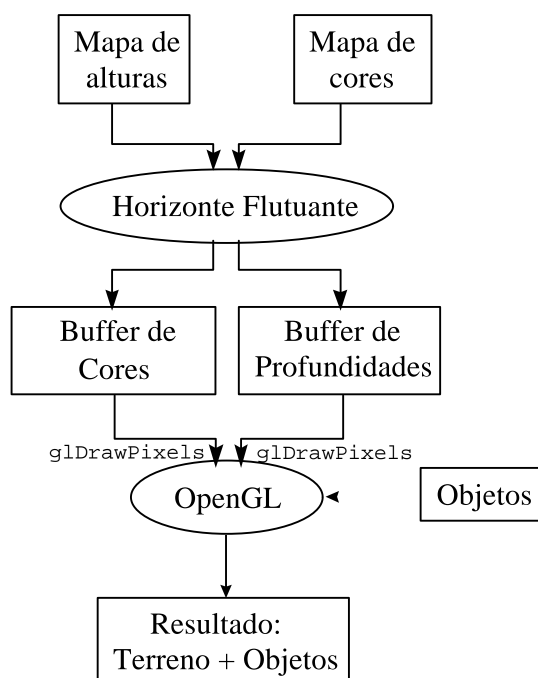


Figura 5.1 - Estratégia para combinar ambos os algoritmos.

5.1. Modelo de Câmara

A projeção no OpenGL é definida em um modelo de câmara pelos seguintes parâmetros: ***eye***, ***ref***, ***up***, *left*, *right*, *bottom*, *top*, *near* e *far*. Serão usadas as notações descritas na página VIII para simplificar as formulações das equações. Os três primeiros parâmetros definem a posição (*viewpoint*), inclinação e direção de visualização do observador, conforme ilustrado na Figura 5.2. Os demais definem o sólido de visualização (em sistema de coordenadas do observador), chamado também de frustum de visão, como é visto na Figura 5.3.

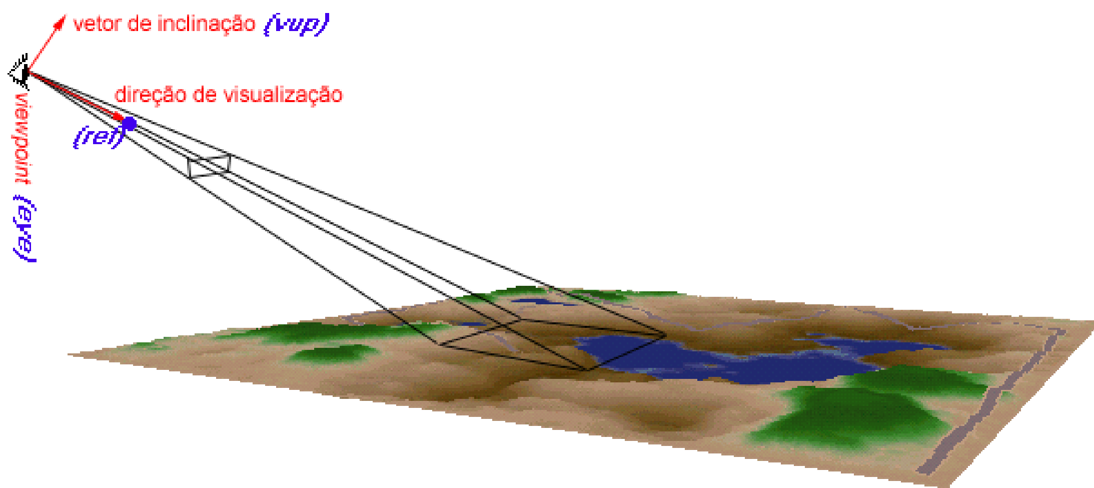


Figura 5.2 - Parâmetros da câmara.

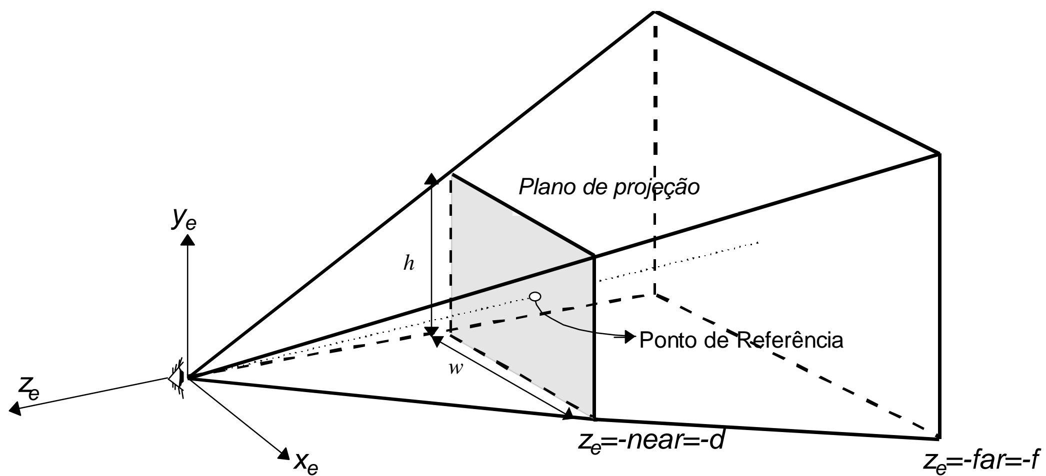


Figura 5.3 - Frustum de visão.

A projeção cônica usada no algoritmo de Horizonte Flutuante é definida por: [a] a posição do observador, equivalente ao ***eye***; [b] uma direção de visão, que adicionando vetorialmente por ***eye***, fica definido facilmente o ponto de referência ***ref***; [c] a primeira fatia de *voxels* que pode ser visível (i = 1), definindo o plano *near*; [d] o número de fatias do terreno na direção de $raio_k$, que pelo fato de estarem a uma distância de uma unidade entre si define a distância do plano de cerceamento *far* ; e [e] um ângulo de câmara e um *incl* que podem ser usados para calcular a janela do OpenGL, conforme mostrado na Figura 5.4.

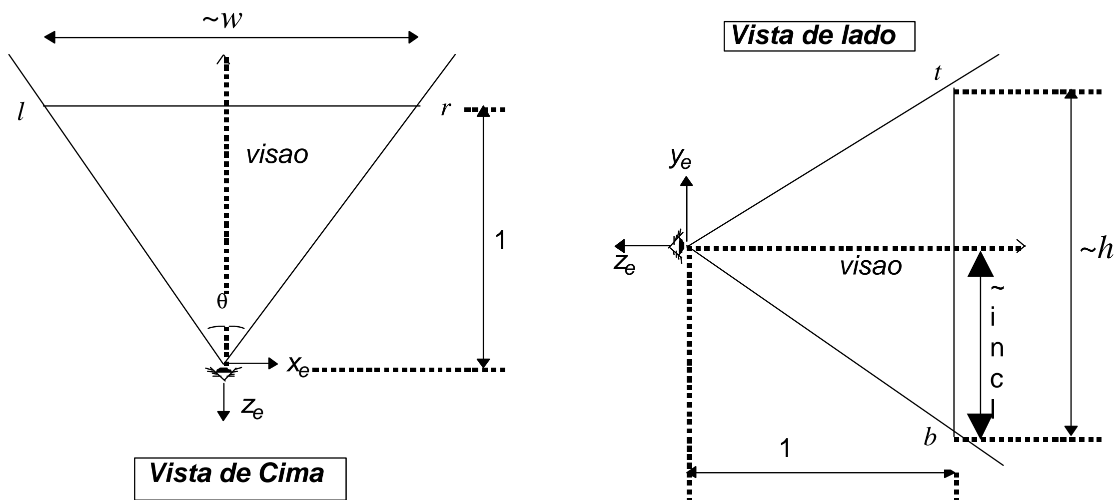


Figura 5.4 - Definição do plano de cerceamento *near*.

Altura (h) e largura (w) são dimensões reais da janela e *incl* é um valor relativo à altura em sistemas de coordenadas da tela, enquanto que outras dimensões citadas na Figura 5.4 estão em um sistema de coordenadas do observador. O sistema de coordenadas do observador (Figura 5.2) é gerado pela base ortogonal formado pelo vetores que definem a inclinação do observador *vup*, uma direção de visualização *dir* = (*ref* - *eye*) e um vetor perpendicular a estes de calculado pelo produto vetorial de *dir* por *vup*. Sua origem localiza-se na posição do observador. Isto quer dizer que b não necessariamente vale -*incl* nem que a distância de t a b seja igual a h. E nem que a distância de r a l seja igual à w. Mas a razão entre b e -*incl* deve ser a mesma que a razão entre a distância entre t a b e h, e a mesma que a razão entre a distância de r a l e w. Com isto temos que os planos de visualização utilizados nos algoritmos são proporcionais.

O valor de $\boldsymbol{q}$ é igual ao valor do ângulo do cone de visão (frustum) na horizontal em relação ao observador. Portanto

$$l = -\tan\left(\frac{\boldsymbol{q}}{2}\right) \tag{5.1}$$

e

$$r = \tan\left(\frac{\boldsymbol{q}}{2}\right) \tag{5.2}$$

Os valores de t e b devem ser tais que a razão de aspecto do plano *near*, $(t - b)/(r - l)$, seja igual à da janela, h/w. Com isto temos que

$$\frac{t-b}{r-l} = \frac{h}{w} \tag{5.3}$$

Devemos ter também

$$\frac{incl}{-b} = \frac{h}{t-b},$$

ou seja,

$$t = b \times \left(1 - \frac{h}{incl}\right) \tag{5.4}$$

Substituindo a equação (5.4) em (5.3) e usando (5.1) e (5.2)

$$b = -\frac{2 \times \tan\left(\frac{\boldsymbol{q}}{2}\right) \times incl}{w} \tag{5.5}$$

Substituindo a equação (5.5) em (5.4)

$$t = \frac{2 \times \tan\left(\frac{\mathbf{q}}{2}\right) \times (-\,incl + h)}{w} \tag{5.6}$$

Com isto definimos o plano de cerceamento *near* (definido por *l*, *r*, *t*, *b* e *n*).

5.2. Cálculo de Profundidade

Para combinar imagens geradas pelo Algoritmo de Horizonte Flutuante com as primitivas do Z-Buffer, é necessário o valor de profundidade de cada *pixel* na imagem de terreno. Como mostrado abaixo, esta profundidade é uma função da distância do ponto correspondente no terreno ao plano de projeção. Isto sugere uma breve modificação em Algoritmo 1. Na versão descrita na seção 4, um passo ao longo do terreno é dado em passos unitários; assim, os pontos onde o terreno é amostrado pertencem aos círculos pontilhados mostrados em Figura 5.5. Com a finalidade do cálculo de profundidade, é mais conveniente que amostrarmos o terreno em fatias que são paralelas à tela, representadas pelas linhas tracejadas em Figura 5.5.

Isto também torna o processo de cada coluna de pixel mais eficiente. Na versão original, cada coluna *k* da tela tem seu próprio conjunto inclinações e variações de alturas que são inversamente proporcional à distância d_{pk} entre o observador e a coluna *k*. Portanto, estes coeficientes diferem dos correspondes coluna central pelo fator $cos(\mathbf{q}_k)$. Porém, se o terreno é amostrado de acordo com as linhas tracejadas, os passos horizontais são corrigidos pelo fator $1/cos(\mathbf{q}_k)$, causando as mesmas variações de altura em relação aos passos relativos à coluna central. Então, inclinação e a atualização da altura se torna o mesmo para todas as colunas, contanto que o vetor unitário $raio_k = (dx, dy)$ seja multiplicado pelo fator $1/cos(\mathbf{q}_k)$ para cada coluna *k*. Nota que isto produz a mesma melhoria em eficiência quanto as simplificações propostas por [LaMothe95] e [Freese+95] sem as respectivas distorção.

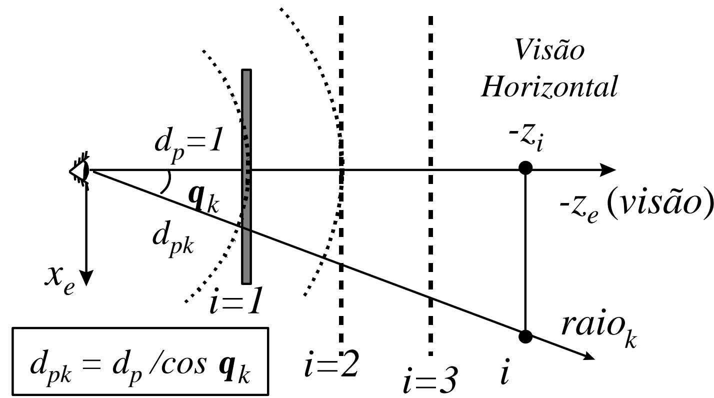


Figura 5.5 - Distâncias na direção de $raio_k$.

Até mesmo com a simplificação acima, o cálculo de profundidade para todas as fatias *i* ainda é caro, devido à natureza não linear da projeção cônica. Esta projeção pode ser obtida pela matriz homogênea, *P*, dada pela equação (5.7). Essa matriz é utilizada pelo OpenGL ([Neider+93] e [Martha+94]) para transformar o sistema de coordenadas do observador para o sistema de coordenadas normalizadas da tela.

$$[P] = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{5.7}$$

Assim, a coordenada de tela, z_s, pode ser calculada a partir da coordenada do observador, z_e, através da expressão:

$$z_s = \frac{(f+n) + \frac{2fn}{z_e}}{f-n} \tag{5.8}$$

O OpenGL também fornece uma função chamada `glDepthRange`, que especifica um mapeamento linear entre o domínio de profundidade [-1,1] e um domínio de profundidade escolhido. Os valores *default* para este novo domínio são 0.0 para *near* e 1.0 para a distância *far*. Com estes valores, a Equação (5.8) deve ser modificada por:

$$z_s' = \frac{z_s + 1}{2} \tag{5.9}$$

Com as modificações acima, o Algoritmo 1 percorre o terreno modelado em fatias paralelas ao plano de projeção, como mostrado na Figura 5.5. A coordenada z_e, como uma função da fatia i, é então determinada por:

$$z_e = -i \tag{5.10}$$

Substituindo as equações (5.10) em (5.8) e (5.8) em (5.9) podemos ter o valor de profundidade de cada fatia i, dado por:

$$prof_i = \frac{f - \frac{f \times n}{i}}{f - n} \tag{5.11}$$

É importante notar que esta relação é invariante para a coluna de *pixels*, a posição do observador e a direção de visão. Ou seja, podemos pré-calcular todas as profundidades e armazená-las em um vetor. É importante notar, porém, que todas essas simplificações só são válidas no caso onde a direção de visão é horizontal, como mostrado na Figura 4.1.

O Algoritmo 2, mostrado abaixo, implementa o cálculo da equação (5.11).

```
  VetorProf(n,f) {
1     para (i=0; i <= (f-n); i++)
2       Prof[i]=(f-(f*n)/(i+1))/(f-n);
3 } fim de VetorProf
```

Algoritmo 2 - Cálculo do Vetor de Profundidade.

Capítulo 6

6. Aperfeiçoamento da Imagem do Algoritmo de Terreno

Podem surgir alguns problemas na visualização de terrenos, especialmente quando adicionamos objetos. Eles podem estar relacionados com a baixa resolução dos dados de entrada (mapas de cores e alturas), causada pela baixa taxa de amostragem na geração desses dados, ou então relacionados com a taxa de amostragem feita pelo algoritmo de Horizonte Flutuante (se o espaço entre as fatias de terreno é grande demais, algoritmo pode descartar dados relevantes do terreno). Um problema grave em relação a esse espaço entre fatias seria a visualização de um objeto que se encontra nesse espaço e que estaria por debaixo da superfície do terreno, não podendo então ser visto.

Podemos ter ainda outro problema relacionado ao tamanho de cada *voxel* do terreno e o tamanho do *pixel* da tela. Um mesmo *voxel* seria desenhado em mais de um *pixel* quando, por exemplo, visualizarmos um terreno muito de perto. Isto resulta em "retângulos grandes" de mesma cor na tela e em uma visão muito descontínua das alturas dos *voxels* na tela.

Veremos neste capítulo algumas soluções para esses problemas. A primeira solução é o aumento do número de fatias e a segunda solução é uma subdivisão dos *voxels* onde as alturas e cores de cada um desses *sub-voxels* são dadas por interpolações bilineares a partir de *voxels* vizinhos. A primeira solução está diretamente relacionada com a amostragem feita no terreno enquanto que a segunda solução está relacionada com a reconstrução do terreno. Na seção 6.3 é feita uma combinação destas duas soluções, isto é, tenta-se melhorar a visualização aumentando a taxa de amostragem e melhorando a técnica de reconstrução.

6.1. Aumento do Número de Fatias

Um dos problemas relatados acima diz respeito à desconsideração de alguns *voxels* ao percorrer um raio, pelo fato do passo dado no terreno, gerando as fatias, ser grande. Este problema é melhor observado na Figura 6.1. Nela podemos notar que existe uma região entre dois raios que terá uma só cor (verde). Os *voxels* de cor vermelha não serão interceptados ao longo desses raios, devido ao tamanho do passo dado. Vemos também as regiões vazias entre as fatias. As linhas coloridas representam as fatias de terrenos geradas.

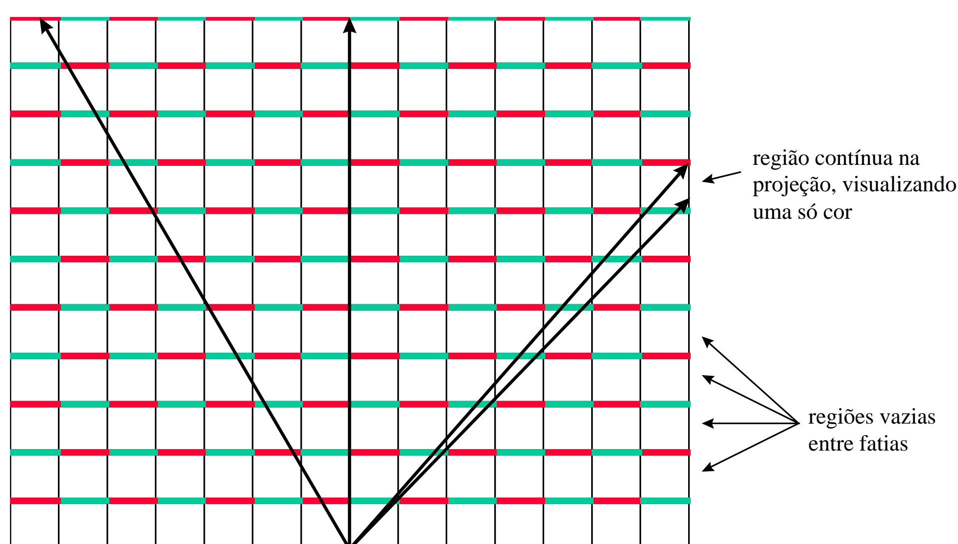


Figura 6.1 - Problemas com as regiões entre fatias.

Essas regiões também fazem gerar o erro de nelas poder visualizar um objeto que estaria por debaixo da superfície do terreno.

Cobrir essas regiões estaria totalmente fora da idéia do algoritmo proposto, que, como já foi dito antes, representa o terreno como fatias perpendiculares à grade *xy* do terreno e também perpendiculares à direção de visão.

Uma solução imediata, então, é reduzir os passos dados no terreno. Com isto teremos uma maior amostragem do terreno ao longo dos horizontes. Alterando o tamanho do passo, deve-se apenas modificar os cálculos de incremento do coeficiente angular do horizonte e atualização das alturas dos *voxels* no Algoritmo 1. Para isto deve-se alterar as linhas 3, 6, 13 e 14 para:

```
3*         m = FATOR*incl/dpk;  primeiro horizonte
6*              y += dy; x += dx; z -= FATOR*m;  Passo em dir. de raio_k
13*                       m -= FATOR/dpk;  atualiza coeficiente do horizonte
14*                       z += FATOR/dpk;  corrige a altura do horizonte
```

onde FATOR é inversamente proporcional ao número de intervalos novos. Isto é, se incluirmos mais uma fatia em cada região, este fator será igual a 0,5.

O cálculo da profundidade do Algoritmo 2 também é alterado para

```
  VetorProf(n,f) {
1     for (i=0; i <= (f-n)*FATOR; i++)
2        Prof[i]=(f-(f*n)/(i*FATOR + 1)/(f-n);
3  }  fim de VetorProf
```

Algoritmo 2b - Cálculo do Vetor de Profundidade com intervalo menor entre fatias.

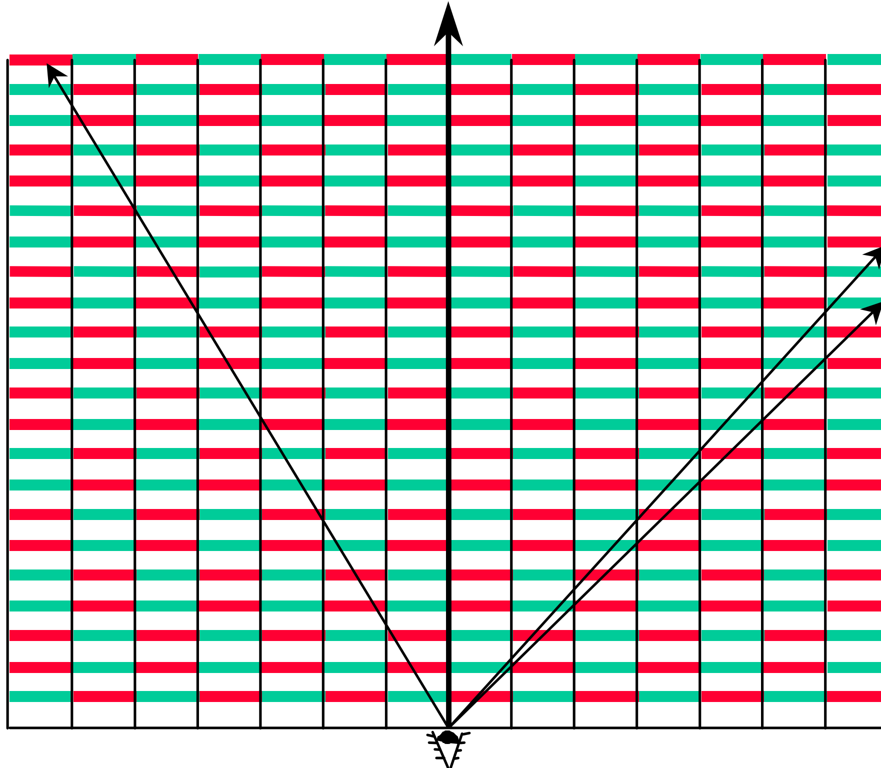
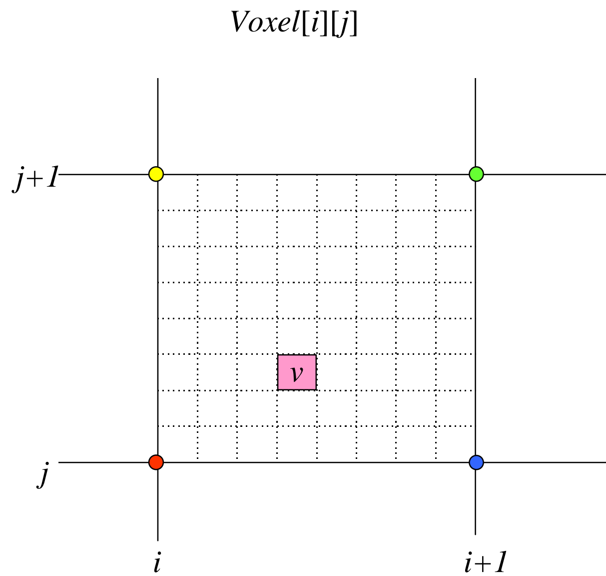


Figura 6.2 - Inclusão de mais fatias.

A Figura 6.2 ilustra o aumento do número de fatias em relação à Figura 6.1. Note que agora não há mais uma região contínua entre dois raios com a mesma cor, o que aconteceu na Figura 6.1.

6.2. Divisão dos Voxels

Os retângulos grandes que aparecem na visualização devem-se ao fato de mais de um horizonte interceptar o mesmo *voxel* do terreno quando este está muito próximo do observador. Uma solução para isto seria reamostrar o terreno, refinando mais a grade. Isto nem sempre é viável devido à resolução do equipamento de sensoriamento do terreno. Outra solução é usar uma interpolação no terreno. Isto é, imaginar o terreno como sendo uma superfície contínua e calcular a altura e cor do ponto do terreno interceptado pelo horizonte com a contribuição dos pontos vizinhos do terreno. Com isto, cada *voxel* terá alturas variáveis, calculadas a partir de uma interpolação bilinear. Como uma interpolação em cada uma das interseções seria muito cara, pode-se pensar no *voxel* como um inteiro dividido em partes, isto é, um *voxel* dividido, por exemplo, em 64 *subvoxels* (uma divisão em 8 partes iguais em cada direção x e y). Cada um desses *subvoxels* teria sua cor e altura calculadas com a contribuição dos *voxels* vizinhos segundo uma matriz de contribuições. Este esquema de divisão de um *voxel* é mostrado na Figura 6.3.



$inf_v = p_1 * inf[i][j] + p_2 * inf[i+1][j] + p_3 * inf[i+1][j+1] + p_4 * inf[i][j+1]$

Figura 6.3 - Subdivisão de um *voxel*, com contribuição dos *voxels* vizinhos.

A matriz *inf* na Figura 6.3 é a informação de cor e altura no *voxel ij*. As constantes p_1, p_2, p_3 e p_4 são os pesos ou contribuições das informações dos *voxels*. Pelo fato desses pesos serem invariantes em relação ao *voxel*, pode ser pré-calculada e armazenada uma matriz chamada matriz de pesos ou matriz de contribuição de informações. Os valores contidos nesta matriz são os valores necessários para a interpolação, porém na forma discretizada. O nível da discretização deve depender do grau de resolução da grade do terreno.

A matriz de contribuição é calculada pelo algoritmo a seguir:

```
   monta_matriz_amostragem {
1      para (i=0; i<num_divisoes; i++) {
2          contrib_ij[0][i] = 1.0 - i/(num_divisoes -1);
3          contrib_ij[num_divisoes-1][i] = 0.0;
4      }  fim do para
5      para (j=0; j<num_divisoes; j++) {
6          para (i=1; i<(num_divisoes -1); i++) {
7              contrib_ij[i][j] = ((num_divisoes -1) - i)*
8                                  contrib_ij[0][j]/
9                                  (num_divisoes -1);
10         }  fim do para
11     }  fim do para
12 }  fim de monta_matriz_amostragem
```

Algoritmo 3 - Cálculo da matriz de amostragem.

O peso p_1 é dado diretamente por esta matriz, calculando-se apenas em qual subdivisão a interseção do horizonte com o *voxel* ocorreu e tomando-se o valor na posição equivalente à matriz. Para os outros pesos, podemos aproveitar a mesma matriz, mas o acesso à matriz difere, como mostram as equações a seguir, supondo-se que a interseção se deu na subdivisão *p* na direção *x* e *q* na direção *y*.

p_1 = contrib_ij[p][q];
p_2 = contrib_ij[p][num_divisoes - q];
p_3 = contrib_ij[num_divisoes - p][num_divisoes - q];
p_4 = contrib_ij[num_divisoes - p][q];

6.3. Combinação das Soluções

Cada uma das soluções apresentadas são independentes uma da outra. Isto é, podemos usar as duas soluções juntas ou então somente uma delas. Isto irá depender da necessidade em relação aos dados de entrada.

Como cada uma dessas soluções acarreta alguma perda de eficiência em animação, podemos dividir os terrenos em regiões e aplicar estas soluções apenas nestas regiões. Por exemplo, podemos dividir o terreno em 3 partes em relação à distância do ponto do terreno ao observador. Com isto podemos aplicar a solução da divisão dos *voxels* nas duas regiões que estejam mais próximas ao observador: na região mais próxima dividimos mais os *voxels* do que na segunda. Na terceira região, que está mais distante do observador, não há muita necessidade de usarmos esta solução, pois, quanto mais distante, a projeção de um *voxel* diminui de tamanho e assim os retângulos se reduzem, podendo chegar a um ponto. Note que a eficiência não depende do número de subdivisões de um *voxel*. O motivo de termos duas regiões consecutivas com números diferentes de divisões de *voxels* é apenas evitar uma grande diferença na visualização, caso as duas primeiras regiões forem pouco espessas e, assim, a terceira ficar perto do observador.

Do mesmo modo feito para a solução da divisão de *voxels*, podemos dividir o terreno em regiões e fazer com que cada uma dessas regiões tenha um tamanho de passo diferente,

sendo que a região mais distante pode ter um passo maior que uma região mais próxima. O conjunto de todas estas soluções está relacionado com a idéia de multiresolução.

Capítulo 7

7. Implementação

A implementação de renderização de terreno da estratégia proposta na Figura 5.1 segue o Algoritmo 1, com as seguintes mudanças: [a] é dada uma escala na cena e multiplicada pela inversa da constante d_{pk}, substituindo-se uma divisão por multiplicação; [b] a norma de passo (*dx*, *dy*) não é *1*, mas $1/cos(\theta_k)$; e [c] toda vez que um *voxel* é pintado, sua profundidade é armazenada no *buffer* de profundidade. Isto é feito incluindo a seguinte linha de código logo após a linha 11 no Algoritmo 1:

```
11A BufferProf[col,j] = Prof[i];
```

Estamos assumindo aqui que o vetor de profundidade já foi pré-calculado no início do programa pelo procedimento mostrado no Algoritmo 2.

É definida uma escala na vertical a partir dos dados do terreno. Isto é, é definida uma escala da forma s_h/s_w, onde uma unidade de altura de *voxel* equivale a s_h metros e uma unidade de largura de *voxel* equivale a s_w metros. Esta escala é dada diretamente para o algoritmo de Z-Buffer e é chamada de *EscalaZB*.

Para o algoritmo de Horizonte Flutuante, esta escala é dada junto a d_{pk}, eliminando as divisões do Algoritmo 1. Esta escala é denominada de *EscalaHF* e é calculada por:

$$EscalaHF = \frac{1}{EscalaZB \times d_{pk}}.$$

A constante d_{pk}, definida como distância do observador ao plano de visão, é dada por

$$d_{pk} = \frac{w}{2 \times \tan\left(\frac{q}{2}\right)}.$$

No algoritmo, é lançado um raio para a base de cada coluna da tela, e em seguida este raio percorre o terreno segundo uma direção.

No trecho de código a seguir é mostrado o uso do algoritmo de Horizonte Flutuante com cálculo de profundidades, implementado pela função `HorizFlutcomProf` (veja Figura 7.1).

```
incr_x = (long)( SIN(visao)*kFatorIncr);
incr_y = (long)(-COS(visao)*kFatorIncr);
dx = COS(visao + kAnguloVisao) * InvCOSAngVis;
dy = SIN(visao + kAnguloVisao) * InvCOSAngVis;
for (col=0; col < larg; col++) {
    HorizFlutcomProf(col, incl, dx, dy);
    dx += incr_x; /* novas direcoes para visao - raios no terreno*/
    dy += incr_y;
} /* fim do for */
```

Ao contrário do que é usualmente feito em jogos, onde os raios lançados são igualmente espaçados em relação aos ângulos, gerando uma imagem distorcida, temos que lançar raios igualmente espaçados em relação à interseção com o plano de visão. A variável *visao* é o valor do ângulo que define a direção de visão; com isto, o observador está direcionando a visão segundo o vetor (*cos*(*visao*), *sin*(*visao*)). A constante *kAnguloVisao* é igual à metade do valor do ângulo do cone de visão (metade deste ângulo para esquerda e metade para direita). O vetor (*incr_x*, *incr_y*) é perpendicular ao vetor (*dx*, *dy*), pois ele está no plano da tela e a tela é perpendicular a (*dx*, *dy*).

A constante *kFatorIncr* é calculada por

$$kFatorIncr = \frac{2 \times \tan\left(\frac{q}{2}\right)}{w}.$$

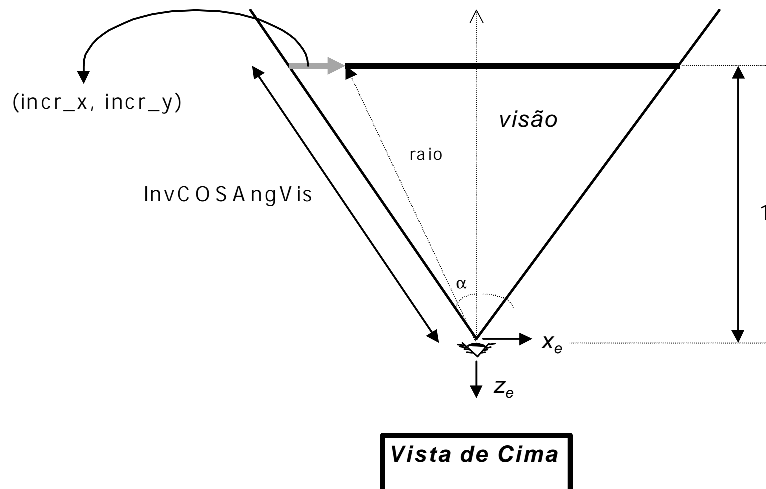


Figura 7.1 - Definições dos raios.

A constante *InvCOSAngVis* é a inversa do coseno de α (metade do ângulo de visão - θ), para evitar a distorção na visualização, conforme descrito anteriormente.

Com isso, lançamos raios, por incremento, chamando a função do algoritmo de Horizonte Flutuante:

```
void HorizFlutcomProf(int col, int Incl, int dx, int dy)
{
  int           x, y, z, dz, h, ph, pos;
  unsigned char c, c_red, c_green, c_blue;
  float         p, *Prof;

  x = pos_x; // posicao do observador no terreno.
  y = pos_y;
  z = pos_z;
  dz = Incl*EscalaHF; // inclinacao inicial do horizonte.
  ph = 0; // incremento inicial da inclinacao do horizonte
  Prof = profundidade;  // profundidade da primeira fatia do terreno no Z-Buffer
  MC = &MatrizCor[3*col]; // matriz de cores da tela
  MP = &MatrizProf[col]; // matriz de profundidade para o Z-Buffer
  while (ph < ProfMax) {
    x += dx; // caminha para o voxel da proxima fatia do terreno.
    y += dy;
    z -= dz; // atualiza a altura do horizonte na proxima fatia.
    ph += EscalaHF; // atualiza o incremento da altura do horizonte.
    pos = _calc_pos(x, y); // posicao do voxel (x, y) no mapa dado por um vetor.
```

```
      h = MapaAlt[pos]; // altura do voxel.
      if (h > z) { // o voxel e mais alto que o horizonte.
        c = MapaCor[pos]; // cor do voxel.
        c_red   = red[c];
        c_green = green[c];
        c_blue  = blue[c];
        p = *Prof; // profundidade do voxel no Z-Buffer.
        do {
          dz -= EscalaHF; // atualiza a inclinacao do horizonte.
          *MC = c_red;   MC++; // pinta o voxel na tela.
          *MC = c_green; MC++;
          *MC = c_blue;  MC += passo_MC;
          *MP = p; MP += largura;
          if (MC > MC_max) return;  // o raio ultrapassou o limite sup. da tela.
          z += ph; // atualiza a altura do horizonte flutuante.
        } // fim do do
        while (h > z); // enquanto o voxel eh mais alto que o horizonte.
      } // fim do if
      Prof++; // profundiodade da proxima fatia no Z-Buffer.
    } // fim do while
}
```

Na função acima, as variáveis *MC* e *MP* são ponteiros para as matrizes de cor (*MatrizCor*) e profundidade (*MatrizProf*), respectivamente e apontam para a posição relativa a um *pixel*. A constante *passo_MC* é igual à 3**largura*-2, onde *largura* é igual à largura da janela de visão. A variável *Prof* aponta para a posição do vetor de profundidades para a fatia do terreno corrente. O vetor MapaCor contém os índices para os vetores *red*, *green* e *blue*, que determinam as cores do terreno.

O plano *far* está localizado na fatia de ordem 256, isto é, cada raio vai percorrer 256 fatias e *ProfMax* = 256**EscalaHF*.

Com a imagem gerada e os valores de profundidades calculados e armazenados nas matrizes *MatrizProf* e *MatrizCor*, pode-se de passá-los para o OpenGL. O trecho de código responsável por isto é:

```
/* impede alteracoes no buffer de cores */
  glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
```

```
/* escreve a matriz de profundidade */
  glRasterPos2d (0.0, 0.0);
  glDrawPixels (larg, alt, GL_DEPTH_COMPONENT, GL_FLOAT,
                MatrizProf);

/* habilita alterações no buffer de cores */
  glPixelTransferi (GL_MAP_COLOR, GL_TRUE);
  glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_FALSE);
/* nao realiza teste de profundidade */
  glDisable(GL_DEPTH_TEST);
/* escreve a matriz de cores (imagem) */
  glRasterPos2d (0.0, 0.0);
  glDrawPixels (larg, alt, GL_COLOR_INDEX,
                GL_UNSIGNED_BYTE, MatrizCor);
```

Os três primeiros comandos são responsáveis por passar para o OpenGL a matriz de profundidade, impedindo alterações no *buffer* de cores. Os comandos restantes passam para o OpenGL a matriz de cores. No caso de visualização somente do terreno, os três primeiros comandos não se aplicam.

Em seguida, desenham-se os objetos usando o OpenGL, utilizando o algoritmo de Z-Buffer implementado. São definidos o frustum (através da função `glFrustum`) e a câmara (através da função `gluLookAt`) como vistos nos capítulos anteriores.

```
/* definicao da projecao perspectiva */
  glMatrixMode (GL_PROJECTION);
  glLoadIdentity ();
/* definicao do frustum de visualizacao */
  glFrustum (-tan30 /* esq */, tan30 /* dir */,
             -2*tan30*(incl)/w /* base */,
                 2*tan30*(h-incl)/w /* topo */,
             1.0 /* near */, 256.0 /* far */);

/* definicao da camera (observador) */
  glMatrixMode (GL_MODELVIEW);
  glLoadIdentity ();

/* posicao da camara */
  eye_x = (GLdouble)(xOrg >> 16);
  eye_y = (GLdouble)(yOrg >> 16);
  eye_z = (GLdouble)(gAltitude*EscalaZB);
```

```
/* ponto de referencia */
  ref_x = (GLdouble)(eye_x + COS(visao));
  ref_y = (GLdouble)(eye_y + SIN(visao));
  ref_z = (GLdouble)(eye_z);
/* inclinacao da camara */
  vup_x = 0.0;
  vup_y = 0.0;
  vup_z = 1.0;
  gluLookAt (eye_x, eye_y, eye_z,
             ref_x, ref_y, ref_z,
             vup_x, vup_y, vup_z);
```

Uma observação importante é o uso de representação de ponto fixo ao invés de ponto flutuante. Operação feitas com números representados por pontos fixos são bem mais rápidas. Foram implementadas duas versões, uma com ponto flutuante e outra com ponto fixo, respeitando a seguinte norma:

- As variáveis *x* e *y*, relacionadas à localização no mapa do terreno, utilizam os 9 *bits* mais à esquerda para a parte inteira e os 23 *bits* à direita para representar a parte fracionária. Isto limita o mapa do terreno em um intervalo entre 0 e 512.

- As demais variáveis utilizam os 16 *bits* mais à esquerda para a parte inteira e os 16 *bits* à direita para representar a parte fracionária. Para torná-las compatíveis com as variáveis do terreno, faz-se um deslocamento de 7 *bits* para a esquerda.

As diferenças de tempos com estas duas notações podem ser notadas na Tabela 8.1 e na Tabela 8.2.

Para implementar o aperfeiçoamento nas imagens do terreno usando *subvoxels*, foram necessárias mais três funções, _calc_viz, _calc_alt e _calc_cor, para determinar os *voxels* vizinhos ao *subvoxel* resultante da interseção do horizonte com o terreno, a altura e a cor do *subvoxel*, respectivamente, como ilustrado na Figura 6.3

```
void _calc_viz (float x, float y)
{
  int resto_x, resto_y;

 /* indice do subvoxel  - o voxel eh subdividido em:
    64 (contrib_tam = 8) ou
    16 (contrib_tam = 4)
 quadrados iguais. */
```

```
  resto_x = (int)((x  - (int)x)*contrib_tam);
  resto_y = (int)((y  - (int)y)*contrib_tam);
  if (resto_x < 0) resto_x += contrib_tam;
  if (resto_y < 0) resto_y += contrib_tam;

 /* busca na matriz de contribuicao os pesos de cada vertice na interpolacao. */
  pesop1 = contrib[resto_x][resto_y];
  pesop2 = contrib[(contrib_tam - 1)-resto_x][resto_y];
  pesop3 = contrib[(contrib_tam - 1)-resto_x][(contrib_tam -
1)-resto_y];
  pesop4 = contrib[resto_x][(contrib_tam - 1)-resto_y];

 /* calcula as posicoes dos 4 vertices que definem o voxel (x, y) no mapa. */
  pos1 = _calc_pos(x, y);
  pos2 = _calc_pos(x+1, y);
  pos3 = _calc_pos(x+1, y+1);
  pos4 = _calc_pos(x, y+1);
}


void _calc_cor(void)
{
  unsigned char c1, c2, c3, c4;

  c1 = MapaCor[pos1];
  c2 = MapaCor[pos2];
  c3 = MapaCor[pos3];
  c4 = MapaCor[pos4];

 /* cor do subvoxel calculada por uma interpolacao bilinear - media geometrica. */
  c_red   = pesop1*red[c1] + pesop2*red[c2] +
            pesop3*red[c3] + pesop4*red[c4];
  c_green = pesop1*green[c1] + pesop2*green[c2] +
            pesop3*green[c3] + pesop4*green[c4];
  c_blue  = pesop1*blue[c1] + pesop2*blue[c2] +
            pesop3*blue[c3] + pesop4*blue[c4]);
}


void _calc_alt(void)
{
 /* altura do subvoxel calculada por uma interpolacao bilinear - media geometrica. */
  h = pesop1*MapaAlt[pos1] + pesop2*MapaAlt[pos2] +
      pesop3*MapaAlt[pos3] + pesop4*MapaAlt[pos4];
}
```

Assim, com estas funções, a função do algoritmo do Horizonte Flutuante com os aperfeiçoamento nas imagens é dada a seguir:

```
void HorizFlutcomProfInterp(int col, int Incl, int dx, int dy)
{
  int          x, y, z, dz, ph;
  float        p, *Prof;

  x = pos_x; // posicao do observador no terreno.
  y = pos_y;
  z = pos_z;
  dz = Incl*EscalaHF; // inclinacao inicial do horizonte.
  ph = 0; // incremento inicial da inclinacao do horizonte.
  Prof = profundidade; // profundidade da primeira fatia do terreno no Z-Buffer.
  MC = &MatrizCor[3*col]; // matriz de cores da tela.
  MP = &MatrizProf[col]; // matriz de profundidade para o Z-Buffer.
  while (ph < ProfMax) {
    x += dx; // caminha para o voxel da proxima fatia do terreno.
    y += dy;
    z -= dz; // atualiza a altura do horizonte na proxima fatia.
    ph += EscalaHF; // atualiza o incremento da altura do horizonte.
    _calc_viz(x, y); //determina os vizinhos de (x, y) com suas contribuicoes.
    _calc_alt();// calcula a altura pela interpolacao bilinear.
    if (h > z) { // o subvoxel e mais alto que o horizonte.
      _calc_cor();// cor do subvoxel.
      p = *Prof; // profundidade do voxel no Z-Buffer.
      do {
        dz -= EscalaHF; // atualiza a inclinacao do horizonte.
        *MC = c_red;   MC++; // pinta o voxel na tela.
        *MC = c_green; MC++;
        *MC = c_blue;  MC += passo_MC;
        *MP = p; MP += largura;
        if (MC > MC_max) return; // o raio ultrapassou o limite sup. da tela.
        z += ph; // atualiza a altura do horizonte flutuante.
      } // fim do do
      while (h > z); // enquanto o voxel eh mais alto que o horizonte.
    } // fim do if.
    Prof++; // profundidade da proxima fatia no Z-Buffer.
  } // fim do while.
}
```

A implementação dos aperfeiçoamentos relativos à diminuição dos intervalos entre as fatias é feita de modo direto, apenas alterando as iniciações das variáveis descritas no Capítulo 6.

Capítulo 8

8. Exemplos

Foram gerados 3 exemplos de visualização: um exemplo com a visualização de um terreno fictício com objetos, com uma tabela mostrando o desempenho do algoritmo; os outros dois exemplos com a visualização apenas do terreno, um equivalente a um tabuleiro de xadrez e outro a uma região da cidade de São José dos Campos - SP, obtida por satélite.

8.1. Terreno com Objetos

Para exemplificar e testar as idéias apresentadas neste trabalho, foi gerado um modelo de terreno, ilustrado na Figura 2.1. A sua visualização usando o algoritmo de Horizonte Flutuante é mostrada na Figura 8.1.

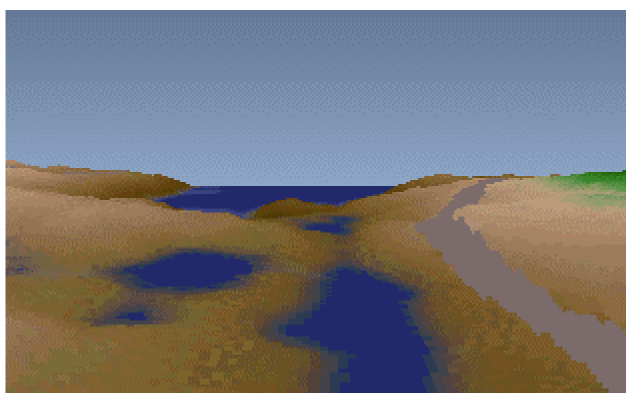


Figura 8.1 - Imagem gerada pelo Horizonte Flutuante.

Quando são adicionados os objetos mostrados na Figura 8.2 no modelo de terreno de acordo com a planta mostrada na Figura 8.3, o resultado do algoritmo proposto é a imagem mostrada na Figura 8.4.

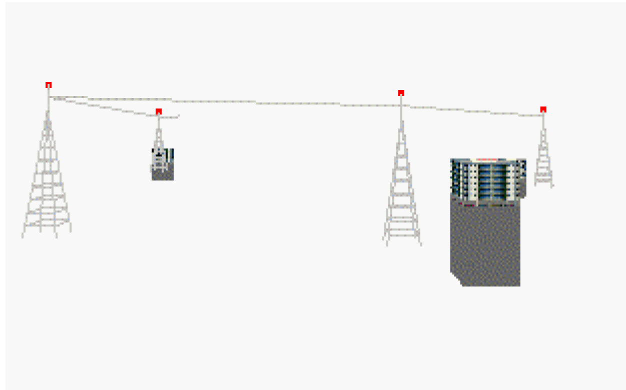


Figura 8.2 - Objetos a serem adicionados ao terreno.

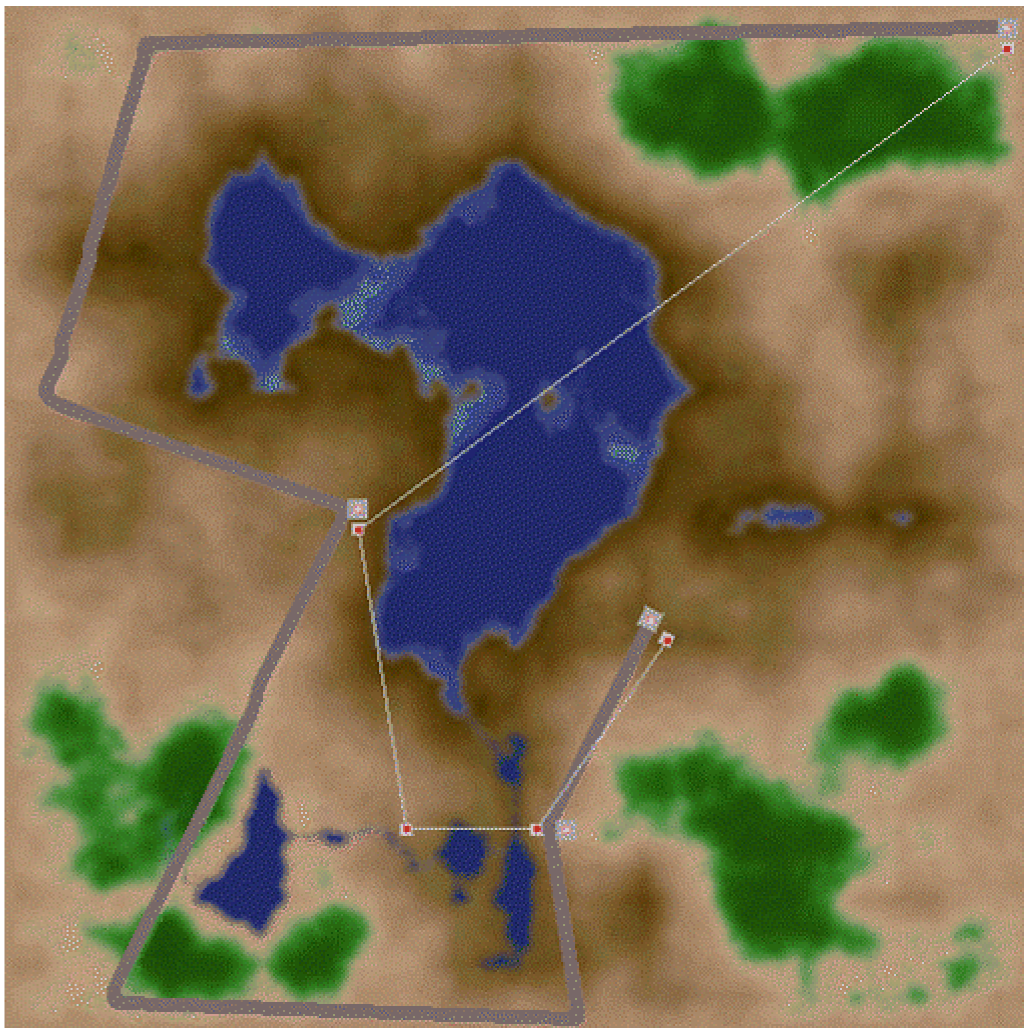


Figura 8.3 - Visão da planta do terreno com objetos.

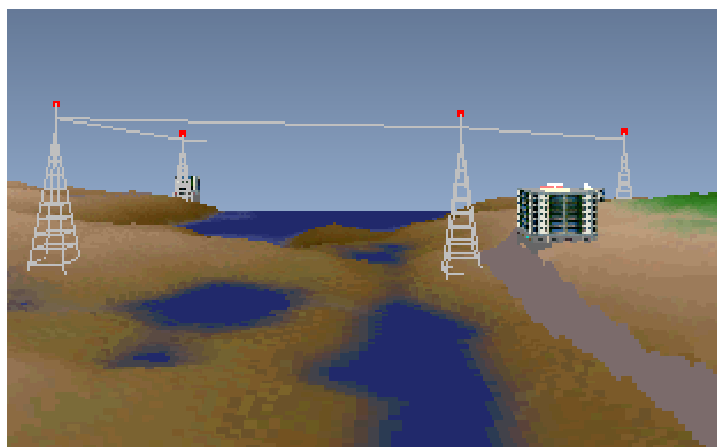


Figura 8.4 - Imagem com terreno e objetos.

Para avaliar a eficiência da estratégia proposta, medimos o desempenho do algoritmo em dois computadores diferentes: um PC PENTIUM 166 MHz e uma Silicon Indigo 2. O tempo médio para gerar um quadro em duas seqüências animadas é mostrado na Tabela 8.1. Esta tabela refere-se à implementação com uso de representação de ponto fixo. A primeira seqüência visualiza o terreno com os objetos (Figura 8.4) e a segunda visualiza o terreno sem os objetos (Figura 8.1). Foram calculados tempos médios usando 10 quadros.

Note, na Tabela 8.1, que o tempo gasto no cálculo de profundidade no algoritmo de Horizonte Flutuante é muito pequeno. O tempo maior é gasto transferindo os *buffers* para o OpenGL. Para carregar o *buffer* de cor gastamos tanto tempo quanto fizemos com a renderização de terreno. Para carregar o *buffer* de profundidade, tivemos que gastar o dobro do tempo gasto pelo Horizonte Flutuante. A eficiência da função responsável por estas transferências, `glDrawPixels`, é alvo de muitas discussões no grupo de notícias (*news group*) `news:comp.graphics.api.opengl`. É possível obter melhores resultados com uma versão mais eficiente desta função.

Na Tabela 8.2 são mostrados os tempos na implementação usando a representação de ponto flutuante ao invés de ponto fixo. Note a grande diferença na animação em relação à Tabela 8.1.

As taxas em quadros por segundo da mesma animação usada, mas usando somente o algoritmo de Z-Buffer para visualizar tanto o terreno quanto os objetos, segundo as representações descritas no Capítulo 3, são mostradas na Tabela 8.3.

	Terreno com objetos				Terreno sem objetos			
	PC		SGI		PC		SGI	
Passos no Algoritmo	**t(s)**	**t(%)**	**t(s)**	**t(%)**	**t(s)**	**t(%)**	**t(s)**	**t(%)**
Horizonte Flutuante	0,05	23	0,18	51	0,04	50	0,13	76
Carga de Z-Buffer	0,11	50	0,11	31	-	-	-	-
Carga do Buffer de cores	0,04	18	0,04	12	0,04	50	0,04	24
Objetos	0,02	9	0,02	6	-	-	-	-
Tempo total	0,22	100	0,35	100	0,08	100	0,17	100
Quadros/seg.	4,6		3,0		14,1		5,9	

Tabela 8.1 - Tempo em segundos para cada passo da estratégia proposta (ponto fixo).

	Terreno com objetos				Terreno sem objetos			
	PC		SGI		PC		SGI	
Passos no Algoritmo	**t(s)**	**t(%)**	**t(s)**	**t(%)**	**t(s)**	**t(%)**	**t(s)**	**t(%)**
Horizonte Flutuante	0,11	39	0,22	57	0,10	71	0,21	84
Carga de Z-Buffer	0,11	39	0,11	28	-	-	-	-
Carga do Buffer de cores	0,04	15	0,04	10	0,04	29	0,04	16
Objetos	0,02	7	0,02	5	-	-	-	-
Tempo total	0,28	100	0,39	100	0,14	100	0,25	100
Quadros/seg.	3,7		2,8		7,2		4,2	

Tabela 8.2 - Tempo em segundos para cada passo da estratégia proposta (ponto flutuante).

Terreno representado por	**PC**	**SGI**
[a] Pontos nas faces de topo	0,63	0,83
[b] Faces de topo	0,29	0,50
[c] Faces Frontais	0,25	0,48
[d] Segmentos de reta	0,43	0,46
[e] Paralelepípedos	0,07	0,17
[f] Triângulos *flats*	0,39	0,46
[g] Triângulos suaves	0,36	0,44

Tabela 8.3 - Taxa em quadros por segundo da animação com Z-Buffer.

Usando as propostas para aperfeiçoamento de imagens discutidas no Capítulo 6, temos as seguintes imagens:

Figura 8.5 - Visualização do terreno com mais fatias.

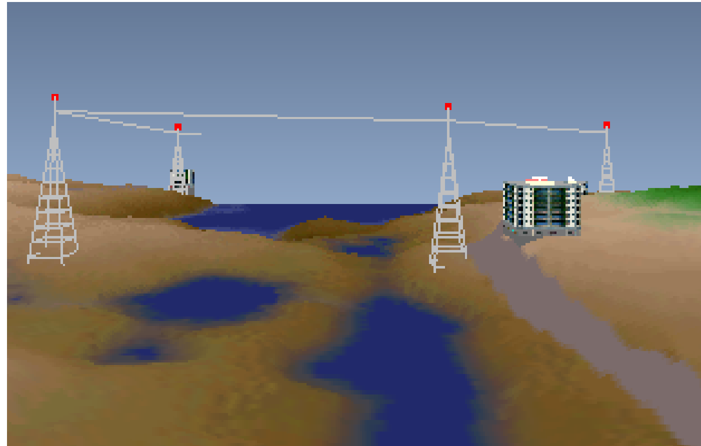


Figura 8.6 - Visualização do terreno com interpolação.

8.2. Tabuleiro de Xadrez

Este exemplo fictício de terreno planar com a textura de um tabuleiro de xadrez foi gerado com o intuito de visualizar um problema, descrito no capítulo 6, com o algoritmo de Horizonte Flutuante, no que diz respeito ao tamanho do passo, ignorando informações que neste caso são de cores (preto ou branco), gerando uma região contínua no terreno de mesma cor, o que não existe em um tabuleiro de xadrez. A imagem com o algoritmo original é ilustrada na Figura 8.7. As soluções são mostradas na Figura 8.8, Figura 8.9 e Figura 8.10.

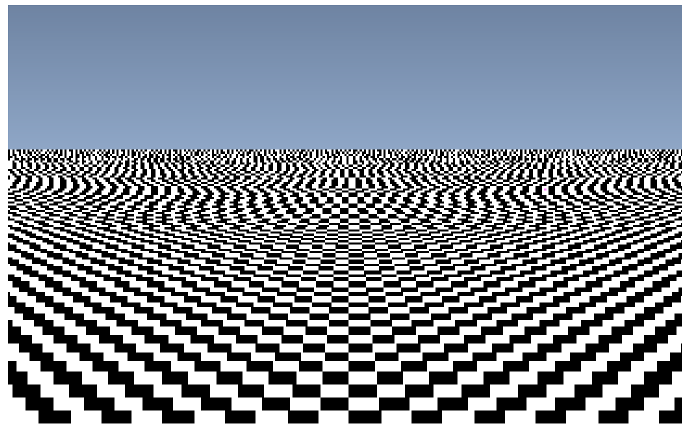
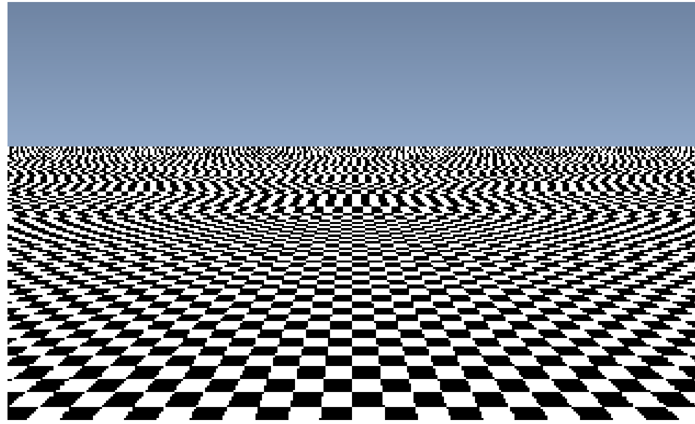
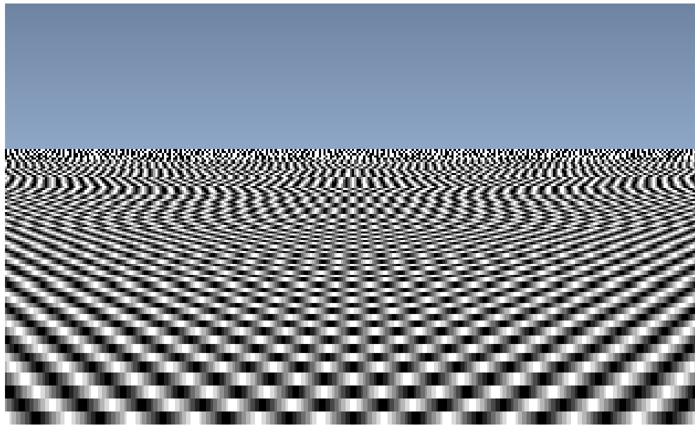
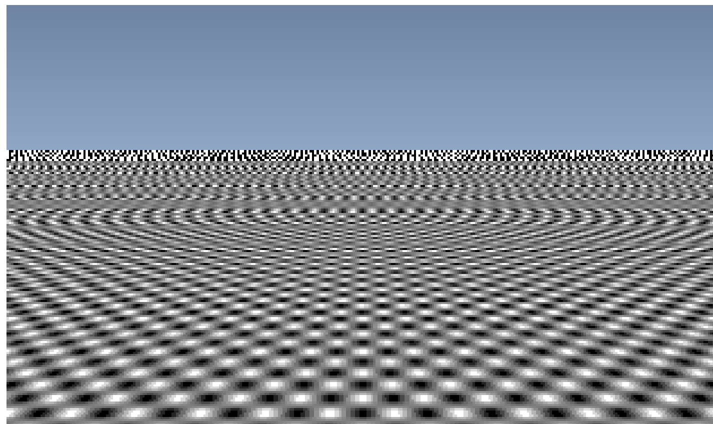


Figura 8.7 - Visualização de um terreno xadrez.







$$y = g(x) = sign\left(\sin\left(\frac{h_{obs} * \boldsymbol{p}}{incl - x}\right)\right)$$

onde: *sign(x)* vale *1* se *x>=0* e *-1*, caso contrário; h_{obs} é a altura do observador e *incl* o deslocamento do plano de visão, como mostra a figura. Nas visualizações mostradas acima estes valores são *190* e *40*, respectivamente. O valor *1* corresponde ao branco do xadrez, e o valor *-1* o preto. A Figura 8.12 mostra o gráfico da função *g(x)* gerada no MapleV. Note-se que esta figura apresenta um erro quando x está próximo de *140* devido as altas freqüências. A Figura 8.13, mostra outro gráfico, também obtido pelo Maple V, forçando uma amostragem e reconstrução para cada valor inteiro de *x.* Note-se que, apesar da amostragem ser mais fina, quando *x* se aproxima de *incl* o gráfico volta a apresentar problemas. Estes problemas denominados de *aliasing* estão relacionados com a freqüência de amostragem que deve ser acima da freqüência de Nyquist [Gomes+97].

Para o caso da projeção de terreno apresentada aqui, sempre teremos o fenômeno de *aliasing*, pois a função *g(x)* não possui banda limitada. Uma solução para o problema seria aplicar um filtro de passa-baixa na textura de forma a limitar a banda da função.

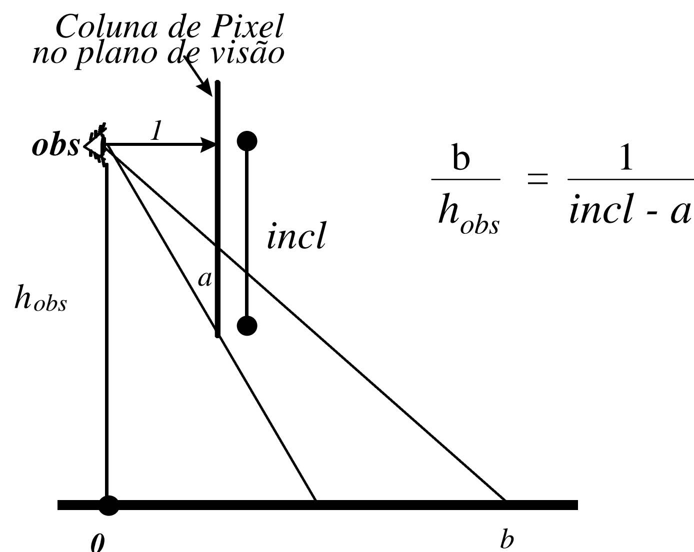


Figura 8.11 – Relação entre os pixels (*a*) do plano de visão com os pontos do terreno (*b*).

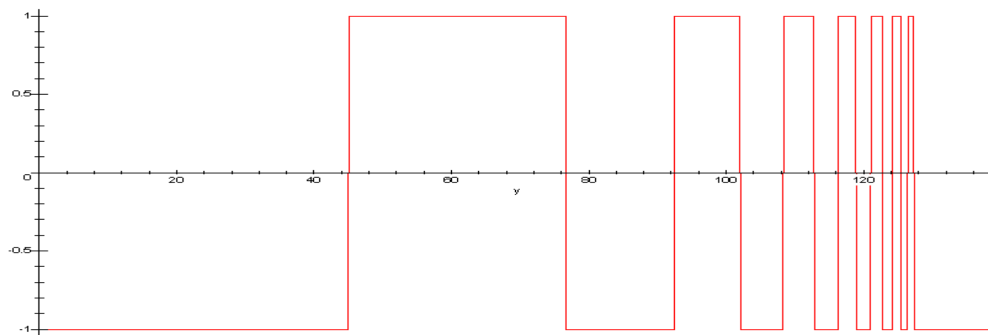


Figura 8.12 - Gráfico de *g(x)* gerada automaticamente pelo Maple V.

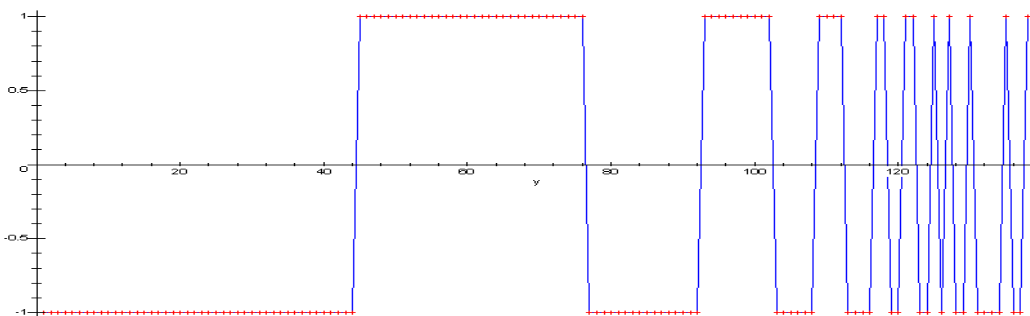


Figura 8.13 - Gráfico de *g(x)* gerado pelo Maple V com intervalo de $\Delta x=1$.

8.3. São José dos Campos

Outro exemplo foi gerado por um satélite e refere-se a uma região de São José dos Campos - SP. Por tratar-se de um dado provindo de satélite, as cores não retratam a realidade. Outro problema é a escala: a largura de cada célula da grade equivale a aproximadamente 30 metros (muito grande para o nosso caso). Neste exemplo pode-se notar os "retângulos grandes" mostrados na Figura 8.14. Na Figura 8.15, Figura 8.16 e Figura 8.17 são ilustradas as propostas de soluções para estes retângulos através da inclusão de mais fatias, da interpolação e da união das duas soluções, respectivamente. Os mapas de alturas e cores são

ilustrados na Figura 8.18. Neste mapa de alturas, as cores mais claras indicam áreas mais elevadas.

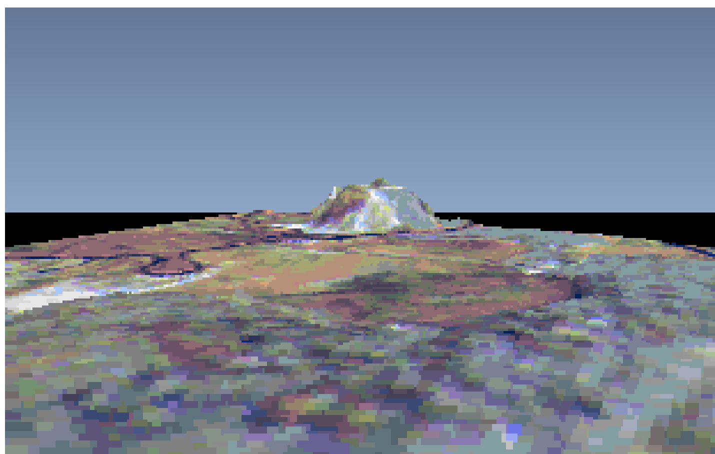


Figura 8.14 - Visualização de São José dos Campos.

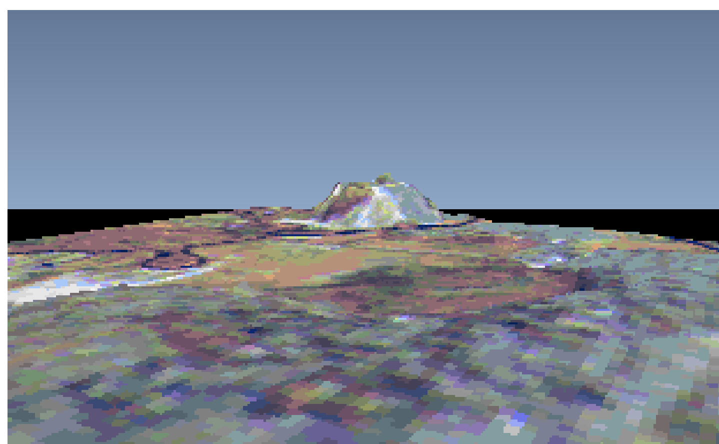


Figura 8.15 - Visualização de São José dos Campos com mais fatias.

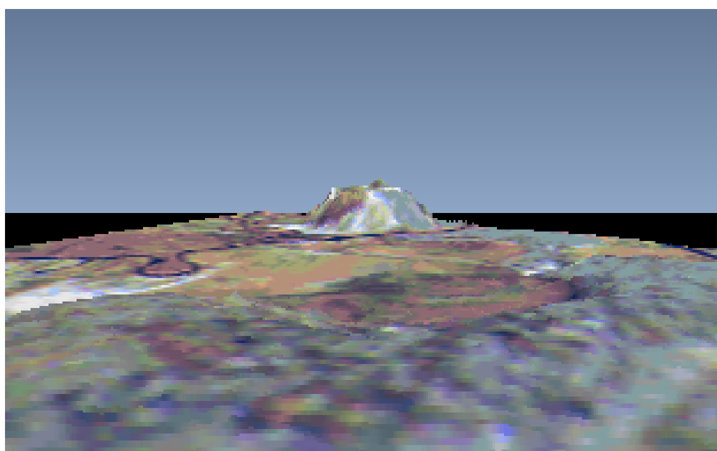


Figura 8.16 - Visualização de São José dos Campos com interpolação.

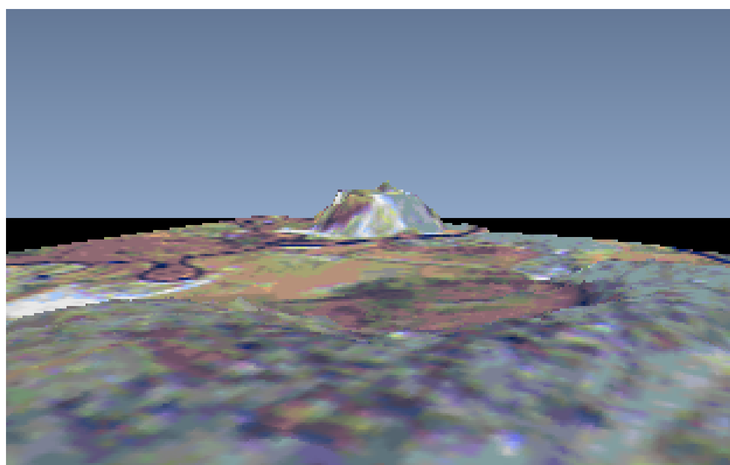
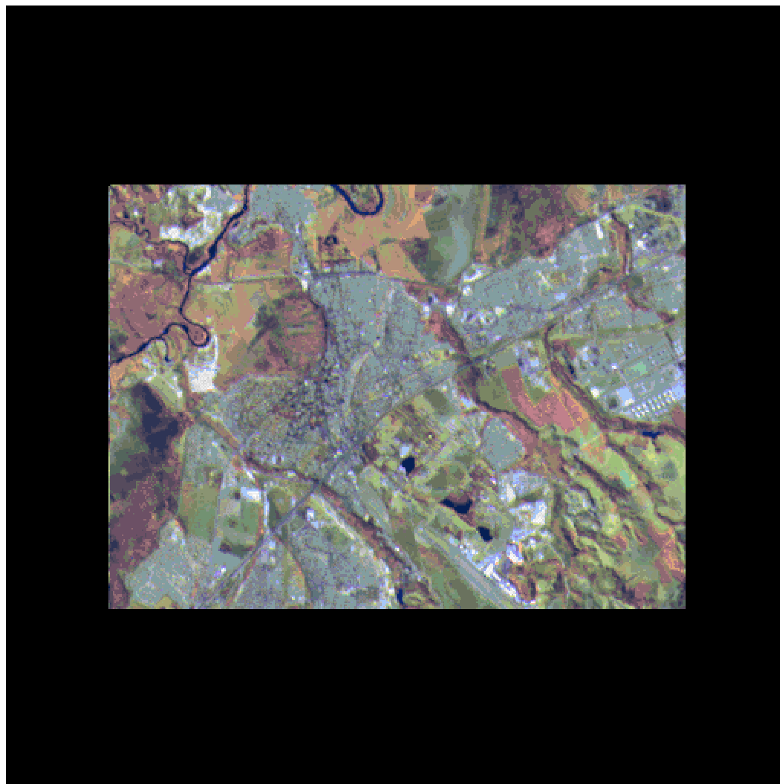


Figura 8.17 - Visualização de São José dos Campos com mais fatias e interpolação.

(a) Mapa de alturas.



(b) Mapa de cores.

Figura 8.18 - Mapas de um terreno de São José dos Campos - SP.

Capítulo 9

9. Conclusões

O método proposto para calcular informações de profundidade no algoritmo de Horizonte Flutuante é muito eficiente e não altera seu desempenho, uma vez que pode ser pré-calculado pois não varia em função da coluna da tela nem da posição do observador.

Até mesmo com o baixo desempenho da função `glDrawPixels`, os resultados obtidos com a estratégia proposta são bem melhores do que fazendo uso de Z-Buffer para visualizar o terreno. Este fato reforça a necessidade de um algoritmo específico para esta classe de problemas.

Embora o problema de *aliasing* não tenha sido severo nos exemplos apresentados aqui, com exceção do terreno xadrez, que é um caso atípico e fora da realidade, acreditamos que ele possa se tornar uma preocupação séria se usarmos fotografias aéreas reais (lembrando que as cores do terreno relativo a São José dos Campos não são reais).

É importante previamente analisar as características do terreno em questão para ver a necessidade das melhorias apresentadas no Capítulo 6. Por exemplo, para o terreno descrito na Figura 2.1, não houve muita melhoria na visualização em relação ao algoritmo proposto inicialmente. Já no terreno de São José dos Campos, a melhoria foi substancial e dá uma impressão melhor da topologia do terreno. Na Figura 8.9 e na Figura 8.10, a solução de interpolação não se aplicou bem, pois o interesse neste caso era visualizar os "enormes retângulos" que fazem parte do terreno.

Para os casos onde o algoritmo é usado em aplicações do tipo jogos (simuladores de vôo), não são necessárias estas melhorias, que diminuem a velocidade da animação. É mais eficiente fazer um tratamento nos mapas que descrevem o terreno.

Cada uma dessas soluções de aperfeiçoamento diminuirá o tempo de visualização da cena. Por exemplo, para a inclusão de novas fatias, o tempo para visualizar o terreno é

diretamente proporcional ao número de fatias novas, isto é, se adicionarmos uma fatia nova em cada região vazia, o tempo de visualização será dobrado, no pior dos casos.

Este trabalho resume bem a necessidade de utilizarmos algoritmos de visualização que tiram proveito do tipo específico de dados. E, também, responde à seguinte pergunta: até onde podemos deixar a tecnologia suprir as necessidades de velocidade de visualização, ao invés de pesquisarmos algoritmos específicos? Esta pergunta pode ser respondida com os resultados da visualização de terrenos usando o OpenGL, que, apesar de estar implementado em *hardware*, é bem mais lento que uma visualização com o algoritmo de Horizonte Flutuante, que é específico e não implementado em *hardware*.

9.1. Sugestões para Trabalhos Futuros

Como sugestões de trabalhos futuros, podemos citar:

- Implementação de um processo de iluminação no terreno: sugere-se desenvolver um pré-processamento no mapa de cores em função do mapa de alturas, isto é, aplicar uma "pré-iluminação", como é feito nos terrenos gerados pelo VistaPro, em função da inclinação do terreno no ponto. Com isto, teremos uma iluminação com ponto de luz no céu.
- Implementação de técnicas de multiresolução: para poder acelerar a visualização, sugere-se usar a técnica de multiresolução, onde região do terreno mais próxima do observador, teríamos uma resolução maior do terreno que em uma região mais afastada.
- Desenvolvimento de técnicas de visualização de terrenos com objetos com uma maior liberdade da câmara, isto é, liberdade para o observador inclinar a cabeça. Neste trabalho, assumiu-se que o observador sempre está direcionando a visão na horizontal. Usamos uma técnica para simular uma visão inclinada simplesmente deslocando o frustum para cima ou para baixo. Isto faz com que a visualização sofra uma deformação. Na inclinação da cabeça, as equações para os cálculos de profundidades formuladas neste trabalho não valem mais, pois cada *voxel* do terreno não estará mais paralelo ao plano de visão (ao plano *near* do frustum de

visão). O algoritmo de Horizonte Flutuante proposto também não funciona neste caso corretamente, pois os planos formados por cada coluna da tela com o observador não são mais perpendiculares à grade xy do terreno.

Referências

[Camara+96] Câmara, G. et al. *Anatomia de Sistemas de Informações Geográfica*. 10ª Escola de Computação, 1996.

[Cohen+94] Cohen, D. et al. Photorealistic Terrain Imaging and Flight Simulation. *IEEE Computer Graphics and Applications*, Vol. 14, No. 2, pp. 10-12, março de 1994.

[Frederick+96] Frederick, P. et al. Visualização Interativa Tridimensional de Modelos de Terreno com Textura. *Anais do IX SIBGRAPI*, pp. 341-342, 1996.

[Freese+95] Freese, P. *More Tricks of the Game Programming Gurus*. SAMS Publishing, 1995.

[Gomes+97] Gomes, J., Velho, L. *Image Processing for Computer Graphics*. Springer-Verlag, 1997.

[Graf+94] Graf, K. Ch. et al. Perspective Terrain Visualization - A Fusion of Remote Sensing, GIS, and Computer Graphics. *Comput. & Graphics*, Vol. 18, No. 6, pp. 795-802, 1994.

[GuGaCa97] Guedes, L., Gattass, M., Carvalho, P.C.P. *Real Time Rendering of Photo-Texture Terrain Height Fields*, SIBGRAPI'97.

[Kaneda+89] Kaneda, K. et al. Three Dimensional Terrain Modeling and Display for Environmental Assessment, *Computer Graphics*, Vol. 23, No. 23, Julho de 1989, pp. 207-214.

[LaMothe95] La Mothe, A. *Black Art of 3D Game Programming*. Waite Group Press, 1995.

[Martha+94] Martha, L. F. et al. Um Resumo das Transformações Geométricas para Visualização em 3D. *Caderno de Comunicações do VII SIBGRAPI*, pp. 9-12, 1994.

[Neider+93] Neider, J. et al. *OpenGL Programming Guide: the Official Guide Learnning OpenGL, release 1*. Addison-Wesley Publishing Company, 1993.

[Paglieroni+94] Paglieroni, D. et al. Height Distributional Distance Transform Methods for Height Field Ray Tracing. *ACM Transactions on Graphics*, Vol. 13, No. 4, pp. 376-399, outubro de 1994.

[Sawyer97] Sawyer, B. Skimming the Voxel Surface with NovaLogic's Commanche 3. *Game Developer*, pp. 62-70, abril-maio de 1997.

[Szenberg+97] Szenberg, F., Gattass, M., Carvalho, P.C.P. *An Algorithm for the Visualization of a Terrain with Objects*, SIBGRAPI'97.

[tvcg+96] Cohen-Or, D. et al. *A Real-Time Photo-Realistic Visual Flytrough*. ftp://ftp.math.tau.ac.il/pub/daniel/tiltan.ps.gz.

[VistaPro] VistaPro. http://www.callamer.com/vrli/vp.html.