Simulação Gráfica de Modelos Estocásticos com Suporte Volumétrico

por

Marcela Souto Maior da Silva

DCC/IM/NCE – UFRJ

2006

Marcela Souto Maior da Silva

Simulação Gráfica de Modelos Estocásticos com Suporte Volumétrico

Dissertação de Mestrado Submetida ao Programa de Pós-Graduação em Informática do Instituto de Matemática/Núcleo de Computação Eletrônica da Universidade Federal do Rio de Janeiro - UFRJ, como parte dos requisitos necessários para obtenção do título de mestre em Ciências em Informática.

Orientador: Ernesto Prado Lopes
Co-orientador: Eliana Prado Lopes Aude

Rio de Janeiro, RJ – Brasil
Fevereiro – 2006

AGRADECIMENTOS

Devo os meus agradecimentos a um conjunto de pessoas que contribuíram de forma concreta para a realização deste trabalho. Mesmo com todos os obstáculos, que não foram poucos, foi possível provar que o sucesso de um projeto se deve à força e à vontade daqueles que participam, ou mesmo torcem, para a sua conclusão.

Para estas pessoas, minha eterna gratidão.

À professora Eliana Aude, que me deu a valiosa oportunidade de participar do grupo CONTROLAB, do qual obtive não só conhecimentos técnicos, mas também contribuiu para a minha formação pessoal.

Ao meu orientador, Ernesto Lopes, pela honra de ter compartilhado comigo seus conhecimentos e por ter me encorajado a seguir nos meus estudos durante o mestrado.

Ao meu amigo e colega Flavio Signorelli, meu companheiro em todas as horas, sempre me incentivando a prosseguir com persistência durante a nossa vida acadêmica. Obrigada por estar ao meu lado.

Aos meus amigos e colegas Júlio Tadeu e Henrique Serdeira, devo também a vocês os meus agradecimentos na construção deste trabalho. Obrigada pelo apoio que sempre obtive de vocês.

Aos outros integrantes do grupo CONTROLAB e à todos os meus colegas de mestrado como a amiga Patricia Lopes, muito obrigada estarem comigo.

Aos meus pais, Wiliams Júlio da Silva e Sônia Maria Souto Maior da Silva, obrigada pela participação e incentivo nos meus estudos. Devo o sucesso deste trabalho a vocês.

À minha irmã, Marília Souto Maior da Silva, e à toda a minha família, agradeço pelo apoio e carinho constantes.

E a Deus, por tudo.

RESUMO

Simulação Gráfica de Modelos Estocásticos com Suporte Volumétrico

Marcela Souto Maior da Silva

Orientador: Ernesto Prado Lopes

Este trabalho consiste na definição e na implementação de um sistema para a simulação e visualização de Processos Pontuais Indexados *ou Marked Point Processes*, mais especificamente, dos processos deste tipo, onde o Processo Pontual envolvido é um Processo de Poisson homogêneo ou não homogêneo, com suporte espacial tridimensional. Neste trabalho o volume do espaço que define o suporte do *Processos Pontuais Indexados* é a união de paralelepípedos cuja a interseção de seus interiores é vazia. Uma linguagem foi desenvolvida para a especificação, pelo usuário, dos volumes onde a simulação é feita. Ela impõe algumas restrições, esperamos não muito fortes para a maioria das aplicações práticas, na disposição dos paralelepípedos, além da restrição de não interseção já mencionada. Foi criada uma rotina de geração de números aleatórios para o sistema, evitando-se usar as rotinas de geração de números aleatórios fornecidas pelos fabricantes de computadores que, em geral, não são confiáveis. Para isso, procurou-se seguir as recomendações da literatura especializada, tanto na definição do algoritmo de geração de números pseudo-aleatórios, como no cuidado com a validação das simulações através de testes estatísticos de ajuste e de independência. Uma realização do processo de Poisson homogêneo ou não-homogêneo espacial é uma população de pontos no volume definido. A visualização destes pontos

permite avaliar a intensidade, a dispersão e posicionamento dos possíveis agrupamentos de pontos, etc. Depois, por um processo de coloração destes pontos, ou de vizinhanças deles é possível ter-se uma idéia clara da realização do *Processos Pontuais Indexados* simulado. Para a visualização gráfica da simulação, usou-se a biblioteca gráfica *OpenGL* que é atualmente amplamente utilizada em sistemas de computação gráfica. O resultado visual das simulações, obtido através do subsistema de visualização, permite um acesso fácil, rápido e abrangente das informações sobre o modelo. É possível rotacionar o volume em torno dos eixos x, y e z, transladá-lo, ampliá-lo ou diminuí-lo e os recursos de iluminação permitem destacar o caráter tridimensional da imagem. Foi implementada a opção de obter e visualizar a envoltória convexa do volume definido para a simulação. A envoltória convexa é construída a partir dos vértices dos blocos que compõem este volume. Para isso foi programado o algoritmo proposto em [MdBS98]. Em geral, o volume definido pelos blocos é apenas uma aproximação do verdadeiro corpo em que o fenômeno estudado acontece. A envoltória convexa deste volume é importante para avaliação da diferença entre o volume definido para simulação e o volume real. O último capítulo desta tese traz uma avaliação do trabalho feito e propostas de trabalhos futuros.

ABSTRACT

Graphic Simulation of Stochastic Models With Volumetric Suport

Marcela Souto Maior da Silva

Supervisor: Ernesto Prado Lopes

This work consists of a definition and an implementation of a Marked Point Processes visualization and simulation system, specially about Point Processes like homogeneous and non-homogeneous Poisson Processes with tridimensional suport. The space volume that defines the Marked Point Process is the union of blocks which interior intersections are null. We developed a kind of file that says to the system the volume´s dimension which tha simulation is applied. It has some restrictions that me WISH be not very strong. A random number generator rotine was created to be aplied on the system, because we want to avoid the random variables generator that exists in the most computer´s producers, which usually are untrustly. For this reason, we had to be carefull and we aplied the recommendation of specialized literature on pseudo-random variables generator algorithm and on the simulation validation by stathistical tests (hipothetical tests and tests for independence). One sample of homogeneous or non-homogeneous spatial Poisson Process is a point population in a defined volume. The view of these points allow the viewer to evaluate the intensity, dispersion and position of the possible points groups. After that, is possible to get a precise idea of the Point Process sample simutated, making use of a colouration process of these points. The simulation view is based on the OpenGL packet application, which is extensively used on computer graphic´s area. This result, that we also

call visualization subsystem, allow us to get a faster, easiear and more extensive informations about the model. It´s possible to rotate, translate, increase and decrease the volume. We used the ilumination resource to get the tridimensional feeling of the image. The user can also opt to view the convex hull of the simulation volume. For this work, the algorithm in [MdBS98] was implemented. Usually, the volume defined by the blocks is only the real size aproximation where tha phenomena happens. The importance of the convex hull is that it can show the diference between the simulation and the real volume. The last chapter of this work shows an evaluation of the present work and future proposes.

Lista de Figuras

4.1 Visão 3D 34
4.2 Translação de um objeto 40
4.3 Rotação de um objeto em torno do eixo z 46
4.4 O escalamento de um objeto 47
4.5 Projeção de uma imagem 49
4.6 A projeção perspectiva 50
4.7 A projeção paralela 50
4.8 Projeções de um cubo com um ponto de fuga 51
4.9 Cálculo da projeção perspectiva 52
4.10 Projeção oblíqua de um cubo (P' é a projeção de $P(0,0,1)$) 56
4.11 Projeção oblíqua de $P(0,0,1)$ em $P'(l\cos\alpha, l\sin\alpha, 0)$) 57
4.12 Projeção oblíqua de (x,y,z) em $(x_p,y_p,0)$ 57
4.13 A visão do observador e a normal da face 62
4.14 Spot de luz 64
4.15 Visualização gerada pelo aplicativo 68
4.16 Simulação vista somente com o domínio de entrada 69
4.17 Sistema de coordenadas padrão do OpenGL 71

5.1 Sentido anti-horário das arestas da face 78
5.2 A face f é visível a p, mas não é para q. 78
5.3 O *horizonte* de $\mathcal{CH}(P_{r-1})$ em relação a p_r. 79
5.4 A aresta $\overrightarrow{e}$ e twin($\overrightarrow{e}$). 80
5.5 Conectando as arestas do *horizonte* a p_r. 83
5.6 Inclusão da Face f 83
5.7 O fecho convexo gerado pelo algoritmo 5.3.1 86

Lista de Tabelas

2.1 Resultados da experiência com $n = 100$ rodadas do algoritmo 1 23
2.2 Resultado da experiência com $n = 140$ rodadas do algoritmo 2 26

4.1 Códigos do arquivo de entrada 70

Sumário

Lista de Figuras **7**

Lista de Tabelas **8**

1 Introdução **11**
1.1 Modelos Estocásticos com Suporte Espacial 11
1.2 Descrição do Trabalho 13

2 Geração de Números Aleatórios **14**
2.1 A Geração de Números Aleatórios 14
2.2 Teste de Hipóteses Estatísticas 15
2.2.1 O Valor P 18
2.2.2 Procedimento Geral para Testes de Hipóteses 18
2.2.3 A Distribuição Multinomial 19
2.3 Teste de Adequação de Ajuste 20
2.3.1 Teste de Ajuste para uma Distribuição Supostamente Uniforme 21
2.3.2 Teste de Ajuste para uma Distribuição Supostamente de Poisson 24
2.4 Teste de Independência 27

3 Processos Pontuais e Processos Pontuais Indexados **29**
3.1 Descrição dos Modelos Estocásticos 29
3.1.1 Modelagem dos Processos Pontuais de Poisson 30
3.2 Simulação dos Processos Pontuais e dos Processos Pontuais Indexados 31

4 Visualização Espacial **33**
4.1 O Ambiente Tridimensional 33
4.2 Transformações Geométricas 35
4.2.1 Coordenadas Homogêneas 35
4.2.2 Translação 40
4.2.3 Rotação 41
4.2.4 Escala 46
4.3 Projeção 48
4.3.1 Projeção Perspectiva 50

4.3.2 Projeção Paralela 54
4.4 O que é *OpenGL*? 57
4.4.1 Transformações e Perspectiva em OpenGL 59
4.4.2 Remoção de Superfícies Ocultas 60
4.4.3 A Iluminação em OpenGL 62
4.5 A Visualização Volumétrica Gerada pelo Aplicativo 66
4.5.1 Dimensão da grade 69
4.5.2 Ambiente de desenvolvimento 73

5 Fecho Convexo **74**
5.1 O Fecho Convexo no Plano e no Espaço 74
5.2 Computando o Fecho Convexo no Espaço Tridimensional 75
5.3 Estruturas e Algoritmo 79

6 Conclusão e Trabalhos Futuros **87**

Referências Bibliográficas **89**

Capítulo 1

Introdução

1.1 Modelos Estocásticos com Suporte Espacial

Os modelos estocásticos com suporte espacial são largamente usados em geologia, ciência dos solos, processamento de imagens, epidemiologia, agronomia, ecologia, astronomia, meteorologia, engenharia de reservatórios etc. Esses modelos são famílias de variáveis aleatórias indexadas por conjuntos não ordenados, como treliças e subconjuntos contínuos do R^n, definidos por processos aleatórios ou simplesmente fixos. Nestas áreas, lida-se com dados fortemente influenciados pela posição no espaço em que eles são coletados. Este fato aconselha o uso de tais modelos estocásticos com suporte espacial. Eles diferem dos modelos da estatística clássica (modelos de regressão, por exemplo) pelo fato de que neles a dependência espacial dos dados não é totalmente explicada pela parte determinística do modelo. Eles incluem também uma estrutura de covariâncias que depende da posição das variáveis aleatórias no espaço. Cressie [Cre93] descreve esta família de modelos como:

$$\{Z(x)|\ x \in D\}\,,$$

onde $Z(x)$ são variáveis aleatórias indexadas pelos elementos do conjunto D, que

é em geral não ordenado e que pode ser um conjunto fixo ou um conjunto aleatório. Ele considera três classes de modelos:

1. Os que são objeto de estudo da Geoestatística. Neste caso, D é um subconjunto do R^n que contém um retângulo com volume positivo, $Z(x)$ são variáveis Gaussianas e a estrutura de covariâncias é descrita pelo variograma ;

2. Os em que D é uma treliça. Estes têm uma relação com as funções aleatórias espaciais equivalente ao dos processos estocásticos a tempo discreto com os processos estocásticos a tempo contínuo;

3. Os em que D é uma realização de um processo pontual espacial em um subconjunto do R^n e $Z(x)$, para $x \in D$ é uma variável aleatória. Estes processos são em geral chamados de *Processos Pontuais Indexados*.

Em seu livro, Cressie descreve detalhadamente modelos nestas três classes e discute os avanços feitos até a data em que o livro foi escrito, no desenvolvimento de uma inferência estatística para esses modelos.

O presente trabalho tem como objetivo desenvolver um sistema computacional para a simulação de processos estocásticos com suporte espacial pertencentes à terceira classe descrita por Cressie, ou seja, para os *Processos Pontuais Indexados*. Mais especificamente, para aqueles em que o processo pontual envolvido é um processo de Poisson homogêneo ou não homogêneo e em que os parâmetros das variáveis aleatórias $Z(x)$ são uma função não aleatória de x.

A simulação deste tipo de processo, nas áreas descritas no início desta seção, é usada principalmente para a inferência sobre os parâmetros do modelo. Isso é feito usando-se os resultados das simulações e os dados reais disponíveis em cada caso. Estas questões não serão discutidas neste trabalho.

1.2 Descrição do Trabalho

Esse trabalho apresenta uma ferramenta para a simulação de *Processos Pontuais* e de *Processos Pontuais Indexados* e para a visualização gráfica tridimensional desta simulação, utilizando a biblioteca gráfica *OpenGL*. Foi criada uma linguagem que permite a definição do suporte espacial do processo a ser simulado, como a união de blocos cuja interseção de seus interiores é vazia. O sistema criado permite a visualização deste volume, assim como sua rotação, translação e escalamento. Para este volume é possível ainda achar e visualizar o seu fecho convexo.

A seguir é apresentado um resumo do conteúdo de cada um dos capítulos:

Capítulo 2 Nesse capítulo é discutida a geração de números aleatórios e são apresentados testes de hipóteses utilizados para a validação dos algoritmos de simulação e de suas implementações.

Capítulo 3 Nesse capítulo será apresentada a descrição dos *Processos Pontuais* e dos *Processos Pontuais Indexados*. Serão discutidos, mais detalhadamente os Processos de Poisson homogêneos e não-homogêneos, com suporte espacial, e os algoritmos para sua simulação.

Capítulo 4 Nesse capítulo serão apresentados aspectos teóricos da computação gráfica, a biblioteca de visualização gráfica OpenGL e uma descrição da estrutura do subsistema gráfico desenvolvido.

Capítulo 5 Nesse capítulo será descrito o algoritmo de geração do fecho convexo e a sua implementação.

Capítulo 6 Neste capítulo serão apresentadas as conclusões e propostas de extensão do trabalho.

Capítulo 2

Geração de Números Aleatórios

2.1 A Geração de Números Aleatórios

Qualquer simulação em computador de um sistema físico que envolve aleatoriedade deve incluir um método para geração de seqüências de números aleatórios. A simulação computacional de qualquer fenômeno aleatório envolve a geração de variáveis aleatórias com distribuições pré-definidas. Uma vez que o modelo probabilístico tenha sido escolhido, um algoritmo para geração das variáveis aleatórias envolvidas deve ser utilizado. Os métodos de geração destas variáveis aleatórias (v.as.) são, em geral, baseados na geração de números aleatórios, ou seja, nas realizações de seqüências de v.as. independentes e uniformemente distribuídas no intervalo $[0, 1]$. Os números pseudo-aleatórios são elementos de uma seqüência determinística de números que devem possuir as mesmas propriedades estatísticas que uma seqüência de números aleatórios. Na prática da simulação de sistemas aleatórios por computador, a geração de números pseudo-aleatórios substitui a geração de números aleatórios. Existe na literatura uma grande quantidade de algoritmos propostos para geração de números pseudo-aleatórios no intervalo $[0, 1]$ [Rip87]. Neste trabalho, foi implementada uma rotina de geração de números aleatórios (*Mrand()*) utilizando, para isso, um algoritmo

multiplicativo congruencial definido por

$$X_i = (aX_{i-1} + c) \bmod M$$

onde a é o multiplicador, c é o incremento e M é o módulo. A seqüência pseudo-randômica de variáveis uniformes U_i é dada, então, por $U_i = X_i/M$ onde X_0 é a *semente* (dada através da função *Msrand()*). Na literatura ([Mar72] e [Rip87]), encontram-se diversos valores possíveis atribuídos à a, c e M, dentre eles $a = 69069$, $c = 1$ e $M = 2^{32}$. Estes foram os valores utilizados neste trabalho.

As seqüências geradas têm período 2^{32} e os números pseudo-aleatórios retornados são inteiros entre 0 e $2^{32} - 1$.

É amplamente aceito, na área de simulação estocástica, que um bom gerador de números pseudo-aleatórios deva usar um algoritmo simples e portanto rápido, com período de no mínimo 2^{27} e tomando valores uniformemente distribuídos em $(0, 1)$. Considerando-se k valores sucessivos, eles devem ser uniformemente distribuídos em $(0, 1)^k$, para $k \in \{1, 2, ..., 10\}$. Teoricamente o algoritmo usado no *Mrand()* satisfaz estes requisitos. No entanto, é recomendável fazer-se alguns testes empíricos para verificar se a versão do algoritmo programada realmente satisfaz essas condições. Na próxima seção serão discutidos aspectos da teoria de Teste de Hipótese e de Testes Não-Paramétricos que são normalmente usadas para este tipo de teste estatístico.

2.2 Teste de Hipóteses Estatísticas

Muitos problemas requerem que decidamos entre aceitar ou rejeitar uma afirmação acerca de algum parâmetro de alguma distribuição ou sobre a natureza mesmo de uma distribuição de acordo com [MR03] e [Lar82]. Tal afirmação é chamada de **hipótese**

e o procedimento de tomada de decisão sobre a hipótese é chamado de **teste de hipóteses**. Podemos imaginar um teste estatístico de hipóteses como o estágio de análise dos dados de um experimento, onde se está interessado, por exemplo, em comparar a média de uma população com um certo valor especificado. Neste caso, uma **hipótese estatística** é uma afirmação sobre um parâmetro de uma população, ou seja, uma afirmação acerca de um parâmetro da distribuição de probabilidade de uma variável aleatória. Procedimentos de teste de hipóteses usam informações de uma amostra aleatória proveniente da população de interesse. Se essa informação for consistente com a hipótese, então concluiremos que a hipótese é verdadeira; no entanto, se essa informação for inconsistente com a hipótese, concluiremos que a hipótese é falsa.

Na estrutura do teste de hipóteses faz-se, em geral, duas hipóteses. A **hipótese nula** é aquela que desejamos testar, ou seja, a que não rejeitaremos com facilidade. A rejeição da hipótese nula sempre leva à aceitação da **hipótese alternativa**. Por exemplo, no teste de hipóteses acerca do parâmetro de uma distribuição, a hipótese nula pode ser estabelecida de modo que ela especifique o valor exato esperado do parâmetro e a hipótese alternativa permitiria ao parâmetro assumir outros valores.

Testar a hipótese envolve considerar uma amostra aleatória, computar uma estatística de teste a partir de dados amostrais e, então, usar o resultado para tomar uma decisão a respeito da hipótese nula.

O processo de decisão do teste é definido pela escolha da **região crítica**. A região crítica é o conjunto de resultados possíveis da estatística de teste para os quais devemos rejeitar a hipótese nula. O complementar da região crítica em relação ao conjunto de resultados possíveis do teste é a **região de aceitação** para a qual aceitaremos a hipótese nula.

O procedimento de decisão pode conduzir a uma de duas conclusões erradas. Uma

delas seria a rejeição da hipótese nula H_0 quando ela for verdadeira, definida como **erro tipo I**. A outra seria aceitar a hipótese nula, quando ela é falsa, definida como **erro tipo II**. A partir disto, temos duas medidas para o erro tipo I e erro tipo II respectivamente:

α = P(erro tipo I) = P(rejeitar H_0 quando H_0 for verdadeira)

β = P(erro tipo II) = P(aceitar H_0 quando H_0 for falsa)

Algumas vezes, a probabilidade do erro tipo I é chamada de **nível de significância** do teste.

Os limites entre as regiões críticas e a região de aceitação são chamados de **valores críticos**. A probabilidade α do erro tipo I é controlada quando os valores críticos são selecionados. Assim, geralmente é fácil estabelecer a probabilidade de erro tipo I em (ou perto de) qualquer valor desejado. Uma vez que se pode controlar diretamente a probabilidade de rejeitar erroneamente H_0, sempre pensamos na rejeição da hipótese H_0 como uma **conclusão forte**. Por outro lado, a probabilidade β do erro tipo II não é constante, mas depende da natureza da hipótese alternativa H_1. Ela depende também do tamanho da amostra que tenhamos selecionado. Pelo fato da probabilidade β do erro tipo II ser uma função do tamanho da amostra e da extensão com que a hipótese nula H_0 é falsa, costuma-se pensar na aceitação de H_0 como uma **conclusão fraca**, a menos que saibamos que β seja aceitavelmente pequena. Aceitar H_0 implica que não encontramos evidência suficiente para rejeitar H_0, ou seja, para fazer uma afirmação forte. Aceitar H_0 não significa necessariamente que haja uma alta probabilidade de que H_0 seja verdadeira. Isso pode significar simplesmente que mais dados são requeridos para atingir uma conclusão forte, o que pode levar a implicações importantes para a formulação das hipóteses.

2.2.1 O Valor P

O teste de hipóteses apresenta a aceitação ou a rejeição de uma hipótese nula H_0 com um nível de significância fixo segundo [MR03]. No entanto, apenas com a informação do nível de significância não é possível constatar o quão significativa é a aceitação da hipótese nula para o valor da estatística de teste calculado. Neste caso, torna-se importante o cálculo do **valor P**, que é uma forma de avaliar a força com que a hipótese nula foi aceita. O valor P é o valor mínimo do nível de significância α em que rejeitaríamos a hipótese nula para o valor obtido da estatística do teste. Nos testes de hipóteses a seguir, fizemos o valor de P igual a $1 - F_{\chi_0^2}(x)$, onde χ_0^2 é a estatística do teste, $F_{\chi_0^2}$ sua função de distribuição cumulativa e x seu valor observado.

2.2.2 Procedimento Geral para Testes de Hipóteses

Em geral, devemos aplicar a seguinte seqüência de etapas na metodologia do uso de teste de hipótese, onde as etapas de 1 a 4 devem ser completadas antes de examinar os dados amostrais.

1. A partir do contexto do problema, identifique o parâmetro de interesse ou a distribuição da população.
2. Estabeleça a hipótese nula H_0.
3. Especifique uma hipótese alternativa, H_1.
4. Escolha um nível de significância, α.
5. Estabeleça uma estatística apropriada de teste.
6. Estabeleça a região de rejeição para a estatística.

7. Calcule as grandezas amostrais necessárias, substitua na equação para a estatística de teste e calcule seu valor.

8. Decida se H_0 deve ser ou não rejeitada e reporte isso no contexto do problema usando, se necessário, o valor P.

A fim de apresentar os testes estatísticos que faremos, enunciaremos a seguir alguns conceitos e teoremas importantes.

2.2.3 A Distribuição Multinomial

Um *experimento multinomial* é um experimento aleatório com $k \geq 3$ resultados distintos possíveis. Trata-se da generalização do experimento de *Bernoulli* onde $k = 2$. As probabilidades de sucesso para k diferentes saídas são $p_1, p_2, p_3, \ldots, p_k$, onde $\sum p_i = 1$. Se são feitos n experimentos multinomiais independentes, cada um com as probabilidades $p_1, p_2, p_3, \ldots, p_k$, e define-se X_i = número de vezes em que a saída i é observada nas n tentativas, $i = 1, 2, \ldots, k$, então $(X_1, X_2, \ldots, X_k)$ é chamado *vetor aleatório multinomial* com parâmetros $n, p_1, p_2, \ldots, p_k$. A função de probabilidade para $(X_1, X_2, \ldots, X_k)$ é

$$p_{X_1,X_2,\ldots,X_k}(x_1, x_2, \ldots, x_k) = \frac{n!}{x_1!x_2!\ldots x_k!} p_1^{x_1} p_2^{x_2} \ldots p_k^{x_k},$$

onde $x_i = 0, 1, 2, \ldots, n$, $i = 1, 2, \ldots, k$ e $\sum p_i = 1$. Cada X_i individualmente é uma variável aleatória binomial com parâmetros n e p_i.

Uma gama de procedimentos de teste (dentre eles o teste de adequação de ajuste anunciado na próxima seção) usam o seguinte resultado:

Teorema A. Se $(X_1, X_2, \ldots, X_k)$ é um vetor aleatório multinomial com parâmetros $n, p_1, p_2, \ldots, p_k$, então a função de distribuição da variável aleatória

$$U = \sum_{i=1}^{k} \frac{(X_i - np_i)^2}{np_i}$$

se aproxima da função de distribuição χ^2 com $k-1$ graus de liberdade quando $n \to \infty$. Ou seja,

$$\lim_{n\to\infty} F_U(t) = F_{\chi^2}(t), \text{ para todo t,}$$

onde $F_{\chi^2}(t)$ é a função de distribuição χ^2 com $k-1$ graus de liberdade.

2.3 Teste de Adequação de Ajuste

O procedimento que descreveremos agora pode ser aplicado quando não conhecemos a distribuição da população considerada e desejamos testar se uma distribuição particular é satisfatória como modelo para essa população.

O procedimento de teste requer uma amostra aleatória de tamanho n, proveniente da população cuja distribuição de probabilidade é desconhecida. Essas n observações são arranjadas em um histograma de freqüência, tendo k intervalos de classe. Considere O_i a freqüência observada no i-ésimo intervalo de classe. A partir da distribuição de probabilidades utilizada na hipótese, calculamos a freqüência esperada no i-ésimo intervalo de classe, denotada por E_i. A estatística de teste é:

$$\chi_0^2 = \sum_{i=1}^{k} \frac{(O_i - E_i)^2}{E_i}$$

Se a população seguir a distribuição utilizada na hipótese, χ_0^2 terá, aproximadamente, uma distribuição qui-quadrado com $k-p-1$ graus de liberdade, sendo p

o número de parâmetros da distribuição utilizada na hipótese, que foram estimados pelas estatísticas amostrais. Essa aproximação melhora à medida que n aumenta. Rejeitaremos a hipótese de que a distribuição da população é a distribuição utilizada na hipótese H_0, se o valor calculado da estatística de teste for $\chi_0^2 > \chi_{1-\alpha,k-p-1}^2$.

Um ponto a ser notado na aplicação desse procedimento de teste se refere à magnitude das freqüências esperadas. Se essas freqüências esperadas forem muito pequenas, então a estatística de teste χ_0^2 não refletirá o desvio entre observado e esperado, mas somente a pequena magnitude das freqüências esperadas. Não há concordância geral relativa ao valor mínimo das freqüências esperadas, mas valores de 3, 4 e 5 são largamente utilizados como mínimos. Se uma freqüência esperada for muito pequena, ela poderá ser combinada com a freqüência esperada em um intervalo de classe adjacente. As freqüências observadas correspondentes seriam então também combinadas e k seria reduzido de um. Intervalos de classe não necessitam ter a mesma largura.

Neste trabalho, foram aplicados os testes de ajuste aos algoritmos 1 e 2 descritos abaixo, de modo a validar as rotinas do software que os implementa.

2.3.1 Teste de Ajuste para uma Distribuição Supostamente Uniforme

Vamos agora descrever o algoritmo utilizado pelo programa para gerar variáveis com distribuição uniforme e, em seguida, faremos o procedimento de adequação de ajuste com $\alpha = 0.05$ a fim de sabermos se a distribuição uniforme é apropriada como modelo de probabilidade para essas variáveis geradas.

Algoritmo 1: geração de variáveis $U_1, U_2, \ldots, U_k$ supostamente com distribuição uniforme

1. Forneça um inteiro $X_0 \geq 0$ para semente;

2. Para $i = 1$ até n faça:

3. $X_i = (aX_{i-1} + c) \bmod M$ para $a = 69069$, $b = 1$ e $M = 2^{32}$

4. $U_i = X_i/M$

5. Retorne U_i.

Utilizando $\alpha = 0.05$, desejamos saber se o programa está adequadamente gerando uma distribuição uniforme. Usaremos uma amostra aleatória de tamanho $n = 100$.

Uma prática comum na construção de intervalos de classe para a distribuição de freqüências usada no teste de adequação de ajuste de qui-quadrado é escolher os limites das células de modo que as freqüências esperadas $E_i = np_i$ sejam iguais para todas as células. Para usar esse método, queremos escolher os limites das células $a_0, a_1, \ldots, a_k$ para as k células de modo que todas as probabilidades

$$p_i = P(a_{i-1} \leq X \leq a_i) = \int_{a_{i-1}}^{a_i} f(x)\, dx$$

sejam iguais. Usaremos oito células ($k = 8$) conforme a tabela 2.1. Feito o teste, obtivemos: média amostral = 0.486, variância amostral = 0.074 e estatística de teste = 6.560. Atribui-se $\hat{a}$ = menor valor da amostra e $\hat{b}$ = maior valor da amostra. Os intervalos de classe vão de $\hat{a} = 0.014$ a $\hat{b} = 0.957$, e o tamanho dos intervalos de classe é $\dfrac{\hat{b} - \hat{a}}{k} = 0.118$. Conta-se a partir daí quantos valores da amostra se encontram para cada um destes intervalos obtidos, obtendo-se as freqüências observadas (ver tabela 2.1).

Podemos aplicar o procedimento das oito etapas do teste de hipóteses para esse problema.

1. A amostra (U_i) é gerada pelo algoritmo 1.

Intervalo de Classe	Freqüência Observada ABS	Freqüência Observada REL	Freqüência Esperada ABS	Freqüência Esperada REL
$0.014 \leq x < 0.132$	17	0.170	12.500	0.125
$0.132 \leq x < 0.250$	7	0.070	12.500	0.125
$0.250 \leq x < 0.367$	11	0.110	12.500	0.125
$0.367 \leq x < 0.486$	14	0.140	12.500	0.125
$0.486 \leq x < 0.604$	14	0.140	12.500	0.125
$0.604 \leq x < 0.722$	14	0.140	12.500	0.125
$0.722 \leq x < 0.839$	11	0.110	12.500	0.125
$0.839 \leq x < 0.957$	8	0.080	12.500	0.125

Tabela 2.1: Resultados da experiência com $n = 100$ rodadas do algoritmo 1

2. H_0: A forma da distribuição é uniforme.

3. H_1: A forma da distribuição não é uniforme.

4. $\alpha = 0.05$

5. A estatística de teste é
$$\chi_0^2 = \sum_{i=1}^{k} \frac{(O_i - E_i)^2}{E_i}$$

6. Uma vez que os dois parâmetros na distribuição uniforme tenham sido estimados, a estatística qui-quadrado anterior terá $k - p - 1 = 8 - 2 - 1 = 5$ graus de liberdade (tem-se $p = 2$ por causa da média e do desvio padrão). Conseqüentemente, rejeitaremos H_0 se $\chi_0^2 > \chi_{0.95,5}^2 = 11.07$

7. Cálculos:
$$\chi_0^2 = \sum_{i=1}^{k} \frac{(O_i - E_i)^2}{E_i} = 6.560$$

8. Conclusões: Já que $\chi_0^2 = 6.560 < \chi_{0.95,5}^2 = 11.07$, não rejeitaremos a hipótese nula e não há evidência forte para indicar que o algoritmo 1 não gera uma variável aleatória uniformemente distribuída. O valor P calculado é 0.847.

2.3.2 Teste de Ajuste para uma Distribuição Supostamente de Poisson

O processo de Poisson $N = N(t), t \in [0, +\infty)$ é um processo de contagem a tempo contínuo, cujos tempos entre duas chegadas consecutivas são variáveis aleatórias exponencialmente distribuídas com parâmetro λ e independentes. A variável $N(t)$ é uma variável de Poisson de parâmetro $\mu = \lambda t$.

Suponhamos que queremos gerar os primeiros n eventos de um processo de Poisson $N = N(t), t \in [0, +\infty)$, com taxa λ. Como os tempos entre sucessivos eventos deste processo são variáveis exponenciais e independentes com taxa λ, uma forma de gerar o processo é gerando esses intervalos de tempo. Para isto geramos n números pseudo-randômicos $U_1, U_2, ..., U_n$ e atribuímos $X_i = -1/\lambda(\log U_i)$. X_i será o tempo entre o $(i-1)$-ésimo e o i-ésimo evento do processo de Poisson. O tempo do n-ésimo evento é a soma dos primeiros n intervalos de tempo, ou seja, ele é gerado por $\sum_{i=1}^{n} X_i$. O algoritmo 2 descrito abaixo gera a variável de Poisson $N(t)$, simulando um processo de Poisson no intervalo de tempo $[0, t]$.

Primeiramente descreveremos o algoritmo 2. Em seguida, aplicaremos o teste de adequação de ajuste, como foi feito na subseção anterior, para decidirmos se ele efetivamente gera uma variável aleatória com distribuição de Poisson de parâmetro μ. O algoritmo 2 funciona da seguinte forma: ele gera sucessivas variáveis exponenciais de parâmetro λ até que sua soma exceda $T = \mu/\lambda$. O valor dessas variáveis representam os tempos entre chegadas sucessivas do processo de Poisson. O número I dessas variáveis é o valor simulado da variável de Poisson de parâmetro μ.

Algoritmo 2: geração da variável I supostamente com distribuição de Poisson

1. $t = 0$.

2. Gera-se uma variável randômica uniforme U (Algoritmo 1).

3. $t = t - \frac{1}{\lambda}(\log U)$. Se $t > T$, pare.

4. $I = I + 1$, $S(I) = t$.

5. Vá para o *passo 2*.

Observações: no algoritmo, t refere-se ao tempo acumulado das sucessivas chegadas e I é o número de eventos que ocorreram até o tempo t.

O valor final de I no algoritmo acima irá representar o número de eventos que ocorre até o tempo T, e os valores $S(1), ..., S(I)$ serão os tempos de chegada em ordem crescente.

Foram realizadas 140 rodadas da rotina que supostamente gera uma variável aleatória de Poisson com o parâmetro igual a $\mu = \lambda T = 4.0$, e cada um dos valores gerados é armazenado em um vetor em ordem crescente. Em seguida, é calculada a freqüência absoluta (ou freqüência observada) de cada um dos valores das variáveis de Poisson, na ordem armazenada no vetor. Esses valores podem ser conferidos na tabela 2.2 onde o total de intervalos de classe é 9, a média amostral é 4.036, a média real é 4.000, o grau de liberdade é 7 e a estatística de teste é 4.759.

Como μ é desconhecido, estimamos seu valor pela média amostral. A estimativa do número médio de chegadas no tempo T do algoritmo 2 é a média amostral, ou seja:

Média amostral $= M = \sum_{i=1}^{9} \frac{N_i F_i}{140}$

onde N_i é o valor da classe, F_i é a freqüência observada para N_i.

A partir da distribuição de Poisson com parâmetro M, podemos calcular p_i, a freqüência esperada relativa, associada com o i-ésimo intervalo de classe. Uma vez que

Número de Poisson	Freqüência Observada ABS	Freqüência Observada REL	Freqüência Esperada ABS	Freqüência Esperada REL
0.000	4.000	0.029	2.474	0.018
1.000	8.000	0.057	9.985	0.071
2.000	18.000	0.129	20.149	0.144
3.000	30.000	0.214	27.105	0.194
4.000	32.000	0.229	27.347	0.195
5.000	16.000	0.114	22.073	0.158
6.000	14.000	0.100	14.847	0.106
7.000	10.000	0.071	8.560	0.061
8.000 ou mais	8.000	0.057	7.036	0.050

Tabela 2.2: Resultado da experiência com $n = 140$ rodadas do algoritmo 2

cada intervalo de classe corresponde a um número particular gerado pelo algoritmo, podemos encontrar p_i, da forma:

$$p_i = P(X = k) = \frac{e^{-M} M^k}{k!}$$

As freqüências esperadas absolutas são calculadas pela multiplicação do tamanho da amostra $n = 140$ vezes as probabilidades p_i; isto é, $E_i = np_i = 140p_i$.

A estatística de teste qui-quadrado terá $k-p-1 = 9-1-1 = 7$ graus de liberdade, porque a média da distribuição de Poisson foi estimada a partir desses dados.

O procedimento de oito etapas para o teste de adequação de ajuste pode ser agora aplicado, usando $\alpha = 0.05$, conforme a seguir:

1. A variável de interesse é a saída do algoritmo 2.

2. H_0: A forma de distribuição é Poisson.

3. H_1: A forma de distribuição não é Poisson.

4. $\alpha = 0.05$

5. A estatística de teste é
$$\chi_0^2 = \sum_{i=1}^{k} \frac{(O_i - E_i)^2}{E_i}$$

6. Rejeitar H_0 se $\chi_0^2 > \chi_{0.95,7}^2 = 14.067$

7. Cálculos:
$$\chi_0^2 = \sum_{i=1}^{k} \frac{(O_i - E_i)^2}{E_i} = 4.759$$

8. Conclusões: Uma vez que $\chi_0^2 = 4.759 < \chi_{1-0.05,7}^2 = 14.067$, não rejeitaremos a hipótese nula. O valor P calculado para este teste de estatística é 0.699.

2.4 Teste de Independência

Existem diversos testes não-paramétricos que podem ser aplicados quando desejamos saber se um conjunto de números randômicos são independentes [Lar82]. Com esta finalidade aplicaremos o teste *run* em uma amostra de tamanho $n = 100$ do algoritmo 1. Em [Rip87] estão maiores detalhes sobre a formulação do teste. O teste *run* foi escolhido por ser um teste relativamente simples de ser aplicado conforme veremos a seguir:

Dada a amostra $U_1, U_2, \ldots, U_{100}$ do algoritmo 1, calcula-se o valor da mediana, ou seja, deseja-se saber o valor intermediário em que teremos metade da amostra com valores inferiores à mediana e metade com valores superiores à mediana. Em seguida, para cada termo $U_i, i = 1, \ldots, 100$ da amostra, associa-se o valor "0" para os termos cujo valor é inferior ao da mediana calculada e, analogamente, associa-se "1" para os termos maiores que o valor da mediana. Assim, teremos uma seqüência de "0´s" e "1´s" para a amostra. Denomina-se *run* cada subseqüência de "0´s" e "1´s" dentro de uma seqüência (por exemplo, a seqüência 1100011100 possui quatro *runs*: o primeiro

é 11, o segundo é 000, o terceiro é 111 e o quarto é 00). Denota-se m o total de zeros e n total de uns na seqüência (no exemplo acima, $m = n = 5$). Para que seja aceita a independência dos valores da amostra, basta que o número total de *runs* da amostra esteja entre os valores t_p e u_q, onde t_p é o maior inteiro tal que $P(R \leq t_p) \leq p$, para $p < 0.5$ e u_q é o menor inteiro tal que $P(R \geq u_q) \leq q$, para $q < 0.5$:

$$t_p = \frac{2mn}{m+n}(1 + \frac{1}{\sqrt{m+n}}z_p)$$

e

$$u_q = 1 + \frac{2mn}{m+n}(1 + \frac{z_{1-p}}{\sqrt{m+n}}),$$

z_p é o p-ésimo quantil da distribuição normal reduzida (ou seja, $N(z_p) = p$).

Não entraremos nos detalhes dos cálculos que levaram aos resultados das equações apresentadas acima [Rip87]. Estamos apenas interessados na sua utilização no teste de aleatoriedade cujo resultado mostraremos a seguir.

Este teste foi programado para o algoritmo 1 (geração de variáveis uniformes) numa amostra de tamanho $n = 100$. A mediana encontrada possui o valor 0.476266, $m = 50$, $n = 50$ e a seqüência gerada possui $r = 57$ *runs*. Rejeitaremos a hipótese de que o algoritmo produz variáveis aleatórias se o total de runs observados for $r \leq t_p$ ou $r \geq u_q$. Para $p = 0.05$ e $q = 0.05$, obtivemos $t_{0.05} = 41.80$ e $u_{0.05} = 59.20$. Como $41.80 < 57 < 59.20$, aceitaremos a hipótese de aleatoriedade nestas observações.

Capítulo 3

Processos Pontuais e Processos Pontuais Indexados

3.1 Descrição dos Modelos Estocásticos

Dois modelos estocástico são importantes neste trabalho: os *Processos Pontuais* (ou *Point Processes*) e os *Processos Pontuais Indexados* (ou *Marked Point Processes*).

Um *Processo Pontual* modela a distribuição aleatória de pontos em um domínio pré-fixado. O número de tais pontos e a posição desses pontos no domínio são aleatórios.

Um *Processo Pontual Indexado* modela a distribuição aleatória de pontos em um domínio pré-fixado e atribui aleatoriamente a cada um desses pontos valores de um conjunto determinado. Um *Processo Pontual Indexado* pode ser definido como um par ordenado (X, C), onde X é um *Processo Pontual* referente à localização dos pontos no espaço em que um determinado fenômeno ocorre e C é uma função aleatória a valores reais que representa a intensidade do fenômeno em cada ponto do espaço.

Neste trabalho trataremos da simulação dos *Processos Pontuais* de Poisson homogêneos e não-homogêneos e da simulação dos *Processos Pontuais Indexados* relacionados a estes. A simulação dos últimos é feita a partir da simulação dos primeiros.

Os *Processos Pontuais Indexados* são até mais importantes para as aplicações práticas.

3.1.1 Modelagem dos Processos Pontuais de Poisson

Seja Γ um boreliano do $\Re^n$ e $\aleph$ o conjunto dos borelianos de Γ então um *Processo Pontual* N é dito ser um processo de Poisson não-homogêneo em Γ, se para cada B $\in \aleph$, as v.as. N(B), que contam o número de pontos que ocorrem em B, satisfazem as seguintes condições:

i) $P(N(B) \in \{0, 1, ...\}) = 1$

ii)$\forall (B_i)_{i\in\{1,2,...,n\}}$, coleção de conjuntos disjuntos onde $B_i \in \aleph$, as v.as. $N(B_i)$ são independentes.

iii) $\forall s \in \Gamma$, temos:

$P\{N(ds) = 0\} = 1 - \mu(ds) + o(\mu(ds))$,

$P\{N(ds) = 1\} = \mu(ds) + o(\mu(ds))$,

$P\{N(ds) > 1\} = o(\mu(ds))$,

onde ds é uma região infinitesimal em torno de s e $0 \leq \mu(B) < +\infty$, $\forall$ B $\in \aleph$.

Estes três postulados implicam que $\forall$ B $\in \aleph$ a v.a. $N(B)$ tem distribuição de Poisson de média $\mu(B)$, dada por:

$$P\{N(B) = n\} = \frac{(\mu(B))^n e^{-\mu(B)}}{n!}, \text{ para } n = 0, 1, ...$$

Quando $\mu(B) = \mu \times |B|$, onde μ é uma constante positiva e $|B|$ é o volume de B então o processo de Poisson é dito *homogêneo*.

3.2 Simulação dos Processos Pontuais e dos Processos Pontuais Indexados

Como dissemos anteriormente, trataremos apenas da simulação dos *Processos Pontuais* de Poisson homogêneos e não-homogêneos e da simulação dos *Processos Pontuais Indexados* relacionados a estes.

O volume do espaço que define o suporte dos Processos Pontuais a serem simulados é a união de paralelepípedos cuja a interseção de seus interiores é vazia, isso indica que devemos primeiro definir um algoritmo para simular um *Processo Pontual* de Poisson Homogêneo, de intensidade λ, em um cubo de arestas unitárias V. Para isso, devemos simular uma variável de Poisson de parâmetro $\mu = \lambda$ e, se o valor simulado for n, simular n vetores aleatórios uniformemente distribuídos no cubo unitário. O algoritmo que executa esta simulação é:

Algoritmo 3: geração de uma realização do *Processo Pontual* de Poisson homogêneo em um cubo unitário

1. Gera-se uma variável de Poisson N de parâmetro $\mu = \lambda$ usando o Algoritmo 2;
2. Para $i = 1, \ldots, N$, faça:
3. Gera-se um vetor aleatório (U_1, U_2, U_3) usando o Algoritmo 1;

Quando a simulação for num paralelepípedo K, primeiro devemos achar a transformação linear T tal que TK seja o cubo unitário. T será representada por uma matriz diagonal, portanto facilmente invertível. Em seguida aplicamos o algoritmo 3 ao cubo unitário TK fazendo $\mu = \lambda|K|$, onde $|K|$ é o volume de K. Para acharmos finalmente os pontos buscados, aplicamos aos pontos produzidos pelo algoritmo 3 a transformação linear T^{-1}.

No caso mais geral, o suporte do Processo de Poisson não-homogêneo é descrito por uma lista K_i, $i = 1, \ldots, M$ de paralelepípedos e para cada paralelepípedo K_i é associado um λ_i que define a intensidade do processo em K_i. Neste caso o algoritmo se escreve:

Algoritmo 4: geração de uma realização do *Processo Pontual* de Poisson.

1. Para $i = 1, \ldots, M$, faça:

2. Achar a transformação linear T_i de K_i em um cubo unitário.

3. Aplicar o algoritmo 3 com $\mu = \lambda_i |K_i|$.

4. Aplicar para cada ponto achado a transformação T_i^{-1}.

Como um *Processo Pontual Indexado* é um par ordenado (X, C), onde X é um *Processo Pontual* e C é uma função aleatória a valores reais (ou vetoriais), podemos supor que a definição de C é feita pela escolha de uma classe de v.as. para cada $C(x)$ e de uma função determinística f, definida no suporte do processo e a valores no espaço dos parâmetros possíveis de $C(x)$. Uma vez definido o Processo Pontual X, a classe das variáveis $C(x)$ e a função f, o algoritmo de simulação do *Processo Pontual Indexado* é:

Algoritmo 5: geração de uma realização do *Processo Pontual Indexado.*

1. Aplicar o algoritmo 3 com os dados necessários, como definido.

2. Para cada ponto x gerado pelo algoritmo 3, simular a v.a. $C(x)$ com parâmetros $f(x)$.

Capítulo 4

Visualização Espacial

Neste capítulo serão apresentadas as técnicas de programação necessárias ao desenvolvimento do subsistema de visualização, ou seja, da parte do software dedicada à visualização gráfica das simulações. Inicialmente, faremos uma breve exposição dos conceitos teóricos da computação gráfica que foram usados. Em seguida, apresentaremos a biblioteca gráfica OpenGL e discutiremos sua aplicação neste trabalho.

4.1 O Ambiente Tridimensional

Chamamos cena uma porção finita qualquer do mundo real. A percepção de espacialidade de uma imagem pode ser entendida como a capacidade do ser humano de distinguir as formas, as cores, as texturas, e a relação espacial entre objetos pertencentes à cena através desta imagem. Na visão humana, para obter uma representação de uma cena, o cérebro leva em conta diversas informações obtidas por ambos os olhos. Os movimentos do globo ocular, as cores e as sombras dos objetos contribuem decisivamente para a construção desta representação. Cada olho forma uma imagem bidimensional, que é uma fotografia temporária dos objetos na retina. Estas duas imagens bidimensionais são ligeiramente diferentes porque elas são recebidas em dois ângulos diferentes pelos olhos, que são para isso mesmo, separados. O cérebro com-

bina estas imagens para produzir um único quadro composto 3D.

A Figura 4.1 mostra um objeto visto pelos os dois olhos.

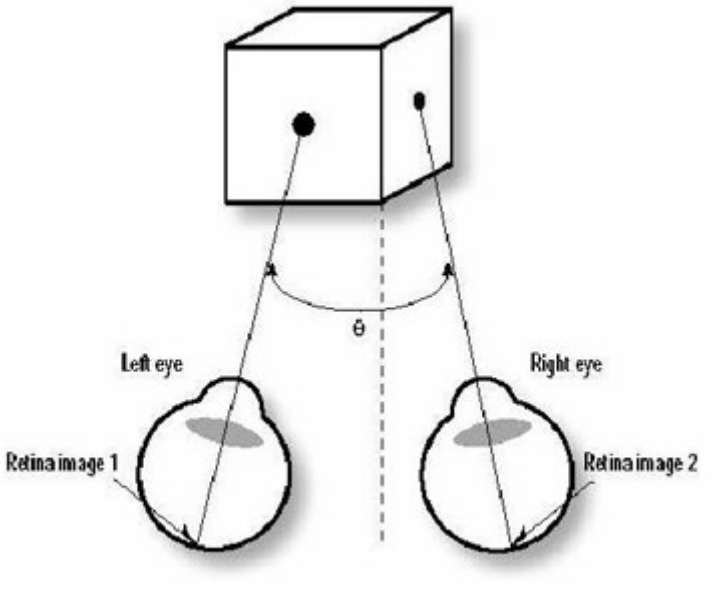

Figura 4.1: Visão 3D

A computação gráfica procura criar, para as cenas, imagens bidimensionais com boa percepção de espacialidade. Os sistemas gráficos devem transformar formas 3D em imagens bidimensionais para serem apresentadas em uma tela plana. Elas deverão dar a ilusão de profundidade, ou de tridimensão. Isto é feito através da projeção. Além disso, esses sistemas devem procurar facilitar o acesso do observador às informações contidas na cena e possibilitar a criação de novas cenas, isso é feito através das transformações geométricas e da iluminação. Inicialmente discutiremos as transformações geométricas e em seguida apresentaremos as projeções. E, por fim, mostraremos a iluminação como um recurso do OpenGL.

4.2 Transformações Geométricas

As transformações geométricas são ferramentas importantes na manipulação das cenas tridimensionais. Elas são utilizadas, por exemplo, para transladar, rotacionar, escalar e refletir objetos, atuando no conjunto dos vértices ou pontos que definem esses objetos, transformando-os em um novo conjunto de vértices ou pontos. Nesta seção apresentaremos primeiro as *coordenadas homogêneas*, que são a representação adequada para pontos e vetores e para as suas transformações geométricas em um sistema gráfico. Em seguida, apresentaremos as transformações geométricas.

4.2.1 Coordenadas Homogêneas

A representação de pontos e vetores na álgebra linear ordinária é ambígua, no sentido de que, uma vez fixado um sistema de coordenadas retangulares, um elemento do R^3 representa ao mesmo tempo um ponto e um vetor. Isto é fonte de problemas para os sistemas computacionais. O uso das *coordenadas homogêneas* permite resolver este problema. Em coordenadas homogêneas, pontos de um espaço afim de dimensão 3 e vetores do espaço vetorial associado são representados univocamente por elementos do R^4. As transformações geométricas de rotação, translação e de escala assumem uma representação matricial como uma transformação linear em espaços vetoriais de dimensão 4. Em um espaço afim, um *marco* é definido por um ponto, chamado origem e pelos vetores de uma base do espaço vetorial associado, ou seja, no caso de dimensão 3, que será o único tratado neste trabalho, pela especificação: (v_1, v_2, v_3, P_0), onde os três primeiros elementos são vetores da base e o quarto é a origem. Qualquer ponto P do espaço afim pode ser escrito de uma maneira única como [MdBS98]:

$$P = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 + P_0.$$

assim, podemos expressar esta relação, usando um produto formal de matrizes, como:

$$P = \begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 & 1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{bmatrix}.$$

A matriz linha de quatro dimensões do lado direito da equação é a representação das coordenadas homogêneas do ponto P no marco determinado por $v_1, v_2, v_3,$ e P_0. Equivalentemente, podemos dizer que P é representado pela matriz coluna:

$$\mathbf{p} = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ 1 \end{bmatrix}.$$

A representação de um vetor tridimensional segue uma lógica similar. No mesmo marco, qualquer vetor w pode ser escrito como:

$$w = \delta_1 v_1 + \delta_2 v_2 + \delta_3 v_3$$

$$= \begin{bmatrix} \delta_1 & \delta_2 & \delta_3 & 0 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{bmatrix}.$$

Equivalentemente, w pode ser representado pela matriz coluna:

$$\mathbf{w} = \begin{bmatrix} \delta_1 \\ \delta_2 \\ \delta_3 \\ 0 \end{bmatrix}.$$

Existem numerosas formas de interpretar esta formulação geometricamente. Ressaltaremos apenas a simplicidade das operações com pontos e vetores utilizando suas representações em coordenadas homogêneas e a álgebra matricial ordinária. Vamos agora considerar a operação de mudança de marco. Sejam (v_1, v_2, v_3, P_0) e (u_1, u_2, u_3, Q_0) dois marcos, a expressão do segundo marco em função do primeiro é:

$$u_1 = \gamma_{11}v_1 + \gamma_{12}v_2 + \gamma_{13}v_3$$

$$u_2 = \gamma_{21}v_1 + \gamma_{22}v_2 + \gamma_{23}v_3$$

$$u_3 = \gamma_{31}v_1 + \gamma_{32}v_2 + \gamma_{33}v_3$$

$$Q_0 = \gamma_{41}v_1 + \gamma_{42}v_2 + \gamma_{43}v_3 + P_0$$

Estas equações, na forma de matrizes, podem ser escritas como:

$$\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ Q_0 \end{bmatrix} = \mathbf{M} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{bmatrix},$$

onde $\mathbf{M}$ é uma matriz 4×4

$$\mathbf{M} = \begin{bmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} & 0 \\ \gamma_{21} & \gamma_{22} & \gamma_{23} & 0 \\ \gamma_{31} & \gamma_{32} & \gamma_{33} & 0 \\ \gamma_{41} & \gamma_{42} & \gamma_{43} & 1 \end{bmatrix}.$$

$\mathbf{M}$ é chamada de *matriz de representação* da mudança de marco. Podemos utilizá-la para computar diretamente as alterações de representação. Suponhamos que $\mathbf{a}$ e $\mathbf{b}$ são representações em coordenadas homogêneas de um ponto ou um vetor, em dois marcos distintos. Então,

$$\mathbf{b}^T \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ Q_0 \end{bmatrix} = \mathbf{b}^T \mathbf{M} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{bmatrix} = \mathbf{a}^T \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{bmatrix}.$$

Assim,

$$\mathbf{a} = \mathbf{M}^T \mathbf{b}.$$

Claro que $\mathbf{M}^T$, é

$$\mathbf{M}^T = \begin{bmatrix} \gamma_{11} & \gamma_{21} & \gamma_{31} & \gamma_{41} \\ \gamma_{12} & \gamma_{22} & \gamma_{32} & \gamma_{42} \\ \gamma_{13} & \gamma_{23} & \gamma_{33} & \gamma_{43} \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

que, assim como $\mathbf{M}$, é determinada por 12 coeficientes.

É importante observar que a composição das transformações geométricas pode ser representada pela multiplicação das representações matriciais dessas transformações. Embora se tenha que trabalhar em quatro dimensões para resolver problemas tridimensionais, o uso esse tipo de representação simplifica muito os cálculos e a programação. Os hardwares modernos implementam as operações em coordenadas homogêneas em VLSI de forma paralela, garantindo alta velocidade nos cálculos.

4.2.2 Translação

A *translação* é uma operação que desloca um ponto, em uma dada direção, por uma distância fixada, como ilustrado na Figura 4.2. Se movermos o ponto p, na direção d, por uma distância $|d|$, então: $p' = p + d$.

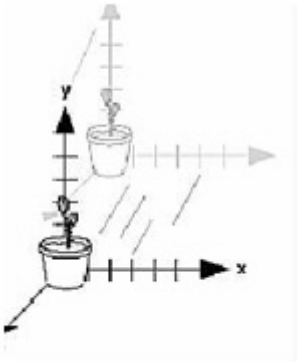

Figura 4.2: Translação de um objeto

Em coordenadas homogêneas temos:

$$p = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}, \; p' = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} \text{ e } d = \begin{bmatrix} \alpha_x \\ \alpha_y \\ \alpha_x \\ 0 \end{bmatrix}$$

Logo temos:

$$x' = x + \alpha_x$$

$$y' = y + \alpha_y$$

$$z' = z + \alpha_z$$

Vemos que uma translação é obtida pela multiplicação de uma matriz por um vetor:

$$\mathbf{p}' = \mathbf{T}\mathbf{p},$$

onde

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & \alpha_x \\ 0 & 1 & 0 & \alpha_y \\ 0 & 0 & 1 & \alpha_z \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

T é chamada *matriz de translação*. Às vezes a escrevemos como $T(\alpha_x, \alpha_y, \alpha_z)$ para enfatizar os três parâmetros de que ela depende, os demais estando fixados.

4.2.3 Rotação

A especificação da *rotação* é mais complexa que a da translação, pois mais parâmetros estão envolvidos. Inicia-se pela rotação de um ponto em torno de outro ponto em um plano. Especificando-se este último ponto como origem e uma base ortogonal com um vetor perpendicular ao plano definiremos um marco particular. Suponhamos que o ponto (x, y) do plano é rotacionado em torno da origem de um ângulo θ até a posição (x', y'). As equações desta rotação no plano são obtidas como segue:

Escrevemos (x, y) e (x', y') na *forma polar*

$$x = \rho \cos \phi,$$

$$y = \rho \sin \phi,$$

$$x' = \rho \cos(\theta + \phi),$$

$$y' = \rho \sin(\theta + \phi),$$

Usando as identidades trigonométricas para o seno e o cosseno da soma de dois ângulos, temos:

$$x' = \rho \cos \phi \cos \theta - \rho \sin \phi \sin \theta = x \cos \theta - y \sin \theta,$$

$$y' = \rho \cos \phi \sin \theta + \rho \sin \phi \cos \theta = x \sin \theta + y \cos \theta.$$

Estas equações podem ser escritas na forma matricial como:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Essa matriz é a da rotação num plano em torno de um ponto. A rotação tridimensional de um ângulo θ em torno de um eixo perpendicular ao plano se expressa como:

$$x' = x \cos \theta - y \sin \theta,$$

$$y' = x \sin\theta + y \cos\theta,$$

$$z' = z;$$

e na forma matricial

$$\mathbf{w}' = M_z w,$$

onde

$$\mathbf{M}_z = \mathbf{M}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Verifica-se que w' (ponto ou vetor) é o resultado da rotação de w em torno do eixo z. Pode-se derivar as matrizes de rotação em torno dos demais eixos ortogonais com uma argumentação idêntica. As demais matrizes são:

$$\mathbf{M}_x = \mathbf{M}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix},$$

$$\mathbf{M}_y = \mathbf{M}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}.$$

Os sinais dos termos de senos são consistentes com a definição da rotação positiva com a "regra da mão direita".

A matriz de rotação em coordenadas homogêneas, como a de todas as transformações lineares no espaço tridimensional, tem a forma:

$$\begin{bmatrix} \alpha_{11} & \alpha_{21} & \alpha_{31} & 0 \\ \alpha_{12} & \alpha_{22} & \alpha_{32} & 0 \\ \alpha_{13} & \alpha_{23} & \alpha_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

Quando

$$\begin{bmatrix} \alpha_{11} & \alpha_{21} & \alpha_{31} \\ \alpha_{12} & \alpha_{22} & \alpha_{32} \\ \alpha_{13} & \alpha_{23} & \alpha_{33} \end{bmatrix},$$

é $\mathbf{M}_x, \mathbf{M}_y$ ou $\mathbf{M}_z$, denota-se $\mathbf{R}_x, \mathbf{R}_y$ ou $\mathbf{R}_z$ as respectivas matrizes de rotação em coordenadas homogêneas. A notação para a rotação em torno de um eixo genérico é $\mathbf{R}_\bullet$.

Uma rotação $\mathbf{R}_\bullet$ de um ângulo θ pode ser desfeita através da mesma rotação de ângulo $-\theta$, então

$$\mathbf{R}_\bullet^{-1}(\theta) = \mathbf{R}_\bullet(-\theta).$$

Além disto, como todos os termos, em $\mathbf{M}_\bullet$, com cosseno estão na diagonal e os termos com seno não estão na diagonal. Pode-se utilizar as identidades trigonométricas

$\cos(-\theta) = \cos\theta$ e $\sin(-\theta) = -\sin\theta$ para encontrar que:

$$\mathbf{R}_{\bullet}^{-1}(\theta) = \mathbf{R}_{\bullet}^{T}(\theta).$$

Para construir-se a matriz de uma rotação qualquer, em torno de um ponto fixado como origem, basta fazer-se o produto das rotações individuais sobre os três eixos

$$\mathbf{R} = \mathbf{R}_z\mathbf{R}_y\mathbf{R}_x.$$

Como a transposta do produto de matrizes é o produto das transpostas na ordem inversa, para uma rotação qualquer, temos também:

$$\mathbf{R}^{-1} = \mathbf{R}^{T}.$$

Uma matriz cuja inversa é igual a sua transposta é chamada de *matriz ortogonal*. A vantagem desta propriedade é que não é necessário nenhum cálculo adicional para se obter a matriz inversa, bastando apenas alterar os índices de linhas e colunas da matriz.

A Figura 4.3 ilustra uma rotação feita em torno do eixo z. Neste caso, os pontos originais são multiplicados por uma matriz de rotação ao redor do eixo definido pelo vetor 0z no sentido anti-horário.

No caso da rotação em torno de um eixo genérico (e_x, e_y, e_z), basta transladar a figura a ser rotacionada para a origem (calcula-se, para isto, a matriz de translação em relação ao eixo (e_x, e_y, e_z)), aplicar as rotações necessárias e, finalmente, transladar a figura para a sua posição inicial, aplicando a inversa da matriz de translação calculada inicialmente.

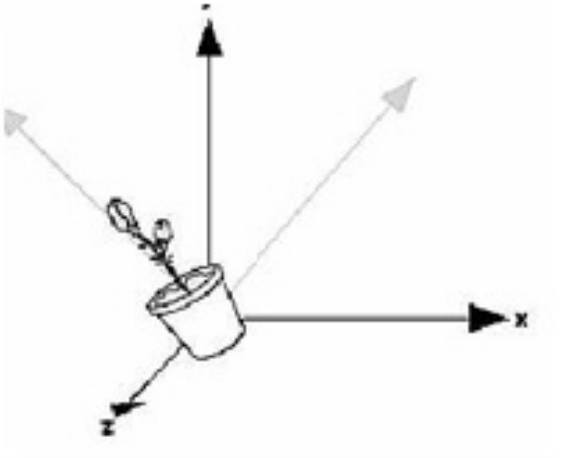

Figura 4.3: Rotação de um objeto em torno do eixo z

4.2.4 Escala

O *escalamento* altera o tamanho de um objeto, podendo torná-lo maior ou menor em relação ao seu tamanho original. As transformações com escala possuem um ponto fixo, e para especificar uma escala, precisaremos deste ponto fixo, de uma direção na qual desejamos fazer a escala e um fator de escala (α). Para $\alpha > 1$, o objeto "cresce" em uma direção específica; para $0 \leq \alpha < 1$, o objeto "encolhe" nesta direção. Valores negativos de α nos fornece a *reflexão* sobre um ponto fixo, na direção de escalamento, como podemos observar na Figura 4.4. Nesta figura o ponto fixo é a origem, $\alpha = -1$ e a direção de escalamento é $(0, 1, 0)$.

Consideraremos primeiro o ponto fixo como sendo a origem. Iremos mostrar, mais tarde, como obter transformações concatenadas a fim de obtermos uma transformação para um ponto fixo arbitrário. Uma matriz de escalamento com um ponto fixo na origem permite o escalamento independente sobre os eixos das coordenadas. As três equações são

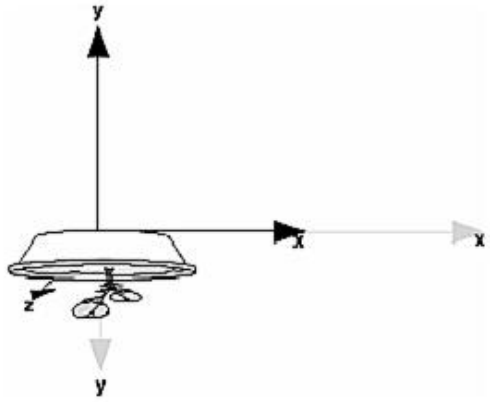

Figura 4.4: O escalamento de um objeto

$$x' = \beta_x x,\ y' = \beta_y y,\ z' = \beta_z z.$$

Estas três equações podem ser combinadas na forma homogênea como $\mathbf{p}' = \mathbf{Sp}$, onde

$$\mathbf{S} = \mathbf{S}(\beta_x, \beta_y, \beta_z) = \begin{bmatrix} \beta_x & 0 & 0 & 0 \\ 0 & \beta_y & 0 & 0 \\ 0 & 0 & \beta_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Note que aqui, como para toda a transformação linear expressa em coordenadas homogêneas, a última linha da matriz força o quarto componente do ponto transformado a reter o seu valor.

Obteremos o inverso da matriz de escala aplicando os recíprocos dos fatores de

escala:

$$\mathbf{S}^{-1}(\beta_x, \beta_y, \beta_z) = \mathbf{S}\left(\frac{1}{\beta_x}, \frac{1}{\beta_y}, \frac{1}{\beta_z}\right).$$

Para o escalamento de um objeto em um ponto $P' = (x', y', z')$ arbitrário, teremos que utilizar a transformação na seguinte ordem:

1. Translada-se o ponto P' para o ponto de origem do marco;

2. Faz-se o escalamento do objeto em relação à origem e

3. Translada-se o objeto escalonado da origem para P' (para isto, basta aplicar a matriz inversa obtida no ítem 1).

4.3 Projeção

Após a criação de cenas e objetos tridimensionais o próximo passo é efetuar a sua apresentação. Conforme mencionamos no início do capítulo, nos deparamos com o problema de apresentar uma entidade tridimensional num meio bidimensional (2D), que é a tela bidimensional. A esse processo de transformação de objetos tridimensionais em um *plano de projeção* bidimensional denominamos de *projeção*.

Este processo tem sido tratado exaustivamente por desenhistas, artistas, arquitetos, que buscaram técnicas e artifícios para sistematizar e solucionar este problema. Pode-se dizer que o olho do observador coloca-se no *centro de projeção*, e o plano que deve conter o objeto ou cena projetada transforma-se no plano de projeção. Os segmentos de reta que saem do centro de projeção e atingem o objeto projetado no plano de projeção, são chamadas de *projetantes* (ou *projetores*), que podem ser vistos na Figura 4.5.

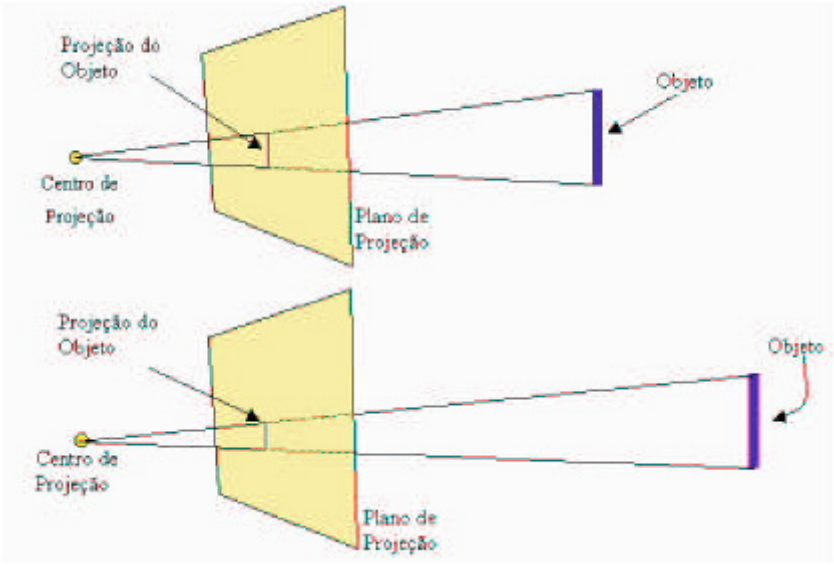

Figura 4.5: Projeção de uma imagem

A classe das projeções é conhecida como *projeção geométrica planar*, uma vez que a projeção é feita sobre um plano e não sobre uma superfície curva. Projeções geométricas planares, também referidas simplesmente como projeções, podem ser divididas em duas classes básicas: *perspectiva* e *paralela*. A diferença entre essas duas classes está na relação entre o centro de projeção e o plano de projeção. Se a distância entre eles for finita, então a projeção é perspectiva, enquanto que se essa distância for infinita então a projeção é paralela. As Figuras 4.6 e 4.7 ilustram estes dois casos. A projeção paralela possui este nome porque, com o centro de projeção infinitamente distante, as projetantes são paralelas. Quando definimos uma projeção perspectiva, o centro de projeção é explicitado, enquanto que na projeção paralela deve-se informar apenas a direção da projeção.

As projeções perspectiva e paralela podem ser definidas através de matrizes 4×4, o que é interessante para a composição de transformações juntamente com a projeção. A seguir, vamos detalhar cada uma destas projeções e suas matrizes.

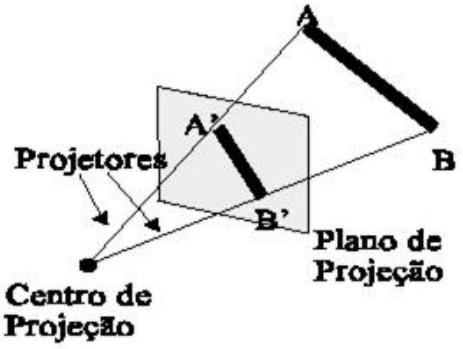

Figura 4.6: A projeção perspectiva

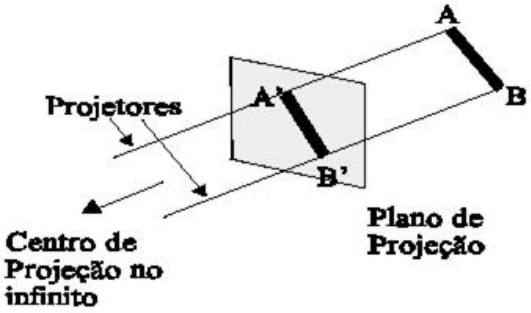

Figura 4.7: A projeção paralela

4.3.1 Projeção Perspectiva

A projeção em perspectiva cria um efeito visual similar ao de sistemas fotográficos e ao da visualização humana. É utilizado quando se deseja um certo grau de realismo, uma vez que é possível se ter a percepção tridimensional da cena [FD84].

Os desenhos em perspectiva são caracterizados pelo *encurtamento perspectivo* e pelos *pontos de fuga*. O encurtamento perspectivo é a ilusão de que os objetos e comprimentos são cada vez menores à medida que sua distância ao centro de projeção aumenta. A projeção em perspectiva de qualquer conjunto de linhas que não são paralelas ao plano de projeção convergem para um ponto de fuga.

Projeções perspectivas são categorizadas pelo seu número de pontos de fuga principais, ou seja, o número de eixos que o plano de projeção intercepta. A Figura 4.8 mostra uma projeção perspectiva de um cubo vista por dois ângulos diferentes. Observe que esta projeção possui apenas um ponto de fuga, pois somente as linhas paralelas ao eixo z converge, e as linhas paralelas aos eixos x e y continuam paralelas.

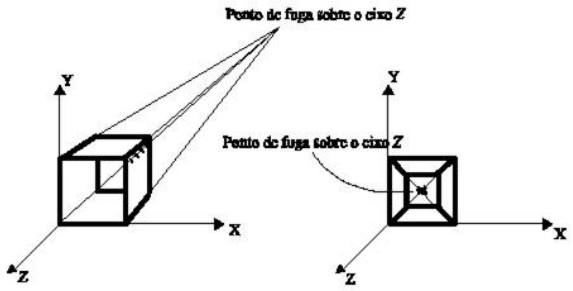

Figura 4.8: Projeções de um cubo com um ponto de fuga

Projeções perspectivas com dois pontos de fuga (quando dois eixos principais são interceptados pelo plano de projeção) são mais comumente usadas em arquitetura, engenharia, desenho publicitário e projeto industrial. Já as projeções perspectivas com três pontos de fuga são bem menos utilizadas, pois adicionam muito pouco em termos de realismo comparativamente às projeções com dois pontos de fuga, e são mais difíceis de serem construídas.

Vamos supor agora que, no caso de projeção perspectiva, o plano de projeção é normal ao eixo z, em $z = d$. Assim, seja o plano de projeção que se encontra a uma distância d da origem, e P o ponto que será projetado sobre ele. Iremos calcular o ponto $P_p = (x_p, y_p, z_p)$, que é a projeção perspectiva de $P = (x, y, z)$ sobre o plano de projeção em $z = d$.

Usando a semelhança de triângulos da Figura 4.9, temos:

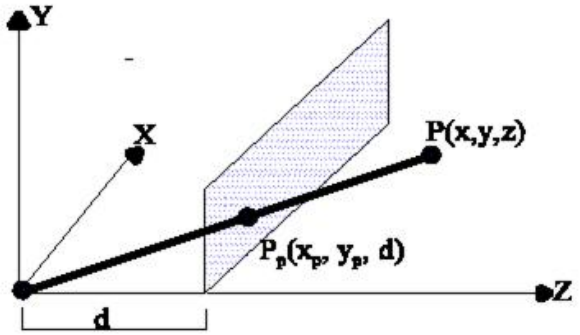

Figura 4.9: Cálculo da projeção perspectiva

$$\frac{x_p}{d} = \frac{x}{z},\ x_p = \frac{dx}{z} = \frac{x}{z/d} \tag{4.1}$$

e

$$\frac{y_p}{d} = \frac{y}{z},\ y_p = \frac{dy}{z} = \frac{y}{z/d} \tag{4.2}$$

A distância d pode ser vista como o fator de escala que se aplica a x_p e y_p. Na projeção perspectiva, a divisão por z faz com que os objetos mais distantes do plano de projeção pareçam menores do que os que estão mais próximos. Além disso, z pode assumir qualquer valor (com exceção de $z = 0$). Consideremos agora a seguinte matriz, que é a matriz da projeção perspectiva:

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}.$$

Multiplicando a matriz $\mathbf{M}$ pelo ponto P em coordenadas homogêneas, teremos um ponto $Q = [x\ y\ z\ z/d]^T$:

$$Q = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix} = \mathbf{M}P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}.$$

Dividindo Q por z/d iremos obter a representação equivalente do ponto Q:

$$Q' = \begin{bmatrix} \frac{x}{z/d} \\ \frac{y}{z/d} \\ d \\ 1 \end{bmatrix} = \begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix}.$$

Então, para obter as coordenadas cartesianas, temos:

$$(x_p, y_p, z_p) = \left(\frac{x}{z/d}, \frac{y}{z/d}, d \right)$$

Vemos que a formulação deriva diretamente do resultado das equações 4.1 e 4.2, e

a coordenada $z_p = d$ resulta da posição do plano de projeção sobre o eixo z (normal ao eixo z).

4.3.2 Projeção Paralela

A projeção paralela é uma visualização menos realística comparada à projeção perspectiva, e a razão disto é que a projeção paralela não possui encurtamento perspectivo. No entanto, assim como na perspectiva, os ângulos são preservados somente nas faces do objeto que são paralelas ao plano de projeção.

As projeções paralelas são categorizadas em dois tipos, baseado na relação da direção de projeção e na *normal*[1] do plano de projeção. Nas projeções paralelas *ortográficas*, estas direções são as mesmas, enquanto que nas projeções paralelas *oblíquas* não são. Ou seja, na projeção ortográfica a direção da projeção é normal ao plano de projeção, enquanto que na projeção oblíqua, as projetantes são inclinadas em relação ao plano de projeção formando um ângulo α.

Para a projeção paralela, suponhamos que o plano de projeção está em $z = 0$. Na projeção ortográfica, como a direção da projeção é a mesma direção da normal ao plano de projeção, será o eixo z neste caso. Assim, o ponto P se projeta como

$$x_p = x,\ y_p = y,\ z_p = 0.$$

Sua projeção é expressa pela matriz:

[1] A normal é o vetor que é perpendicular a um determinado plano

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Poderemos escrever este resultado no nosso sistema de coordenadas homogêneas:

$$\begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Consideremos agora a projeção oblíqua. Poderemos escrever a sua matriz em termos de α e l mostrados na Figura 4.10. Nesta figura, o cubo unitário é projetado no plano xy. Note que o ponto $P(0, 0, 1)$, que é um ponto atrás do cubo, pode ser projetado no plano xy como $(l\cos\alpha, l\sin\alpha, 0)$.

A reta projetante, que não é perpendicular ao plano de projeção, deve passar por P e P', conforme podemos observar na Figura 4.11. A direção desta reta projetante é dada por $P' - P = (l\cos\alpha, l\sin\alpha, -1)$ que faz um ângulo β com o plano xy.

Consideremos agora um ponto genérico (x, y, z) e a sua projeção oblíqua dada por $(x_p, y_p, 0)$. As equações para os valores de projeção de x e y em função de z são obtidas por semelhança de triângulos como descrito na Figura 4.12.

$$\frac{x - x_p}{l\cos\alpha} = \frac{y - y_p}{l\sin\alpha} = \frac{z}{-1}.$$

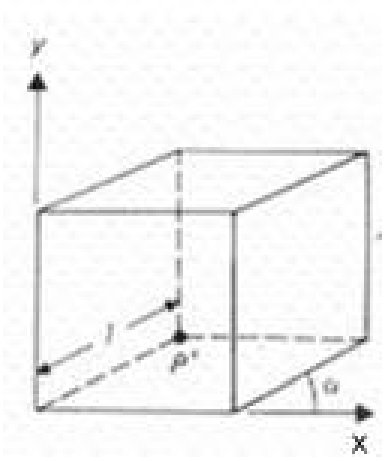

Figura 4.10: Projeção oblíqua de um cubo (P' é a projeção de $P(0,0,1)$)

Desta relação, temos

$$x_p = x + zl\cos\alpha$$

e

$$y_p = y + zl\sin\alpha.$$

Chegamos, assim, à matriz de projeção oblíqua

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & l\cos\alpha & 0 \\ 0 & 1 & l\sin\alpha & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

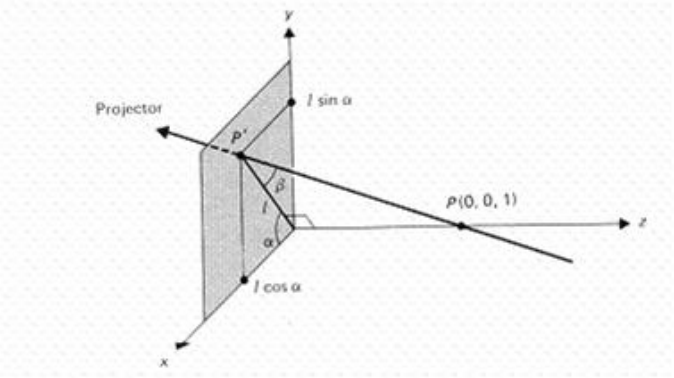

Figura 4.11: Projeção oblíqua de $P(0,0,1)$ em $P'(l\cos\alpha, l\sin\alpha, 0))$

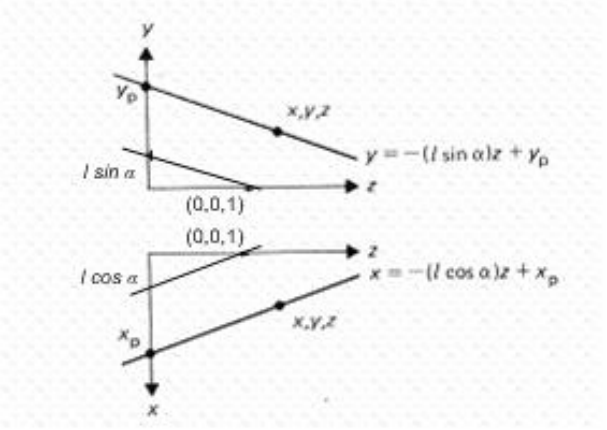

Figura 4.12: Projeção oblíqua de (x, y, z) em $(x_p, y_p, 0)$

4.4 O que é *OpenGL*?

OpenGL é uma biblioteca de rotinas gráficas de modelagem, manipulação de objetos e exibição tridimensional que permite a criação de aplicações que usam computação gráfica. Seus recursos permitem ao usuário criar objetos gráficos com qualidade, de modo rápido, além de incluir recursos avançados de animação, tratamento de imagens e texturas, onde é possível ter visualização em vários ângulos.

O OpenGL também pode ser definido como uma interface de software para dispositivos de hardware. Esta interface consiste em cerca de 150 comandos distintos

usados para especificar os objetos e operações necessárias para produzir aplicativos tridimensionais interativos. O OpenGL foi desenvolvido independente de interface de hardware para ser implementado em múltiplas plataformas.

A biblioteca OpenGL foi introduzida em 1992 pela *Silicon Graphics*, no intuito de conceber uma API (Interface de Programação de Aplicação) gráfica independente de dispositivos de exibição. Com isto, seria estabelecida uma ponte entre o processo de modelagem geométrica de objetos, situadas em um nível de abstração mais elevado, e as rotinas de exibição e de processamento de imagens implementadas em dispositivos (hardware) e sistemas operacionais específicos. A função utilizada pelo OpenGL para desenhar um ponto na tela, por exemplo, possui o mesmo nome e parâmetros em todos os sistemas operacionais nos quais OpenGL foi implementada, e produz o mesmo efeito de exibição em cada um destes sistemas.

Diante das funcionalidades providas pelo OpenGL, tal biblioteca tem se tornado um padrão amplamente utilizado na indústria de desenvolvimento de aplicações. Este fato tem sido adotado também pela facilidade de aprendizado, pela estabilidade das rotinas, pela boa documentação disponível e pelos resultados visuais consistentes para qualquer sistema de exibição concordante com este padrão. Diversos jogos, aplicações científicas e comerciais tem utilizado OpenGL como ferramenta de apresentação de recursos visuais, principalmente com a adoção deste padrão por parte dos fabricantes de placas de vídeo destinadas aos consumidores domésticos.

As especificações do OpenGL não descrevem as interações entre OpenGL e o sistema de janelas utilizado (MS Windows, X Window etc). Assim, tarefas comuns em uma aplicação, tais como criar janelas gráficas, gerenciar eventos provenientes de mouse e teclado, e apresentação de menus ficam a cargo de bibliotecas próprias de cada sistema operacional. Neste trabalho será utilizada a biblioteca GLUT (OpenGL ToolKit) para gerenciamento de janelas.

Desde sua introdução, OpenGL transformou-se num padrão extensamente utilizado pelas indústrias. OpenGL promove a inovação e acelera o desenvolvimento de aplicações incorporando um grande conjunto de funções. Entre os recursos gráficos disponíveis pelo OpenGL, temos os modos de desenho de pontos, ajuste de largura de linhas, aplicação de transparência, atenuação de serrilhamento (anti-aliasing), mapeamento de superfícies com textura, seleção de janela de desenho, manipulação de fontes/tipos de iluminação e sombreamento, transformação de sistemas de coordenadas e transformações em perspectiva e combinação de imagens (blending).

4.4.1 Transformações e Perspectiva em OpenGL

Sistemas gráficos como o OpenGL trabalham com o conceito de matriz corrente para tratar as transformações. Desta forma, apenas uma matriz fica armazenada no sistema gráfico para esta função. Todos os vértices das primitivas que estão sendo definidas são transformados por ela. O sistema gráfico fornece funções para iniciar e alterar esta matriz. No OpenGL esta matriz é chamada de *matriz de modelagem e visualização (model view)* e as transformações são acumuladas à direita. Ou seja, ao fornecermos ao sistema uma nova matriz **M**, ela é multiplicada pela esquerda pela matriz corrente **C** e a nova matriz corrente assume o valor **CM**. Geometricamente isto significa que a transformação **M** ocorre antes das transformações acumuladas em **C**.

O sistema OpenGL implementa um mecanismo de *pilha* para a matriz de transformação. Isto porque a idéia central das transformações em OpenGL é que elas são cumulativas, ou seja, podem ser aplicadas umas sobre as outras. A cada transformação a matriz de transformação é alterada e usada para desenhar os objeto a partir daquele momento, até que seja novamente alterada. A pilha é uma estrutura

muito utilizada em computação [SM94]. Ela é denominada por **LIFO** (*Last In First Out*), ou seja, o último elemento que entra é o primeiro que sai da pilha. Com ela o programador pode recuperar matrizes através de mecanismos de *push* (inserção na pilha) e *pop* (remoção da pilha). Com o mecanismo de *push* e *pop* na pilha podemos garantir que a função retorna sem alterar o estado corrente das transformações, ou seja, sem efeitos colaterais indesejados.

Já para as projeções basta definirmos uma matriz 4×4, apresentada na seção anterior, a qual deveremos aplicar depois da matriz de modelagem e visualização. Entretanto, no caso das projeções perspectivas, deve ser feita uma *divisão perspectiva* no final (que é a divisão pelo fator w apresentado na subseção referente à projeção perspectiva). Nas projeções ortogonais essa divisão não é necessária.

4.4.2 Remoção de Superfícies Ocultas

Dado um objeto tridimensional e uma especificação de visualização definindo o tipo de projeção, o plano de projeção, etc, deseja-se agora determinar quais as arestas e superfícies do objeto que são visíveis a partir do centro de projeção (para projeções em perspectiva) ou ao longo de uma direção de projeção (para projeções paralelas), de forma que apenas serão exibidas as arestas e as superfícies visíveis. Existem dois casos possíveis quando uma face (ou aresta) está oculta; está oculta pelo próprio objeto ou por outros objetos. Apesar da idéia fundamental ser relativamente simples, sua implementação é custosa, o que encorajou a construção cuidadosa de numerosos algoritmos [FD84]. Não se pode afirmar que uma técnica é melhor do que a outra pois irá depender da aplicação (complexidade da cena, tipos de objetos, equipamento disponível, entre outros). Alguns algoritmos fornecem soluções mais rápidas, outros fornecem soluções mais lentas mas que, em compensação, possuem mais realismo e

detalhes (incluem sombras, transparência, etc).

O *OpenGL* possui um depth buffer que trabalha através da associação de uma profundidade, ou distância, do plano de visualização (geralmente o plano de corte mais próximo do observador) com cada pixel da window. Inicialmente, os valores de profundidade são especificados para serem o maior possível através do comando *glClear(GL_DEPTH_BUFFER_BIT)*. Entretanto, habilitando o depth-buffering através dos comandos *glutInitDisplayMode(GLUT_DEPTH)* e *glEnable(GL_DEPTH_TEST)*, antes de cada pixel ser desenhado é feita uma comparação com o valor de profundidade já armazenado. Se o valor de profundidade for menor (está mais próximo) o pixel é desenhado e o valor de profundidade é atualizado. Caso contrário as informações do pixel são desprezadas.

Vamos aqui mostrar como o OpenGL trabalha para remover faces ocultas de um mesmo objeto. O OpenGL fornece a opção de desenhar apenas as faces visíveis em relação ao observador. Para situações em que não podemos ver algumas faces do objeto, é possível reduzir o trabalho requerido em se remover faces ocultas apenas eliminando todas as faces invisíveis deste mesmo objeto (*back-face culling*) antes de ser aplicado qualquer outro algoritmo de remoção de superfícies ocultas.

Para se encontrar a face oculta de um polígono, basta analisarmos a normal de cada uma de suas faces. A face será frontal se a normal estiver apontando para o observador, como podemos ver na Figura 4.13. Considerando θ o ângulo entre a normal e o observador, então o polígono estará com a face visível se e somente se $-90^0 < \theta < 90^0$ ou, equivalentemente, $\cos\theta > 0$. Por causa desta segunda condição, ao invés de calcularmos o cosseno, poderemos utilizar o *produto interno*, ou seja, $\mathbf{n}.\mathbf{v} > 0$, onde $\mathbf{n}$ é a normal da face a ser analisada e $\mathbf{v}$ é o vetor de visão do observador.

No OpenGL, a função **glEnable(GL_CULL)** realiza a eliminação das faces ocul-

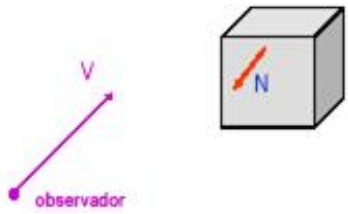

Figura 4.13: A visão do observador e a normal da face

tas do próprio objeto.

4.4.3 A Iluminação em OpenGL

Após a remoção de faces ocultas, o próximo passo é produzir a *iluminação* das faces visíveis pelo observador, levando em conta a(s) fonte(s) de luz, características da superfície dos objetos a serem iluminados, a posição e a orientação destas superfícies e da(s) fonte(s). A iluminação [MWS99] é um recurso gráfico essencial para produzir a sensação de tridimensão na visualização bidimensional da cena. O princípio da iluminação consiste em simular como os objetos refletem as luzes.

Ao observarmos um ponto em um objeto, a cor que vemos é determinada pela soma de energia luminosa portada pelos raios de luz que o atingem diretamente ou após múltiplas reflexões nas superfícies que o cercam.

Uma fonte de luz pode ser descrita de acordo com a função de iluminação

$$\mathbf{I} = \begin{bmatrix} I_r \\ I_g \\ I_b \end{bmatrix}$$

onde I_r, I_g e I_b são os componentes de intensidade das cores vermelha, verde e azul respectivamente. As fontes de luz também podem ser classificadas em:

- Luz Ambiente: geralmente oriunda de fontes largas as quais dissipam luz em todas as direções, possibilitando uma iluminação uniforme da cena. A iluminação ambiente é caracterizada pela intensidade

$$\mathbf{I}_a = \begin{bmatrix} I_{ar} \\ I_{ag} \\ I_{ab} \end{bmatrix}$$

a qual, embora idêntica para todos os pontos da cena, cada superfície pode refletir esta luz diferentemente. As componentes de I_a representam as intensidades das freqüências vermelha, verde e azul da luz ambiente.

- Fonte Pontual: emite luz igualmente em todas as direções. Uma fonte pontual localizada em um ponto p_0 pode ser escrita como

$$\mathbf{I}(p_0) = \begin{bmatrix} I_r(p_0) \\ I_g(p_0) \\ I_b(p_0) \end{bmatrix}$$

onde $I(p_0)$ denota qualquer um dos componentes de $I(p_0)$. A intensidade proveniente de uma fonte pontual é proporcional ao inverso do quadrado da distância entre a fonte e a superfície, ou seja, em um ponto p da superfície a intensidade de luz recebida de uma fonte pontual é dada pela matriz $i(p, p_0) = \dfrac{1}{|p - p_0|^2} I(p_0)$.

- Spot de luz: é caracterizado por um feixe de luz emitido, como um cone de origem p_s apontando na direção l_s e cuja abertura é dada por um ângulo θ.

Assim, a intensidade proveniente desta fonte em um ponto p de uma superfície é uma função do ângulo ϕ entre a direção da fonte (l_s) e a direção da reta que, partindo de p_s encontra a superfície em p. Esta função é, usualmente, definida por $\cos^e \phi$ para $\phi \in (-\theta, \theta)$, onde e é uma constante maior que zero que determina o quão rapidamente a intensidade da luz decresce com $|\phi|$. A Figura 4.14 ilustra um exemplo de spot de luz.

Figura 4.14: Spot de luz

- Fonte de Luz Distante: neste caso, a fonte de luz é caracterizada por uma direção e uma intensidade constantes na cena. Assim, o ângulo de incidência desta luz nas superfícies planas é constante.

O modelo de reflexão utilizado pelo OpenGL é o modelo de *Phong*. Este modelo é uma simplificação da equação integral que descreve a distribuição de energia luminosa nas superfícies de uma cena. O modelo de *Phong* usa quatro vetores para calcular a energia luminosa em cada freqüência (vermelha, verde e azul) para um ponto arbitrário p da superfície. Os vetores são: $\vec{n}$ (vetor normal à superfície em p), $\vec{v}$ (vetor direção da reta com origem em p e passando pelo observador), $\vec{l}$ (vetor direção da reta com origem em p e na direção da fonte de luz) e $\vec{r}$ (vetor direção de reflexão de $\vec{l}$). Assume-se também que cada superfície é composta de um material com várias propriedades. O material pode emitir luz, refletir parte da luz incidente em todas as direções, ou refletir uma parte da luz incidente numa única direção, tal como um espelho.

O modelo de *Phong* é baseado na combinação dos seguintes componentes:

- Reflexão Ambiente: resultado da luz refletida no ambiente; é uma luz que vem de todas as direções. São reflexões indiretas em que todos os objetos são iluminados da mesma forma. A sua contribuição é dada por $I_a = L_a k_a$, onde L_a é a intensidade da luz ambiente em cada um dos pontos da superfície e k_a o coeficiente de reflectividade ambiente da superfície, sendo $0 \leq k_a \leq 1$.

- Reflexão Difusa: também chamada de reflexão *lambertiana*, é a luz que vem de uma direção, atinge a superfície e é refletida em todas as direções; assim, parece possuir o mesmo brilho independente de onde a câmera está posicionada. É o caso das reflexões de objetos lisos foscos. A luminosidade aparente da superfície não depende do observador, mas sim do cosseno do ângulo de incidência da luz. A sua contribuição é dada por $I_d = \frac{k_d}{a + bd + cd^2}(\mathbf{l} \cdot \mathbf{n})L_d$, onde k_d representa a fração da luz difusa refletida, L_d a intensidade da luz difusa incidente em cada um dos pontos da superfície, $\mathbf{l}$ o vetor direção da fonte de luz normalizado, $\mathbf{n}$ o vetor direção da normal do ponto de interesse normalizado e, além disto, foi incorporado o termo quadrático relativo à atenuação que a luz sofre a uma distância d entre a fonte de luz e a superfície.

- Reflexão Especular: luz que vem de uma direção e tende a ser refletida numa única direção, que é o caso das reflexões de objetos lisos brilhantes (metálicos ou polidos). Esta reflexão depende da disposição entre observador, objeto e fonte de luz. Em um espelho perfeito, a reflexão se dá em ângulos iguais e o observador só enxergaria a reflexão de uma fonte pontual se estivesse na posição certa. No modelo de *Phong* simula-se refletores imperfeitos assumindo que a luz é refletida segundo um cone cujo eixo passa pelo observador. Sua contribuição é dada por $I_s = k_s L_s (\mathbf{r} \cdot \mathbf{v})^\alpha$, onde k_s é o coeficiente de reflectividade especular,

L_s a intensidade da fonte de luz, $\mathbf{r}$ o vetor direção de um refletor perfeito normalizado, $\mathbf{v}$ o vetor direção do observador normalizado e α o *coeficiente de especularidade*, que indica quão polida é a superfície. O espelho ideal tem o coeficiente de especularidade igual a infinito. Na prática, usam-se valores entre 5 e 100.

Teremos, finalmente, o modelo de *Phong* para o cálculo da intensidade luminosa de um ponto visível da cena, considerando os componentes de luz ambiente, difusa e especular:

$$I = I_a + I_d + I_s = \frac{1}{a + bd + cd^2}(k_d L_d \mathbf{l} \cdot \mathbf{n} + k_s L_s (\mathbf{r} \cdot \mathbf{v})^{\alpha}) + k_a L_a$$

Este modelo assume que cada fonte de luz na cena pode ter as componentes difusa, ambiente e especular separadamente, para cada uma das três cores primárias (vermelho, verde e azul).

4.5 A Visualização Volumétrica Gerada pelo Aplicativo

Parte do software foi desenvolvido com a finalidade da visualização, através da computação gráfica, de uma simulação de Processo de Poisson homogêneo ou não homogêneo, cujo volume é dado como entrada. Além de selecionar qual a dimensão da grade em que ocorrerá a simulação, o usuário deverá determinar quais os parâmetros de entrada dos Processos Pontuais de Poisson homogêneo e não-homogêneo, e dos Processos Pontuais Indexados, conforme discutimos no capítulo 3. No caso do processo de Poisson não-homogêneo, este parâmetro deverá estar relacionado com a coleção

de conjuntos disjuntos onde $B_i \in \aleph$, em que as v.as. $N(B_i)$ são independentes. Para detalharmos melhor como é feita esta entrada no software, iremos organizar o assunto nas próximas subseções. Antes, porém, descreveremos como está organizado a grade no aplicativo.

Esse volume selecionado está internamente representado por uma matriz (x, y, z) preenchida com zeros, onde cada um desses zeros representa um bloco da grade. A partir daí, teremos tratamentos diferentes no preenchimento desta matriz dependendo do tipo de simulação requerida, ou seja, se for o Processo de Poisson homogêneo ou não-homogêneo.

No caso de Processo de Poisson homogêneo, o domínio escolhido pelo usuário, através de um arquivo de entrada e dentro do volume fixo, será preenchido na matriz por "1´s". Posteriormente, o programa simulará o Processo de Poisson com $\mu(B) = \mu \times |B|$, onde μ é uma constante positiva e $|B|$ é o volume de B. Após gerado o número de pontos, eles serão distribuídos dentro do domínio selecionado pelo usuário. As coordenadas (x, y, z) destes pontos seguirão a distribuição uniforme, conforme foi apresentado no capítulo 3. Para cada ponto gerado (ou seja, para cada coordenada (x, y, z) calculada), verifica-se se tal ponto pertence ao domínio dado pelo usuário. Se pertencer, a posição deste ponto na matriz estará preenchida com "1", logo este ponto aparecerá na grade e estará preenchido com "2" na matriz do programa. Se o ponto gerado não estiver no domínio, a posição dele na matriz estará preenchida com "0" e um novo ponto deverá ser gerado pois este será ignorado. Ao final, teremos um conjunto de pontos gerados pelo Processo de Poisson dentro do domínio requerido pelo usuário.

O caso do Processo de Poisson não-homogêneo é semelhante ao do Processo de Poisson homogêneo, com a diferença de que o domínio escolhido pelo usuário é dividido em subdomínios e cada um desses subdomínios terá o seu próprio parâmetro

(μ) para o processo de Poisson, ou seja, cada subdomínio $S_1, S_2, S_3, \ldots$ terá o seu próprio número de pontos dados pelo Processo de Poisson com parâmetros individuais ($\mu_1, \mu_2, \mu_3, \ \ldots$). Internamente, a matriz será preenchida por inteiros distintos para cada um dos subdomínios e as coordenadas dos pontos serão calculadas e posicionadas na grade analogamente ao processo de Poisson homogêneo.

A grade resultante mostrará os pontos gerados pelo Processo de Poisson preenchidos, o domínio selecionado pelo usuário e o volume fixo conforme podemos observar na Figura 4.15. Pode-se, ainda, visualizar a simulação somente no domínio requerido pelo usuário conforme mostra a Figura 4.16. Pode-se rotacionar a grade em torno dos três eixos e transladá-lo tanto para a direita/esquerda quanto para frente/trás. A fim de obter uma maior percepção tridimensional, foi implementada a iluminação.

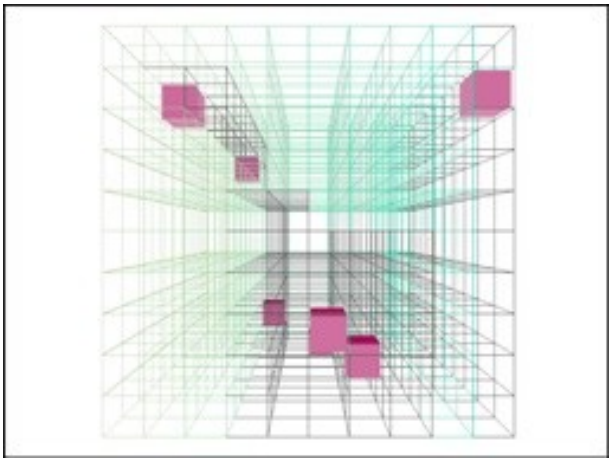

Figura 4.15: Visualização gerada pelo aplicativo

Agora vamos apresentar qual é a linguagem utilizada pelo programa para a entrada dos dados da dimensão da grade e como é feito o desenho de cada um dos blocos que constituem esta grade. No final, será descrito o ambiente de desenvolvimento.

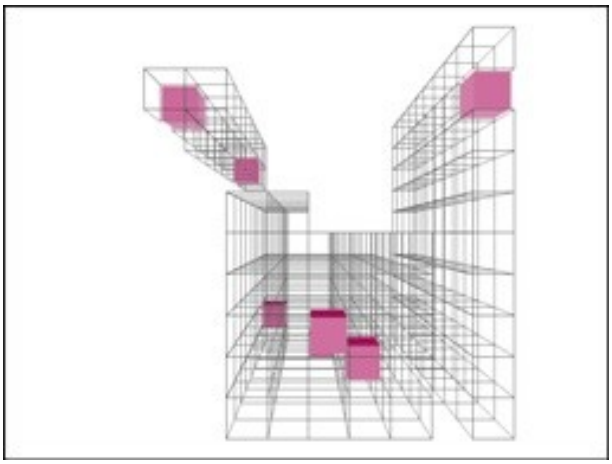

Figura 4.16: Simulação vista somente com o domínio de entrada

4.5.1 Dimensão da grade

Introduziremos agora a linguagem utilizada para a entrada de dados no programa. Estes dados estão relacionados com a dimensão do volume em que será feita a simulação de Processos Pontuais de Poisson homogêneo e não-homogêneo. Além disto, foi implementado o algoritmo 4 apresentado no capítulo 3 deste trabalho.

A primeira linha deste arquivo informará quantos blocos existem para cada um dos eixos, ou seja, para o eixo x, para o eixo y e para o eixo z. Por exemplo, se a primeira linha for 20 10 15, teremos 20 blocos no eixo x, 10 blocos no eixo y e 15 blocos no eixo z; ou seja, no total teremos 3000 blocos para este exemplo.

A segunda linha do arquivo é um código da dimensão dos blocos da grade e, assim como na primeira linha, é constituído por três números consecutivos. Chamando-os de KDX(código especificando a dimensão da grade na direção x), KDY(código especificando a dimensão da grade na direção y) e KDZ(código especificando a dimensão da grade na direção z), o valor que cada um deles pode assumir é 0, 1 ou -1 e cada um destes valores informa uma determinada dimensão conforme podemos analisar na Tabela 4.1. Chamaremos de *camada* o conjunto de blocos pertencentes ao mesmo

plano xz. Voltando ao nosso exemplo, se a segunda linha for $-1-1-1$ significa que os blocos terão a mesma dimensão nos eixos x, y e z.

Código	Valor	Especificações
KDX	-1	As dimensões da grade na direção x são as mesmas para todos os blocos da grade.
KDX	0	As dimensões da grade na direção x são lidas para cada bloco na primeira linha da primeira camada. Estas mesmas dimensões na direção x são atribuídas para todas as outras linhas e todas as outras camadas na grade.
KDX	1	As dimensões na direção x são lidas para todos os blocos da grade na camada um. Estas mesmas dimensões na direção x são atribuídas para todas as outras linhas e todas as outras camadas na grade.
KDY	-1	As dimensões da grade na direção y são as mesmas para todos os blocos da grade.
KDY	0	As dimensões da grade na direção y são lidas para cada bloco na primeira coluna da primeira camada. Estas mesmas dimensões na direção y são atribuídas para todas as outras colunas e todas as outras camadas na grade.
KDY	1	As dimensões na direção y são lidas para todos os blocos da grade na camada um. Estas mesmas dimensões na direção y são atribuídas para todas as outras linhas e todas as outras camadas na grade.
KDZ	-1	As dimensões da grade na direção z são as mesmas para todos os blocos da grade.
KDZ	0	Lê-se uma constante especificando a espessura de cada camada na grade.
KDZ	1	As dimensões da grade na direção z são lidas para cada bloco.

Tabela 4.1: Códigos do arquivo de entrada

As próximas linhas do arquivo descreverão o tamanho do bloco nas direções x, y e z. Os eixos x, y e z podem ser vistos na Figura 4.17. Chamaremos o tamanho do bloco na direção x de *largura* do bloco, o tamanho do bloco na direção y de *altura* do bloco e o tamanho do bloco na direção z de *profundidade* do bloco. A dimensão de cada um dos blocos dependerá do código assumido para *KDX*, *KDY* e *KDZ*.

Para o nosso exemplo, como $KDX = -1$, teremos apenas um valor para a largura de todos os blocos da grade, que no nosso exemplo será 20.0 (assumiremos que os

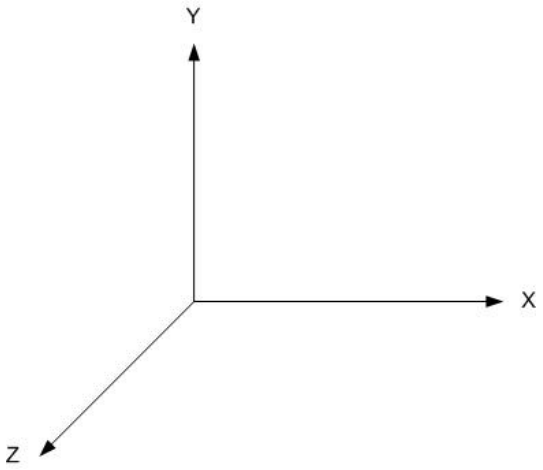

Figura 4.17: Sistema de coordenadas padrão do OpenGL

valores relativos ao tamanho de cada bloco será dado em ponto flutuante). Da mesma forma, $KDY = -1$, então a altura de todos os blocos da grade terá um mesmo valor, que será igual a 15.0 neste exemplo e, similarmente, $KDZ = -1$ o que indica que todos os blocos terão a mesma profundidade que, no nosso caso, será igual a 7.0.

Agora iremos analisar como serão desenhados cada um dos blocos, ou *paralelepípedos*, da grade. Para isto, foi desenvolvida uma rotina específica cujos parâmetros de entrada são a altura, largura e a profundidade do bloco a ser desenhado. Além disto, é importante salientarmos que o ponto em que se começa a desenhar o bloco é o mesmo em que se termina o seu desenho, e chamaremos tal ponto de *cursor*. Com isto, para desenharmos os blocos pertencentes a uma mesma linha (ou fileira), chamaremos esta rotina para o desenho do bloco (informando a sua largura, altura e profundidade de acordo com o arquivo de entrada).

Após desenhado o bloco, deveremos deslocar o cursor de acordo com o valor da largura do bloco desenhado no sentido positivo do eixo x e inicia-se o desenho do próximo bloco. Isto deverá ser feito sucessivamente até que todos os blocos pertencentes a esta linha tenham sido desenhados. Depois, basta transladarmos o cursor

para a posição de desenho do primeiro bloco desta fileira de blocos e, caso exista mais uma fileira de blocos para ser desenhada, deslocarmos o cursor de acordo com o valor da profundidade deste primeiro bloco no sentido negativo do eixo z. Com isto, é possível desenharmos todos os blocos pertencentes a uma mesma camada. Para desenharmos a próxima camada teremos que retornar o cursor na posição inicial de desenho da camada, ou seja, na posição inicial de desenho do primeiro bloco da primeira fileira desenhada desta camada. A partir deste ponto, desloca-se o cursor no sentido negativo do eixo y de forma que o valor desse deslocamento seja a altura deste bloco. Com isto, estaremos prontos para desenharmos a próxima camada. Fazemos este procedimento enquanto houverem camadas a serem desenhadas.

Finalmente iremos descrever como é o procedimento de desenho do bloco. O bloco é formado por seis faces. Cada uma das faces, por sua vez, é formada por quatro vértices e cujas dimensões (altura, largura e profundidade) são definidas pelo arquivo de entrada conforme foi definido anteriormente. A ordem dos vértices no desenho da face é feita no sentido anti-horário. Ou seja, para a face frontal (e a primeira a ser desenhada) teremos como primeiro ponto o superior mais a esquerda, seguido pelo inferior mais a esquerda, o inferior mais a direita e, finalmente, o superior mais a direita. Com esta ordenação, a face estará visível ao observador.

Ao usar iluminação em OpenGL, é necessário especificar qual é a normal do objeto desenhado (através da primitiva **glNormal3f**). A normal indica ao OpenGL qual o sentido em que a face está visível. Se não for especificada a normal, poderemos obter imperfeições nos resultados, como faces que não deveriam ser iluminadas passam a ser iluminadas, entre outros. A normal deverá apontar "para fora" do polígono.

Voltemos ao nosso procedimento de desenho do bloco. Observando a face frontal, teremos então a normal no sentido positivo no eixo z. Isto significa que a normal está apontando para o observador, que é exatamente a direção desejada. Na face posterior,

a normal deverá apontar no sentido oposto ao do observador, ou seja, "para dentro" da tela. Este procedimento com a normal é similar ao problema de remoção de superfícies ocultas apresentado na seção anterior. Se rotacionarmos o bloco 180 graus em torno do eixo x ou em torno do eixo y, a face frontal estará com a face voltada para dentro da tela e a face posterior estará com a face voltada para o observador. Não importa qual das faces estará voltada para o observador, contanto que a normal esteja apontando para ele. Uma vez que a luz se encontra perto do observador, a qualquer momento que a normal estiver apontando para o observador também estará apontando para a luz. Quando isto ocorre, a face fica iluminada. Quanto mais a normal estiver apontando para a luz, mais iluminada a face estará. Como as normais estão apontando para fora do bloco, então não existirá luz dentro do bloco, que é o efeito desejado.

4.5.2 Ambiente de desenvolvimento

O aplicativo foi desenvolvido em um PC com processador AMD K6-2 550Mhz com 384 MB de RAM PC 133, placa de vídeo NVIDIA GeForce 256 com 64 MB de RAM, sistema operacional Windows 98 SE e com plataforma de desenvolvimento Visual C++ 6.0. Também foi testado em um PC com processador AMD Duron com 512 MB de RAM, placa de vídeo NVIDIA GeForce4 MX 440 com AGP8X, sistema operacional Windows XP Professional versão 2002 com plataforma de desenvolvimento Visual C++ 6.0.

Capítulo 5

Fecho Convexo

Será apresentado neste capítulo o algoritmo do *fecho convexo* (ou *convex hull*) numa aplicação tridimensional. A implementação do fecho convexo neste trabalho possibilita uma cobertura espacial no grid gerado pelo aplicativo apresentado no capítulo anterior. Para algumas aplicações, é interessante a existência de tal cobertura, sendo uma alternativa de aproximação dos volumes a serem estudados. Iniciaremos com o conceito do fecho convexo apresentado a seguir.

5.1 O Fecho Convexo no Plano e no Espaço

O fecho convexo é um dos mais fundamentais conceitos da geometria computacional [MdBS98]. Primeiramente introduziremos a sua definição no plano para, logo em seguida, o apresentarmos no espaço tridimensional, o qual estamos interessados.

Um subconjunto $S \in R^n$ é chamado *convexo* se e somente se para qualquer par de pontos $p, q \in S$ o segmento de linha $\overline{pq}$ está completamente contido em S. O fecho convexo $\mathcal{CH}(S)$ de um conjunto S é o menor conjunto convexo que contém S. Mais precisamente, é a interseção de todos os conjuntos convexos que contêm S. É possível visualizarmos o fecho convexo através do seguinte experimento: imagine que os pontos são pregos espalhados em um plano. Em seguida, envolvemo-los com

um elástico de forma que os pregos fiquem todos dentro da área envolvida por ele. Minimizando esta área, teremos formado o fecho convexo onde a fronteira é o elástico.

Similarmente, para o fecho convexo no espaço, teremos apenas que submeter o conceito apresentado para uma dimensão maior. Sendo assim, o fecho convexo de um conjunto P de n pontos é um *politopo* convexo cujos vértices são pontos em P e, desta forma, possui no máximo n vértices. Um politopo é a interseção de hiperplanos no espaço. Formalmente, uma *face* de um politopo convexo é definida como o subconjunto máximo de pontos coplanares na sua fronteira. Uma face de um politopo convexo é necessariamente um polígono convexo. Uma *aresta* de um polígono convexo é uma aresta de qualquer uma de suas faces.

5.2 Computando o Fecho Convexo no Espaço Tridimensional

Seja P um conjunto de n pontos no espaço tridimensional. Especificamente, seja P o número de vértices dos blocos que compõem o grid descrito no capítulo anterior. Iremos agora computar $\mathcal{CH}(P)$, o fecho convexo de P, utilizando um algoritmo randômico incremental descrito a seguir.

A construção incremental inicia-se pela escolha de quatro pontos em P *não-coplanares*, de forma que o seu fecho convexo seja um tetraedro. Quatro pontos são não-coplanares quando estes quatro pontos não pertencem ao mesmo plano. Isto pode ser feito da seguinte maneira: são escolhidos dois pontos, p_1 e p_2, em P. O próximo ponto, p_3, deverá ser escolhido de tal forma que seja *não-colinear* à p_1 e p_2. Ou seja, p_3 não deverá pertencer a mesma reta formada por p_1 e p_2. Para isto, iremos testar se o vetor formado por p_1 e p_2 é *linearmente dependente* do vetor formado por

p_1 e p_3. Se for, os pontos p_1, p_2 e p_3 serão colineares, logo iremos escolher outro ponto em P, p_3 é devolvido a P e um novo teste deverá ser feito. Se não forem linearmente dependentes, p_1, p_2 e p_3 formarão um triângulo. Agora iremos escolher um quarto ponto em P, p_4, que não esteja no mesmo plano formado por p_1, p_2 e p_3 para que seja possível formar o tetraedro. Neste caso, deveremos recorrer ao *teste de coplanaridade* [San73] dada pela Definição 5.2.1.

Definição 5.2.1 *Teste de Coplanaridade*

Quatro pontos p_1, p_2, p_3 e p_4 são coplanares se e somente se os vetores $\vec{p_1p_2}$, $\vec{p_1p_3}$ e $\vec{p_1p_4}$ são linearmente dependentes. Sabe-se ainda que esses três vetores são linearmente dependentes se e somente se $\left[\vec{p_1p_2}, \vec{p_1p_3}, \vec{p_1p_4}\right] = 0$ (produto misto entre os vetores $\vec{p_1p_2}$, $\vec{p_1p_3}$ e $\vec{p_1p_4}$). Assim, uma condição necessária e suficiente para que um ponto p_4 pertença ao plano determinado pelos pontos p_1, p_2 e p_3 é que

$$\left[\vec{p_1p_2}, \vec{p_1p_3}, \vec{p_1p_4}\right] = 0.$$

Logo, para o nosso fecho convexo, o próximo ponto escolhido, p_4, não deverá satisfazer as condições do teste de coplanaridade ($\left[\vec{p_1p_2}, \vec{p_1p_3}, \vec{p_1p_4}\right] \neq 0$) pois queremos que p_4 não pertença ao plano do triângulo formado por p_1, p_2 e p_3. Teremos, por fim, um tetraedro formado por p_1, p_2, p_3 e p_4.

A partir de agora iremos computar o fecho convexo para os pontos restantes, ou seja, para p_i, $5 \leq i \leq n$. Esses pontos serão escolhidos de forma aleatória e a cada p_i escolhido, este ponto deverá ser descartado de P. Imagine que estamos em algum passo do algoritmo e o próximo ponto a ser escolhido de P é p_r. Logo, o nosso fecho convexo até então formado é $\mathcal{CH}(P_{r-1})$ e iremos construir $\mathcal{CH}(P_r)$. Note que existem dois casos a serem analisados.

- p_r está dentro de $\mathcal{CH}(P_{r-1})$. Neste caso, nada será feito pois p_r já pertence ao fecho convexo formado e não precisaremos alterar $\mathcal{CH}(P_{r-1})$. Outro ponto (p_{r+1}) deverá ser escolhido.

- p_r está fora de $\mathcal{CH}(P_{r-1})$. Como p_r não pertence ao fecho convexo $\mathcal{CH}(P_{r-1})$, então p_r consegue ver algumas das faces de $\mathcal{CH}(P_{r-1})$. Estas faces visíveis por p_r formam a fronteira entre a região visível por p_r e a região invisível por p_r (formada pelas faces de $\mathcal{CH}(P_{r-1})$ que estão "atrás" das faces visíveis por p_r). Chamaremos tal fronteira de *horizonte* de p_r.

Agora vamos mostrar o que é uma face ser *visível* em relação a um ponto p em termos geométricos. Por convenção, uma face f é definida pelo sentido anti-horário de suas arestas, como pode ser observado na Figura 5.1. Para testarmos se p consegue ver f, verificaremos em qual região p se encontra em relação ao plano que contém f, uma vez que este plano separa o espaço em duas regiões distintas (Figura 5.2). Apenas em uma destas regiões p conseguir á ver f necessariamente. Se p estiver no mesmo plano de f, ele não verá f. E para sabermos se p se encontra na região que vê f, basta analisarmos o ângulo θ formado entre a normal $\vec{N}$ de f e o vetor $\vec{V}$ que liga a origem da normal a p. Assim como no cálculo das faces visíveis de um objeto apresentado no capítulo anterior, basta calcularmos o *produto interno* entre $\vec{N}$ e $\vec{V}$ (por definição, $\vec{N} \,.\, \vec{V} = \left\|\vec{N}\right\| . \left\|\vec{V}\right\| . \cos\theta$, onde $0 \leq \theta \leq \pi$). Se $\vec{N} \,.\, \vec{V} > 0$, significa que $0 \leq \theta < \frac{\pi}{2}$ e p conseguirá ver f. Caso contrário, se $\vec{N} \,.\, \vec{V} \leq 0$, então $\frac{\pi}{2} \leq \theta \leq \pi$ e p não verá f.

O *horizonte* faz um importante papel na transformação de $\mathcal{CH}(P_{r-1})$ para $\mathcal{CH}(P_r)$. Ela será o limite entre as faces que permanecerão inalteradas no fecho convexo (as faces invisíveis a p_r) e as faces que serão substituídas por novas faces que conectarão p_r a $\mathcal{CH}(P_{r-1})$, conforme podemos verificar na Figura 5.3. É importante para a

Figura 5.1: Sentido anti-horário das arestas da face

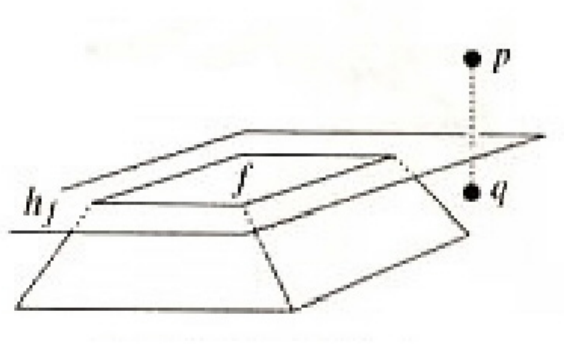

Figura 5.2: A face f é visível a p, mas não é para q.

implementação deste algoritmo mantermos estruturas eficientes para armazenarmos todas essas informações. Um exemplo destas estruturas é a criação de uma lista encadeada que armazena as faces do fecho convexo em construção. Maiores detalhes das estruturas utilizadas e de como identificar o *horizonte* serão descritos na próxima seção.

Pelo raciocínio descrito acima, é possível determinarmos quais as faces são visíveis por p_r. Basta então selecionarmos estas faces na lista encadeada, removermos tais faces da lista e acrescentarmos novas faces que conectam p_r o *horizonte*.

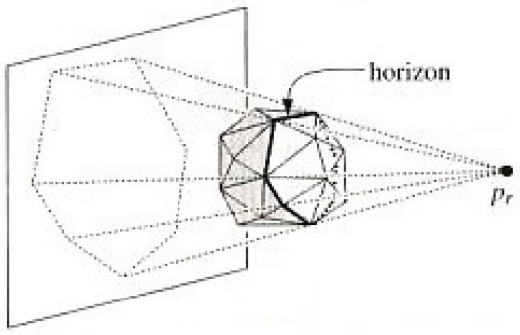

Figura 5.3: O *horizonte* de $\mathcal{CH}(P_{r-1})$ em relação a p_r.

Este procedimento deverá ser feito enquanto existirem pontos a serem inseridos no fecho convexo. Na próxima seção, além da descrição das estruturas deste problema será apresentado o algoritmo do fecho convexo tridimensional.

5.3 Estruturas e Algoritmo

Iremos começar com a descrição da lista encadeada que armazena as faces do fecho convexo, denominada *DCEL* (*Double-Connected Edge List*).

Como foi dito anteriormente, uma face é caracterizada pela orientação anti-horária de suas arestas. Logo, para armazenar as informações de uma face, basta nos referirmos a apenas uma aresta da face e indicar onde encontrar as informações da próxima aresta e da aresta anterior.

Chamaremos de *twin(a)* a aresta com o sentido inverso à aresta a. Um exemplo de *twin* podemos ver na Figura 5.4. Note que a face adjacente f' a uma face f é caracterizada por twin(a), onde $a \in f$. Com estas propriedades formamos a DCEL, cujos campos estão descritos a seguir:

1. *HalfEdge*: índice da face, mais precisamente o índice de uma das arestas da

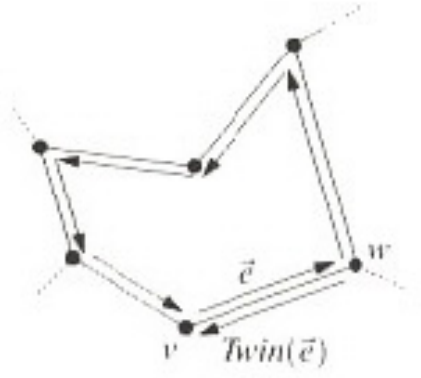

Figura 5.4: A aresta $\vec{e}$ e twin($\vec{e}$).

face;

2. *Twin*: índice da aresta com o sentido oposto da aresta indicada por HalfEdge;

3. *Next*: índice da próxima aresta da face;

4. *Prev*: índice da aresta anterior da face;

5. *Prox*: endereço do próximo nó da lista;

Inicialmente, o programa possui apenas os pontos dos vértices dos blocos do grid. As coordenadas desses pontos são armazenadas em uma lista encadeada chamada *PNT*. A ordem de inserção dos pontos na PNT é feita à medida que são calculadas as dimensões dos blocos, sendo que os últimos pontos inseridos (que são os primeiros da lista, pois esta lista funciona como uma pilha, onde os últimos elementos inseridos serão os primeiros a serem retirados) pertencem à face invisível do bloco em relação ao observador na posição inicial do grid, devido à própria construção do bloco conforme pudemos observar no capítulo anterior. Esta informação se torna importante para o início do algoritmo do fecho convexo: a primeira face do algoritmo será constituída

pelos três primeiros pontos de PNT, ou seja, estará na face invisível para o observador em seu primeiro instante. Logo, a ordem das arestas que constituirão tal face deverá respeitar o sentido horário para o usuário inicialmente. Isto porque esta face só estará visível para o observador se rotacionarmos o grid no ângulo de $\theta = \pi$ em relação ao eixo x ou em relação ao eixo y.

Assim conseguiremos identificar a ordem de inserção das arestas da primeira face do fecho convexo formado na DCEL, o que é o suficiente para sabermos a ordem de inserção de todas as arestas das faces que serão construídas em sua seqüência pelo algoritmo. Isto se deve ao fato de, pela construção do algoritmo, serão inseridas na DCEL apenas faces adjacentes às faces já previamente existentes no fecho convexo, e essas novas faces inseridas estão caracterizadas por arestas presentes na DCEL, o que significa que estarão orientadas de acordo com o sentido das arestas as quais formam o *horizonte*.

Existe também uma estrutura que, pelo o que iremos apresentar a seguir, torna o nosso algoritmo mais eficiente. Esta estrutura, chamada *Lista de Conflitos*, é formada por duas listas encadeadas: a primeira delas, chamada de *lista de pontos em conflito* ou simplesmente CPONTOS, armazenará os pontos que ainda não foram inseridos no fecho convexo até o momento além de, para cada um desses pontos, as faces visíveis deste fecho convexo. A segunda lista, chamada *lista de faces em confito* ou CFACES, armazena as faces presentes no fecho convexo até então formado, e para cada uma destas faces são armazenados os pontos que vêem estas faces, caso exista algum. Sempre que for inserido um ponto p_r em $\mathcal{CH}(P_{r-1})$, basta procurarmos o nó referente a p_r em CPONTOS para sabermos quais as faces visíveis por p_r e, assim, substituí-las pelas novas faces que conectam p_r o seu *horizonte* (vamos identificar esse nó como CPONTOS(p_r)).

O termo *conflito* indica que, em relação ao ponto e a face que se referem um ao

outro, ambos não podem existir ao mesmo tempo na estrutura, ou seja, uma vez a face presente na Lista de Conflitos o ponto não estará presente na lista e vice-versa.

Na inicialização da Lista de Conflitos, para cada um dos pontos p_i, $5 \leq i \leq n$ que são os pontos que ainda não foram inseridos no fecho convexo, armazenam-se as faces do tetraedro que são visíveis para p_i de acordo com o critério apresentado na seção anterior.

Ao adicionarmos um ponto p_r em $\mathcal{CH}(P_{r-1})$, a primeira ação a ser feita é identificar as faces visíveis por p_r através de CPONTOS(p_r). O *horizonte* é constituída pelas arestas das faces visíveis cujas faces adjacentes não são visíveis por p_r (de acordo com a definição de *horizonte*). Para identificarmos estas arestas, deveremos verificar para cada aresta a das faces visíveis, se *twin*(a) (armazenada na DCEL) pertence ou não a uma face visível por p_r. Se não pertencer, a fará parte do *horizonte*.

Após identificarmos o *horizonte*, poderemos descartar p_r e as faces visíveis por p_r da Lista de Conflitos. As novas faces se formarão ao conectarmos cada uma das arestas do *horizonte* a p_r, que podemos observar na Figura 5.5. A partir daí, teremos duas situações possíveis em relação da nova face f': que ela esteja no mesmo plano de alguma face f de $\mathcal{CH}(P_{r-1})$ ou não. Observe que f' só estará no mesmo plano de f quando f não é vista por p_r. Se isto ocorrer, f continuará inalterada em $\mathcal{CH}(P_{r-1})$ e um novo triângulo (f') será formado conectando p_r à aresta comum a f e a f' e que faz parte do horizonte. Neste caso, a Lista de Conflitos da nova face f' será a mesma de f, uma vez que o plano que contém a nova face não se altera. Para sabermos se f e f' se encontram no mesmo plano, basta utilizarmos a Definição 5.2.1 da seção anterior.

Analisaremos agora o caso em que a nova face não se encontra no mesmo plano de nenhuma outra face. Supondo que um certo ponto p_t, $r < t \leq n$ possa ver a nova face, que agora chamaremos de f. Logo, p_t também poderá ver a aresta e oposta

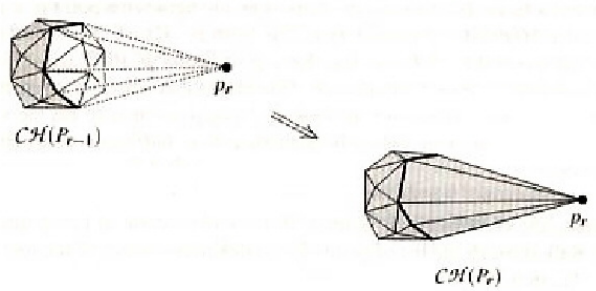

Figura 5.5: Conectando as arestas do *horizonte* a p_r.

ao ponto p_r. Esta aresta e pertence ao *horizonte* de e em $\mathcal{CH}(P_{r-1})$. Uma vez que $\mathcal{CH}(P_{r-1}) \subset \mathcal{CH}(P_r)$, a aresta e pode ser vista por p_t em $\mathcal{CH}(P_{r-1})$, ou seja, antes de inserirmos a nova face f. Isto acontece quando pelo menos uma das faces incidentes à e em $\mathcal{CH}(P_{r-1})$ é vista por p_t. Isto implica que a Lista de Conflitos da face f pode ser calculada apenas testando os pontos na Lista de Conflitos das duas faces f_1 e f_2 incidentes a aresta e pertencente ao *horizonte* em $\mathcal{CH}(P_{r-1})$, conforme podemos visualizar na Figura 5.6. Isto torna eficiente a atualização da Lista de Conflitos, já que não precisaremos ao longo do algoritmo estarmos sempre testando todos os pontos que ainda serão inseridos no fecho convexo.

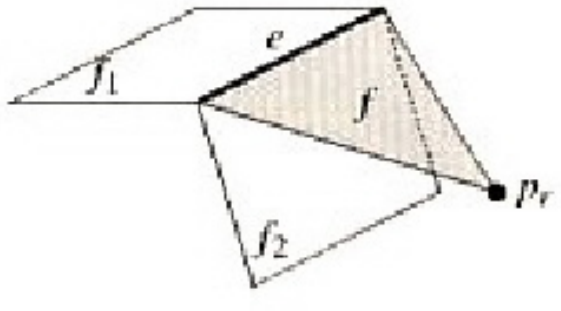

Figura 5.6: Inclusão da Face f

O Algoritmo 5.3.1 apresenta a construção do fecho convexo. A Figura 5.7 mostra

uma ilustração do fecho convexo gerado pelo algoritmo.

Algoritmo 5.3.1 *FECHO CONVEXO*

Entrada: Conjunto P de n pontos no espaço.

Saída: O fecho convexo $\mathcal{CH}(P)$ de P.

1. *Encontre, em P, quatro pontos p_1, p_2, p_3 e p_4 que formem um tetraedro.*
2. $\mathcal{C} \leftarrow \mathcal{CH}(p_1, p_2, p_3, p_4)$.
3. *Faça a permutação dos pontos restantes $(p_5, p_6, \ldots, p_n)$.*
4. *Inicialize a Lista de Conflitos $\mathcal{LG}$ com todos os pares visíveis (p_t, f), onde f é uma face de $\mathcal{C}$ e $t > 4$.*
5. *Para $r := 5$ até n faça*
6. *Se existir pelo menos uma face vista por p_r, faça*
7. *Adicione na lista $\mathcal{L}$ todas as arestas visíveis por p_r, cujos* twins *destas arestas pertençam a faces não visíveis por p_r. No final, $\mathcal{L}$ terá o* horizonte *relativa a p_r.*
8. *Armazenam-se os índices das faces que vêem p_r da Lista de Conflitos em um vetor. Estas faces serão excluídas mais adiante.*
9. *Remova p_r da Lista de Conflitos, ou seja, remove-se o nó relativo a p_r de CPONTOS. Além disto, atualize CFACES onde para cada face f visível por p_r, remova p_r da relação dos pontos visíveis por f.*
10. *Para todo $e \in \mathcal{L}$, faça*

11. *Cria-se um novo nó do tipo CFACES, que será a nova face que conecta e a* p_r*. Chamemos este novo nó de* novo_no*, e terá como índice e. Este nó só será adicionado em CFACES no final do algoritmo.*

12. *Cria-se também uma lista auxiliar, AUX, que irá atualizar CFACES no final do algoritmo, que terá novas faces inseridas.*

13. *Se esta nova face constituída por e e* p_r *for coplanar com a face constituída por* twin*(e), então*

14. *Os pontos que vêem esta nova face serão os mesmos que vêem* twin*(e). Acrescentamos estes pontos na relação dos pontos visíveis por* p_r *em* novo_no*.*

15. *Senão deveremos verificar, para cada ponto que vê a face que contém e (a qual será removida adiante) e* twin*(e), se esses pontos vêem a nova face. Se sim, acrescentamos este ponto no vetor dos pontos visíveis por* p_r *em* novo_no*.*

16. $AUX \leftarrow$ novo_no*.*

17. *Fim de para todo* $e \in \mathcal{L}$*.*

18. *Remova todas as faces que vêem* p_r *da Lista de Conflitos (que são as faces cujos índices estão no vetor dado pelo passo 8). Para cada uma destas faces removidas, atualiza CPONTOS.*

19. *Atualize CPONTOS de acordo com as novas faces que serão inseridas (AUX).*

20. $CFACES \leftarrow AUX$*.*

21. $\mathcal{C} \leftarrow \mathcal{CH}(p_r)$*.*

22. *Fim de para* $r := 5$ *até* n*.*

23. Retorne $\mathcal{C}$.

Figura 5.7: O fecho convexo gerado pelo algoritmo 5.3.1

Capítulo 6

Conclusão e Trabalhos Futuros

Foi desenvolvido, através deste trabalho, a definição e implementação de um sistema para a simulação e visualização de Processos de Poisson homogêneos e não homogêneos em um determinado volume. O volume onde é aplicada a simulação é denominado *grid*, que é particionado em blocos cujas dimensões são previamente definidas. Além disto, foi criada uma rotina de geração de números aleatórios para o sistema e realizados testes estatísticos de ajuste e de independência a fim de validar os resultados obtidos.

A simulação pode ser visualizada com o uso da biblioteca gráfica OpenGL, que é uma ferramenta amplamente usada na computação gráfica e com o atrativo da portabilidade, já que é independente de plataforma. Foi também desenvolvida uma cobertura convexa, denominada *envelope convexo* (ou *convex hull*). O algoritmo do envelope convexo é aplicado no conjunto de pontos formados pelos vértices dos blocos que constituem o grid.

Vemos como extensões possíveis desse trabalho a otimização das estruturas de dados, como no caso da matriz onde está armazenada a localização dos pontos gerados pela simulação. Se a matriz for esparsa, ou seja, poucos pontos foram gerados na simulação, então sugere-se o armazenamento desses pontos em uma lista encadeada, por ser menos custosa.

Uma outra alternativa seria desenvolver diferentes texturas para a cobertura, de forma que o usuário pudesse optar pela textura que fosse mais conveniente para a sua aplicação. Um efeito interessante é o da transparência (ou blending) aplicado no envelope convexo, cuja vantagem é a visualização da simulação no seu interior.

Também seria interessante um estudo sobre a aplicação de superfícies mais suaves no conjunto de pontos formado pelos vértices dos blocos do grid. O OpenGL suporta curvas e superfícies de Bézier através de mecanismos denominados *avaliadores* (ou *evaluators*) [MdBS98], os quais não requerem espaçamento uniforme dos pontos de controle e, por usarem curvas e superfícies de Bézier para gerar outros tipos de curvas e superfícies polinomiais apenas com a geração de novos pontos de controle, este mecanismo torna-se flexível. Os avaliadores são utilizados para gerar curvas e superfícies em uma, duas, três e até quatro dimensões.

Além disso, outra possibilidade que está em fase de construção é a alteração no padrão de linguagem em que os dados são fornecidos pelo arquivo de entrada. Pelo padrão atual, não é possível ter números de blocos diferentes em cada camada. Seria interessante, então, desenvolver um padrão mais abrangente, em que seja possível desenhar a forma de um diamante, por exemplo.

Referências Bibliográficas

[Cre93] N. A. C. Cressie. *Statistics for Spatial Data*. Wiley-Interscience Publication, 1993.

[FD84] J. D. Foley and A. Van Dam. *Fundamentals of Interactive Computer Graphics*. Addison Wesley, USA, 1st edition, 1984.

[Lar82] H.J. Larson. *Introduction to Probability Theory and Statistical Inference*. John Wiley and Sons, 3rd edition, 1982.

[Mar72] G. Marsaglia. *The structure of linear congruential sequences*. Academic Press, 1972.

[MdBS98] M. Overmars M. de Berg, M. van Kreveld and O. Schwarzkopf. *Computational Geometry - Algorithms and Applications*. Springer, New York, USA, 1998.

[MR03] D. C. Montgomery and G. C. Runger. *Estatística Aplicada e Probabilidade para Engenheiros*. Editora LTC, 2nd edition, 2003.

[MWS99] T. Davis M. Woo, J. Neider and D. Shreiner. *OpenGL Programming Guide*. Addison Wesley, USA, 1999.

[Rip87] B.D. Ripley. *Stochastic Simulation*. Wiley Series in Probability and Mathematical Statistics, 1987.

[San73] N. M. Santos. *Vetores e Matrizes*. Instituto de Matemática Pura e Aplicada, Rio de Janeiro, Brasil, 1st edition, 1973.

[SM94] J.L. Szwarcfiter and L. Markenzon. *Estruturas de Dados e Seus Algoritmos*. LTC Editora, Rio de Janeiro, Brasil, 1994.