UNIVERSIDADE FEDERAL DE GOIÁS – UFG

CAMPUS CATALÃO – CaC

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO – DCC

Bacharelado em Ciência da Computação

Projeto Final de Curso

Pipeline de Visualização Gráfica

Autor: Alex Fernando de Araújo

Orientador: Prof. Acrísio José do Nascimento Júnior

Catalão - 2007

Alex Fernando de Araújo

Pipeline de Visualização Gráfica

Monografia apresentada ao Curso de
Bacharelado em Ciência da Computação da
Universidade Federal de Goiás Campus Catalão
como requisito parcial para obtenção do título de
Bacharel em Ciência da Computação

Área de Concentração: Visualização de Sólidos
Orientador: Prof. Acrísio José do Nascimento Júnior

Catalão - 2007

Araújo, A. F.

Pipeline de Visualização Gráfica/Prof. Acrísio José do Nascimento Júnior- Catalão - 2007

Número de paginas: 79

Projeto Final de Curso (Bacharelado) Universidade Federal de Goiás, Campus Catalão, Curso de Bacharelado em Ciência da Computação, 2007.

Palavras-Chave: 1. Projeção. 2. Câmera. 3. Clipping e Detecção/Remoção de Superfícies Escondidas

Alex Fernando de Araújo

Pipeline de Visualização Gráfica

Monografia apresentada e aprovada em _____ de _____
Pela Banca Examinadora constituída pelos professores.

_____

Prof. Acrísio José do Nascimento Júnior – Presidente da Banca

_____

Prof. Ms. Márcio Antônio Duarte - Avaliador

_____

Prof. Veríssimo Guimarães Júnior - Avaliador

Dedico este trabalho à minha família por estar sempre disposta a me ajudar nos momentos que mais preciso, por ser o estímulo que me faz crescer sempre, buscando uma vida melhor, e por abrir mão da minha companhia, concedendo a mim a oportunidade de ir atrás dos meus objetivos. Dedico também a todas aquelas pessoas, que por não me conhecerem direito e/ou por terem pouco conhecimento sobre computação gráfica, criticaram a linha de pesquisa e os métodos que escolhi para o desenvolvimento desta monografia.

AGRADECIMENTOS

Em primeiro lugar a Deus, pois sem ele eu não estaria aqui e vocês não teriam a oportunidade de ver este trabalho.

Ao meu orientador Prof. ACRÍSIO JOSÉ DO NASCIMENTO JÚNIOR pelo incentivo, presteza no auxílio às atividades e discussões sobre o andamento e normatização desta monografia de conclusão de curso.

Especialmente ao Professor CARLOS ALBERTO MORENO BARBOSA, pelo apoio, auxílio às atividades desenvolvidas durante esta monografica e pelas dicas sobre a importância da computação gráfica e sobre a quantidade de boas pesquisas que podem ser desenvolvidas nesta área.

À NATHASA RODRIGUES PIMENTEL, que além de ser muito especial na minha vida e uma grande incentivadora para a conclusão deste trabalho, é a minha revisora particular, sem a qual esta monografia não teria a mesma qualidade.

Ao grande amigo LEONARDO GARCIA MARQUES, pelo auxílio no uso do LaTex para a formatação desta monografia e dos artigos produzidos.

Ao Professor FERNANDO GOULART, pelo incentivo e agradável recepção durante os simpósios e congressos que tivemos a oportunidade de participar juntos.

A todos os professores do departamento de Ciência da Computação e de outros departamentos que de alguma forma contribuíram com a minha formação.

A todos os demais professores, funcionários e colaboradores da Universidade Federal de Goiás pela dedicação e entusiasmo demonstrado ao longo do curso.

Aos colegas de classe pela espontaneidade e alegria na troca de informações e materiais numa rara demonstração de amizade e solidariedade.

E à minha família, que sempre foi fundamental na minha vida, me apoiando e incentivando o tempo todo.

“De nada adianta correr: é preciso partir no momento preciso.”(Jean de la Fontaine)

RESUMO

Araújo, A. F. Pipeline de Visualização Gráfica. Curso de Ciência da Computação, Campus Catalão, UFG, Catalão, Brasil, 2007, 79p.

Este trabalho apresenta um pouco do *pipeline* de visualização gráfica, que é composta pelo sistema de câmeras virtuais, projeção, *clipping* e detecção e remoção de superfícies ocultas, apresentando seus fundamentos, aplicações, vantagens e desvantagens. Também trata mais especificamente de *clipping* com a utilização de janelas de recorte (ou *viewport* no formato circular, que é pouco conhecida pela literatura, mostrando que a área de visualização de sólidos é pouco explorada na computação gráfica e permite muitas pesquisas neste segmento.

Palavras-Chaves: Projeção, Câmera, Clipping e Detecção/Remoção de Superfícies Escondidas

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Capítulo 1

Introdução

A visualização de sólidos é parte fundamental da computação gráfica, pois permite a exibição de objetos (tanto 2D quanto 3D) com as mesmas características que estes possuem no mundo real. Uma das principais tarefas da visualização de sólidos dentro da computação gráfica é proporcionar ao observador a interação com um mundo virtual, dando-lhe a sensação de estar vivenciando uma cena real. Para isso, há uma seqüência de passos e operações que devem ser seguidos para a obtenção de resultados satisfatórios que recebe o nome de *pipeline de visualização gráfica* e é mostrado na Figura 1.1.

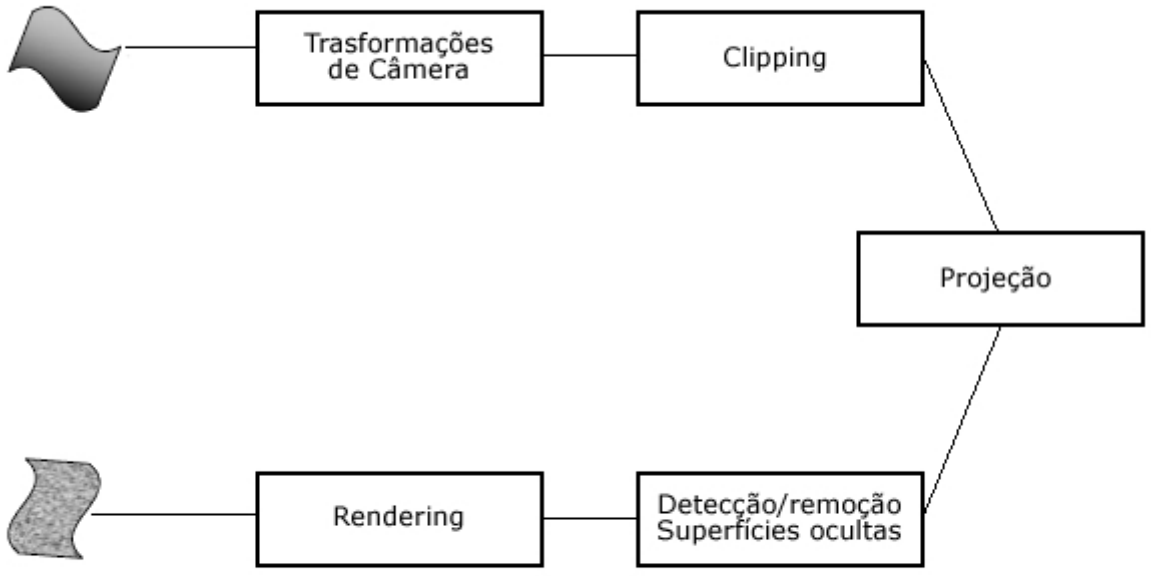

Figura 1.1: Pipeline de Visualização gráfica

Em uma cena virtual as câmeras do ambiente são responsáveis por tudo que é mostrado

ao observador, a projeção gráfica possibilita a transformação de cenas 3D para 2D, para que estas possam ser visualizadas no monitor. O *clipping* e a detecção e remoção de superfícies escondidas são aplicados para otimização, retirando os pixels que não são vistos, reduzindo assim o tempo de processamento e aumentando o realismo da cena. Porém, para que isto seja possível, é importante conhecer os fundamentos de cada parte do *pipeline* de visualização gráfica. Este trabalho tem como objetivo abordar estas técnicas, apresentado a importância da utilização de cada uma.

1.1 Contexto

As transformações de câmera, o *clipping*, as principais projeções gráficas e a detecção/remoção de superfícies ocultas serão abordadas no decorrer deste trabalho. Estas operações são responsáveis pela redução no tempo de processamento da cena (*clipping*), pela renderização correta dos objetos, evitando que um objeto localizado atrás de outro seja mostrado na frente (remoção de pontos escondidos), permite também a exibição dos cenários tridimensionais em meios de visualização 2D (projeção) e o posicionamento do observador na cena (transformações de câmera).

1.1.1 Câmera

Tudo que é visto em uma cena computacional é determinado pelo sistema de câmeras. O sistema de lentes e as operações das câmeras virtuais seguem os mesmos conceitos das reais. No entanto, para que o computador consiga simular o funcionamento de uma câmera, deve-se aplicar várias técnicas e conceitos, tais como: (a) transformação entre sistemas de coordenadas, (b) simulação de efeitos de foco e (c) manipulação da câmera.

Para efetuar a transformação do sistema de coordenadas global (coordenadas do mundo real) para o sistema de coordenadas de visualização (coordenadas do dispositivo gráfico) (a), as transformações geométricas (translação, rotação e escala) são fundamentais. Para reduzir esforço computacional, recomenda-se efetuar uma composição destas transformações antes de aplicá-las aos *pixels* dos objetos. Isto é feito através da multiplicação das matrizes de cada transformação, gerando uma única matriz que deve ser multiplicada às coordenadas de cada ponto.

O efeito de foco (b) ocorre quando um objeto fica fora do foco de visão. Quando isto ocorre, este objeto fica embaçado (desfocado), devendo, no sistema virtual, ser borrado para que a cena fique mais real. Para isso, aplica-se uma mistura das cores dos objetos que ficam fora de foco.

A manipulação da câmera virtual (c) é feita através das seguintes operações:

- *Azimuth*;

- *Elevation*;
- *Roll*;
- *Yaw*;
- *Pitch*;
- *Dolly*;
- *Zoom*.

1.1.2 Projeção

Os dispositivos de exibição de imagens são bidimensionais (monitores, fotografias, TV's). Como as cenas têm que manter a percepção de tridimensionalidade, a projeção gráfica tem uma importância muito grande no processo de visualização gráfica.

As técnicas de projeção são utilizadas na computação gráfica para transformar os pontos do sistema de coordenadas 3D para o 2D. Desta forma, torna-se possível exibir as cenas computacionalmente. As principais projeções são a paralela e a perspectiva. Na paralela, as arestas dos objetos mantêm relação de proporcionalidade, uma vez que as linhas projetoras são paralelas entre si. Assim, ela não gera cenas realistas, porém sua aplicação é muito forte na engenharia e arquitetura para a visualização de maquetes. A projeção perspectiva consegue reduzir a dimensão da cena e ainda manter a noção de profundidade desta, dando ao observador a sensação de estar vendo um cenário tridimensional. No entanto, este encontra-se em um sistema de coordenadas 2D. Para conseguir este efeito, determina-se um ponto de referência e faz com que todos os raios projetores se encontrem neste ponto. Com isso, consegue-se uma redução gradativa dos objetos que localizam-se mais distante do observador, simulando o sistema de visão humano.

1.1.3 Clipping

Ao criar um cenário virtual, o sistema de câmeras define uma área onde os agentes internos a ela podem ser vistos. Esta área recebe o nome de *viewport* (ou janela de visualização).

Por muito tempo, os objetos localizados fora desta janela eram renderizados e, mesmo não sendo vistos pelo observador, aumentava muito o esforço para tratar cenas que possuíam muitos polígonos fora da área visível. Diante deste fato, surgiram os algoritmos de *clipping* (ou recorte) para retirar estas partes e reduzir o processamento gasto para desenhá-las.

1.1.4 Remoção de Superfícies Ocultas

Remover as superfícies que estão sobrepostas em uma cena reduz a quantidade de pontos a serem renderizados e aumenta o realismo da mesma. Seu principal objetivo é aumentar o realismo, uma vez que se os objetos estiverem muito próximos uns aos outros, dependendo da ordem de renderização, pode acontecer de um objeto mais ao fundo ser desenhado na frente de outro, como mostra a Figura 1.2.

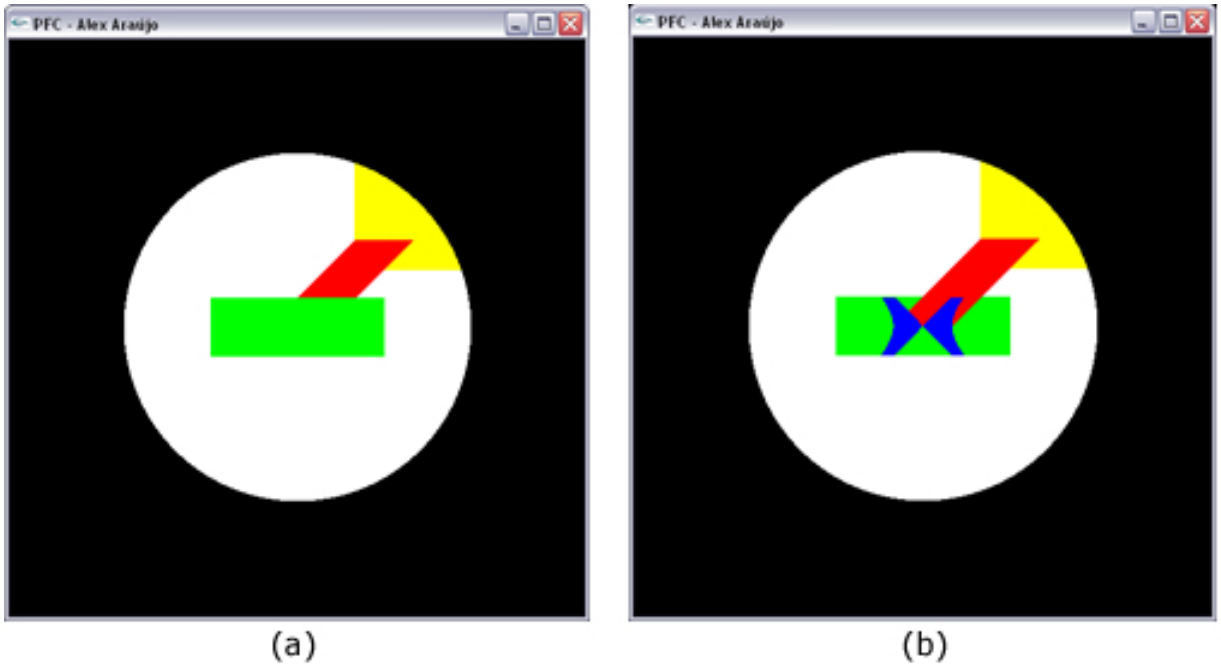

Figura 1.2: (a) Cena renderizada sem a remoção de pontos escondidos e (b) cena renderizada após a remoção dos pontos escondidos.

Vários algoritmos já foram desenvolvidos para tratar esta particularidade da computação gráfica. Há abordagens que rasterizam os objetos e verifica a validade de cada ponto, como o método *z-buffer*, que é utilizado pelo pacote gráfico OpenGL (uma biblioteca livre e compatível com a maioria dos *hardwares* gráficos). Outras técnicas menos utilizadas preferem transformar cada polígono em um conjunto de arestas, encontrar as interseções destas e preencher novamente as partes que são visíveis. A utilização destas técnicas depende da aplicação a ser desenvolvida e dos recursos disponíveis. A primeira técnica citada necessita de mais memória, mas em contrapartida, é mais simples de ser implementada e pode ser aplicada a polígonos de qualquer forma.

1.2 Objetivos

Este trabalho está focado em apresentar os conceitos mais importantes da visualização de sólidos dentro da computação gráfica, apresentando os principais fundamentos do sistema vir-

tual de câmeras e das projeções gráficas, o processo de detecção e remoção de superfícies escondidas, além das técnicas de *clipping*, dando ênfase à utilização de uma circunferência como *viewport*).

Com este trabalho pretende-se adquirir um conhecimento mais detalhado sobre o funcionamento dos sistemas de câmera, projeções, clipping e remoção de faces escondidas dentro da Computação Gráfica, bem como o conhecimento de alguns algoritmos adotados nestas técnicas e, principalmente, obter (desenvolver) um algoritmo capaz de recortar os segmentos de reta e as superfícies que encontram-se fora da janela de *clipping* circular.

1.3 Organização da Monografia

Este trabalho está organizado da seguinte forma:

- Fundamentos do sistema de câmera é abordado no Capítulo 2.
- O Capítulo 3 apresenta os principais tipos, assim como a características e definições das projeções gráficas.
- Os principais algoritmos de recorte consideram o *viewport* formado por uma área retangular. O Capítulo 4 apresenta de forma mais detalhada uma descrição dos conceitos e as aplicações do *clipping*. Já no Capítulo 6, é apresentado uma discussão aprofundada sobre a utilização de janelas de visualização circular, mostrando os algoritmos para recorte de linhas e de superfícies para estas janelas, bem como os resultados de suas execuções.
- Os detalhes sobre a remoção de superfícies escondidas e seus principais algoritmos são apresentados no Capítulo 5.

Capítulo 2

Câmera

2.1 Introdução

Uma das operações básicas efetuadas durante a produção de cenas tridimensionais em computação gráfica é a modelagem de um sistema de câmera [Kolb et al., 1995]. A câmera é considerada uma entidade que encapsula uma série de atributos que permitem modelar a visão de um observador da cena virtual [da Silva, 2002]. Seu comportamento depende destes atributos, podendo por exemplo, ser fixa ou capaz de deslocar-se junto com um agente pelo cenário [Brunstein et al., 2003].

Após a criação de cenas e objetos computacionais deve-se efetuar a sua apresentação para o usuário. Para obter a visualização de uma cena o primeiro passo é atentar para os parâmetros de exibição, ou seja, definir os parâmetros da câmera [Azevedo e Conci, 1986]. Estes definem a posição e a orientação do plano de projeção que correspondem à câmera da cena. Neste processo, as coordenadas do objeto são transferidas para as coordenadas de referência e projetadas sobre o plano de visualização. As operações de câmeras [Marquis-Bolduc et al., 2008] podem ser aplicadas tanto a formas aramadas quanto a formas sólidas, permitindo ainda a aplicação de técnicas de *rendering* nas superfícies visíveis para aumentar o realismo das cenas.

Rendering - Processo do pipeline de visualização gráfica onde aplica-se técnicas como iluminação e sombreamento para permitir uma visualização mais realista de um ambiente virtual.

2.2 Características da Câmera Virtual

2.2.1 Uma Analogia com a Câmera Real

Para facilitar o entendimento do conceito de câmera deve-se fazer uma analogia com uma máquina fotográfica. O fotógrafo define a posição, a orientação e o ponto focal da sua câmera. A fotografia que ele obtém é uma projeção da cena em um plano 2D. Analogamente, a imagem

obtida de uma câmera virtual depende de vários fatores que determinam como esta é projetada no monitor, que é um meio de exibição 2D. Os principais destes parâmetros são a posição, a orientação e o ponto focal da câmera, além do tipo de projeção e do posicionamento dos *clipping planes (planos de clipping)*.

2.2.2 O sistema de Lentes

Formado por uma ou mais camadas de vidro ou às vezes plástico (Figura 2.1) com um orifício de convergência sobre um dos eixos do sistema de coordenadas, a lente da câmera é responsável pela captura das imagens. Este orifício, quando aberto, permite a passagem dos raios de luz refletidos pelos objetos da cena e é o responsável pela determinação da abertura da câmera. A quantidade *(determinada pela abertura da câmera)* e a intensidade dos raios de luz que atingem a lente da câmera definem o posicionamento do objeto em relação à fonte de luz e a sua profundidade, considerando a posição do observador *(posição da câmera)* [Spindler et al., 2006].

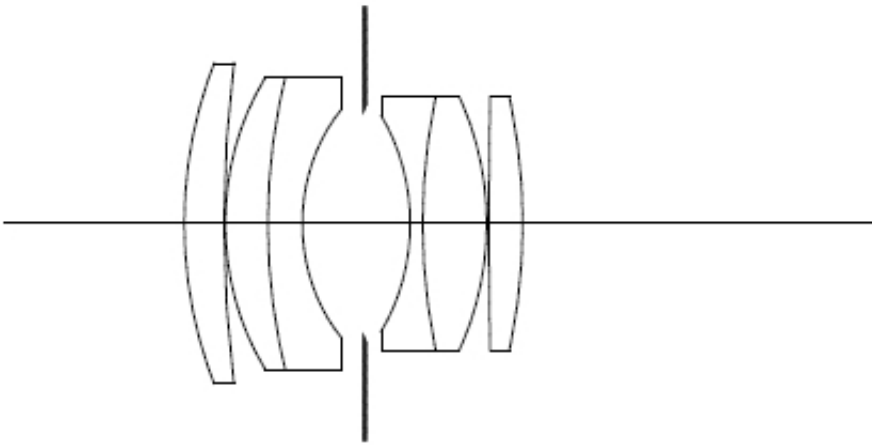

Figura 2.1: Arquitetura da lente de uma câmera.

2.3 Especificação das Coordenadas de Visualização

Para representar computacionalmente as câmeras virtuais deve-se definir um plano de projeção (explicado com mais detalhes no Capítulo 3), perpendicular ao eixo z do sistema de coordenadas. A cena é gerada no sistema de coordenadas global (ou coordenadas global, ou ainda *World Coordinates*). Assim, o próximo passo é transformar as coordenadas desta cena para o sistema de coordenadas de visualização *(SCV)* (*Viewing Coordinate System*). O sistema de coordenadas global representa o mundo em que o objeto é gerado, ou seja, são as coordenadas reais do objeto antes destas serem passadas para coordenadas do ambiente virtual.

A determinação do sistema de coordenadas de observação é feita através da especificação de sua origem, da seleção da direção positiva para o eixo z do novo sistema e da especificação do vetor normal *(N)* ao plano de observação (ou plano de projeção). Por fim, determina-se o vetor *view up* (V) que representará a direção do eixo *y* do *SCV*. O ponto de referência de visualização (PRV) é definido no sistema de coordenadas global e é utilizado para a determinação da origem do *SCV*. Geralmente, este ponto é considerado como sendo a posição da câmera [Traina e de Oliveira, 2003]. A Figura 2.2 apresenta o SCV em relação ao sistema de coordenadas global.

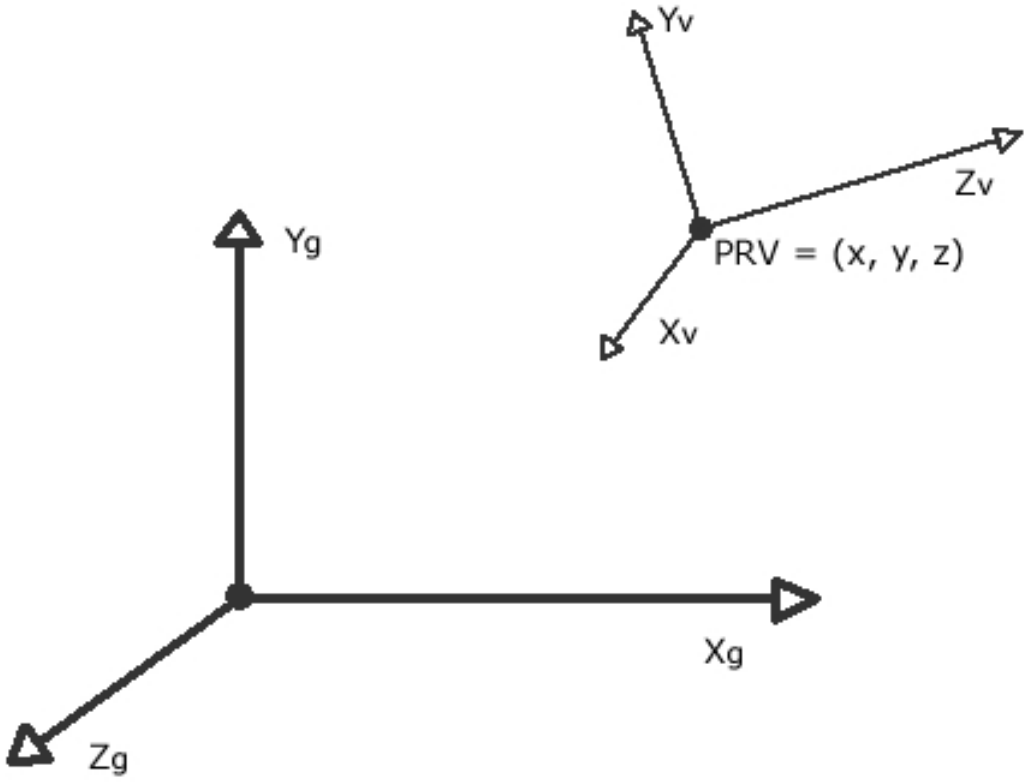

Figura 2.2: Um SCV baseado no sistema de coordenadas global, onde X_V, Y_V e Z_V são coordenadas do sistema de coordenadas de visualização e X_g, Y_g e Z_g são do sistema global.

O vetor *V* deve ser perpendicular ao plano *xz* do *SCV* e isto pode tornar-se um problema para o programador na hora de implementar seu sistema de câmera virtual. Para amenizar este problema os métodos de transformação de visualização implementados pelos atuais pacotes gráficos ajustam *V* de forma a torná-lo perpendicular a *N*. Assim, o programador tem a liberdade de escolher qualquer direção conveniente para o vetor *view up*, tomando cuidado apenas para que este não seja paralelo ao eixo z do sistema de observação. O vetor *(U)* que determina a direção do eixo *x* do *SCV* é calculado a partir dos vetores *N* e *V*. Lembrando que *U* deve ser perpendicular a *V* e a *N*.

2.3.1 Transformação do Sistema de Coordenadas Global para o Sistema de Coordenadas de Visualização

A projeção dos objetos de uma cena é feita quando as coordenadas deste pertencem ao *SCV*. Portanto, antes de efetuar as operações de projeção (que serão apresentadas no Capítulo 3) deve-se transformar as coordenadas da cena criada do sistema global para o *SCV*. Esta conversão é efetuada através de transformações geométricas *(translação e rotação)*, sobrepondo o sistema de coordenadas global. A translação é usada para levar o ponto de referência de visualização para a origem do sistema global [Lam et al., 2006]. Logo após, rotações são executadas com o objetivo de alinhar os eixos principais do novo sistema de coordenadas *(SCV)* com os eixos principais do sistema antigo.

2.3.2 Transformação entre Sistemas de Coordenadas

Para se trabalhar com câmera, deve-se fazer algumas transformações antes de tratar diretamente com o objeto a ser observado. Entre as diversas transformações que devemos fazer, a transformação entre sistemas de coordenadas é o princípio para se ter uma visualização adequada.

Um objeto, quando apresentado, nem sempre se encontra no sistema de coordenadas Cartesiano devendo ser convertido para esta. Em aplicações de design e modelagem, objetos individuais são definidos em seu próprio sistema de coordenadas, e as coordenadas locais devem ser transformadas para posicionar os objetos no sistema de coordenadas global da cena. Esta seção trata especificamente de transformações entre sistemas de coordenadas Cartesianos.

Para transformar descrições de um objeto dadas em um sistema de coordenadas xy para um sistema x'y' com origens em (0,0) e (x_0, y_0), com um ângulo de orientação entre os eixos x e x', deve-se determinar a transformação que sobrepõe os eixos xy aos eixos x'y'. Então, as seguintes etapas podem ser seguidas:

- Transladar o sistema x'y' de modo que a origem coincida com a origem do sistema xy: T($-x_0$, $-y_0$)
- Rotacionar o eixo x' de forma que ele coincida com o eixo x: R($-\theta$)

Concatenando as duas matrizes de transformação obtém-se a matriz de composição (Matriz 2.1) que descreve a transformação necessária:

$$M_{xy,x'y'} = R(-\theta).T(-x_0, -y_0) \tag{2.1}$$

Um método alternativo para estabelecer a orientação do segundo sistema de coordenadas consiste em especificar um vetor V na direção positiva do eixo y', passando pela origem do sistema xy. Para isso, basta ter um vetor unitário indicando a direção desejada, que é a do

vetor V. Então encontra-se o vetor unitário $\overrightarrow{u}$ (Vetor 2.3) ao longo do eixo x' rotacionando v (Vetor 2.2) de 90º no sentido horário.

$$v = \frac{V}{|V|} = (v_x, v_y) \tag{2.2}$$

$$u = (v_y, -v_x) = (u_x, u_y) \tag{2.3}$$

Os elementos de uma matriz de rotação podem ser expressos como elementos de um conjunto de vetores ortogonais. Esta, é dada pela Matriz 2.4.

$$M_{rot} = \begin{bmatrix} u_x & u_y & 0 \\ v_x & v_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.4}$$

2.3.3 Composição de Transformações

Uma composição de transformações se faz necessária para aumentar a eficiência da operação com a imagem; visto que seria necessário várias transformações, uma de cada vez. Quando estas são agrupadas em uma única operação, obtém-se o mesmo resultado final e diminui o esforço computacional. Considerando a rotação de um objeto em torno de um ponto P_1, ocorre que a realização desta operação é mais simples na origem do sistema. Então, algumas operações devem ser realizadas:

- Efetuar uma translação de P_1 até a origem;
- Efetuar a rotação desejada e;
- Efetuar a translação inversa à realizada no passo 1.

Realizando matematicamente a operação mostrada na Equação 2.5.

$$T(x_1, y_1).R(\theta).T(-x_1, -y_1) = \begin{bmatrix} 1 & 0 & x_1 \\ 0 & 1 & y_1 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} 1 & 0 & -x_1 \\ 0 & 1 & -y_1 \\ 0 & 0 & 1 \end{bmatrix} =$$
$$\begin{bmatrix} \cos\theta & -\sin\theta & x_1.(1-\cos\theta) + y_1.\sin\theta \\ \sin\theta & \cos\theta & y_1(1-\cos\theta) - x_1.\sin\theta \\ 0 & 0 & 1 \end{bmatrix} \tag{2.5}$$

Esse procedimento também pode ser utilizado, de maneira similar, para operação de escalamento de um objeto em relação a um ponto arbitrário P_1. Como mostrado na Equação 2.5, basta executar uma operação de escalamento ao invés de uma rotação. Matematicamente, o procedimento ficará da seguinte forma, como mostrado na Equação 2.6.

$$T(x_1, y_1).S(s_x, s_y).T(-x_1, -y_1) = \begin{bmatrix} 1 & 0 & x_1 \\ 0 & 1 & y_1 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} 1 & 0 & -x_1 \\ 0 & 1 & -y_1 \\ 0 & 0 & 1 \end{bmatrix} =$$

$$\begin{bmatrix} s_x & 0 & x_1.(1-s_x) \\ 0 & s_y & y_1(1-s_y) \\ 0 & 0 & 1 \end{bmatrix} \tag{2.6}$$

2.3.4 Transformações em 3D

A capacidade para representar e visualizar um objeto em três dimensões é fundamental para a percepção de sua forma. Porém, em muitas situações é necessário mais do que isto, ou seja, poder ”manusear”o objeto, movimentando-o através de rotações, translações e mesmo escala.

Uma característica das transformações a serem mostradas é a utilização de coordenadas homogêneas. Nas transformação de vetores a operação é diferente da transformação de pontos. As coordenadas homogêneas permitem unificar este tratamento, colocando uma coordenada extra w, que tem valor igual a zero, para o tratamento de vetores, e valor igual a um, para o tratamento de pontos. Termos independentes formam uma coluna extra da matriz. A translação em 3D pode ser vista simplesmente como uma extensão a partir da translação 2D, Tabela 2.1.

Eixo de Rotação	Direção da Rotação Positiva
x	y para z
y	z para x
z	x para y

Tabela 2.1: Sentido de Rotação

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{2.7}$$

A Matriz 2.7 é a matriz de translação para o sistema de coordenadas 3D ,que também pode ser representada como em 2.8.

$$P' = T(d_x, d_y, d_z).P \tag{2.8}$$

Similarmente, a escala em 3D, fica (2.9 ou 2.10):

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{2.9}$$

$$P' = T(s_x, s_y, s_z).P \tag{2.10}$$

Finalmente, verifica-se como ficam as equações de rotação em 3D, pois estas podem ser efetuadas em relação a qualquer um dos eixos cartesianos, gerando as matrizes de rotação em torno dos eixos *z* (2.11), *x* (2.12) e *y*(2.13).

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{2.11}$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{2.12}$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{2.13}$$

Todas as matrizes de transformações foram retiradas do livro de Geometria Analítica [da Silva e dos Reis, 1996].

2.3.5 Composição de Transformações em 3D

A composição de transformações em 3D pode ser entendida mais facilmente através do exemplo indicado na Figura 2.3. Esta mostra que os segmentos de reta P_1P_2 e P_1P_3 saem de uma posição inicial, longe da origem do sistema (na posição inicial em a), para uma posição final, na origem do sistema (na posição final em b). As transformações a serem feitas podem ser indicadas conforme os seguintes passos:

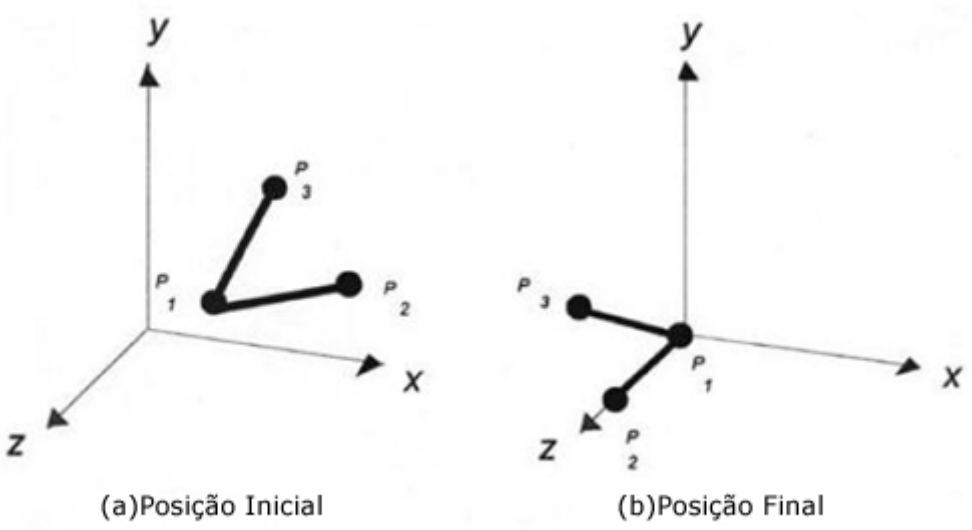

Figura 2.3: Transladando P_1, P_2 e P_3 da posição inicial em (a) para a origem (b).

- Transladar P_1 para a origem;
- Rotacionar o segmento P_1P_2 em relação ao eixo y, de forma que ele (P_1P_2) fique no plano yz;
- Rotacionar o segmento P_1P_2 em relação ao eixo x, de forma que ele (P_1P_2) fique sobre o eixo z;

- Rotacionar o segmento P_1P_3 em relação ao eixo z, de forma que ele (P_1P_3) fique no plano yz.

Após o quarto passo, o resultado obtido é o mesmo mostrado na Figura 2.3(b). As Figuras 2.4, 2.5 e 2.6 mostram passo a passo a operação de rotação no sistema 3D.

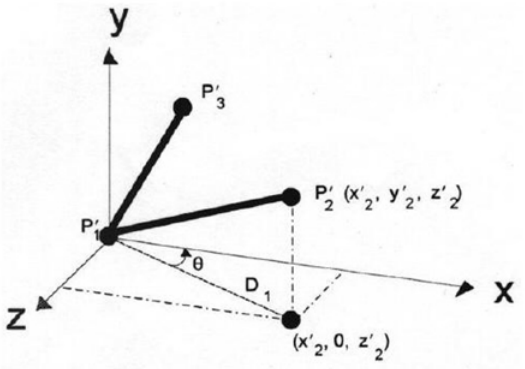

Figura 2.4: Rotação do segmento de reta P_2P_3.

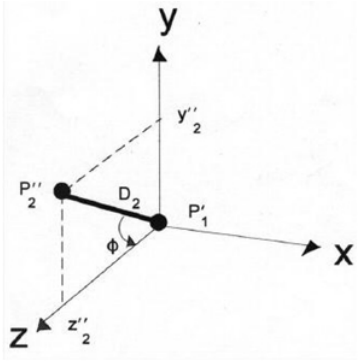

Figura 2.5: Rotação em relação ao eixo x. P_1 e P_2 de comprimento D_2 é rotacionado em direção ao eixo z, pelo ângulo positivo ϕ.

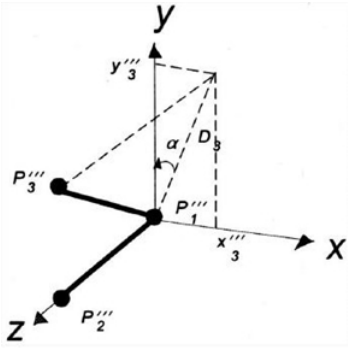

Figura 2.6: Rotação em relação ao eixo z.

2.3.6 A Câmera Virtual

A localização da câmera e para onde esta aponta são definidos por sua posição e seu ponto focal, respectivamente, e a direção dos raios projetores é definida pelo vetor que vai da posição da câmera até o seu ponto focal, onde geralmente, localiza-se o plano no qual a cena é projetada. Este plano tem sua orientação em relação às linhas projetoras dependente do tipo de projeção utilizada. A posição, o ponto focal e o vetor *view up* definem a orientação da câmera sintética. A forma como os raios de luz do ambiente atingem a lente da câmera, ou seja, o mapeamento dos agentes da cena no plano de visualização, é definido pelo tipo de projeção adotada [Ware e Osborne, 1990]. O Capítulo 3 aborda os principais tipos de projeção, seus conceitos e características.

Os planos de recorte normalmente são perpendiculares ao vetor de projeção e são usados para eliminar os objetos que encontram-se muito próximos ou muito distantes da câmera. Suas localizações dependem do intervalo de recorte definido durante sua implementação. Estes planos recebem o nome de plano anterior e plano posterior, como mostra a Figura 2.7. Seu uso é fundamental, pois os agentes que se encontram muito próximos (praticamente encostados na lente da câmera) ou muito distantes do observador, ou seja, da câmera, não possuem boa definição, tornando-se desnecessários em muitas aplicações.

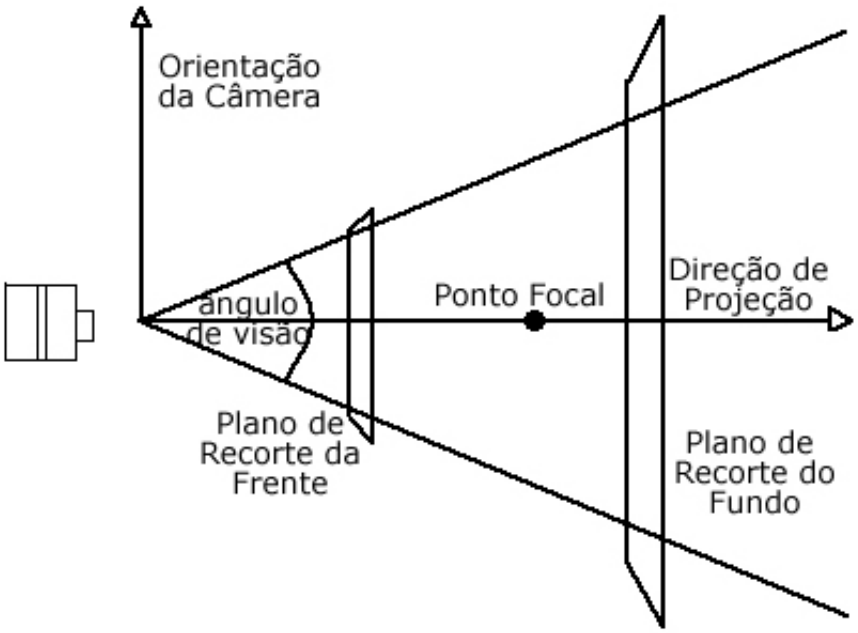

Figura 2.7: Atributos da câmera [Rogers, 1985].

A câmera vai funcionar como os olhos do observador, isto significa que este enxerga apenas o que a câmera vê. Assim, a posição de visualização é escolhida de acordo com o que se deseja mostrar; (a) a frente, (b) a parte de trás, (c) os lados, (d) o topo e/ou (e) a base do objeto [Drucker e Zeltzer, 1994]. Além disso, há a possibilidade de colocar a câmera entre vários

objetos ou até mesmo dentro destes. Portanto, o primeiro passo do *pipeline* de visualização gráfica é definir a orientação da câmera [Sutherland, 1964].

A abertura ou o tipo de lente estão entre os parâmetros de maior influência do sistema de câmeras. Esta abertura é determinada pela janela de *clipping*, isto é, quanto menor esta janela, menor é a abertura da lente e vice-versa. Porém, apenas o uso da câmera não torna possível a visualização nítida do cenário, deve-se também determinar a intensidade correta de luz que incide na cena, pois é a reflexão da luz que forma a imagem na lente da câmera [Yang et al., 2002]. Devido a estas características do sistema de câmeras percebe-se que quanto maior a distância entre a câmera e a cena, mais objetos são mostrados, porém com menos detalhes.

As câmeras virtuais têm a vantagem de ser mais flexíveis do que as reais [Ware e Osborne, 1990]. As primeiras podem ser instaladas em qualquer parte da cena sem muito esforço. Por exemplo, para obter uma visão aérea de um determinado cenário a câmera deve ser suspensa a uma altura desejada. Em um cenário real, a câmera deve ser suspensa por algum dispositivo, o que muitas vezes aumenta o nível de dificuldade, já as câmeras virtuais basta alterar seus parâmetros de posicionamento.

Desta forma, a nitidez dos objetos visualizados está diretamente ligada ao posicionamento da câmera responsável por sua visualização. Quando um cenário é fotografado utilizando-se uma máquina fotográfica (seja ela analógica, ou digital), percebe-se que alguns objetos aparecem nítidos enquanto outros não. Este fenômeno é decorrente da distribuição dos objetos na cena, quanto maior a profundidade do objeto em relação à câmera (ou observador) menos nítido ele é. A falta de nitidez também aparece em objetos localizados muito próximos à câmera. Em computação gráfica, este efeito é obtido desfocando os objetos de acordo com sua profundidade. Isto deve ser feito misturando-se os *pixels* dos objetos que encontram-se fora de foco com os *pixels* de seus vizinhos, fazendo com que as cores de um apareçam dentro de outro, borrando as imagens. Estes efeitos de câmera podem ser incorporados a algoritmos *scan-line* ou *ray-tracing* [Hearn e Baker, 1986]. Estes efeitos de foco tornam as cenas computacionais bem mais reais, porém demandam muito processamento.

2.4 Simulando Efeitos de Foco em Câmera

Ao modelar os efeitos de câmera em uma cena deve-se especificar o tamanho focal e outros parâmetros das lentes convexas, ou seja, a abertura da lente da câmera (conhecida também como abertura da câmera) que será posicionada em frente ao plano de projeção. Estes parâmetros fazem com que alguns objetos fiquem no foco enquanto outros fiquem fora dele.

A Figura 2.8 apresenta a estrutura da lente de uma câmera. O tamanho focal **f** é a distância do centro da lente até o foco (ponto focal), que é a posição de convergência das linhas de luz

que entram paralelas na lente. A abertura da câmera é dada pelo parâmetro **n** (Equação 2.14).

$$n = \frac{f}{2 \times raio\ \ da\ \ lente} \tag{2.14}$$

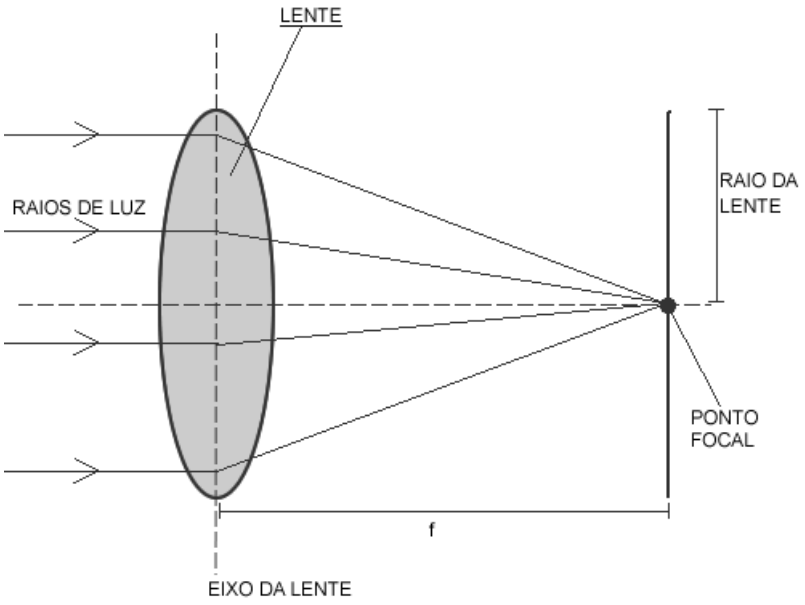

Figura 2.8: Esquema de uma lente convexa.

O algoritmo de *ray-tracing* determina os efeitos de foco através da Equação 2.15. Como mostrado na Figura 2.9, o parâmetro **d** é a distância do centro da lente até o objeto e d_i é a distância do centro da lente até o plano da imagem, local onde o objeto encontra-se no foco.

$$\frac{1}{d} + \frac{1}{d_i} = \frac{1}{f} \tag{2.15}$$

O objeto e sua imagem estão de lados opostos da lente, um em cada extremidade da linha que passa pelo centro desta e, geralmente $d > f$.

Ao gerar a visualização de uma cena tridimensional em computação gráfica, o desenvolvedor pode movimentar sua câmera em qualquer direção da cena. A imagem que será projetada para o observador é composta pela região que aparece na lente da câmera. Por isso, as partes da cena que aparecem na projeção final são determinadas pelo tipo e tamanho da lente da câmera. As principais API's gráficas 3D possuem funções e métodos que utilizam-se desta idéia, como o DirectX e o OpenGL [Hill, 2000]. A próxima seção aborda algumas operações que facilitam esta manipulação de câmeras virtuais.

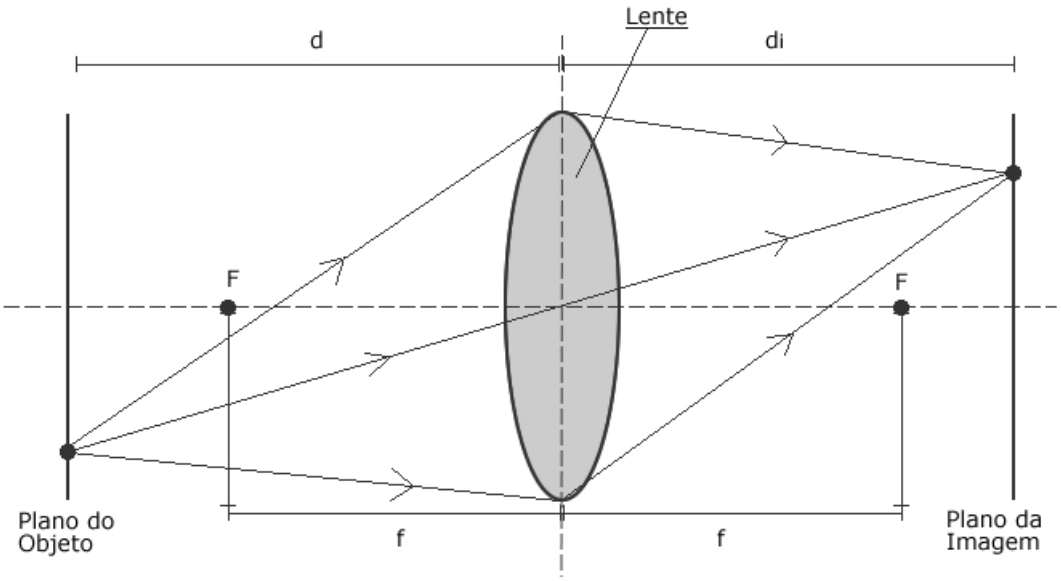

Figura 2.9: Esquema de uma lente convexa considerando a distância até os planos do objeto e da imagem.

O Pacote Gráfico OpenGL - Biblioteca de uso gratuito para modelagem, manipulação e exibição de objetos 3D, a OpenGL permite criar cenas computacionais com qualidade, incluindo recursos de animação, tratamento de imagens e texturas. Desenvolvida pela Silicon Graphics esta API (Interface de Programação de Aplicação) tornou-se independente de dispositivos de exibição. As funções deste pacote independem também de sistema operacional, ou seja, produzem o mesmo efeito em qualquer um. Esta API encapsula aproximadamente 250 funções e comandos através dos quais torna-se possível a construção de modelos e cenas mais complexos. Um detalhe importante é que a OpenGL não gerencia eventos de controle, como tratamento de eventos do mouse ou teclado. Quando este gerenciamento for necessário deve-se utilizar uma biblioteca auxiliar. A vantagem de utilizar esta API para a criação de cenas tridimensionais é a rapidez com que a cena é renderizada, isto porque a Silicon Graphics utiliza algoritmos cuidadosamente desenvolvidos e otimizados para executar as tarefas da OpenGL.

2.5 Manipulando a Câmera

Da mesma forma que o fotógrafo pode movimentar sua câmera em qualquer direção, o desenvolvedor (fotógrafo virtual) também movimenta sua câmera sintética (Figura 2.10). Isto é possível devido aos graus de liberdade desta. Três graus de liberdade em relação à posição na cena, três em relação à orientação e um referente ao zoom da câmera [Ware e Osborne, 1990].

Além dos parâmetros já citados, há algumas operações diretamente ligadas aos graus de

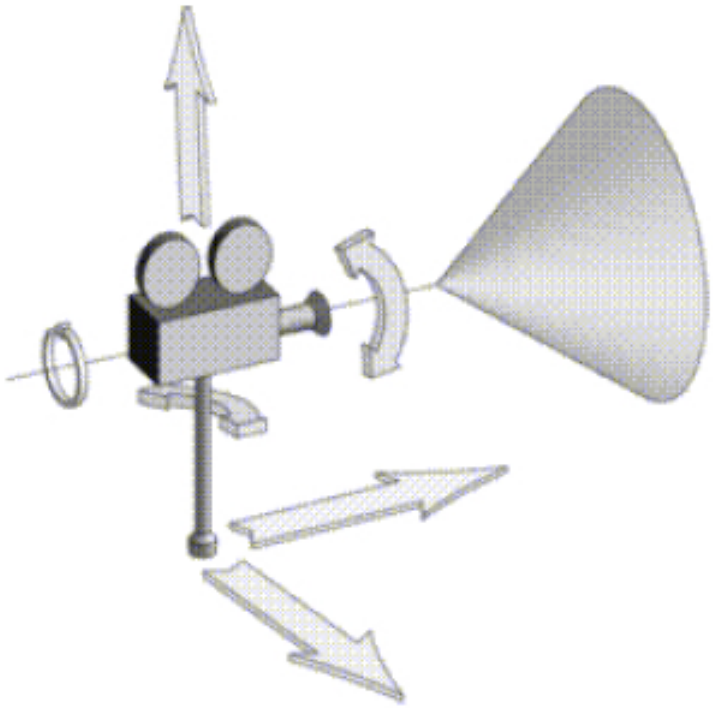

Figura 2.10: Os sete graus de liberdade na especificação da câmera virtual [Gomes e Velho, 1998]

liberdade da câmera que facilitam sua manipulação durante a exibição de uma cena computacionalmente:

Azimuth - O Azimute é a direção horizontal no sentido horário, em relação ao norte para algum ponto no ambiente. De origem da palavra as-sumut (árabe) significa a direção. Um azimute de 45º, por exemplo, significa 45º à direita do norte. Computacionalmente falando, ele considera o ponto focal como centro da cena e rotaciona a câmera ao redor do vetor *view up*.

Elevation - efetua o produto vetorial dos vetores *view up* e de projeção e rotaciona a câmera em torno do vetor resultante desta multiplicação, considerando o ponto focal como centro.

Roll - também conhecida como *twist* esta operação rotaciona o vetor *view up* em torno do vetor normal ao plano de projeção.

Yaw - nesta operação o centro de rotação deixa de ser o ponto focal e passa a localizar-se na posição da câmera e o ponto focal é rotacionado em torno do vetor *view up*.

Pitch - semelhante à operação *elevation*. Porém, o centro de rotação localiza-se na posição da câmera e o ponto focal é rotacionado ao redor do vetor resultante do produto vetorial entre o *view up* e o vetor direção da projeção.

Dolly - permite a movimentação da câmera na direção do vetor de projeção, aproximando-a ou afastando-a do ponto focal.

Zoom - permite o aumento ou a diminuição de uma determinada parte da cena. Para conseguir este efeito, o ângulo de visão da câmera deve ser alterado de modo a localizar esta área na região que é potencialmente visível.

2.6 Conclusão

A visualização de cenas criadas virtualmente só é possível por causa do sistema de câmeras implantadas na cena. Algumas aplicações não exigem a modelagem de câmeras precisas, mas para muitas situações a precisão deste sistema é fundamental [Kolb et al., 1995]. A simulação de câmera é uma das responsáveis pela grande evolução da visualização de sólidos dentro da computação gráfica, pois possibilitou o desenvolvimento de sistemas que simulem a movimentação do sistema de visão humano [Ware e Osborne, 1990]. Os chamados *cyberspaces* que são locais onde o usuário interage diretamente com o ambiente virtual são exemplos desta aplicação. [Sutherland, 1998] apresenta um dos trabalhos pioneiros nesta área, onde são usados capacetes com sensores para capturar os movimentos da cabeça do usuário e movimentar a câmera no ambiente, permitindo que o observador navegue e visualize todo o cenário. As pesquisas mais recentes utilizam-se de outros sensores para determinar as coordenadas das câmeras, como por exemplo, dados obtidos dos sensores de luvas.

Os jogos de computadores também utilizam-se de recursos de câmeras para permitir ao jogador visualizar o cenário. Tudo que é mostrado na tela do computador é capturado pela câmera virtual. Às vezes esta é fixa, como em um jogo de xadrez, onde há apenas um ângulo de visão. No entanto, uma grande quantidade dos jogos computacionais *(e também das aplicações de computação gráfica)* necessitam que seu sistema de câmeras seja móvel, permitindo o deslocamento do observador dentro do cenário tridimensional.

O aumento do grau de realismo tem se tornado uma tendência nas pesquisas de computação gráfica, e para conseguir este efeito deve-se ter modelos de câmeras com alta precisão. As aplicações de efeitos especiais e realidade aumentada são exemplos de situações que exigem a modelagem de um sistema de câmeras preciso, uma vez que os dados da cena sintética devem ter as mesmas características do mundo real, ou seja, as câmeras virtuais devem ter as mesmas propriedades das câmeras reais.

Capítulo 3

Projeção

3.1 Introdução

Projeção é o ato de representar uma figura geométrica obtida pela incidência sobre um plano de perpendiculares tiradas dos extremos do objeto que se quer representar [Bueno, 1998]. Trazendo para o meio computacional tal termo ganha significado mais amplo. Neste projetar é transformar os pontos de um sistema de coordenadas de dimensão *n* para pontos em um sistema de dimensão menor que *n* (geralmente *n-1*) [Foley et al., 1994]. Seu uso se dá com mais frequência na visualização de imagens tridimensionais em meios bidimensionais como fotografias, imagens médicas, filmagens, etc. A visualização de imagens na forma tridimensional ainda é muito limitada, sendo possível apenas por meio de holografia, por isso a projeção é extremamente importante [Novins et al., 1990]. É ela que torna possível a exibição de imagens capturadas por câmeras e/ou modeladas computacionalmente. Para isso, cada ponto do objeto é redesenhado com uma coordenada a menos que o seu correspondente original. Será considerado para estudo a projeção de corpos 3D em 2D.

3.1.1 Para que serve

A projeção é feita traçando-se linhas imaginárias, chamadas de projetoras que passam por cada ponto do objeto e incidem no centro de projeção, um ponto que fica a uma distância *d* do plano de projeção [Wang et al., 2007]. Onde estas linhas interceptam o plano de projeção os ponto devem ser desenhados (Figura 3.1). Após efetuar esta operação para todos os pontos, obtém-se a imagem projetada. Os principais tipos de projeção são apresentados na próxima seção.

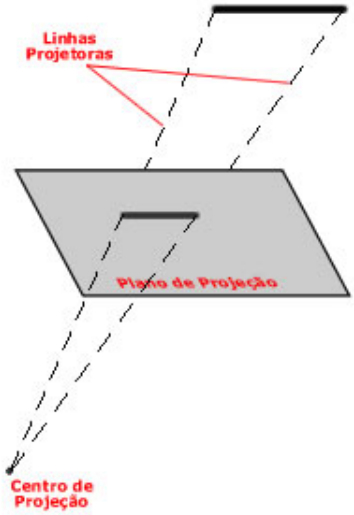

Figura 3.1: Projeção de uma linha.

3.1.2 Tipos de Projeções

As projeções geométricas dividem-se em dois tipos: (a) projeção paralela e (b) projeção em perspectiva. Na paralela (a) as linhas projetoras são paralelas entre si. De acordo com a geometria projetiva, todo par de reta possui pelo menos um ponto em comum, inclusive as paralelas, onde o ponto de encontro está localizado no infinito. Por isso, nesta abordagem, o centro de projeção é colocado no infinito. Neste caso, o objeto projetado mantém o seu tamanho independente da distância que ele se encontra do observador. Na projeção em perspectiva (b) o centro de projeção localiza-se a uma distância d finita. Desta forma é possível obter visualizações mais realistas, uma vez que a figura projetada depende diretamente da distância em que o observador se encontra. Em termos de programação e matemática, o tamanho do objeto é inversamente proporcional ao valor de d.

Como pode ser observado na Figura 3.2, existem várias abordagens para cada uma das projeções geométricas. Tais divisões ocorrem porque em um sistema tridimensional existem várias possibilidades (ou ângulos) de visualização diferentes, exigindo a aplicação de conceitos matemáticos diferentes. Tais abordagens serão estudadas mais profundamente nas seções subseqüentes.

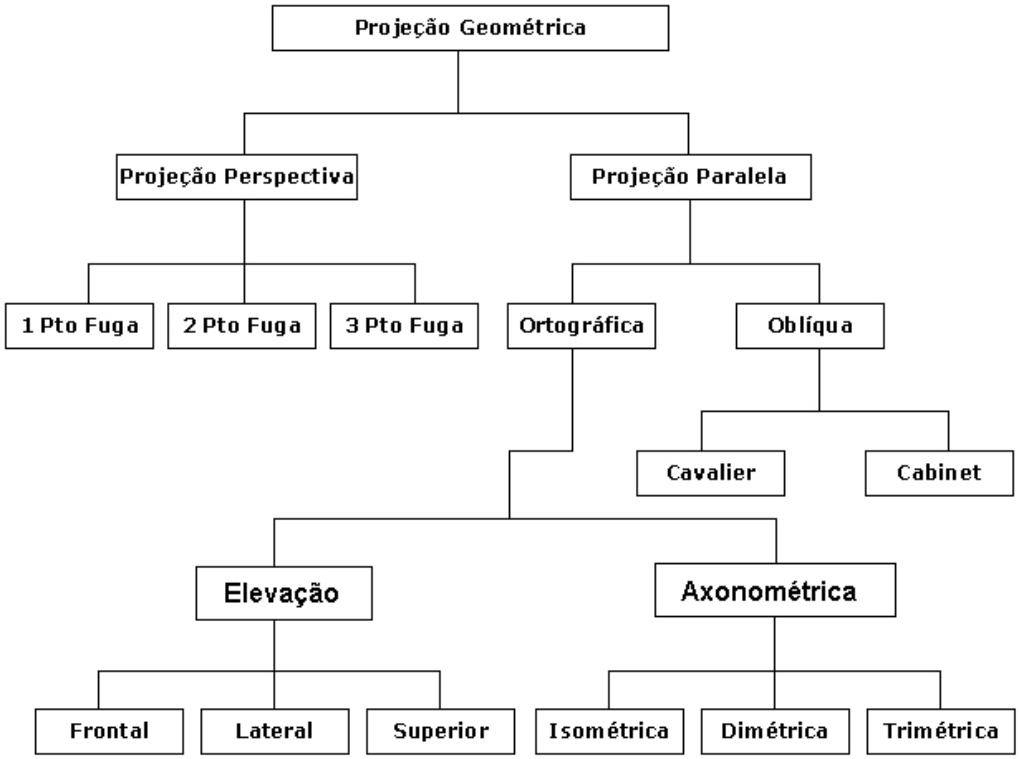

Figura 3.2: Tipos de projeções.

3.2 Projeção Paralela

A projeção paralela é a mais simples de implementar, nela o centro de projeção localiza-se no infinito fazendo com que as linhas projetoras fiquem paralelas entre si. É aplicada em várias áreas, como arquitetura e engenharia por possibilitar o cálculo das medidas proporcionalmente reais dos objetos. Porém, esta não consegue gerar visualizações realistas, uma vez que as medidas das faces não dependem da distância que o observador encontra-se do objeto, como mostrado na Figura 3.3.

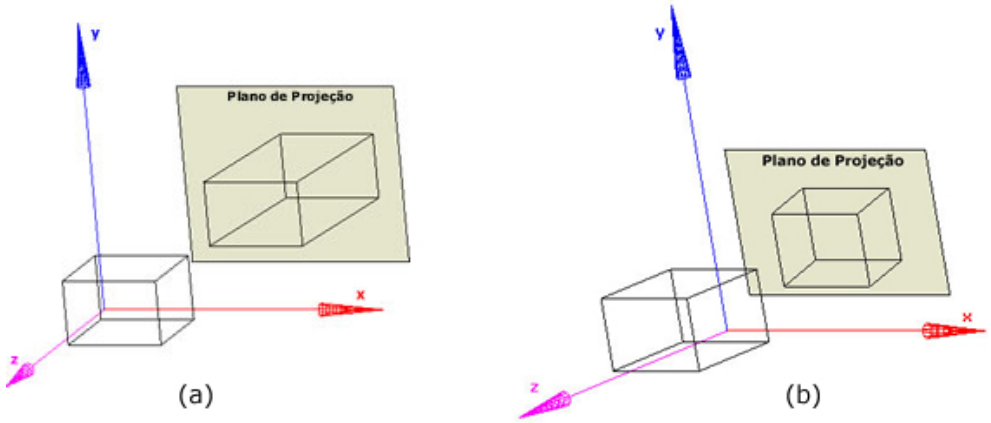

Figura 3.3: Projeção paralela ortográfica cavalier (a) e cabinet (b) de um cubo.

Existem dois tipos de projeção paralela: ortográfica e oblíqua. Esta diferenciação ocorre devido ao ângulo formado entre o plano de projeção e as linhas projetoras. Na projeção ortográfica este ângulo é de 90º. É comum pensar que esta abordagem da projeção paralela exibe apenas uma face do objeto de cada vez. Porém, isto não é um fato verídico, uma vez que é possível visualizar mais de uma face, que é feito alterando o alinhamento do plano de projeção (projeção ortográfica axonométrica).

A projeção oblíqua é semelhante à ortográfica, com a diferença de que na oblíqua as linhas projetoras e o plano de projeção não são perpendiculares, em alguns pacotes gráficos ela é feita utilizando-se dois ângulos de rotação, um em cada direção. Ela divide-se em *cavalier* e *cabinet*. Na projeção *cavalier* as linhas projetoras formam um ângulo de 45º com o plano de projeção, preservando a medida original das faces não paralelas este plano. A *cabinet* tem o objetivo de reduzir o tamanho do objeto original pela metade, deixando apenas a face paralela ao plano de projeção sem distorção. Para isto, o ângulo formado com o plano de projeção é tal que a tangente seja igual a 2, o que leva a um ângulo de aproximadamente 63,4º [Azevedo e Conci, 1986]. A Figura 3.3 mostra a projeção ortográfica cavalier de um cubo cujas dimensões são:

- altura = 2.5 unidades de medida;
- largura = 3.0 unidades de medida;
- comprimento = 3.0 unidades de medida;
- $\alpha = 30^{o}$ e $\phi = 45^{o}$.

O cubo desenhado no plano de projeção da Figura 3.3 é resultado da execução do Algoritmo 3.1 utilizando-se os dados acima. Apenas para permitir a visualização do cubo original (que está em 3 dimensões) foram efetuadas duas rotações no sistema de coordenadas. Uma de 20º em torno do eixo *Y* e outra de 15º no eixo *X*. No entanto, estas operações não interferem na projeção do cubo.

3.2.1 Projeção Paralela Ortográfica

A projeção ortográfica mantém as linhas de projeção dispostas de forma que elas sejam perpendiculares ao plano de projeção. Seu uso é mais freqüente em arquitetura e engenharia para visualizações frontal, lateral e superior dos objetos. Nestas abordagens só é possível visualizar um lado de cada vez, isso é feito colocando-se o plano de projeção perpendicular ao eixo principal, que é o eixo que possui a mesma direção das linhas projetoras. No entanto, existem três outras abordagens menos usadas onde é possível a exibição de mais de uma face de uma só vez, que são as projeções ortográficas axonométricas (isométrica, dimétrica e trimétrica). Nestas, o plano de projeção é realinhado permitindo a visualização de até três faces em uma projeção [Hearn e Baker, 1986].

Algoritmo 3.1 Projeções Paralelas Oblíquas

Entrada: Um vetor V de pontos tridimensionais, o ângulo α e o tipo de projeção ortográfica p.

Saída: A projeção oblíqua do objeto.

1: **se** $(p = \text{“}cavalier\text{”})$ **então**
2: $\quad \phi \leftarrow 45^{\text{o}}$
3: **senão**
4: $\quad$ **se** $(p = \text{“}cabinet\text{”})$ **então**
5: $\quad\quad \phi \leftarrow 63,4^{\text{o}}$
6: $\quad$ **fim se**
7: **fim se**
8: $L_1 \leftarrow \frac{1}{\tan \phi}$
9: **para** $i \leftarrow 1$ ate $|V|$ **faça**
10: $\quad x_p \leftarrow$ V[i].x $\times$ (V[i].z $\times L_1 \times \cos \alpha$)
11: $\quad y_p \leftarrow$ V[i].y $\times$ (V[i].z $\times L_1 \times \cos \alpha$)
12: **fim para**
13: desenhaPonto(x_p, y_p)

3.2.2 Projeções Ortográficas de Visão Planar

Nesta abordagem as faces dos objetos preservam suas medidas originais, por isso seu uso em atividades onde deseja-se calcular as medidas de faces de objetos através de suas projeções ser tão intenso. Sua implementação é extremante simples, como pode ser verificado no Algoritmo 3.2. Se o eixo *Z* é tomado como principal a projeção é dita frontal, se for o *X* a projeção é lateral e se for o *Y* a projeção é chamada superior, como mostrado na Figura 3.4.

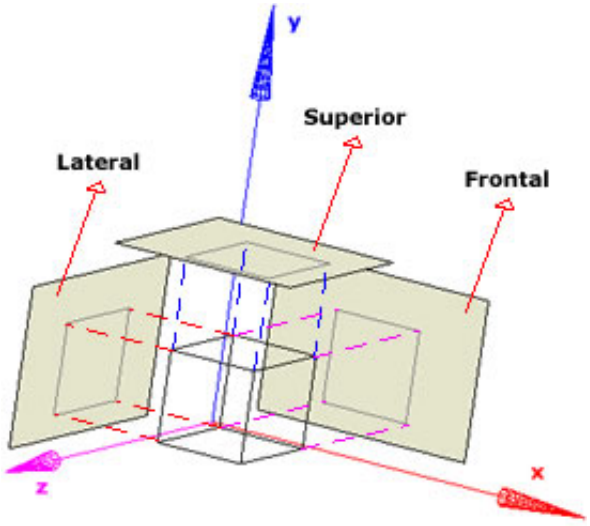

Figura 3.4: Projeções ortográficas frontal, lateral e superior.

Observe, que neste caso, as linhas projetoras (linhas pontilhadas) estão paralelas aos seus

respectivos eixos principais e que a dimensão da parte visível do cubo permanece inalterada.

Algoritmo 3.2 Projeções ortográficas frontal, lateral e superior

Entrada: Um vetor V com os pontos do objeto e o tipo de projeção p.

Saída: As projeções ortográficas frontal, lateral e superior do objeto.

```
1:  para i ← 1 ate |V| faça
2:      x_p ← V[i].x
3:      y_p ← V[i].y
4:      z_p ← V[i].z
5:      se (p = "frontal") então
6:          desenhaPonto(x_i, y_i)
7:      senão
8:          se (p = "lateral") então
9:              desenhaPonto(y_i, z_i)
10:         senão
11:             desenhaPonto(x_i, z_i)
12:         fim se
13:     fim se
14: fim para
```

Outra característica, às vezes um ponto negativo, é que no caso de se observar apenas uma das faces, não é possível se conceber corretamente a forma do objeto. Para solucionar tal problema é comum exibir além das vistas frontal, lateral e superior, uma projeção axonométrica que permite uma visão mais integrada do objeto [Battaiola e Erthal, 1998].

3.2.3 Projeção Axonométrica Isométrica

Ao contrário das abordagens anteriores a projeção axonométrica não possui a direção de projeção e a normal ao plano de projeção coincidentes com a direção de um dos eixos principais. Assim, corrige-se o problema de visualizar apenas um lado por vez e torna possível a observação de várias faces de um mesmo objeto. Na maioria das vezes esta projeção altera a relação dos ângulos e dimensões do objeto distorcendo-os [Hearn e Baker, 1986].

Para conseguir tal projeção são efetuadas rotações no objeto e escolhidos três fatores de redução, um para cada eixo do sistema de coordenadas. As projeções axonométricas são classificadas de acordo com estes fatores de redução. (a) Projeção axonométrica trimétrica onde todos os fatores de redução são diferentes, (b) projeção dimétrica que possui dois fatores de redução iguais e (c) isométrica onde todos os fatores são iguais.

Dentre as projeções axonométricas, a mais usada é a isométrica. Nela, os planos do objeto estão posicionados em relação ao plano de projeção, de forma que os três eixos do objeto

aparentem ter a mesma mudança em suas medidas. A Figura 3.5 deixa essa característica bem clara. Observe que, tanto os objetos tridimensionais (o cubo original - visto de frente - e o cubo rotacionado para projeção isométrica) quanto suas projeções mantêm a mesma proporção entre as medidas. Para conseguir obter um fator de redução igual para os três eixos do sistema da Figura 3.5 foram usados os seguintes dados calculados em [Azevedo e Conci, 1986]:

- Ângulo de rotação em torno do eixo *Y* $\alpha \cong 35.26^{o}$;
- Ângulo de rotação em torno do eixo *X* $\delta \cong 45^{o}$;

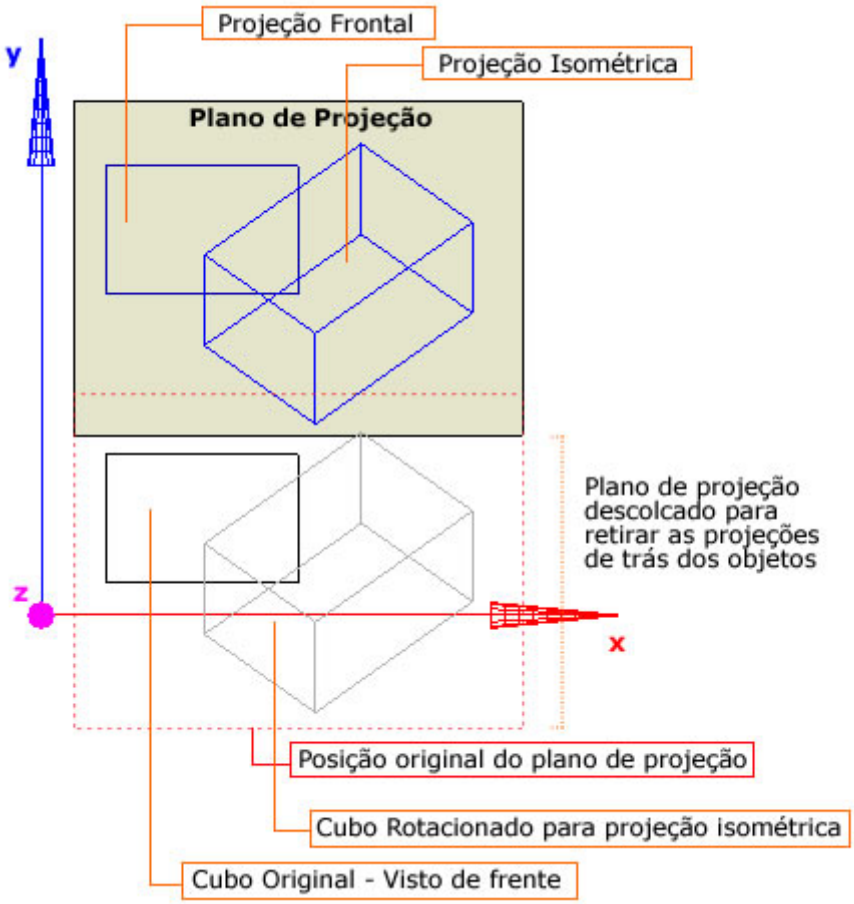

Figura 3.5: Projeção ortográfica frontal e axonométrica isométrica.

Após a descoberta dos ângulos de rotação calcula-se as novas coordenadas dos pontos no sistema cartesiano. Há várias formas de se chegar às equações que geram os novos pontos e uma delas aparece em [Battaiola e Erthal, 1998]. A seguir é apresentada uma forma mais simples de obtê-las.

- Considere um ponto qualquer *P(x, y, z)*. Para facilitar efetue apenas uma rotação de cada vez, ou seja, primeiro rotacione em torno do eixo *Y* e depois em torno de *X*, ou vice-versa. Será considerado a primeira opção;

- Considere α o ângulo de rotação em *Y* e δ em *X*. Rotacinando apenas em *Y* tem-se que:
  $Y^{'}$ = *Y*;
  $X^{'}$ = *X* . $\cos\alpha$ + *Z* . $\sin\alpha$
  $Z^{'}$ = *Z* . $\sin\alpha$ - *X* . $\cos\alpha$

- Rotacionando novamente, mas agora em torno de *X* tem-se:
  $X^{''}$ = $X^{'}$
  $Y^{''}$ = $Y^{'}.\cos\delta - Z^{'}.\sin\delta$
  $Z^{''}$ = $Z^{'}.\sin\delta + Y^{'}.\cos\delta$

- Substituindo:

  1. $X^{''} = X.\cos\alpha + Z.\sin\alpha$
  2. $Y^{''}$ = $Y^{'}.\cos\delta - Z^{'}.\sin\delta$
     $Y^{''} = Y.\cos\delta - (Z.\sin\alpha - X.\cos\alpha).\sin\delta$
     $Y^{''} = Y.\cos\delta - Z.\sin\alpha.\sin\delta + X.\cos\alpha.\sin\delta$
     $Y^{''} = X.\cos\alpha.\sin\delta + Y.\cos\delta - Z.\sin\alpha.\sin\delta$
  3. $Z^{''} = Z^{'}.\sin\delta + Y^{'}.\cos\delta$
     $Z^{''} = (Z.\sin\alpha - X.\cos\alpha).\sin\delta + Y.\cos\delta$
     $Z^{''} = Z.\sin\alpha.\sin\delta - X.\cos\alpha.\sin\delta + Y.\cos\delta$
     $Z^{''} = -X.\cos\alpha.\sin\delta + Y.\cos\delta + Z.\sin\alpha.\sin\delta$

- Logo conclui-se que:
  $P(X^{''}, Y^{''}, Z^{''})$ é o ponto *P(x, y, z)* após a execução das operações.

Como observa-se no Algoritmo 3.3, a implementação de tal projeção também é muito simples. Basta efetuar as operações de rotações de forma adequada e aplicar a projeção frontal tomando o eixo *Z* como eixo principal.

Algoritmo 3.3 Projeção ortográfica axonométrica isométrica

Entrada: Um vetor *V* com os pontos do objeto.

Saída: A projeção axonométrica isométrica do objeto.

1: $V_{AUX}[]$ //vetor auxiliar que conterá os pontos rotacionados
2: **para** $i \leftarrow 1$ ate $|V|$ **faça**
3: $x_p \leftarrow V[i].x \times \cos\alpha + V[i].z \times \sin\alpha$
4: $y_p \leftarrow x_p \times \cos\alpha \times \sin\delta + V[i].y \times \cos\delta - V[i].z \times \sin\alpha \times \sin\delta$
5: $z_p \leftarrow -x_p \times \cos\alpha \times \sin\delta + y_p \times \cos\delta + V[i].z \times \sin\alpha \times \sin\delta$
6: $V_{AUX}[i] \leftarrow (x_p, y_p, z_p)$
7: **fim para**
8: projecaoOrtografica(V_{AUX}, "frontal")

3.2.4 Projeção Axonométrica Dimétrica

Assim como na projeção isométrica, a dimétrica também é obtida através de rotações seguidas em torno de *X* e *Y* a partir de um centro de projeção localizado no infinito (característica fundamental da projeção paralela), com a diferença que em vez de ter os três fatores de redução iguais (como na isométrica) esta possui apenas dois desses fatores equivalentes. Assim, pelo menos dois dos eixos são proporcionais em relação às medidas originais.

3.2.5 Projeção Axonométrica Trimétrica

A projeção axonométrica trimétrica é a menos restritiva de todas, pois são efetuadas rotações em ordem arbitrária em torno dos eixos do sistema de coordenadas, seguidas pela projeção frontal, normalmente tomando-se o eixo *Z* como principal. Nesta abordagem, cada eixo contém um fator de redução diferente, gerando uma distorção para cada face.

3.2.6 Projeção Paralela Oblíqua

A projeção oblíqua se difere da ortográfica pelo fato de que na primeira as linhas projetoras não são perpendiculares ao plano de projeção. Na oblíqua apenas as faces paralelas ao plano de projeção são mostradas em seu tamanho e formas originais, as que não o são ficam distorcidas (o objeto é ilustrado na forma 3D, como nas projeções axonométricas). A projeção destas faces é equivalente a uma vista ortogonal frontal. Assim, existem duas classificações para esta abordagem. Se o ângulo formado entre as projetoras e o plano de projeção for igual a 45º a projeção é dita *cavalier* caso o ângulo seja de aproximandamente 63.4º é chamada de *cabinet* [Hearn e Baker, 1986], nesta última as arestas perpendiculares ao plano de projeção são reduzidas à metade.

A Figura 3.3-(a) e o Algoritmo 3.1 mostram o funcionamento da projeção cavalier. A Figura 3.3-(b) apresenta a projeção cabinet. O algoritmo para as duas projeções é o mesmo, com a diferença que na cavalier $\phi = 45^{o}$ e na cabinet $\phi \cong 63.4^{o}$.

3.3 Projeção Perspectiva

A projeção perspectiva é aplicada em cenas que necessitam manter um alto grau de realismo [Ogata et al., 1998], não sendo aplicável para observação fiel das métricas do espaço ou dos objetos de uma cena. Isto porque ela altera as dimensões dos componentes da cena à medida que estes ficam mais longe do observador, da mesma forma que acontece com o sistema de visão humano. Para obter este efeito aplica-se uma diminuição constante e graduada dos objetos e espaços da cena, dando a idéia de afastamento progressivo para o fundo, resultando na criação da noção de profundidade do espaço. Esta diminuição é conhecida como encurtamento

perspectivo. Nesta abordagem, as linhas projetoras não possuem a propriedade de paralelismo, pois a distância entre o centro de projeção e o plano de projeção é finita. Porém, deve-se ficar atento ao posicionamento do ponto de referência da projeção (ou centro de projeção), pois se este e o objeto estiverem do mesmo lado em relação ao plano de projeção a imagem projetada é invertida. A matriz de projeção perspectiva é dada pela Matriz 3.1.

$$M_{per} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{bmatrix} \tag{3.1}$$

Ao contrário do que muitas pessoas sem conhecimento do assunto pensam, a projeção perspectiva gera cenas com alto grau de realismo por não manter fidelidade na reprodução dos traços originais. Permanecem intactas apenas as faces paralelas ao plano de projeção, nas demais os ângulos entre as arestas são alterados, as linhas paralelas não são projetadas paralelamente e as distâncias não são preservadas (Figura 3.6) [Ogata et al., 1998].

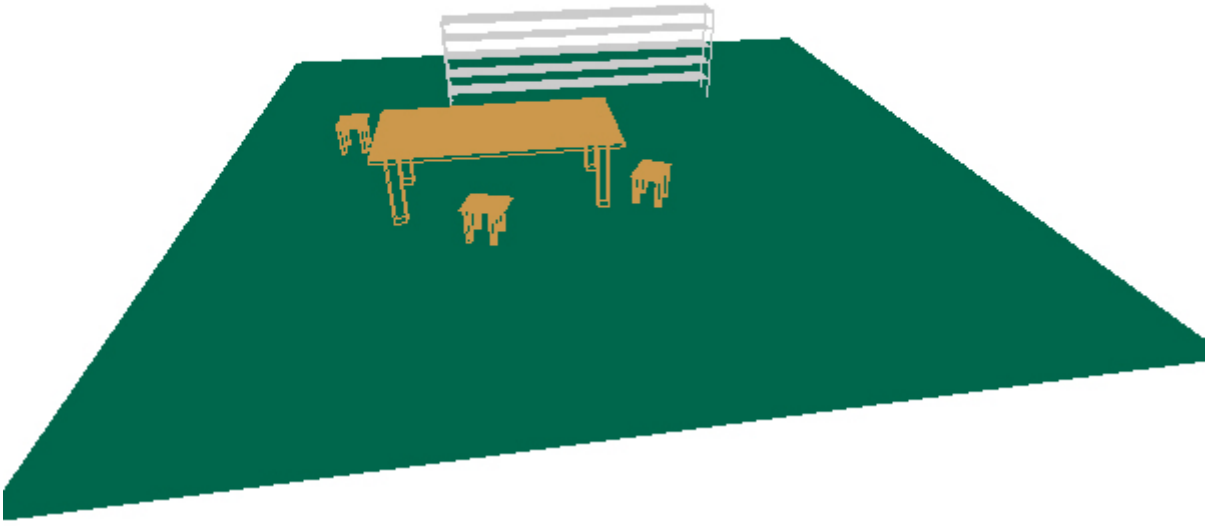

Figura 3.6: Projeção perspectiva de uma cena.

3.3.1 Abordagem Matemática para Projeção Perspectiva

A modelagem matemática da projeção perspectiva é baseada na semelhança de triângulos. Considera-se um plano *S*, paralelo ao plano *XY*, ao qual denomina-se plano de projeção e toma-se, sobre o eixo *Z*, um ponto de coordenadas *(0, 0, -d)* denominado centro de projeção, onde *d* é a distância deste ao plano de projeção. (x_p, y_p, z_p) são as coordenadas do ponto *P* a ser projetado e $(x_{p'}, y_{p'})$ as coordenadas do ponto P' projetado.

As coordenadas do ponto projetado P' são obtidas criando-se uma linha de *P* até o centro projeção e calculando-se a interseção desta com o plano de projeção. A Figura 3.7 exemplifica este cálculo. Olhando o plano de projeção pelo lado direito é possível calcular a coordenada *y* do ponto projetado (Equação 3.2) e observando por cima encontra-se a coordenada *x* (Equação 3.3).

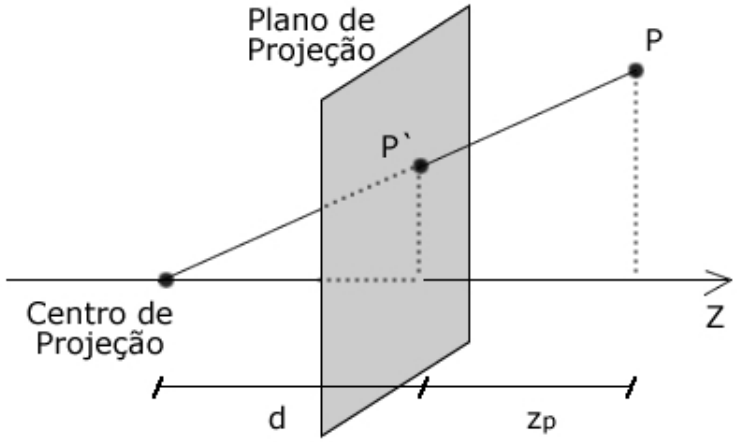

Figura 3.7: Representação da projeção perspectiva de um ponto.

$$\frac{d}{y_{p'}} = \frac{d + z_p}{y_p} \Rightarrow y_{p'} = \frac{y_p}{(\frac{z_p}{d}) + 1} \tag{3.2}$$

$$\frac{d}{x_{p'}} = \frac{d + z_p}{x_p} \Rightarrow x_{p'} = \frac{x_p}{(\frac{z_p}{d}) + 1} \tag{3.3}$$

3.3.2 Pontos de fuga

A projeção perspectiva é caracterizada pelo encurtamento perspectivo e pelos pontos de fuga. Quando o plano de projeção intercepta um eixo principal têm-se a ilusão de que as linhas projetoras paralelas a este eixo convergem para um ponto em comum. Este é chamado de ponto de fuga principal, localiza-se na interseção do eixo com o plano de projeção e é o responsável pela classificação das projeções perspectiva.

- 1 Ponto de Fuga: apenas um eixo principal é interceptado pelo plano de projeção;
- 2 Pontos de Fuga: dois eixos principais são interceptados pelo plano de projeção, é muito usada em desenho industrial e projetos industriais;
- 3 Pontos de Fuga: três eixos principais interceptados pelo plano de projeção, é bem menos utilizada por acrescentar pouco em termos de realismo comparativamente às projeções com 2 pontos de fuga e possuir custo de implementação muito superior.

3.4 Conclusão

Analisando assintoticamente os algoritmos de projeção estudados conclui-se que são algoritmos com tempo de execução linear. Por pertencerem à mesma classe o tempo de execução destes é muito parecido. A Figura 3.8 apresenta o gráfico do resultado da execução dos algoritmos de projeção perspectiva e de projeção paralela oblíqua por um computador pessoal com processador 1.66 GHz e 256 Mb de memória RAM.

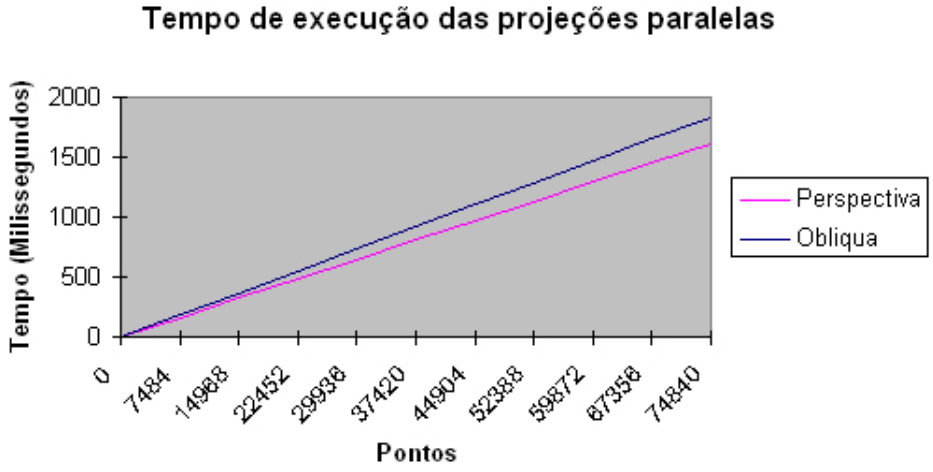

Figura 3.8: Gráfico de execução dos algoritmos de projeção perspectiva e paralela oblíqua.

Algoritmo 3.4 Projeção Perspectiva

Entrada: Um vetor V de pontos tridimensionais e a distância do centro de projeção ao plano de projeção d.

Saída: A projeção Perspectiva do Objeto.

1: **para** $i \leftarrow 1$ ate $|V|$ **faça**
2: $x_p \dfrac{V[i].x}{\dfrac{V[i].z}{d} + 1}$
3: $y_p \dfrac{V[i].y}{\dfrac{V[i].z}{d} + 1}$
4: **fim para**
5: desenhaPonto(x_p, y_p)

Os resultados mostram que a projeção perspectiva (Algoritmo 3.4) possui tempo de execução menor do a projeção paralela (Algoritmo 3.1). Isto deve-se ao fato de que a perspectiva possui menos operações matemáticas do que a paralela, e quanto mais pontos possuir uma cena maior será a diferença de execução das projeções, mesmo a perspectiva gerando cenas mais reais. Esta diferença gera ganhos no pipeline de visualização gráfica quando este é aplicado a animações e a simulação de ambientes reais, uma vez que estes tentam imitar o sistema de visão humano (projeção perspectiva).

Capítulo 4

Clipping

4.1 Introdução

O *clipping* é aplicado em cenas para retirar linhas e superfícies que localizam-se fora da área de visualização [Hearn e Baker, 1986]. Esta técnica tem por objetivo acelerar a representação da cena e divide-se em duas abordagens. A mais tradicional efetua o recorte das partes não vistas antes de fazer a conversão matricial. Para isto, deve-se encontrar geometricamente os vértices que interceptam o contorno do plano de recorte e limitam as superfícies *(geralmente polígonos regulares)* e/ou linhas vistas pelo observador. Como consequência, o processo de conversão matricial torna-se mais rápido, uma vez que deve-se preocupar apenas com a área visível da cena, que às vezes pode ser muito menor que a porção original.

Outra abordagem é recortar a imagem depois de convertê-la para a forma matricial, ou seja, depois de rasterizá-la. Nesta, a imagem da cena é rasterizada e então apenas os pixels visíveis são plotados no plano de visualização. Checa-se as coordenadas de cada pixel, comparando-as com o contorno do plano de *clipping*. Esta abordagem tem a desvantagem de exigir a conversão de todos os *pixels* da cena, até os que não são vistos pelo observador. Porém, geralmente a comparação para verificar se os *pixels* são visíveis ou não é feita por hardwares especializados, deixando esta técnica mais rápida. No entanto, a principal vantagem desta sobre a anterior é que ela pode ser utilizada para *clipping* de qualquer objeto, independente da forma geométrica. O *clipping* também pode ser utilizado em conjunto com o sistema de câmera para ampliar uma porção específica da cena. O método responsável por esta seleção e ampliação é chamado de *windowing* [Harrington, 1987].

Nas duas abordagens o resultado depende da geometria a ser recortada. Os pontos retornam resultados *booleanos*, visível ou invisível. As retas devolvem como resultado um conjunto de segmentos de reta que podem ser visualizadas e a execução de algoritmos de *clipping* para recorte de planos devolve um conjunto de polígonos que podem ser desenhados.

4.1.1 Breve Descrição do Processo de Raster

Rasterizar um objeto é retirar as informações de cada pixel deste e armazenar na memória do dispositivo. Na maioria das vezes, sob a forma de matriz, onde cada posição desta representa um pixel da imagem. Em outras palavras, é o processo de conversão da representação vetorial para a matricial [Azevedo e Conci, 1986]. Os dados armazenados na memória do dispositivo servem de base para determinar quais *pixels* e com qual cor devem ser acesos.

Os polígonos são representados por seus vértices e sua rasterização ocorre através do cálculo da interseção entre as arestas que ligam estes vértices com as *scan-lines* (ou linhas de scaneamento), que são as linhas horizontais do sistema de vídeo. Desta forma, consegue-se preencher a matriz de pixels com a forma exata do polígono, obtendo-se o chamado preenchimento por *scan-line*.

4.2 Clipping de Pontos

Apesar de ser menos utilizado do que o *clipping* de linha ou de superfície, o corte de pontos também tem sua importância no processo de retirada de porções não visíveis de uma cena. Em cenas de explosões ou simulação de nuvens, onde utiliza-se de sistemas de partículas, com cada partícula sendo representada por um ponto, o *clipping* de pontos é muito utilizado. Considerando a janela de visualização formada por *(*$x_{w_{min}}$*,* $y_{w_{min}}$*)* e *(*$x_{w_{max}}$*,* $y_{w_{max}}$*)*, deve-se plotar o ponto *(x, y)* se as inequações 4.1 e 4.2 forem satisfeitas. Caso alguma destas não seja respeitada significa que o ponto localiza-se fora da área de visibilidade.

$$x_{w_{min}} \leqslant x \leqslant x_{w_{max}} \tag{4.1}$$

$$y_{w_{min}} \leqslant y \leqslant y_{w_{max}} \tag{4.2}$$

4.3 Clipping de Linhas

O *clipping* de linhas concentra-se na remoção de segmentos de reta que localizam-se fora da janela de visualização. Nesta abordagem, cada linha a ser mostrada é testada para saber se é interna, externa ou se intercepta os limites do polígono que representa a janela de visualização. Se localizar-se completamente dentro da área visível a linha é desenhada, se estiver fora nada é desenhado. No entanto, se ocorrer a interseção entre a janela e a linha [Adams, 1988], deve-se calcular as interseções e desenhar apenas o segmento de reta limitados por elas, como mostrado pela Figura 4.1.

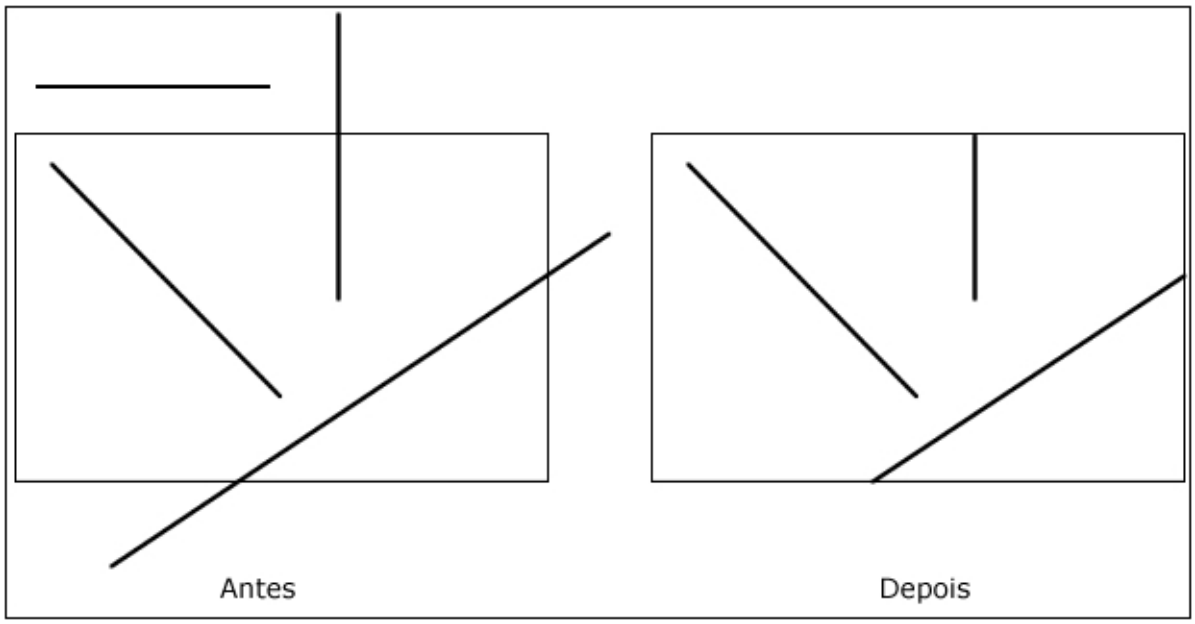

Figura 4.1: *Clipping* de linhas.

4.3.1 Os Principais Algoritmos

Cohen e Sutherland desenvolveram um método rápido para efetuar o recorte de linhas tomando uma janela de *clipping* retangular [Foley et al., 1994]. O algoritmo que recebeu o nome dos autores, tornou-se um dos mais populares e utiliza-se de cálculos matemáticos para obter os pontos finais visíveis da linha que se deseja traçar [Rogers, 1985].

Cohen e Sutherland [Foley et al., 1994] fazem uso de um vetor de inteiros, sendo cada elemento deste correspondente ao posicionamento dos extremos da linha em relação à janela de visualização (0 – visível e 1 – invisível). A primeira posição do vetor recebe valor 1 se o ponto está acima da janela de visualização, a segunda posição recebe 1 se o ponto está abaixo, a terceira e a quarta recebem valor 1 se o ponto localiza-se à direita e à esquerda da área de *clipping*, respectivamente. Estes dados são usados como códigos pelo Algoritmo 4.1 para determinar qual porção da linha deve ser desenhada (Figura 4.2). Desta forma, quando todas as posições

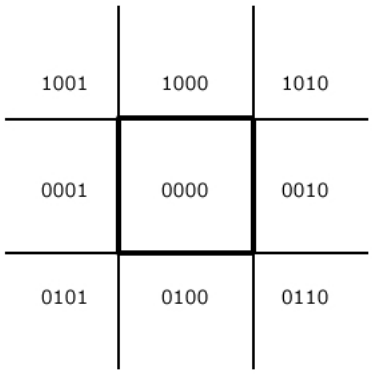

Figura 4.2: Códigos para o recorte de linhas de Cohen e Sutherland.

do vetor são iguais a 0 (zero) o ponto é interno à janela de *clipping*.

Os casos mais complexos ocorrem quando a linha a ser desenhada intercepta o contorno da janela de visualização. Neste caso, calcula-se os pontos de interseção do segmento de reta com a borda da área visível e utiliza-os como pontos finais da semi-reta ser mostrada.

Cohen e Sutherland utilizam-se da abordagem de codificação para determinar quais porções de retas estão visíveis e quais não estão. No entanto, alguns pesquisadores, como Liang e Barsky [Hearn e Baker, 1986], preferem utilizar das equações paramétricas para determinar os pontos finais das semi-retas visíveis. Esta abordagem baseia-se na análise da equação paramétrica da reta que passa pelos pontos que formam o extremos da linha a ser exibida.

O algoritmo Liang-Barsky calcula, para cada linha, os valores dos parâmetros que definem qual parte está dentro da janela de clipping. Enquanto o algoritmo Cohen-Sutherland executa subdivisões para encontrar as interseções Liang e Barskey preferem a técnica de refinamento. A técnica de refinamento consiste em representar a reta na forma paramétrica. Ela consegue ser mais eficiente que o algoritmo anterior porque evita os cálculos de interseções que não são necessárias.

Considerando $\Delta x = x_2 - x_1$ e $\Delta y = y_2 - y_1$, a semi-reta a ser desenhada deve satisfazer as condições apresentadas pelas Equações 4.3 e 4.4

$$x_{min} \leqslant x_1 + t \times \Delta x \leqslant x_{max} \tag{4.3}$$

$$y_{min} \leqslant y_1 + t \times \Delta y \leqslant y_{max} \tag{4.4}$$

4.4 Clipping de Superfícies

Várias aplicações de computação gráfica necessitam de suporte a recorte de superfícies (polígonos) [Greiner e Hormann, 1998]. Geralmente os vértices destes polígonos são armazenados em estruturas de dados (matrizes, listas, arquivos, etc) e há duas abordagens para determinar as porções visíveis: (a) rasterizar o polígono antes de recortá-lo e (b) considerar este formado por um conjunto de linhas, recortá-las e depois fazer o preenchimento apropriado.

4.4.1 Recorte de Polígonos Rasterizados

Esta técnica é muito utilizada pelos algoritmos de *clipping* de superfícies que são usados pelos dispositivos gráficos atuais. Os algoritmos que utilizam-se desta abordagem rasterizam cada polígono que compõe a cena, armazenando as informações de cada ponto em uma estrutura de dados, na maioria das vezes uma matriz bidimensional. Após fatiar (rasterizar) o polígono, cada matriz contendo as informações das superfícies são percorridas e cada fatia (posição da matriz)

Algoritmo 4.1 Algoritmo de Cohen e Sutherland [Hearn e Baker, 1986]

Entrada: Os pontos finais p_1 e p_2 da linha a ser verificada e a janela de *clipping*.

Saída: A semi-reta visível desenhada.

```
1:  enquanto Existir linhas a serem recortadas faça
2:    se p1 e p2 são trivialmente aceitos então
3:      Desenha a linha limitada por p1 e p2
4:      retorna(1)
5:    fim se
6:    se p1 e p2 são trivialmente rejeitados então
7:      retorna(0)
8:    fim se
9:    se p1 é externo à janela de clipping então
10:     se p1 está a esquerda da janela então
11:       Corta o segmento usando a aresta esquerda do viewport
12:     senão
13:       se p1 está a direita da janela então
14:         Corta o segmento usando a aresta direita do viewport
15:       senão
16:         se p1 está abaixo da janela então
17:           Corta o segmento usando a aresta da base do viewport
18:         senão
19:           se p1 está acima da janela então
20:             Corta o segmento usando a aresta superior do viewport
21:           fim se
22:         fim se
23:       fim se
24:     fim se
25:   senão
26:     Efetua os mesmos passos das linhas 9 a 24 para p2
27:   fim se
28: fim enquanto
```

é comparada para verificar se esta localiza-se dentro ou fora da área visível. As fatias internas e as que interceptam o contorno da área de *clipping* são exibidas e as demais descartadas.

Dependendo do tamanho e da quantidade de polígonos que formam a cena, o tempo de execução para os algoritmos de recorte de polígonos rasterizados é maior do que os algoritmos que não rasterizam os objetos. Esta diferença de tempo de execução deve-se não apenas à rasterização dos componentes da cena, mas também ao processo de comparação de cada pixel dos objetos com o contorno da janela de visualização. No entanto, a técnica de rasterizar as superfícies antes de recortá-las ainda é muito utilizada para eliminar as partes da cena que não estão visíveis ao observador, principalmente no processo de detecção e remoção de superfícies escondidas. Parte do pipeline de visualização gráfica que será discutido no Capítulo 5.

4.4.2 Recorte de Polígonos não Rasterizados

Ao contrário da linha de raciocínio anterior, os algoritmos de *clipping* de superfícies não rasterizadas procuram apenas as interseções dos polígonos com as arestas que formam a janela de visualização. Normalmente, esta técnica é mais rápida do que a anterior, porém é menos utilizada.

Às vezes, é difícil entender porque utilizar uma técnica mais lenta sabendo-se que há outras melhores para obter praticamente os mesmos resultados. Porém, no caso do *clipping* de superfícies a explicação baseia-se em três fatores: (a) o grau de dificuldade para desenvolver os algoritmos que recortam as superfícies sem precisar rasterizá-las, (b) a precisão das técnicas utilizadas e (c) o poder de processamento dos dispositivos gráficos atuais.

Os algoritmos que não fazem uso da rasterização são complexos e demandam muito tempo e esforço computacional para serem desenvolvidos. Principalmente, quando trabalham com polígonos complexos, constituídos por muitas arestas. A Figura 4.3 apresenta um polígono com estas características.

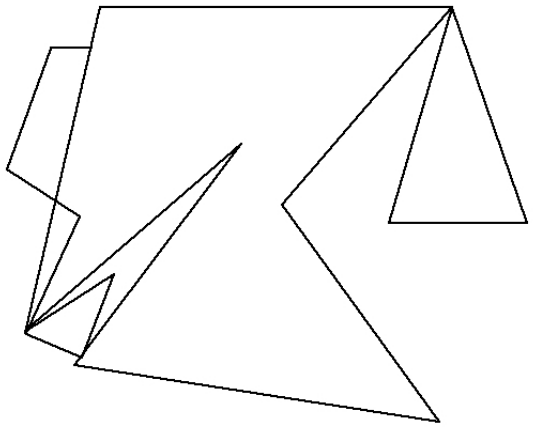

Figura 4.3: Polígono de difícil recorte.

Para polígonos como este, o uso da rasterização é comum pois facilita a implementação e aumenta a precisão do recorte, uma vez que torna possível a determinação exata da posição de cada ponto em relação à janela de *clipping*. Por fim, os dispositivos gráficos atuais têm uma capacidade de processamento muito grande, tornando a operação de *raster* assintoticamente desprezível e determinando a visibilidade dos pixels do polígono de forma ágil.

4.4.3 Os Principais Algoritmos

Esta seção apresenta alguns dos algoritmos de recorte de polígonos mais conhecidos. Todos os algoritmos estudados não encapsulam o processo de *raster*. A proposta dos algoritmos de *clipping* é retornar um conjunto de vértices que formam o contorno do polígono *clipado*. Para isto deve-se desconectar as arestas da superfície de entrada, aplicar os algoritmos estudados na seção anterior e utilizar alguma técnica de preenchimento para solidificar a região recortada.

Sutherland (considerado por muitos como um dos pais da computação gráfica) também desenvolveu um algoritmo para *clipping* de polígonos. Porém, desta vez seu parceiro foi Hodgman [Sutherland e Hodgman, 1974]. Os dois desenvolveram um algoritmo que divide-se em quatro passos. Em cada passo efetua-se o recorte de um lado da janela de visualização, de modo que ao terminar de executar estes passos, apenas as fatias visíveis dos polígonos são exibidas. Para isto, alguns testes devem ser feitos considerando os vértices dos polígonos e o contorno da janela de recorte [Hearn e Baker, 1986].

1º teste - se o primeiro vértice está fora e o segundo dentro da janela de visualização, tanto o segundo vértice quanto a interseção da aresta com o contorno da área visível são inseridos no conjunto de vértices que pertencem à solução;

2º teste - se os dois pontos limitantes da aresta são interno à janela, apenas o segundo vértice é inserido no conjunto solução;

3º teste - se o primeiro vértice é interno e o segundo externo ao *viewport*, insere-se apenas a interseção da aresta com o contorno deste.

4º teste - por fim, se os dois pontos que formam a aresta localizam-se fora da área de visibilidade nenhum dos vértices fazem parte da solução.

Observe que este algoritmo trabalha com arestas sequenciais, por isso não há a necessidade de armazenar todos os pontos válidos em todas as iterações, o que evita a repetição desnecessária de vértices, aumentando sua eficiência e velocidade.

Considere a Figura 4.4 para melhor entendimento do funcionamento do algoritmo de Sutherland e Hodgman.

Observando-a, conclui-se facilmente que os pontos pertencentes à solução são
$s = \{V_1', V_2, V_3, V_3'\}$. Para chegar a este conjunto de vértices utilizando os testes de Hodgman

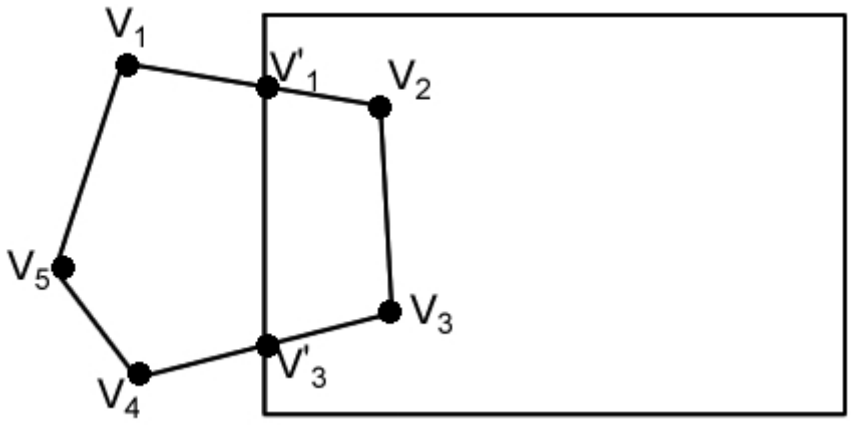

Figura 4.4: Exemplo de execução do Algoritmo Sutherland Hodgeman.

e Sutherland deve-se considerar um ponto de partida; neste caso, o ponto V_1. A aresta $\overrightarrow{V_1V_2}$ encaixa-se no 1º teste. Logo, armazena-se $V_1^{'}$ e V_2 no conjunto solução que passa a ser $s = \{V_1^{'}, V_2\}$. Analisando a aresta subsequente $\overrightarrow{V_2V_3}$ verifica-se a aplicação do 2º teste, adicionando apenas o segundo vértice à solução, uma vez que o primeiro já faz parte da mesma. Logo, tem-se que $s = \{V_1^{'}, V_2, V_3\}$. Continuando, a semi-reta $\overrightarrow{V_3V_4}$ deve ser resolvida pelo 3º teste, inserindo apenas a interseção ao conjunto solução, que passa a ser $s = \{V_1^{'}, V_2, V_3, V_3^{'}\}$. As demais sequências ($\overrightarrow{V_4V_5}$ e $\overrightarrow{V_5V_1}$) encaixam-se no 4º teste, não acrescentando nada à solução. A Figura 4.5 mostra como Sutherland e Hodgman tratam o problema de recorte de polígonos.

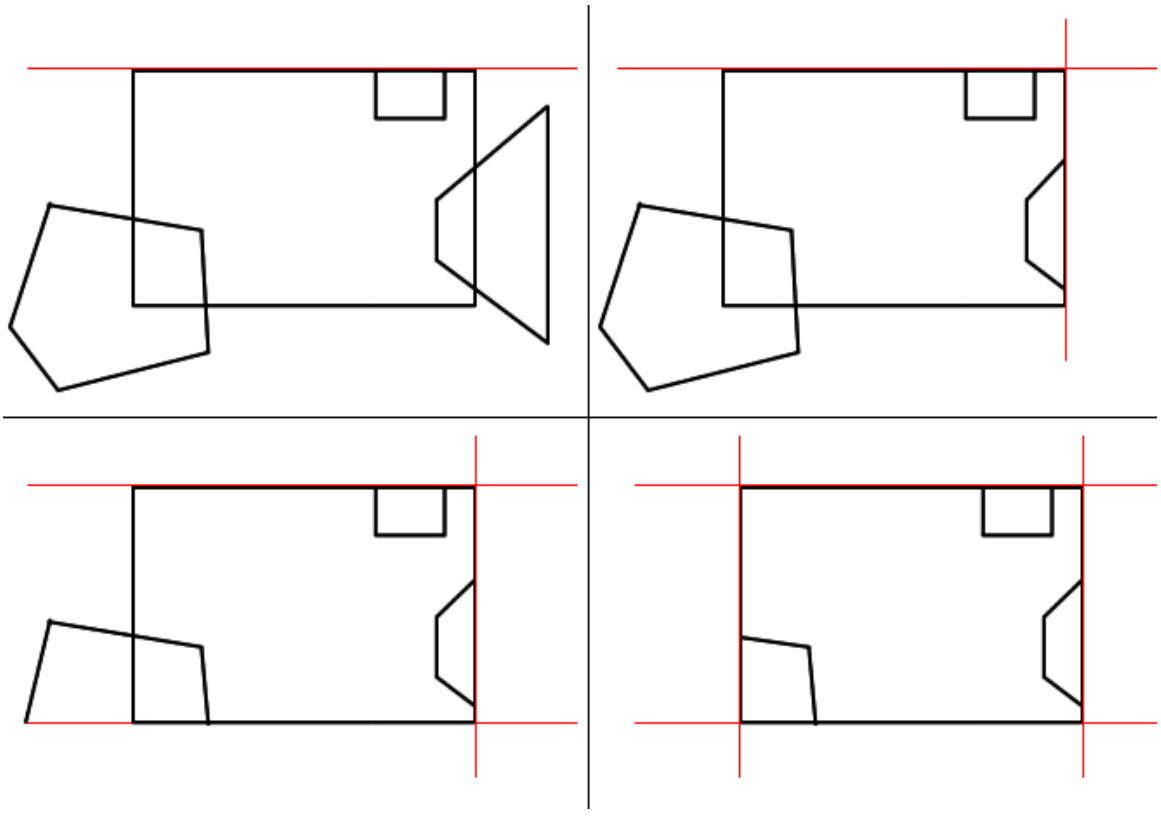

Figura 4.5: *Clipping* de Sutherland e Hodgman.

Este algoritmo é muito eficiente para recorte de polígonos convexos. No entanto, se aplicado a polígonos côncavos seu resultado não é exato [Rogers, 1985], uma vez que o recorte destas formas geométricas geram, no mínimo, *n+1* novos polígonos. E quando o algoritmo de Sutherland e Hodgman é aplicado a elas este tenta conectar os novos polígonos, gerando algumas arestas que não deveriam existir.

Para resolver este problema há duas saídas: adaptar o algoritmo de Sutherland e Hodgman para tentar transformar o polígono côncavo em um conjunto de polígonos convexos, ou então, utilizar algoritmos capazes de tratar as arestas fantasmas (arestas que não deveriam existir após o recorte). Weiler desenvolveu um algoritmo (algoritmo Weiler-Atherton) com esta característica.

Diferente de Sutherland e Hodgeman, Weiler e Atherton não abordam a superfície como sendo um conjunto sequencial de vértices. Para Weiler a cena é formada por uma série de polígonos e para determinar quais partes são visíveis, efetua-se divisões recursivas da imagem [Foley et al., 1990]. Este algoritmo é um dos mais gerais que já foram desenvolvidos para o *clipping* de superfícies [Hearn e Baker, 1986] e pode ser utilizado para recorte tanto de polígonos convexos quanto côncavos. Sua idéia básica é determinar a visibilidade de um polígono em relação a outro através das operações de interseção, união e diferença (operações de conjuntos). Observando a Figura 4.6 fica mais fácil entender o funcionamento do algoritmo de Weiler.

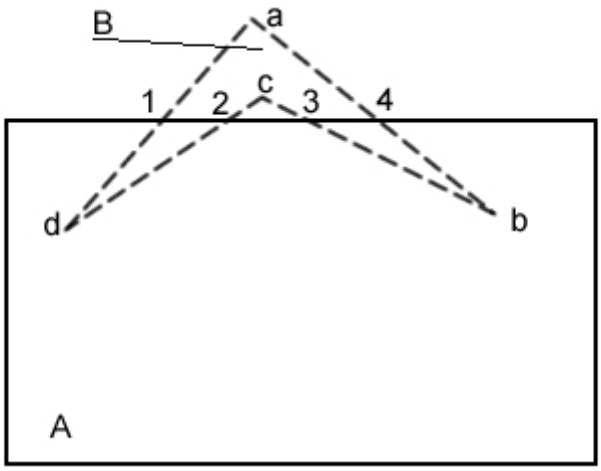

Figura 4.6: *Clipping* de polígono côncavo.

O primeiro passo é encontrar os pontos de interseção entre os polígonos *A* e *B* e armazená-los em uma estrutura de dados (geralmente uma lista). No exemplo da Figura 4.6 há 4 interseções. Estas são encontradas percorrendo-se segmento por segmento de *B* e verificando quais deles interceptam os contornos de *A*. Os pontos de interseção, caso existam, são adicionados ao conjunto solução. Este processo é repetido até que o ponto de partida seja visitado novamente. Quando isto acontece, significa que todos os segmentos foram visitados. O segundo caso, onde um ponto é adicionado à lista que forma a solução, é quando o ponto é interno à janela de *clipping*. Quando os pontos da solução formam um polígono fechado, este é desenhado e procura-se

por novas áreas visíveis. Desta forma, para o exemplo da Figura 4.6 o conjunto solução é composto por dois polígonos distintos: $s_1 = \{d, 1, 2\}$ e $s_2 = \{3, b, 4\}$, que representam o polígono recortado.

Alterando o algoritmo de Weiler e Atherton para recortar um polígono em relação à janela de *clipping* e aos demais polígonos da cena, este deixa de ser apenas um algoritmo de recorte e passa a ser um algoritmo de detecção e remoção de superfícies escondidas. Esta flexibilidade fortalece ainda mais a generalidade deste algoritmo. A detecção e remoção de superfícies ocultas será tratado com mais detalhes no próximo capítulo.

4.5 Clipping Tridimensional

O *clipping* 3D usa volumes de visualização e depende diretamente do tipo de projeção aplicado à cena. A projeção determina a forma como o volume de visualização deve ser calculado, uma vez que se esta for paralela o volume de visão é formado por um cubo e se for perspectiva, por uma pirâmide.

Os algoritmos de *clipping* de linhas apresentados neste capítulo podem ser facilmente adaptados para tratarem linhas que localizam-se no espaço [Foley et al., 1990]. Para alterar o algoritmo de Cohen-Sutherland, por exemplo, deve-se aumentar o número de posições do vetor de codificação de 4 para 6, onde o valor de cada posição assume valor verdadeiro através da análise das seguintes regras:

Posição 1 - se o ponto localiza-se acima do volume de visualização;

Posição 2 - se o ponto localiza-se abaixo do volume de visualização;

Posição 3 - se o ponto está à direita do volume de visualização;

Posição 4 - se o ponto está à esquerda do volume de visualização;

Posição 5 - se o ponto está atrás do volume do visualização;

Posição 6 - se o ponto está em frente ao volume de visualização.

Os critérios para determinar se uma linha é trivialmente rejeitada ou aceita são os mesmos do algoritmo para *clipping* 2D.

Assim como os algoritmos de recorte de linha podem ser adaptados para o sistema tridimensional, os algoritmos de *clipping* de superfícies (Sutherland-Hodgman, Weiler-Atherton, etc) também podem. Bastando algumas poucas alterações, devido ao aumento de dimensão (2D para 3D).

4.6 Conclusão

Diante dos fatos tratados e apresentados por este capítulo, fica claro o motivo de ocorrer um aumento tão grande na complexidade dos algoritmos de *clipping* de superfícies se comparados aos de linha. Enquanto no recorte de linha a semi-reta resultante continua com as mesmas características da entrada do algoritmo, no *clipping* de superfícies isto nem sempre ocorre. Muitas vezes, uma superfície recortada contém mais lados do que a original e, às vezes, um único polígono pode gerar dois novos, com características totalmente diferentes, como observa-se na Figura 4.7.

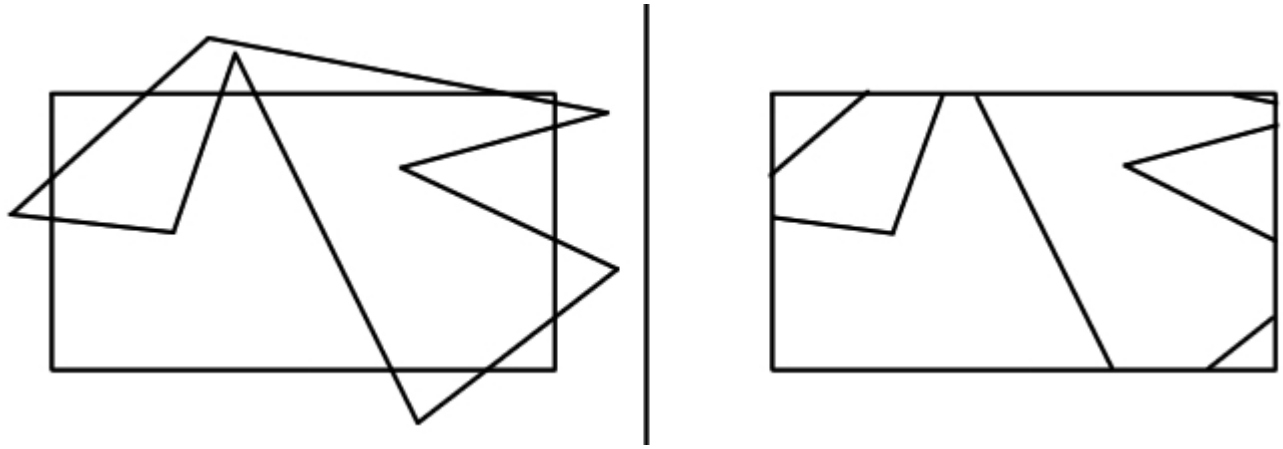

Figura 4.7: *Clipping* de uma superfície côncava.

Este é um exemplo clássico, onde uma única forma geométrica gerou duas outras completamente distintas após a aplicação de um algoritmo de *clipping*. Porém, após entender o funcionamento do recorte de linhas e superfícies 2D, fica fácil trabalhar no sistema tridimensional, pois os conceitos utilizados são praticamente os mesmos, bastando efetuar algumas adaptações nos próprios algoritmos bidimensionais. Por isso, a preocupação maior foi expressar de forma clara e objetiva o *clipping* 2D.

Capítulo 5

Detecção e Remoção de Superfícies Escondidas

5.1 Introdução

Após determinar as características do sistema de câmeras, efetuar as operações de projeção e recortar as partes da cena que extravasam os limites da janela de exibição, o próximo passo do *pipeline* de visualização gráfica é detectar e remover as superfícies que ainda se encontram escondidas na cena. Esta etapa deve ser aplicada antes de exibir a imagem para evitar a renderização das superfícies que estão encobertas por outras. Esta operação, quando bem realizada evita problemas de *renderização* como o mostrado na Figura 5.1. Além disso, suponha que atrás dos cubos desta figura há uma esfera que está encoberta, se não aplicar um algoritmo de remoção de superfícies ocultas, a mesma será *renderizada*, aumentando o tempo de processamento da cena.

Figura 5.1: (a) Erro de interpolação; (b) Interpolação correta - imagem retirada do site http://www.cgmax.com.br.

Os dispositivos gráficos atuais já possuem algoritmos para esta funcionalidade gravados em sua memória fixa, e isto torna o processamento sensivelmente mais rápido [Westermann e Ertl, 1998], uma vez que não é necessário efetuar todos os cálculos via software. Este capítulo apresenta os conceitos, apresentando as vantagens de aplicar tal técnica na exibição de cenas geradas por computador, bem como os principais algoritmos e seus resultados.

5.2 O problema de determinar a visibilidade de uma superfície

Remover as superfícies escondidas de uma cena é considerado por muitos pesquisadores da área de computação gráfica uma das tarefas mais difíceis do *pipeline* de visualização [Bhattacharya et al., 1989]. Isto porque não é apenas remover os *pixels* invisíveis, antes deve-se determinar quais destes pixels não estão visíveis e esta é a maior dificuldade.

O posicionamento da câmera, a quantidade e tamanho dos objetos são fatores que influenciam na visibilidade das superfícies. Por muito tempo a visibilidade foi discutida e pesquisada, gerando várias técnicas que posteriormente foram trazidas para a detecção e remoção de superfícies ocultas, dando origem a diversos algoritmos para efetuar estas operações. Os métodos de determinação de visibilidade dividem-se em dois grupos, (a) métodos que trabalham sobre espaço do objeto e (b) métodos que consideram o espaço da imagem. A primeira abordagem (a), compara entre si todos os objetos que compõem a cena para determinar quais superfícies estão visíveis ao observador [Sechrest e Greenberg, 1981]. Já em (b), cada ponto (pixel) que forma os agentes da cena é comparado com os demais. A implementação de algoritmos baseados no espaço da imagem é bem menos complexa, motivo pelo qual estes aparecem com muito mais frequência do que os outros [Sharir e Overmars, 1992].

5.3 Os algoritmos de visibilidade

Em geral, os algoritmos para detecção e remoção de superfícies escondidas são assintoticamente quadráticos, pois trabalham sobre o espaço matricial, porém isto influencia pouco no tempo de execução, uma vez que estes algoritmos utilizam recursos próprios dos dispositivos gráficos (processador, memória, etc), e estes dispositivos possuem um desenvolvimento muito acelerado.

Aliado e este crescimento, o nível de dificuldade para otimizar os atuais algoritmos de visibilidade geram uma acomodação em relação à melhoria destas rotinas, pois as rotinas atuais conseguem suprir quase todas as exigências impostas a elas. Alguns algoritmos, como o z-buffer, conseguem juntamente com os dispositivos gráficos obter resultados satisfatórios na detecção e remoção de superfícies encobertas, tanto que é utilizado pela biblioteca gráfica

OpenGL [Greene et al., 1993].

5.3.1 Método Back-Face

Este algoritmo é muito simples e é abordado apenas para reforçar a idéia da detecção de superfícies ocultas. Trabalhando sobre o espaço da imagem, este método é simples e rápido para detectar quais superfícies de um poliedro está encobrindo outra. Ele faz um único teste para verificar se o ponto é interno a um polígono ou não [Kumar e Manocha, 1996]. Considerando um ponto 3D *(x, y, z)* este faz parte do polígono se a Inequação 5.1 é satisfeita.

$$a \times x + b \times y + c \times z + d < 0 \tag{5.1}$$

Onde os coeficientes *a*, *b*, *c* e *d* são constantes que descrevem as propriedades espaciais da superfície à qual envolve o ponto.

Considerando um vetor *V* com mesma direção das linhas do sistema de visão (ou da câmera) da cena e *N = (a, b, c)*, um vetor normal à uma face do poliedro, tem-se que esta face está encoberta por alguma outra através da Inequação 5.2, como mostra a Figura 5.2.

$$V.N > 0 \tag{5.2}$$

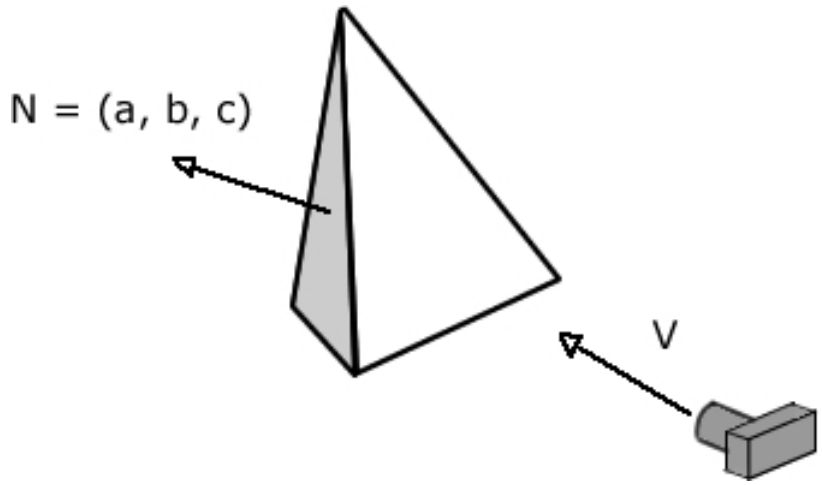

Figura 5.2: Vetores que determinam a visibilidade de uma superfície.

Este algoritmo é muito simples e primitivo, porém apresenta de forma clara e simplificada o problema de visibilidade de superfícies, as próximas seções apresentam algoritmos mais eficientes e completos para tratar, detectar e também remover as superfícies que não são vistas pelo observador.

5.3.2 Método BSP-Tree

Com grande uso nas décadas de 80 e 90, a árvore de particionamento binário do espaço surgiu com o objetivo de determinar a superfície visível em cada ponto da cena. Este método é eficiente para cenas onde o ponto de referência do sistema visualização muda, mas os objetos permanecem fixos. Ele divide o espaço da cena em conjuntos menores, desenhando os polígonos que estão mais ao fundo primeiro, desta forma deve-se ordenar os polígonos para evitar erros de exibição [Tsai e Wang, 2007]. Esta ordenação ocorre durante o processo de pré-processamento da cena para evitar aumento no tempo de execução explícita.

A Figura 5.3 mostra como a árvore de particionamento binário do espaço é montada, observe que o ramo esquerdo da árvore é preenchido com faces que encontram-se em frente ao plano de partição, ao passo que o da direita contém as superfícies localizadas mais ao fundo.

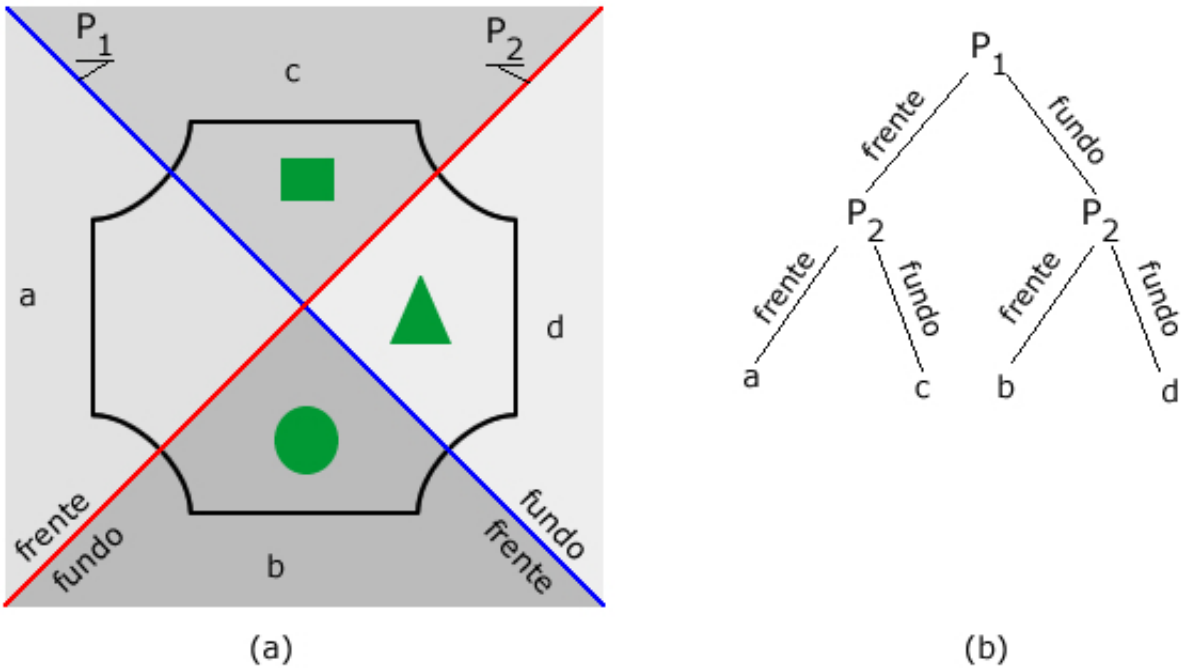

Figura 5.3: Divisão do espaço da cena por dois planos (a) e árvore de partição (b).

Quando for desenhar basta percorrer a árvore lendo primeiro os ramos do fundo e desenhando suas respectivas superfícies. No entanto, este método possui alguns problemas:

- Se um polígono intercepta outro polígono estes não são exibidos corretamente;
- O cálculo da ordem em que os polígonos devem ser desenhados pode ser computacionalmente caro, caso haja muitos polígonos em uma cena.

Este algoritmo é pouco explorado neste trabalho pelo fato de ter perdido espaço na computação gráfica atual devido ao surgimento dos aceleradores gráficos e do método de z-buffer, que aliados conseguem ser mais rápidos e eficientes do que o BSP-tree.

5.3.3 Método Depth-Buffer

Este método é eficiente e muito utilizado para resolver o problema da visibilidade. Por atuar no espaço da imagem, pode ser aplicado para qualquer tipo de polígono ou objeto. Ele compara a profundidade das superfícies através da posição de cada pixel em relação ao plano de projeção. O algoritmo de z-buffer [Han, 2004] utiliza este método para determinar quais superfícies ou parte destas devem ser retiradas da cena. Neste caso, o plano de visualização fica ao longo do *eixo z* do sistema de coordenadas.

Cada superfície é validada individualmente através de cada um de seus *pixels*. As principais vantagens deste algoritmo são: sua velocidade, a não necessidade de ordenação das superfícies e sua simplicidade de implementação [Hearn e Baker, 1986]. Os *pixels* selecionados como visíveis são armazenados na forma bidimensional, ou seja, já projetados, daí a necessidade de usar uma matriz auxiliar na implementação do algoritmo para guardar as informações de profundidade de cada pixel pré-selecionado para o conjunto solução. Além de usar uma área de *buffer* auxiliar, deve-se ter outra área para armazenas as informações de intensidade de cada *pixel*, para permitir a reprodução idêntica do mesmo após o final da execução do método.

Inicialmente, o *buffer* de intensidade é inicializado com a intensidade da cor de fundo e o *buffer* de profundidade inicia-se com 0 (posição do plano de *clipping* traseiro, ou mais afastado) ou com um valor muito pequeno, que é considerado a profundidade mínima. A profundidade máxima é limitada pelo plano de *clipping* frontal, definido por um z_{max}, que depende da aplicação. Desta forma, cada pixel da cena tem sua profundidade comparada com respectiva posição do *buffer* de profundidade (este trabalho considera a área de *buffer* como uma matriz bidimensional), trocando as informações das duas matrizes quando o pixel analisado localizar-se mais próximo do observador do que o anteriormente armazenado em sua posição. A Figura 5.4 apresenta um esquema do funcionamento deste método.

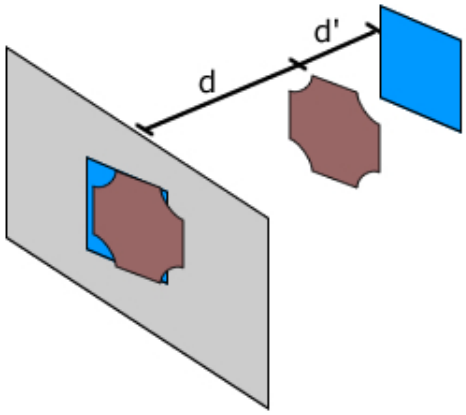

Figura 5.4: Objetivo do algoritmo *z-buffer* (compara todos os pixels e retorna o que tiver menor profundidade).

O algoritmo de z-buffer

O algoritmo de z-buffer (Algoritmo 5.1) é simples de implementar, pode ser aplicado em qualquer cena e gera resultados satisfatórios. No entanto, exige uma área de buffer auxiliar e dependente da resolução do sistema de vídeo e isto pode exigir uma fatia a mais de memória para ser executado.

Algoritmo 5.1 Algoritmo z-buffer

Entrada: Uma estrutura contendo os pontos p_{ij} de todos os objetos na cena.

Saída: A matriz de intensidade, contendo as informações dos pixels visíveis.

1: $w \leftarrow LarguraResolucao()$ // Retorna a largura da resolução do dispositivo de vídeo
2: $h \leftarrow AlturaResolucao()$ // Retorna a altura da resolução do dispositivo de vídeo
3: $zBufferDepth[][] \leftarrow 0.0$ // Matriz de profundidade inicializada com 0.0
4: $zBufferInt[][] \leftarrow IntensidadeFundo()$ // Matriz de intensidade inicializada com a cor de fundo da cena
5: **para** $i \leftarrow 1$ até w **faça**
6: **para** $j \leftarrow 1$ até h **faça**
7: **se** $p_{ij}.getZ()$ ¿ zBufferDepth[i][j] **então**
8: $zBufferDepth[i][j] \leftarrow p_{ij}.getZ()$ // Retorna a componente *z* do ponto
9: $zBufferInt[i][j] \leftarrow p_{ij}.Intensidade()$ // Retorna a intensidade do ponto
10: **fim se**
11: **fim para**
12: **fim para**
13: Retorna $zBufferInt[][]$

A Figura 1.2 - parte (b) mostra o resultado da aplicação do método *z-buffer*. Mesmo usando uma matriz *m* × *n* como *buffer* de profundidade, a qual recebe os valores da coordenada *z* pertencente ao sistema de coordenadas global o algoritmo de z-buffer trabalha sobre o sistema de coordenadas do dispositivo gráfico no que se refere à comparação do posicionamento dos pontos da cena. Na matriz de profundidade o elemento *ij* corresponde ao pixel *ij* da tela, da mesma forma que na matriz de intensidade. Assim, este algoritmo pode passar pelo mesmo ponto várias vezes, porém isto não reduz sua eficiência sua eficiência.

O método *z-buffer* é eficiente, porém não permite a aplicação de *anti-aliasing*, pois armazena apenas uma informação do pixel em cada posição. Para resolver este problema foi proposto o método *a-buffer*. Neste, cada posição da matriz de *buffer* possui uma lista de informações, assim torna possível armazenar os dados necessários para aplicar técnicas de suavização dos contornos dos objetos da cena.

5.3.4 Método Scan-Line

Método que trabalha sobre o espaço da imagem, o *scan-line* para remoção de superfícies escondidas possui os mesmos conceitos do *scan-line* para preenchimento de polígonos [Crocker, 1984]. Ele percorre a cena verificando linha por linha para determinar sua visibilidade. Cada linha traçada intercepta várias superfícies em uma mesma posição, porém o método de *scan-line* retorna as características do pixel da primeira face atingida pela linha, considerando que a origem desta é o olho da cena (ou a câmera). Assim, o algoritmo retorna sempre o pixel mais próximo do observador em uma posição *ij* da matriz da cena.

O Algoritmo scan-line

Ao contrário do algoritmo de z-buffer, o scan-line (Algoritmo 5.2) desenha cada posição da matriz de intensidade uma única vez. No entanto o princípio do método é o mesmo, armazenar as informações do pixel visível em uma matriz, para aumentar sua eficiência esta abordagem utiliza-se de listas ordenadas para proporcionar coerência ao longo da scan-line. Sua idéia básica é ordenar todas as arestas do polígono de forma crescente do valor de y e partir daí determinar o intervalo de x que é interceptado pela scan-line, fazendo uso da idéia de que os vizinhos envolvidos pelos mesmos polígonos possuem características semelhantes (coerência). Analisando com calma, percebe-se que ao final da varredura a junção das scan-lines forma um plano, por isso às vezes faz-se alguns autores preferem chamá-las de plano de varredura.

Algoritmo 5.2 Algoritmo scan-line

Entrada: Uma estrutura contendo os pontos p_{ij} de todos os objetos na cena.

Saída: A matriz de intensidade, contendo as informações dos pixels visíveis.

1: $w \leftarrow LarguraResolucao()$ // Retorna a largura da resolução do dispositivo de vídeo
2: $h \leftarrow AlturaResolucao()$ // Retorna a altura da resolução do dispositivo de vídeo
3: $zBufferInt[][] \leftarrow IntensidadeFundo()$ // Matriz de intensidade inicializada com a cor de fundo da cena
4: **para** $i \leftarrow 1$ até h **faça**
5: **para** $j \leftarrow 1$ até w **faça**
6: Encontra a superfície que contém o primeiro pixel atingido pela scan-line que parte do observador
7: $zBufferInt[i][j] \leftarrow p_{ij}.Intensidade()$ // Retorna a intensidade do ponto
8: **fim para**
9: **fim para**
10: Retorna $zBufferInt[][]$

Enquanto o algoritmo de z-buffer necessita de uma matriz m x n para efetuar o buffer das profundidades da cena, o scan-line utiliza apenas de uma linha desta matriz. Consequentemente,

o espaço de memória utilizado é menor.

Além de pintar cada pixel uma única vez (Figura 5.5), este método explora a coerência entre os pixels localizados ao longo de uma linha de varredura e permite a inclusão anti-aliasing. Por outro lado, não considera a coerência entre as linhas de varredura, ou seja na vertical, se isto fosse considerado poderia eliminar partes, ou às vezes polígonos inteiros com a utilização de apenas uma scan-line. Uma vez que se fosse considerado a semelhança de vizinhos na vertical poderia utilizar algumas heurísticas para prever se a fatia vizinha é visível ou não.

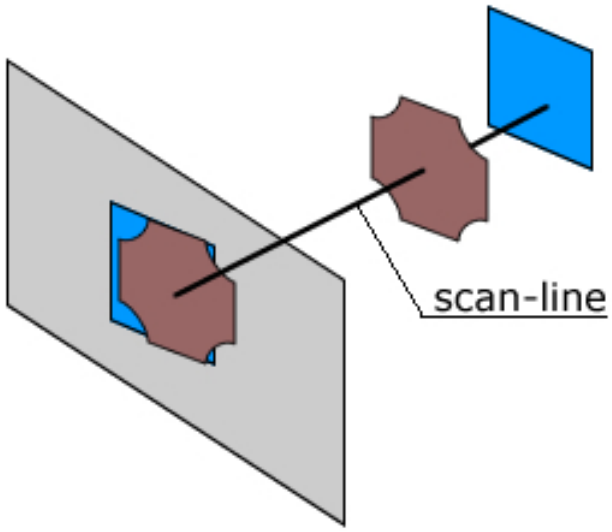

Figura 5.5: Objetivo do algoritmo *scan-line* (retorna o pixel que for interceptado primeiro).

5.3.5 Visibilidade de Superfícies Curvas

Para detectar as partes visíveis de uma superfície curva pode-se convertê-la em uma sequência de polígonos e determinar quais destes são visíveis, porém este processo pode gerar um número muito grande de polígonos, o que exige uma quantidade de memória e um processamento muito grande. Alguns pesquisadores, como Catmull, Binn e Whitted propuseram alguns métodos para solucionar este problema [Hill, 2000].

Desde então muitas sugestões foram apresentadas e em geral para determinar a visibilidade de superfícies curvas utiliza-se os métodos *octree* e *ray-casting* [Hill, 2000]. Utiliza-se o *ray-casting* para determinar o local de interseção mais próximo para os raios que passam por cada *pixel* e o *octree* para refinar (otimizar) o processo de detecção das áreas visíveis da cena.

Ray-Catsting - raios de luz são traçados do observador para o fundo da cena, verificando o que cada pixel da tela consegue enxergar. Caso uma superfície seja transparente o raio vai atravessá-la e interceptar outra face, o método *ray-casting* vai somando a contribuição

de cada uma destas superfície que a linha de luz atinge. Desta forma, este método é capaz de tratar transparência e iluminação eficientemente.

Octree - representação hierárquica baseada na decomposição recursiva do espaço tridimensional em cubos, denominados octantes [Hearn e Baker, 1986].

5.4 Conclusão

Detectar e remover os *pixels* dos objetos que encontram-se fora do alcance dos raios de visão do observador é parte fundamental do processo de visualização de sólidos, principalmente no que diz respeito ao realismo da cena, pois evita que um objeto que localiza-se mais ao fundo da cena apareça na frente de outro que está localizado a uma distância menor do visualizador. Além disso, em cenas complexas e com muitos objetos a redução nos pontos a serem renderizados é significativa. E com o alto desempenho dos dispositivos gráficos modernos o tempo de processamento gasto para remover estes pontos é praticamente desprezível, otimizando o processo computacional para reproduzir cenas tridimensionais com alto grau de realismo.

Capítulo 6

Montando um Pipeline de Visualização Gráfica

6.1 Introdução

Os estudos realizados durante este projeto mostraram que é importante analisar separadamente os conceitos de cada parte do pipeline de visualização gráfica e entender a fundo o funcionamento de cada uma. No entanto, estes conceitos não são aplicados isoladamente. Na prática, um complementa e depende do outro. Este capítulo apresenta os resultados obtidos através da junção de algumas técnicas de visualização. As razões que levaram à escolha destas técnicas também são expostas e justificadas.

Para concluir, este capítulo apresenta os resultados de algoritmos que apresentam visualmente os efeitos das técnicas de *clipping* circular e remoção de superfícies escondidas em uma cena composta por alguns polígonos sobrepostos.

6.1.1 Clipping Circular

O capítulo três apresentou e discutiu os principais conceitos e abordagens de *clipping* aplicado ao *pipeline* de visualização gráfica. Esta técnica é importante para reduzir e esforço computacional envolvido na geração de cenas, principalmente aquelas onde os objetos pertencentes ao cenário são grandes e aparecem em grande número na cena. Só para reforçar a idéia básica desta técnica, o *clipping* tem por finalidade recortar as linhas e superfícies de uma cena que localizam-se fora da área visível (*viewport*). Assim, há duas linhas de raciocínio para efetuar esta operação, uma é recortar as arestas que contornam o objeto e depois aplicar a técnica de preenchimento e a outra é rasterizar toda a cena e verificar a validade de cada pixel.

Outra característica que destacou-se durante a análise deste componente do processo de visualização é o formato do *viewport*. No processo de realização deste trabalho, constatou-se que a grande maioria das literaturas tratam a área visível como um polígono retangular, algumas

consideram como um polígono irregular, e poucas citam o uso de uma circunferência como janela de *clipping*, porém não apresenta o algoritmo para esta solução. Considerando que uma janela de visualização circular (ou elíptica) pode ser aplicada em várias situações, tais como em simulação de uma luneta, de uma mira de uma arma em um jogo de guerra ou ainda na área médica, para simular a visão interna a uma artéria, foi proposto o desenvolvimento de um algoritmo que fosse capaz de recortar linhas e superfícies utilizando uma circunferência como área visível.

6.1.2 Descrição do Algoritmo

O algoritmo de recorte utilizando janela de visualização circular foi desenvolvido em duas fases, a primeira para recorte de linhas e a segunda para recorte de superfícies. O recorte de linha utiliza-se das equações da reta e da circunferência [da Silva e dos Reis, 1996] para definir os segmentos de reta que são visíveis e quais não são. Para obter o algoritmo, o primeiro passo foi determinar qual abordagem utilizar. Após análise de alguns algoritmos, como Cohen-Sutherland [Foley et al., 1990], Liang-Barsky [Hearn e Baker, 1986] e outros, decidiu-se que a forma mais eficiente para o recorte de linhas é o cálculo direto da interseção, pois assim efetua apenas uma iteração. Para isso, torna-se necessário determinar uma formulação matemática para o problema.

6.1.3 Recorte de Linhas

Para determinar a porção visível de uma linha através de uma área circular utiliza-se a Equação 6.1.

$$
\begin{aligned}
a &= -2 \times x_c \\
b &= -2 \times y_c \\
c &= x_c^2 + y_c^2 - r^2 \\
p &= a \times (x_2 - x_1)^2 - 2 \times (-x_1 \times y_2 + x_2 \times y_1) \\
&\times (y_1 - y_2) - b \times (x_2 - x_1) \times (y_1 - y_2) \\
q &= (-x_1 \times y_2 + x_2 \times y_1)^2 + c(x_2 - x_1)^2 \\
\Delta &= p^2 - 4 \times ((x_2 - x_1)^2 + (y_1 - y_2)^2) \times q
\end{aligned}
$$

$$
x_{i_1} = \frac{-p + \sqrt{\Delta}}{2 \times ((x_2 - x_1)^2 + (y_1 - y_2)^2)} \tag{6.1}
$$

$$
x_{i_2} = \frac{-p - \sqrt{\Delta}}{2 \times ((x_2 - x_1)^2 + (y_1 - y_2)^2)}
$$

$$
y_{i_1} = \frac{(-x_1 \times y_2 + x_2 \times y_1) - x_{i_1} \times (y_1 - y_2)}{x_2 - x_1}
$$

$$
y_{i_2} = \frac{(-x_1 \times y_2 + x_2 \times y_1) - x_{i_2} \times (y_1 - y_2)}{x_2 - x_1}
$$

onde:
x_1, y_1, x_2 e y_2 são as coordenadas dos pontos p_1 e p_2, respectivamente;
x_c e y_c são as coordenadas do centro da circunferência e r o raio desta;
(x_{i_1}, y_{i_1}) e (x_{i_2}, y_{i_2}) os pontos de interseção;
p e q são artifícios matemáticos para redução da equação total;
Δ é o discriminante da equação.

Considerando a formulação matemática, emprega-se a Equação 6.1 para encontrar os pontos na extremidade visível da linha. Observa-se que as equações da circunferência e da reta formam a base dos cálculos [da Silva e dos Reis, 1996]. Desta forma, os dados de entrada são: (a) os pontos finais da linha a ser desenhada, (b) o centro da janela e (c) o raio da janela de visualização. Com estes dados, foi estabelecido: a equação da reta que passa pelos dois pontos fornecidos e a equação da circunferência que contorna a área visível. Após determiná-las, efetua o cálculo das coordenadas de cada ponto de interseção, encontrando-os com uma única iteração. Caso a linha não intercepte a circunferência, ou seja, esteja totalmente fora da área visível, o valor de Δ é negativo.

A Figura 6.1 apresenta a modelagem geométrica para uma janela de *clipping* circular. Nesta, os segmentos de linha não visíveis são mostrados em tom de cinza e o segmento visível na cor preta.

O algoritmo aqui apresentado segue a idéia do algoritmo de Cohen e Sutherland

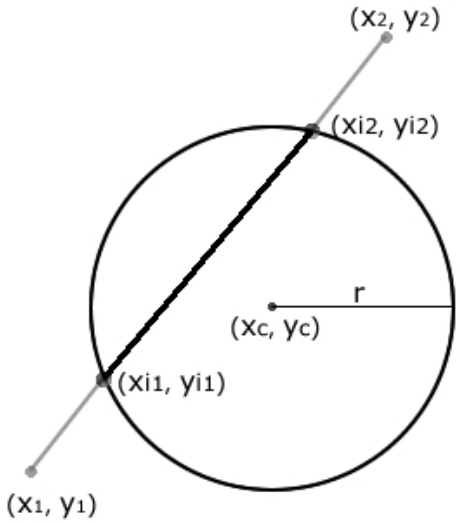

Figura 6.1: Modelo geométrico de uma janela de clipping circular.

[Rogers, 1985], conhecido também como algoritmo de subdivisão de linha. Neste, Cohen e Sutherland, após descartar a possibilidade de que a linha é totalmente invisível, a mesma é dividida fazendo uso das coordenadas que limitam a janela de visualização, para obter os pontos de interseção, utilizando uma única iteração. A Figura 6.2 apresenta o resultado da execução do algoritmo da subdivisão para desenhar algumas linhas.

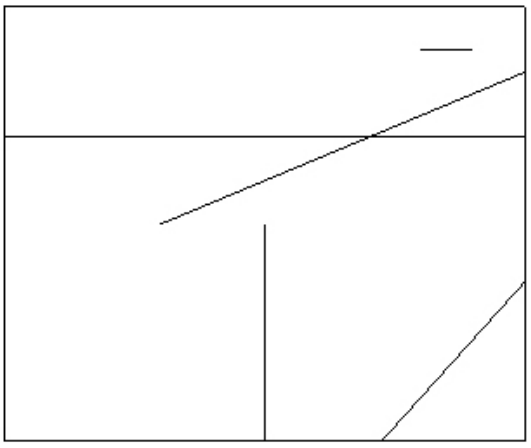

Figura 6.2: Imagem gerada pela execução do algoritmo de Sutherland.

6.1.4 Encontrando os Pontos de Interseção

O Algoritmo 6.1 monta a equação da reta que passa pelos pontos p_1 e p_2, a equação da circunferência que contorna a janela de visualização, a qual possui centro no ponto de coordenadas (x_c, y_c) e raio igual a r; depois, substitui a equação da reta na equação da circunferência, para deixá-la apenas em função de x e encontrar o valor de Δ. Se Δ for negativo, a linha limitada pe-

los pontos p_1 e p_2 é totalmente invisível, pois neste caso a reta que passa por eles não intercepta a circunferência em nenhum ponto. Caso contrário, as coordenadas dos pontos de interseção são calculadas e armazenadas nos pontos (x_{i_1}, y_{i_1}) e (x_{i_2}, y_{i_2}).

Algoritmo 6.1 Encontra Cruzamentos

Entrada: Os pontos $(x_1, y_1), (x_2, y_2)$ (que definem a linha), o ponto (x_c, y_c), centro da circunferência e o raio r.

Saída: Os pontos $(x_{i_1}, y_{i_1}), (x_{i_2}, y_{i_2})$, de interseção.

$a \leftarrow -2 \times x_c$
$b \leftarrow -2 \times y_c$
$c \leftarrow x_c^2 + y_c^2 - r^2$
se $(x_1 = x_2)$ **então**
    $temp_1 \leftarrow y_1 - y_2$
    $temp_2 \leftarrow x_2 - x_1$
senão
    $temp_1 \leftarrow x_2 - x_1$
    $temp_2 \leftarrow y_1 - y_2$
fim se
$temp_3 \leftarrow -x_1 \times y_2 + x_2 \times y_1$
$temp_4 \leftarrow temp_1^2 + temp_2^2$
$temp_5 \leftarrow a \times temp_1^2 - 2 \times temp_3 \times temp_2 - b \times temp_1 \times temp_2$
$temp_6 \leftarrow temp_3^2 + c \times temp_1^2$
$\Delta \leftarrow temp_5^2 - 4 \times temp_4 \times temp_6$
se $\Delta < 0$ **então**
    $flag \leftarrow -1$
    retorne $flag$
fim se
$flag = 0$
$x_{i_1} \leftarrow (-temp_5 + \sqrt{\Delta})/(2 \times temp_4)$
$x_{i_2} \leftarrow (-temp_5 - \sqrt{\Delta})/(2 \times temp_4)$
$y_{i_1} \leftarrow (temp_3 - x_{i_1} \times temp_2)/temp_1$
$y_{i_2} \leftarrow (temp_3 - x_{i_2} \times temp_2)/temp_1$
se $(x_1 = x_2)$ **então**
    retorne $flag, (y_{i_1}, x_{i_1}), (y_{i_2}, x_{i_2})$
fim se
retorne $flag, (x_{i_1}, y_{i_1}), (x_{i_2}, y_{i_2})$

6.1.5 Desenhando a linha

Após encontrar os pontos de interseção, deve-se fazer a plotagem. Porém, é importante verificar antes, se estes são realmente consistentes. Isso deve ser feito porque nem sempre a linha que se deseja desenhar é limitada pelos pontos de interseção, como observa-se na Figura 6.4. O Algoritmo 6.2 valida os pontos e desenha a parte visível da linha, caso exista (Figura 6.3).

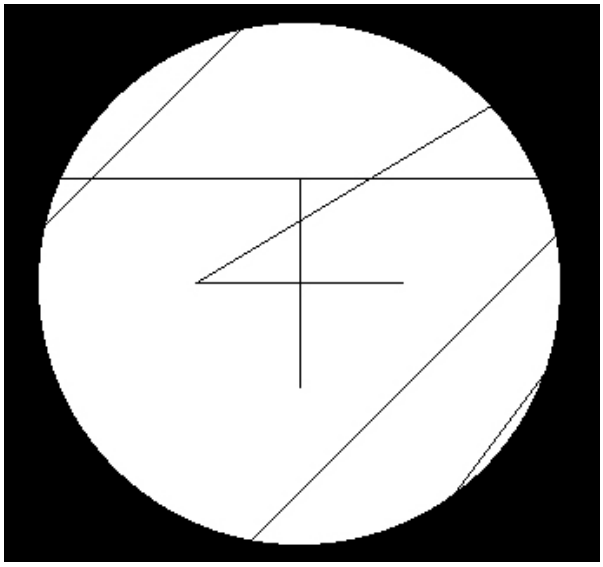

Figura 6.3: Imagem gerada pela execução do Algoritmo 6.2

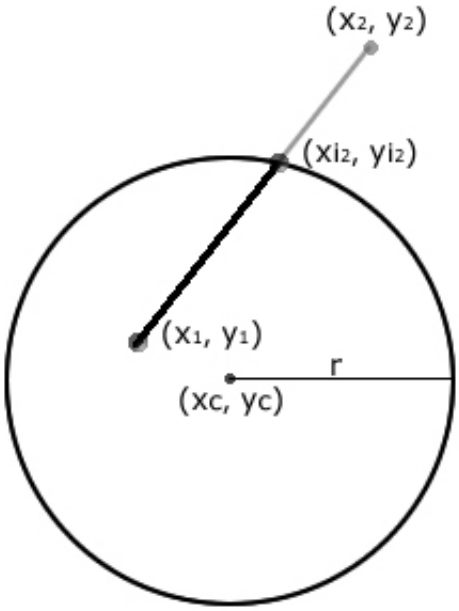

Figura 6.4: Imagem onde uma interseção é inválida

Algoritmo 6.2 Desenha Linha

Entrada: Os pontos $p_1 = (x_1, y_1)$ e $p_2 = (x_2, y_2)$, o centro $c = (x_c, y_c)$ e o raio r

se $(distancia(p_1, c) < r)$ **e** $(distancia(p_2, c) < r)$ **então**
  $(x_{i_1}, y_{i_1}) \leftarrow (x_1, y_1)$
  $(x_{i_2}, y_{i_2}) \leftarrow (x_2, y_2)$
senão
  Encontre (x_{i_1}, y_{i_1}) e (x_{i_2}, y_{i_2}) utilizando o Algoritmo 6.1
  se $flag = 0$ **então**
    se $(distancia(p_1, c) < distancia((x_{i_1}, y_{i_1}), c))$ **então**
      $(x_{i_2}, y_{i_2}) \leftarrow p_1$
    fim se
    se $(distancia(p_2, c) < distancia((x_{i_2}, y_{i_2}), c))$ **então**
      $(x_{i_2}, y_{i_2}) \leftarrow p_2$
    fim se
  fim se
  Desenhe a linha definida por (x_{i_1}, y_{i_1}) e (x_{i_2}, y_{i_2})
fim se

6.1.6 Recorte de Superfícies

Para o *clipping* de linhas, a conclusão tirada é que a forma mais eficiente de fazê-lo é encontrando os pontos onde a reta intercepta a circunferência de visualização e desenhar a semi-reta que os liga. Para isto, segue-se os passos apresentados pela Equação 6.1 para encontrar as raízes do novo sistema quadrático. Para o *clipping* de superfícies a abordagem escolhida foi outra. Após a análise dos algoritmos já existentes, chegou-se à conclusão de que é mais viável rasterizar o objeto antes recortá-lo. Isto deve ser feito para permitir a verificação direta do estado de cada ponto da superfície a ser desenhada. O algoritmo de recorte de superfície para área de visualização circular resultante deste trabalho pode ser visualizado no Algoritmo 6.3.

Algoritmo 6.3 Clipping de Superfície

Entrada: A superfície S, o centro c = (x_c, y_c) e o raio r.

Saída: Os pontos válidos para cena.

$V \leftarrow raster(S)$ //o vetor V recebe os pontos resultantes da rasterização de S
$tolerancia \leftarrow 0.3$ //tolerância para exceder a área visível
para i de 1 até $|V|$ **faça**
  se distancia(c, V[i]) ¡ r + tolerancia **então**
    $desenha(V[i]_x, V[i]_y)$
  fim se
fim para

O Algoritmo 6.3 é simples e eficiente para o *clipping* circular de qualquer superfície. Isto, faz com que seja de grande utilidade para ser implementado no pipeline de visualização gráfica. A Figura 6.5 apresenta o resultado da execução do mesmo, considerando um retângulo como área de entrada.

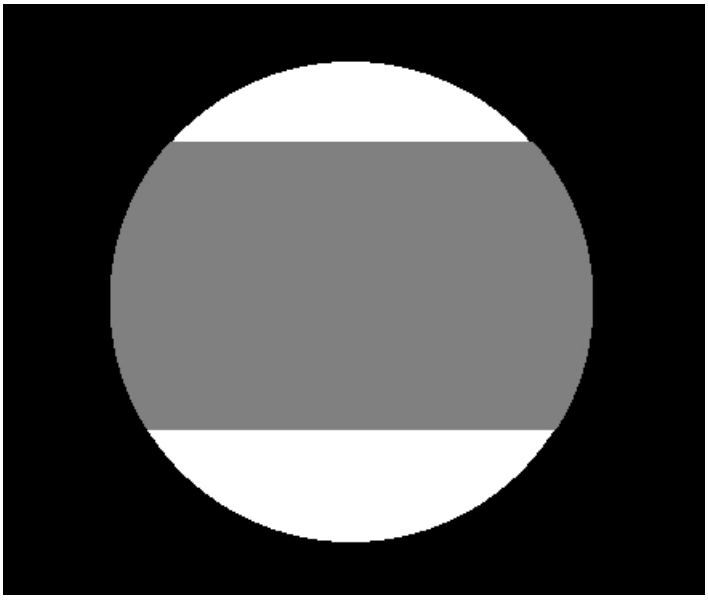

Figura 6.5: Recorte de Superfície

6.1.7 Algoritmo z-Buffer

O último estágio do pipeline de visualização gráfica abordado por este trabalho é a detecção e remoção de superfícies escondidas. Este processo é responsável por dar mais realismo à cena e também por melhorar o desempenho durante a renderização. Há várias abordagens que tratam este problema, no entanto, por motivos já exaustivamente discutidos durante este texto, a técnica escolhida é a de trabalhar com objetos rasterizados, simplificando o algoritmo e explorando a capacidade dos dispositivos de vídeo atuais. Após analisar vários métodos para determinar visibilidade de superfícies, tais como *BSP-Tree*, *depth-buffer* e *scan-line* foi escolhido o algoritmo de *z-buffer* (desenvolvido utilizando o método de *depth-buffer*). Esta escolha baseou-se principalmente na usabilidade deste procedimento, uma vez que este é muito utilizado pelos dispositivos e pacotes gráficos, como o pacote OpenGL.

6.2 Junção dos Algoritmos

Para demonstrar visualmente o funcionamento do pipeline de visualização gráfica abordado por este trabalho optou-se por desenvolver um pseudo-protótipo que mostra passo a passo como é feito o *clipping* e a remoção das superfícies (ou pixels) que encontram-se fora da área visível.

Este pseudo-protótipo foi feito utilizando C/C++ e comandos básicos de OpenGL. A linguagem C/C++ foi escolhida por permitir manipulação mais eficiente dos dispositivos gráficos, e o pacote OpenGL por ser uma ferramenta de uso livre. Utilizou-se uma janela circular como *viewport* para que o algoritmo de *clipping* desenvolvido durante este trabalho pudesse ser aplicado. Porém, antes de remover os pixels não vistos, os polígonos da cena são totalmente desenhados, esta operação é feita apenas para permitir uma melhor visualização de como este processo é realizado, pois na prática a eliminação é feita durante a leitura dos pixels, não sendo necessário a plotagem dos mesmos antes da verificação.

As operações de câmera e projeção são feitas usando diretamente as funções da biblioteca OpenGL específicas para este fim, o mesmo não acontece com o *clipping* e com a remoção de superfícies escondidas porque a intenção é utilizar o recorte circular (algoritmo desenvolvido no decorrer deste) e mostrar como deve-se tratar o problema da visibilidade. Para ficar mais claro para o observador aplicou-se um *alpha* em alguns instantes da cena, para tornar os polígonos translúcidos e permitir a visualização da eliminação dos pixels encobertos pelos demais polígonos.

6.2.1 Considerações Importantes

Durante a execução do algoritmo de recorte circular surgiu a necessidade de aplicar alguma técnica na cena para suavizar os contornos dos polígonos cortados e evitar que fiquem falhas onde deveria ser preenchido. A solução para eliminar estas eventuais falhas que aparecem entre o contorno de visualização e os novos polígonos foi permitir a plotagem de uma faixa com um pixel de largura em torno da circunferência que forma o *viewport*. Todos os pontos utilizados para formar os polígonos da cena foram lidos de arquivos, onde os pixels estão armazenados na forma (x y z). Cada superfície está armazenada em um arquivo que é lido cada vez que o polígono é desenhado no ambiente virtual.

Outro detalhe importante é que a ordem do *pipeline* de visualização gráfica deve ser respeitada para que a cena seja exibida com menor tempo de processamento, pois se o algoritmo de z-buffer for executado antes do de recorte, por exemplo, muitos pixels que não deveriam mais fazer parte da cena serão comparados.

6.3 Os Resultados

A Figura 6.6 apresenta seis estágios do processo de visualização de cenas aplicando-se *clipping* circular e remoção de superfícies escondidas. Em *(a)* aparece a cena inicial, sem aplicar nenhuma das técnicas citadas. Para permitir a visualização através dos polígonos aplicou-se um fator *alpha* com cinquenta por cento de transparência. Note que, alguns polígonos, ou parte deles, encontram-se na mesma linha de visão, como por exemplo, os polígonos azul e verde.

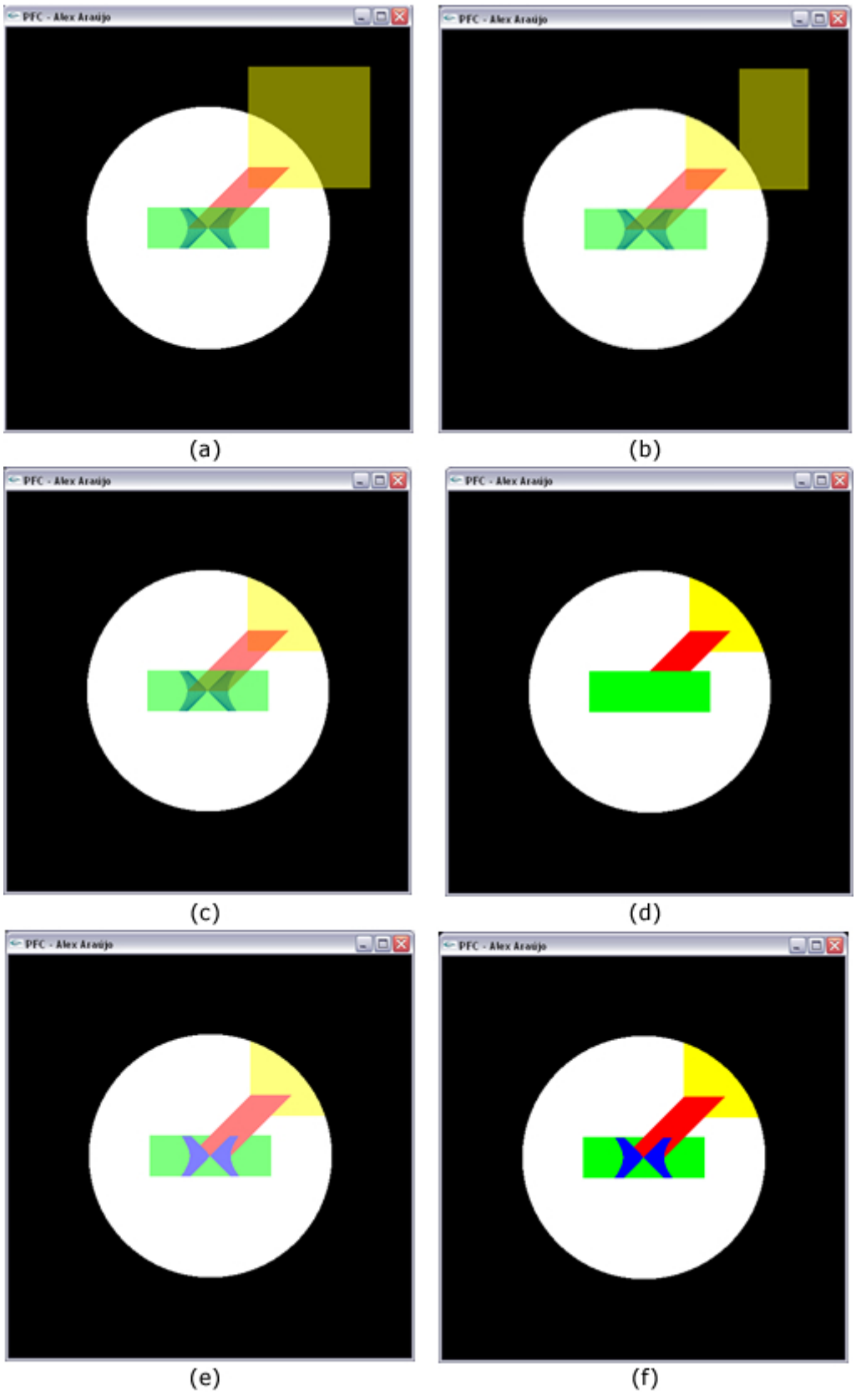

Figura 6.6: Resultados gerados pelo protótipo

Para demonstrar o funcionamento do algoritmo de *z-buffer* cada polígono encontra-se em uma profundidade. A Tabela 6.1 apresenta a posição de cada polígono em relação ao eixo *z*. A diferença de profundidade é muito pequena e foi considerada desta forma para demonstrar os problemas de *rendering* gerados quando não verifica-se a visibilidade das superfícies. Quando a geração dos polígonos da cena não é feita em ordem de profundidade, o resultado obtido costuma ser diferente do esperado, como mostra a Figura 6.6 - (d). Neste caso, de acordo com a Tabela 6.1, o polígono azul deveria estar na frente do verde. Isto ocorre porque os pixels do polígono verde que são encobertos pelo polígono azul não foram removidos da cena.

Polígono	Eixo Z
Amarelo	0.0
Verde	0.02
Vermelho	0.05
Azul	0.06

Tabela 6.1: Posicionamento de cada polígono na cena.

Entre os estágios *(d)* e *(e)* aplicou-se o algoritmo de *z-buffer*, mantendo a mesma ordem de renderização. Após a execução do algoritmo e da aplicação de cinquenta por cento de transparência na cena, observa-se que não possuem mais pixels atrás do polígono azul e das demais partes sobrepostas (Figura 6.6 - (e)). Tornando os objetos da cena novamente opacos obtém-se a cena real *(f)*, com cada objeto posicionado corretamente em relação à sua profundidade.

Além do algoritmo *z-buffer*, o algoritmo de *clipping* circular, que foi desenvolvido no decorrer do trabalho, também foi aplicado para retirar as partes dos polígonos que extravasam a janela de visualização. A circunferência branca da cena é a área visível, tudo que extrapolar seu limite não deverá ser visualizado. Os estágios *(a)*, *(b)* e *(c)* apresentam o processo de retirada destes *pixels* invisíveis. O resultado obtido após estes passos demonstram a eficiência do algoritmo de recorte desenvolvido e apresentado no Capítulo 4.

6.3.1 Conclusão

Os polígonos da cena são formados por uma sequência de pontos, armazenados em arquivos, sob a forma *(x, y, z)*. Os quatro polígonos juntos totalizam trinta e seis mil oitocentos e setenta e oito pontos. A partir dos dados apresentados pela Tabela 6.2, fica claro que após efetuar as operações de *clipping* e remoção de superfície escondida o número de pontos renderizados nesta cena reduziu significativamente, passou a ser quatorze mil novecentos e um pontos a serem desenhados. A Tabela 6.2 apresenta a quantidade exata de pontos de cada polígono, antes e depois de remover os *pixels* não vistos pelo observador.

Polígono	Pontos	Após Clipping	Após z-Buffer	Redução	Redução(%)
Amarelo	22801	5651	4650	18151	79.60
Verde	7701	7701	5266	2435	31.51
Azul	2500	2500	1715	785	31.40
Vermelho	3876	3876	3270	606	15.63
Total	36878	19728	14901	21977	59.60

Tabela 6.2: Tabela de resultados

Tanto o cálculo da profundidade utilizado pelo *z-buffer*, quanto o cálculo da distância de cada ponto ao centro da circunferência que contorna a janela de *clipping* são feitos com auxílio do dispositivo gráfico, ou seja, utiliza-se de aceleradores gráficos para executar tais operações. Assim, levando em consideração o tempo de desenho dos pontos que foram descartados, aliado ao aumento de realismo na cena, conclui-se que a aplicação destes é viável a qualquer cena que tenha como objetivo representar visualmente um fato do cotidiano.

Capítulo 7

Conclusão

O problema de visualização de sólidos resume-se a manipular elementos geométricos e atributos de aparência dos objetos, com o objetivo de gerar, rapidamente e com o menor esforço computacional possível, cenas que mantenham a noção de tridimensionalidade, mesmo após serem projetadas em dispositivos de visualização bidimensionais. Os componentes básicos deste problema são: a imagem (tratada como um conjunto de *pixels*), os elementos geométricos representados na cena e as operações que os transformam em imagens. Com isso, um sólido pode ser considerado uma representação computacional de uma entidade que encapsula atributos visuais e descrições geométricas.

A câmera sintética é a responsável por efetuar a transformação entre o sistema de coordenadas global e o sistema de coordenadas de visualização. Faz-se esta conversão através das transformações geométricas de translação, rotação e escala. Cada uma destas transformações é efetuada através da multiplicação das coordenadas de cada ponto do objeto por uma matriz. A composição destas, reduz o tempo de processamento do sistema de câmeras de um ambiente. É este sistema que permite o movimento em um palco virtual. Assim, sem a utilização correta do mesmo, não é possível transmitir a imagem desejada.

Apenas transformar a cena para o sistema de coordenadas de visualização nem sempre é suficiente para exibí-la no monitor, pois este é um dispositivo 2D. Portanto, torna-se necessário a transformação para o sistema de coordenadas bidimensional com a aplicação de técnicas de projeção gráfica. Esta reduz em uma unidade a dimensão do objeto; no caso da computação gráfica, do sistema 3D para o 2D. A escolha do tipo de projeção a ser usada depende de cada aplicação. Sistemas destinados à engenharia e arquitetura, por exemplo, devem utilizar a projeção paralela. Já sistemas destinados a entretenimento, tais como jogos, filmes animados e realidade virtual, a projeção perspectiva é a mais indicada, pois gera imagens semelhante ao sistema de visão humano, onde quanto mais distante um objeto localiza-se do observador, menor ele é desenhado. Neste estágio, as coordenadas são projetadas no plano de projeção, mapeando-as para o *viewport*.

Quando efetua-se este mapeamento, muitos pontos são colocados além do limite do *viewport*

(ou janela de visualização ou ainda janela de *clipping*). Quando este fato ocorre, os pontos são desenhados desnecessariamente, pois o observador não irá vê-los em seu monitor. Se forem poucos os pontos, o processamento destes não será percebido pelo espectador. No entanto, se forem muitos, o tempo de execução aumenta e começa a interferir na eficiência do sistema. Para amenizar este problema, utiliza-se uma técnica para remover estes pontos do ambiente virtual. Esta técnica chama-se *clipping*.

No processo de *clipping*, os pontos dos objetos são comparados com o contorno da janela de visualização para verificar se são internos a ela ou não. Os que localizam-se dentro da área visível são desenhados e os demais descartados. Durante o desenvolvimento deste trabalho, dois algoritmos para recorte utilizando-se de um *viewport* circular foram desenvolvidos e o algoritmo para recorte de linhas atingiu tempo de processamento semelhante aos métodos utilizados em janelas retangulares. O algoritmo de recorte de superfícies obtém resultados visuais satisfatórios, porém este ainda pode ser melhorado em relação ao tempo de execução. Para isto pode-se utilizar a extração das arestas dos polígonos, recortando-as e depois preenchendo os novos polígonos gerados com o corte.

Para determinar os segmentos de reta que devem ser mostrados, utiliza-se as equações da reta e da circunferência para encontrar os pontos de interseção entre a linha a ser desenhada e a circunferência que contorna a janela de *clipping*. Já no método para recorte de faces, foi adotada a idéia dos principais algoritmos de remoção de superfícies escondidas, onde compara-se cada ponto dos objetos que compõem a cena para determinar sua visibilidade (espaço da imagem). Neste caso, os resultados obtidos foram satisfatórios, funcionando para qualquer tipo de superfície. No entanto, é um processo um pouco lento e tem sua otimização proposta como possível trabalho futuro.

Estes três processos são aplicados para permitir a visualização no monitor e a redução do tempo de processamento das cenas virtuais. O último tópico abordado neste trabalho também teve como um dos seus objetivos melhorar o desempenho do processo de visualização gráfica. Porém, seu foco principal é aumentar o realismo dos ambientes virtuais. Trata-se da detecção e remoção das superfícies ocultas da cena, pois se isto não for feito, os objetos podem aparecer em ordem inversa; por exemplo, se dois corpos estiverem muito próximos ou se interceptarem, dependendo da ordem de renderização destes, o objeto mais ao fundo e que está encoberto por outro pode aparecer na frente.

Para evitar problemas como este, deve-se remover os pontos que encontram-se encobertos por outros. Para isto, o algoritmo de *z-buffer* está entre os mais utilizados. Ele trabalha sobre o espaço da imagem e compara cada ponto da cena com todos os outros que estão na mesma linha de visão. Esta comparação é feita via dispositivos de aceleração gráfica, o que aumenta a velocidade de processamento.

Para finalizar, ficou claro que o importante na pesquisa em computação gráfica não é escolher um pacote gráfico, estudá-lo, aprender utilizar suas funções e começar a modelar cenas

baseadas em entidades reais ou imaginárias. O principal é saber os fundamentos que estão por trás de cada função da biblioteca gráfica e procurar entender porque cada fundamento surgiu, para que serve, quais são suas vantagens e o que deve ser feito para torná-los mais eficientes.

Capítulo 8

Trabalhos Futuros

Este trabalho abordou uma parte do *pipeline* de visualização gráfica (uma vez que o *pipeline* completo é composto também por iluminação, aplicação de texturas, antialiasing, etc.), mostrou seus principais fundamentos e apresentou uma técnica pouco trabalhada pelas literaturas, o *clipping* utilizando janelas de visualização circular. O algoritmo para recorte de superfícies apresentado, pode ser melhorado através do recorte das arestas dos polígonos que compõem a cena. Para isto, deve-se desmembrar as arestas, recortá-las individualmente e depois preencher os novos polígonos obtidos.

As projeções perspectiva e paralela são projeções lineares. Pouca gente já ouviu falar sobre projeção não-linear. Nesta, a imagem final contida no plano de visualização é composta por informações capturadas por mais de uma câmera; por exemplo, captura-se duas ou três fotografias de um objeto, cada uma vista por um ângulo diferente, projeta-se estas e obtém-se uma única imagem com mais detalhes em dois ou três lados. Dependendo da quantidade de imagens utilizadas, pode não ser muito realista. Porém, pode ser útil em aplicações que visam montar uma imagem bidimensional que contenha mais informações visuais do que a obtida pelas projeções lineares.

Referências

Adams, J. A., editor (1988). *Mathematical elements for computer graphics*. McGraw-Hill, Inc., New York, NY, USA.

Azevedo, E. e Conci, A. (1986). *Computação Gráfica Teoria e Prática*. Editora Campus, Upper Saddle River, NJ, USA.

Battaiola, A. L. e Erthal, G. (1998). *Projeções e o seu uso em Computação Gráfica*. XVII Jornada de Atualização em Infirmática da SBC.

Bhattacharya, B., Kirkpatrick, D., e Toussaint, G. (1989). Determining sector visibility of a polygon. In *SCG '89: Proceedings of the fifth annual symposium on Computational geometry*, pages 247–253, New York, NY, USA. ACM.

Brunstein, D., Barequet, G., e Gotsman, C. (2003). Animating a camera for viewing a planar polygon. In Ertl, T., editor, *VMV*, pages 87–94. Aka GmbH.

Bueno, F. S. (1998). *Minidicionário Silveira Bueno*.

Crocker, G. A. (1984). Invisibility coherence for faster scan-line hidden surface algorithms. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 95–102, New York, NY, USA. ACM.

da Silva, M. H. (2002). Tratamento eficiente de visibilidade através de Árvores de volumes envolventes. Master's thesis, Pontifícia Universidade Católica do Rio de Janeiro.

da Silva, V. V. e dos Reis, G. L. (1996). *Geometria Analítica*. Livros Técnicos e Científicos.

Drucker, S. M. e Zeltzer, D. (1994). Intelligent camera control in a virtual environment. In *Proceedings of Graphics Interface '94*, pages 190–199, Banff, Alberta, Canada.

Foley, J. D., Phillips, R. L., Hughes, J. F., van Dam, A., e Feiner, S. K. (1994). *Introduction to Computer Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Foley, J. D., van Dam, A., Feiner, S. K., e Hughes, J. F. (1990). *Computer graphics: principles and practice (2nd ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Gomes, J. e Velho, L. (1998). *Computação Gráfica, Volume 1*. IMPA.

Greene, N., Kass, M., e Miller, G. (1993). Hierarchical z-buffer visibility. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 231–238, New York, NY, USA. ACM.

Greiner, G. e Hormann, K. (1998). Efficient clipping of arbitrary polygons. *ACM Trans. Graph.*, 17(2):71–83.

Han, X. (2004). *The local z buffer algorithms in real time rendering of large complex scenes*. PhD thesis, Iowa City, IA, USA. Supervisor-James Cremer.

Harrington, S. (1987). *Computer graphics: a programming approach, 2nd ed.* McGraw-Hill, Inc., New York, NY, USA.

Hearn, D. e Baker, M. P. (1986). *Computer graphics*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Hill, F. J. (2000). *Computer Graphics Using OpenGL.* Prentice Hall PTR, Upper Saddle River, NJ, USA.

Kolb, C., Mitchell, D., e Hanrahan, P. (1995). A realistic camera model for computer graphics. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 317–324, New York, NY, USA. ACM.

Kumar, S. e Manocha, D. (1996). Hierarchical back-face culling. Technical report, Chapel Hill, NC, USA.

Lam, H., Rensink, R. A., e Munzner, T. (2006). Effects of 2d geometric transformations on visual memory. In *APGV '06: Proceedings of the 3rd symposium on Applied perception in graphics and visualization*, pages 119–126, New York, NY, USA. ACM.

Marquis-Bolduc, M., Deschênes, F., e Pan, W. (2008). Combining apparent motion and perspective as visual cues for content-based camera motion indexing. *Pattern Recogn.*, 41(2):445–457.

Novins, K. L., Sillion, F. X., e Greenberg, D. P. (1990). An efficient method for volume rendering using perspective projection. In *VVS '90: Proceedings of the 1990 workshop on Volume visualization*, pages 95–102, New York, NY, USA. ACM.

Ogata, M., Ohkami, T., Lauer, H. C., e Pfister, H. (1998). A real-time volume rendering architecture using an adaptive resampling scheme for parallel and perspective projections. In *VVS '98: Proceedings of the 1998 IEEE symposium on Volume visualization*, pages 31–38, New York, NY, USA. ACM.

Rogers, D. F. (1985). *Procedural elements for computer graphics*. McGraw-Hill, Inc., New York, NY, USA.

Sechrest, S. e Greenberg, D. P. (1981). A visible polygon reconstruction algorithm. In *SIGGRAPH '81: Proceedings of the 8th annual conference on Computer graphics and interactive techniques*, pages 17–27, New York, NY, USA. ACM.

Sharir, M. e Overmars, M. H. (1992). A simple output-sensitive algorithm for hidden surface removal. *ACM Trans. Graph.*, 11(1):1–11.

Spindler, M., Bubke, M., Germer, T., e Strothotte, T. (2006). Camera textures. In *GRAPHITE '06: Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia*, pages 295–302, New York, NY, USA. ACM.

Sutherland, I. E. (1964). Sketch pad a man-machine graphical communication system. In *DAC '64: Proceedings of the SHARE design automation workshop*, pages 6.329–6.346, New York, NY, USA. ACM.

Sutherland, I. E. (1998). A head-mounted three dimensional display. pages 295–302.

Sutherland, I. E. e Hodgman, G. W. (1974). Reentrant polygon clipping. *Commun. ACM*, 17(1):32–42.

Traina, A. J. M. e de Oliveira, M. C. F. (2003). *Apostila de Computação Gráfica*. ICMC - USP.

Tsai, Y.-Y. e Wang, C.-M. (2007). A novel data hiding scheme for color images using a bsp tree. *J. Syst. Softw.*, 80(3):429–437.

Wang, G., Tsui, H.-T., e Hu, Z. (2007). Structure and motion of nonrigid object under perspective projection. *Pattern Recogn. Lett.*, 28(4):507–515.

Ware, C. e Osborne, S. (1990). Exploration and virtual camera control in virtual three dimensional environments. In *SI3D '90: Proceedings of the 1990 symposium on Interactive 3D graphics*, pages 175–183, New York, NY, USA. ACM.

Westermann, R. e Ertl, T. (1998). Efficiently using graphics hardware in volume rendering applications. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 169–177, New York, NY, USA. ACM.

Yang, J. C., Everett, M., Buehler, C., e McMillan, L. (2002). A real-time distributed light field camera. In *EGRW '02: Proceedings of the 13th Eurographics workshop on Rendering*, pages 77–86, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

Apêndice A

Código Fonte

A.1 Código Fonte do Protótipo

```
1  //@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ Inserção das Bibliotecas @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@//
2  #include <stdio.h>
3  #include <conio.h>
4  #include <stdio.h>
5  #include <math.h>
6  #include <windows.h>
7  #include <GL/GLUT.h>
8  #include <GL/GLU.h>
9  #include <GL/GL.h>
10 #include <GL/GLAUX.h>
11 #include "formasgeo.h"
12
13 //@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ Definição das funções @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@//
14 void leArquivo(char *);
15 void Exibe(void);
16 float distanciaPontos(Ponto2D *, Ponto2D *);
17 void ClippingCircular(char *arq);
18 void desenhaMatriz(bool);
19 void zBuffer(char *);
20 void JanelaClipping(void);
21
22 //@@@@@@@@@@@@@@@@@@@ Variáveis globais utilizadas pelo protótipo @@@@@@@@@@@@@@@@@@@//
23 // Matriz que conterá a profundidade de cada pixel para auxiliar na execução
24 // do algoritmo de z-buffer
25 GLfloat zBufferMat[500][500];
26 // Matriz que conterá a cor de cada pixel visível na cena
27 Ponto3D *corMat[500][500];
28 // Matriz auxiliar que conterá a cor de cada pixel visível na cena. Esta matriz
29 // é utilizada para limpar redesenha a tela quando necessário
30 Ponto3D *corMataux[500][500];
31
32 /*
33  * Função principal do programa
34  */
35 void main(int argc, char *argv[]){
36    // Inicializa a janela
37    glutInit(&argc, argv);
38    // Define o modo de exibição da janela
39    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
40    // Define o tamanho da janela
41    glutInitWindowSize(500,500);
42    // Define a posição inicial da janela
43    glutInitWindowPosition(150,100);
44    // Cria a janela com o título passado no parâmetro
45    glutCreateWindow("PFC - Alex Araújo");
46    // Exibe o conteúdo da janela
47    glutDisplayFunc(Exibe);
48    // Define a cor e a transparência do fundo da janela
49    glClearColor(1.0, 1.0, 1.0, 0.0);
```

```
50      // Define a cor inicial dos pontos da jenela
51      glColor3f(1.0, 1.0, 1.0);
52      // Define a espessura dos pontos a serem desenhados na janela
53      glPointSize(1.0);
54      // Define qual é a forma da matriz corrente
55      glMatrixMode(GL_PROJECTION);
56      // Substitui a matriz corrente pela matriz Identidade (4x4)
57      glLoadIdentity();
58      // Força o refresh da janela a todo momento
59      glutMainLoop();
60  }
61
62  /*
63   * Função que chama as funções que exibirão os pontos na cena
64   */
65  void Exibe(void){
66      // Ativa o OpenGL para aceitar o fator alpha, para que seja possível colcoar
67      // um fator de transparência no objetos da mesma
68      glEnable(GL_BLEND);
69      glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
70      // Limpa a tela
71      glClear(GL_COLOR_BUFFER_BIT);
72      // Desenha a janela de clipping na cor branca, para permitir uma melhor
73      // visualização do processo de remoção dos pontos ocultos
74      JanelaClipping();
75      // Desenha os polígonos sem remover os pontos escondidos
76      glColor4f(1.0, 1.0, 0.0, 0.5);
77      leArquivo("quadrado.txt");
78      glColor4f(0, 0, 1, 0.5);
79      leArquivo("circulo_inverso.txt");
80      glColor4f(1, 0, 0, 0.5);
81      leArquivo("losango.txt");
82      glColor4f(0, 1, 0, 0.5);
83      leArquivo("retangulo.txt");
84      // Chama a função que faz o clipping circular da cena
85      ClippingCircular("quadrado.txt");
86      glFlush();
87      // Pausa a execução do programa por um tempo
88      _sleep(100);
89      ClippingCircular("circulo_inverso.txt");
90      glFlush();
91      _sleep(100);
92      ClippingCircular("losango.txt");
93      glFlush();
94      _sleep(100);
95      ClippingCircular("retangulo.txt");
96      // Desenha a matriz de intensidade usada pelo algoritmo de z-buffer
97      // neste ponto a mesma não possui os pontos do polígono
98      desenhaMatriz(false);
99      // Chama a função com o algoritmo de z-buffer para remover os pontos
100     // encobertos da cena
101     zBuffer("quadrado.txt");
102     glFlush();
103     zBuffer("circulo_inverso.txt");
104     glFlush();
105     zBuffer("losango.txt");
106     glFlush();
107     zBuffer("retangulo.txt");
108     glFlush();
109     // Desenha a matriz final contendo os pontos válidos
110     desenhaMatriz(true);
111 }
112
113 /*
114  * Função que desenha a janela de clipping
115  */
116 void JanelaClipping(){
117     float cont = 0;
118     float raio = 0.600;
119     Ponto2D *centro = new Ponto2D();
120     float xaux, yaux;
121     glBegin(GL_POINTS);
122     for(float i = -250; i < 250; i++){
123         for(float j = -250; j < 250; j++){
```

```
124           xaux = i / 250;
125           yaux = j / 250;
126           if(distanciaPontos(centro, new Ponto2D(xaux, yaux)) >= (raio + 0.003)){
127              glColor4f(0.0, 0.0, 0.0, 1.0);
128              glVertex3f(xaux, yaux, 0.0);
129              glPopMatrix();
130              //inicia os pixels com a cor preta
131              corMat[(int)i+250][(int)j+250] = new Ponto3D(0.0, 0.0, 0.0);
132              //inicia os pixels com a cor preta
133                          corMataux[(int)i+250][(int)j+250] = new Ponto3D(0.0, 0.0, 0.0);
134                    }
135           else{
136              //inicia os pixels com a cor branca
137              corMat[(int)i+250][(int)j+250] = new Ponto3D(1.0, 1.0, 1.0);
138              //inicia os pixels com a cor branca
139                          corMataux[(int)i+250][(int)j+250] = new Ponto3D(1.0, 1.0, 1.0);
140                    }
141           //inicia todos os pixels com profundidade -999
142                   zBufferMat[(int)i+250][(int)j+250] = (float)-9999.0;
143           glFlush();
144            }
145     }
146     glEnd();
147  }
148
149  /*
150   * Função que lê os pontos armazenados em um arquivo recebido como parâmetro
151   * de plota na tela
152   */
153  void leArquivo(char *arq){
154     char caracter;
155     float coordxponto, coordyponto, coordzponto;
156     float raio = 0.500;
157     Ponto2D *centro = new Ponto2D();
158     FILE *arquivo;
159     if((arquivo = fopen(arq,"r")) == NULL){
160        printf("Erro ao abrir arquivo!!!\n\n");
161        exit(1);
162     }
163     rewind(arquivo);
164     glBegin(GL_POINTS);
165     do{
166        fscanf(arquivo,"%f",&coordxponto);
167        fscanf(arquivo,"%f",&coordyponto);
168        fscanf(arquivo,"%f",&coordzponto);
169        glVertex3f(coordxponto, coordyponto, coordzponto);
170        fscanf(arquivo,"%c",&caracter);
171     } while(caracter != 'f');
172     glEnd();
173     fclose(arquivo);
174     glFlush();
175  }
176
177  /*
178   * Função que efetua o clipping circular
179   */
180  void ClippingCircular(char *arq){
181     char caracter;
182     float coordxponto, coordyponto, coordzponto;
183     float raio = 0.600;
184     float corr, corg, corb;
185     Ponto2D *centro = new Ponto2D();
186     FILE *arquivo;
187     if((arquivo = fopen(arq,"r")) == NULL){
188        printf("Erro ao abrir arquivo!!!\n\n");
189        exit(1);
190     }
191     if(arq == "retangulo.txt"){
192        corr = 0.0;
193        corg = 1.0;
194        corb = 0.0;
195     }
196     else if(arq == "losango.txt"){
197        corr = 1.0;
```

```
198         corg = 0.0;
199         corb = 0.0;
200      }
201      else if(arq == "circulo_inverso.txt"){
202         corr = 0.0; corg = 0.0; corb = 1.0;
203      }
204      else if(arq == "quadrado.txt"){
205         corr = 1.0; corg = 1.0; corb = 0.0;
206      }
207      rewind(arquivo);
208      glBegin(GL_POINTS);
209      do{
210         fscanf(arquivo,"%f",&coordxponto);
211         fscanf(arquivo,"%f",&coordyponto);
212         fscanf(arquivo,"%f",&coordzponto);
213         if(distanciaPontos(centro, new Ponto2D(coordxponto, coordyponto)) >= (raio + 0.003)){
214            glColor4f(0.0, 0.0, 0.0, 1.0);
215            glVertex3f(coordxponto, coordyponto, coordzponto);
216             }
217         glFlush();
218         _sleep(1);
219         fscanf(arquivo,"%c",&caracter);
220      } while(caracter != 'f');
221      glEnd();
222      fclose(arquivo);
223   }
224
225   /*
226    * Método z-buffer para remover pontos encobertos
227    */
228   void zBuffer(char *arq){
229      char caracter;
230      float coordxponto, coordyponto, coordzponto;
231      float raio = 0.600;
232      float xaux, yaux;
233      Ponto2D *centro = new Ponto2D();
234      FILE *arquivo;
235      if((arquivo = fopen(arq,"r")) == NULL){
236         printf("Erro ao abrir arquivo!!!\n\n");
237         exit(1);
238      }
239      rewind(arquivo);
240      glBegin(GL_POINTS);
241      do{
242         fscanf(arquivo,"%f",&coordxponto);
243         fscanf(arquivo,"%f",&coordyponto);
244         fscanf(arquivo,"%f",&coordzponto);
245         if(distanciaPontos(centro, new Ponto2D(coordxponto, coordyponto)) <= (raio + 0.003)){
246            xaux = coordxponto * 250 + 250;
247            yaux = coordyponto * 250 + 250;
248            if(zBufferMat[(int)xaux][(int)yaux] < coordzponto){
249               zBufferMat[(int)xaux][(int)yaux] = coordzponto;
250               if(arq == "retangulo.txt")
251                  corMat[(int)xaux][(int)yaux]->setPonto(0,1,0);
252               else if(arq == "losango.txt")
253                  corMat[(int)xaux][(int)yaux]->setPonto(1,0,0);
254               else if(arq == "quadrado.txt")
255                  corMat[(int)xaux][(int)yaux]->setPonto(1.0,1.0,0.0);
256               else if(arq == "circulo_inverso.txt")
257                  corMat[(int)xaux][(int)yaux]->setPonto(0,0,1);
258                    }
259             }
260
261         glFlush();
262         fscanf(arquivo,"%c",&caracter);
263      } while(caracter != 'f');
264      glEnd();
265      desenhaMatriz(false);
266      fclose(arquivo);
267   }
268
269   /*
270    * Função que calcula e retorna a distância entre dois pontos
271    */
```

```
272 float distanciaPontos(Ponto2D *c, Ponto2D *p){
273    float d;
274    d = sqrt(pow(c->getX() - p->getX(), 2) + pow(c->getY() - p->getY(), 2));
275    return d;
276 }
277
278 /*
279  * Função que desenha a cena baseada nas informações de cores contidas na matriz
280  * corMat
281  */
282 void desenhaMatriz(bool transparencia){
283    float xaux, yaux;
284    float transp = 1.0;
285    float r, g, b;
286    if(transparencia)
287       transp = 0.5;
288    for(float i = -250; i < 250; i++){
289       for(float j = -250; j < 250; j++){
290          r = corMat[(int)i+250][(int)j+250]->getX();
291          g = corMat[(int)i+250][(int)j+250]->getY();
292          b = corMat[(int)i+250][(int)j+250]->getZ();
293          glBegin(GL_POINTS);
294          if(distanciaPontos(new Ponto2D(0.0, 0.0), new Ponto2D(i/250, j/250)) <= 0.603)
295             glColor4f(r ,g , b, transp);
296          else
297             glColor4f(r ,g , b, 0.0);
298          xaux = i / 250;
299          yaux = j / 250;
300          _sleep(100);
301          glVertex3f(xaux, yaux, 0);
302          glEnd();
303          glFlush();
304           }
305    }
306    _sleep(1000);
307 }
```

A.2 Biblioteca formasgeo.h

```
1  //@@@@@@@@@@@@@@@@@@ Inserção das Bibliotecas @@@@@@@@@@@@@@@@@@//
2  #include <windows.h>
3  #include <assert.h>
4  #include <math.h>
5  #include <stdlib.h>
6  #include <gl/Gl.h>
7  #include <gl/Glu.h>
8  #include <gl/glut.h>
9
10 //@@@@@@@@@@@@@@@@@@@@@@ Classe Ponto 2D @@@@@@@@@@@@@@@@@@@@@@//
11
12 /**
13  * O objeto Ponto2D é um ponto bidimensional, formado pelas
14  * coordenadas x e y.
15  */
16 class Ponto2D{
17
18 //************************************************************//
19 private:
20    GLfloat x, y;
21
22 //************************************************************//
23 public:
24
25    /*
26     * Construtor da classe que cria um ponto na origem do
27     * sistema de coordenadas
28     */
29    Ponto2D(){
30       x = y = 0.0f;
31    }
32
```

```
33      /*
34       * Construtor da classe que cria um ponto na posição (x, y)
35       */
36      Ponto2D(GLfloat x, GLfloat y){
37         this->x = x;
38         this->y = y;
39      }
40
41      /*
42       * Retorna a coordenada x do ponto
43       */
44      float getX(){ return x;}
45
46      /*
47       * Retorna a coordenada y do ponto
48       */
49      float getY(){ return y;}
50
51      /*
52       * Altera o valor da coordenada x pelo valor recebido como
53       * parâmetro
54       */
55      void setX(GLfloat x){ this->x = x;}
56
57      /*
58       * Altera o valor da coordenada y pelo valor recebido como
59       * parâmetro
60       */
61      void setY(GLfloat y){ this->y = y;}
62
63      /*
64       * Altera as coordenadas x e y pelos valores recebidos como
65       * parâmetros
66       */
67      void setPonto(GLfloat x, GLfloat y){
68         this->x = x;
69         this->y = y;
70      }
71
72      /*
73       * Altera as coordenadas x e y pelas coordenadas do ponto
74       * recebido como parâmetro
75       */
76      void setPonto(Ponto2D *p){
77         this->x = p->getX();
78         this->y = p->getY();
79      }
80  };
81
82  //@@@@@@@@@@@@@@@@@@@@@@ Classe Ponto 3D @@@@@@@@@@@@@@@@@@@@@@@@//
83
84  /**
85   * O objeto Ponto3D é um ponto tridimensional, formado pelas
86   * coordenadas x, y e z.
87   */
88  class Ponto3D{
89
90  //*************************************************************//
91  private:
92     GLfloat x, y, z;
93
94  //*************************************************************//
95  public:
96
97      /*
98       * Construtor da classe que cria um ponto na origem do
99       * sistema de coordenadas
100      */
101     Ponto3D(){
102        x = y = z = 0.0f;
103     }
104
105     /*
106      * Construtor da classe que cria um ponto na posição x, y e z
```

```
107       * sistema de coordenadas
108       */
109      Ponto3D(GLfloat x, GLfloat y, GLfloat z){
110         this->x = x;
111         this->y = y;
112         this->z = z;
113      }
114
115      /*
116       * Retorna a coordenada x do ponto
117       */
118      float getX(){ return x;}
119
120      /*
121       * Retorna a coordenada y do ponto
122       */
123      float getY(){ return y;}
124
125      /*
126       * Retorna a coordenada z do ponto
127       */
128      float getZ(){ return z;}
129
130      /*
131       * Altera o valor da coordenada x pelo valor recebido como
132       * parâmetro
133       */
134      void setX(GLfloat x){ this->x = x;}
135
136      /*
137       * Altera o valor da coordenada y pelo valor recebido como
138       * parâmetro
139       */
140      void setY(GLfloat y){ this->y = y;}
141
142      /*
143       * Altera o valor da coordenada z pelo valor recebido como
144       * parâmetro
145       */
146      void setZ(GLfloat z){ this->z = z;}
147
148      /*
149       * Altera as coordenadas x, y e z pelos valores recebidos como
150       * parâmetros
151       */
152      void setPonto(GLfloat x, GLfloat y, GLfloat z){
153         this->x = x;
154         this->y = y;
155         this->z = z;
156      }
157   };
158
```