Universidade
Federal de
Uberlândia

Uma Proposta para a Triangulação de Delaunay 2D e Localização Planar de Pontos em OCaml

André Luiz Moura

Uberlândia 2006
Tese de Doutorado
Universidade Federal de Uberlândia
Faculdade de Engenharia Elétrica
Programa de Pós-Graduação em Engenharia Elétrica

ANDRÉ LUIZ MOURA

UMA PROPOSTA PARA A TRIANGULAÇÃO DE DELAUNAY 2D E LOCALIZAÇÃO PLANAR DE PONTOS EM OCAML

Tese apresentada ao programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Uberlândia, como parte dos requisitos para obtenção do título de **DOUTOR EM CIÊNCIAS**.

Área de Concentração: Eletricidade Rural e Fontes Alternativas de Energia

Orientador: Prof. José Roberto Camacho (PhD)

Uberlândia – MG
2006

FICHA CATALOGRÁFICA

Elaborada pelo Sistema de Bibliotecas da UFU / Setor de Catalogação e Classificação

M929p Moura, André Luiz, 1967-

Uma Proposta para a Triangulação de Delaunay 2D e Localização Planar de Pontos em OCaml / André Luiz Moura. – Uberlândia, 2006.

114f. : il.

Orientador: José Roberto Camacho.

Tese (doutorado) – Universidade Federal de Uberlândia, Programa de Pós-Graduação em Engenharia Elétrica.

Inclui referência bibliográfica.

1. Engenharia elétrica - Matemática - Teses. I. Camacho, José Roberto. II. Universidade Federal de Uberlândia. Programa de Pós-Graduação em Engenharia Elétrica. III. Título.

CDU: 62:51

ANDRÉ LUIZ MOURA

UMA PROPOSTA PARA A TRIANGULAÇÃO DE DELAUNAY 2D E LOCALIZAÇÃO PLANAR DE PONTOS EM OCAML

Tese apresentada ao programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Uberlândia, como parte dos requisitos necessários para obtenção do título de Doutor em Ciências.

Tese APROVADA pelo Programa de Pós-graduação em Engenharia Elétrica da Universidade Federal de Uberlândia.

Área de Concentração: Eletricidade Rural e Fontes Alternativas de Energia

Banca Examinadora:

José Roberto Camacho, PhD (UFU) – Orientador

Antônio Eduardo Costa Pereira, PhD (UFU)

Edgard Afonso Lamounier Jr., PhD (UFU)

Mário Mourelle Pérez, PhD (UFU)

Renato Cardoso Mesquita, Dr. (UFMG)

Sebastião Camargo Guimarães Jr., Dr. (UFU)

Severino Luiz Guimarães Dutra, Dr. (INPE)

Prof. Darizon Alves de Andrade
Coordenador do Curso de Pós-Graduação em Engenharia Elétrica

Uberlândia, 30 de Junho de 2006

A Deus, à memória de minha mãe, Adélia,
ao meu filho, Arthur, e a minha esposa, Averlíbia.

Agradecimentos

Ao altíssimo Deus, por permitir que tudo acontecesse...

À minha mãe, Adélia Vieira da Costa (*in memorian*), por seu amor e dedicação...

À Universidade Federal de Uberlândia e à Faculdade de Engenharia Elétrica pela oportunidade de realizar este Curso.

Ao Professor Dr. José Roberto Camacho, meu orientador, entusiasta e grande incentivador, pela confiança, amizade e por todo o trabalho de orientação e revisão da tese.

Ao Professor Dr. Antônio Eduardo Costa Pereira, meu primeiro contato na UFU, por toda atenção dispensada, consideração e sugestões.

A minha esposa, Averlíbia Gonçalves Cordeiro Moura, por toda compreensão, incentivo e dedicação no preenchimento de lacunas que surgiram.

Ao meu sogro, Valdave Gonçalves Guimarães, e a minha sogra, Gerantina Cordeiro Guimarães, por todo o apoio provido a minha família durante minhas ausências em casa.

Aos colegas do Curso de Doutorado, João Manoel da Silva, Arquimedes Lopes da Silva e Lindolfo Marra de Castro Neto, pela amizade e companheirismo.

À Joana (anterior secretária da Pós-Graduação) e Marli (atual secretária da Pós-Graduação), pelo excelente desempenho funcional.

Aos meus superiores no Tribunal Regional do Trabalho da 18ª Região, Humberto Magalhães Ayres e Paulo Goiás Cordeiro dos Santos, respectivamente, Diretor da Secretaria de Tecnologia da Informação e Diretor do Serviço de Gestão de Sistemas e Internet, pela deferência em autorizar meus afastamentos ao trabalho para viagens a Uberlândia.

Resumo

Nesta tese, é apresentado um algoritmo dinâmico de localização planar de pontos. O algoritmo foi elaborado sobre dois fundamentos: - o método das “Slabs” para particionar a subdivisão planar, representada por um grafo, e permitir a rápida identificação da região em que se encontra o ponto que está sendo consultado; - a Multiárvore-B Intervalar, uma estrutura de dados derivada árvore-B, aparelhada com mecanismo de pesquisa intervalar e disposta em camadas. O algoritmo é dinâmico porque altera dinamicamente a estrutura de pesquisa, à medida que surgem eventos de inserção ou remoção de segmentos na subdivisão planar que está sendo construída. O algoritmo foi implementado na linguagem OCaml, mas poderia ter sido implementado em qualquer outra linguagem de programação. Para aumentar a eficiência do algoritmo, algumas melhorias podem ser introduzidas, como por exemplo, a substituição do núcleo da Multiárvore-B Intervalar por outros tipos de árvores balanceadas. Adicionalmente, foram discutidos alguns aspectos do processso de construção de malhas de elementos finitos, em que se insere, sobretudo, o problema da localização planar de pontos.

Palavras-chaves: localização planar de pontos dinâmica, árvore balanceada, multiárvore-B intervalar, triangulação de Delaunay incremental, malha bidimensional, OCaml

Abstract

In this thesis, it is presented a planar point location algorithm. The algorithm was developed on top of two elements: - the method of slabs to divide the planar subdivision, is represented by a graph, allowing the fast identification of the region where the point being recalled is; - the Interval Multi-B-tree, a data structure derived from the B-tree, prepared with an interval search structure and disposed in layers. The algorithm is essentially dynamic since the search structure keeps changing dynamically during the process, while the planar subdivision is being built; new events of segment insertion or removal keep appearing. The algorithm was implemented in OCaml, but could be carried out in any other programming language. To increase the algorithm efficiency, some improvements can be introduced, as an example, the substitution of the Interval Multi-B-tree core by other types of balanced trees. Moreover, it was discussed some aspects of the assembling process of the finite element meshing, where it is inserted, mainly, the planar point location problem.

Key-words: dynamic planar point location, balanced tree, interval multi-B-tree, incremental Delaunay triangulation, two dimensional meshing, OCaml

UMA PROPOSTA PARA A TRIANGULAÇÃO DE DELAUNAY 2D E LOCALIZAÇÃO PLANAR DE PONTOS EM OCAML

Sumário

LISTA DE FIGURAS x

LISTA DE TABELAS xiii

LISTA DE FÓRMULAS xiv

LISTA DE SÍMBOLOS xv

TERMINOLOGIA BÁSICA xvi

1. INTRODUÇÃO 18

2. ASPECTOS CRÍTICOS DA GERAÇÃO DA MALHA 35
2.1. Introdução 35
2.2. Tipos de Domínios Geométricos 35
2.3. Tipos de Malhas 36
2.4. Propriedades Desejáveis de uma Malha e de Geradores de Malhas 37
2.5. Triangulação de Delaunay 41
2.6. Inserção de Pontos de Steiner 52
2.7. Localização Planar de Pontos 53
2.8. Refinamento da Malha 62

3. IMPLEMENTAÇÃO DE TRIANGULAÇÃO DE DELAUNAY POR DIVISÃO E CONQUISTA EM OCAML 65
3.1. Introdução 65
3.2. O Algoritmo de Divisão-e-Conquista modificado por Dwyer 65
3.3. Melhoramentos efetuados no Algoritmo de Guibas-Stolfi 66
3.4. A Linguagem Objective Caml 66
3.5. Detalhes da Implementação 67
3.6. Resultados dos Experimentos 71
3.6.1. Amostra da Robustez Numérica de OCaml 72
3.7. Avaliação da Possibilidade de Paralelização 73

4. GERADOR DE MALHAS ESCRITO EM OCAML 78
4.1. Introdução 78
4.2. Características do Gerador 79
4.3. Estrutura de Dados 81
4.4. Notas sobre o Algoritmo 82
4.5. Detalhes da Implementação em OCaml 83
4.6. Resultados da Triangulação de Domínios Geométricos 85

5. LOCALIZAÇÃO PLANAR DE PONTOS USANDO MULTIÁRVORE-B INTERVALAR 88
5.1. Introdução 88
5.2 Árvores Balanceadas 89

5.2.1. A Árvore-B 91
5.2.2. A Árvore-B Intervalar 92
5.2.3. A Multiárvore-B Intervalar 94
5.3. Localização Planar com Multiárvore-B Intervalar 96
5.3.1 Construção Incremental da Estrutura de Pesquisa baseada em Eventos 96
5.3.2. Pré-processamento 101
5.3.3. Consulta de Pontos 102
5.4. Experimentos Computacionais 103

6. CONCLUSÕES 107
6.1. Trabalhos Futuros 109

REFERÊNCIAS BIBLIOGRÁFICAS 110

Lista de Figuras

1.1 Determinação se um ponto *P* está dentro de um triângulo *ABC*. 19
1.2 Se o ponto *P* estiver dentro do triângulo *ABC*, o somatório da área dos três novos triângulos será igual à área de *ABC*. 19
1.3 Se o ponto *P* estiver fora do triângulo *ABC*, o somatório da área dos três novos triângulo será maior que a área de *ABC*. 19
1.4 Um PSLG particionado em *slabs* horizontais. 21
1.5 Um PSLG decomposto em um conjunto de cadeias monótonas (O procedimento de regularização adicionou novas arestas (v2, v3), (v5, v6) e (v7, v8)) ao PSLG. 22
1.6 Um polígono com faces irregulares. 23
1.7 Polígono triangulado para construção da estrutura de dados de Kirkpatrick.. . . 23
1.8 Triangulação envolta por um grande triângulo e conectada aos seus vértices.. . 24
1.9 Triangulação com conjunto independente de vértices com grau ≤ 8. 24
1.10 Triangulação sem o conjunto independe de vértices com grau ≤ 8. 25
1.11 Triangulação refeita. 25
1.12 Estrutura de pesquisa gerada pelo algoritmo de Kirkpatrick. 26
1.13 Um ponto de consulta sobre uma face *F* da subdivisão original. 27
1.14 O caminho percorrido no DAG para localizar a região que contém o ponto de consulta. 27
1.15 O efeito da inserção do ponto *p* no do triângulo T_1 sobre a estrutura de dados *D*. Somente são mostradas as partes que sofreram alterações. 29
1.16 Pior caso da localização de pontos no DAG referente à história de uma triangulação de Delaunay incremental. 30
2.1 Tipos de entradas bidimensionais: (a) polígono simples, (b) polígono com buracos, (c) domínio múltiplo e (d) domínio curvo. 36
2.2 Tipos de Malhas. 37
2.3 Um triângulo de formato inadequado para o método dos elementos finitos. . . . 38
2.4 Passos da triangulação baseada em “circle packing”. 39
2.5 (a) Uma *quadtree*. (b) Uma triangulação de um conjunto de pontos baseada em *quadtree* em que nenhum ângulo é menor que 20º. 40
2.6 A superfície de uma malha tetraédrica derivada de uma *octree*. 40
2.7 Parte da triangulação de uma região com três buracos. 41
2.8 Triangulação de um polígono. 42
2.9 A triangulação de Delaunay sobre uma nuvem de pontos. 42
2.10 A triangulação de Delaunay sobre um conjunto de 400 pontos aleatórios. 43
2.11 Cada triângulo em uma triangulação de Delaunay tem um círculo-circundante vazio. 44
2.12 O teste “InCircle”. 44
2.13 Uso do predicado “InCircle” para decidir qual das duas arestas (*AC* ou *BD*) é aresta de Delaunay. 46
2.14 Triangulações de Delaunay de C_1 e C_2, separadas por uma linha vertical imaginária. 48
2.15 Triangulação de Delaunay de $C_1 \cup C_2$. 48
2.16 Amostra das deficiências da triangulação de Delaunay. 49
2.17 Amostra da correção provida pela inserção de pontos de Steiner. 49

2.18 (a) Um PLSG. (b) Uma triangulação de Delaunay do PSLG. (c) Uma triangulação de Delaunay com restrições do PSLG. 50
2.19 Supertriângulo contendo pontos a serem triangulados. 51
2.20 Os dois casos em que se adiciona um ponto p_r. 51
2.21 Triangulações: (a) sem pontos de Steiner e (b) com pontos de Steiner. 52
2.22 Algoritmo de inserção de pontos de Steiner. 53
2.23 Subdivisão em *slabs*. 54
2.24 Amostra do pior caso da divisão em *slabs*. 55
2.25 Um mapa trapezoidal. 56
2.26 Atualização (incremental) de mapa trapezoidal. 57
2.27 Um mapa trapezoidal de dois segmentos e a estrutura de pesquisa correspondente. 59
2.28 Um algoritmo incremental aleatório. 59
2.29 Um algoritmo de pesquisa em mapa trapezoidal. 60
2.30 Resultado da transformação *shear*. 61
2.31 Movendo um vértice para o centro de massa de seus vizinhos. 63
2.32 Refinamento de uma região da malha com triângulo de qualidade insatisfatória. O vértice *v* é inserido, o triângulo *t* é eliminado, mas a propriedade de Delaunay é mantida. 64
3.1 Exemplo de distribuição uniforme de pontos no quadrado unitário. 66
3.2 Função de multiplicação em OCaml. 67
3.3 Estrutura de dados da triangulação em C. 68
3.4 Estrutura de dados da triangulação em OCaml. 69
3.5 Trecho de código da função *delete_edge* em C. 70
3.6 Trecho de código da função *delete_edge* em OCaml baseado em proposta de Weis. 70
3.7 Estrutura de dados (simplificada) da triangulação em OCaml. 70
3.8 Trecho de código (simplificado) da função *delete_edge* em OCaml. 71
3.9 Resultados visuais de triangulações de PSLG. 72
3.10 Uma estratégia típica de paralelização. 74
3.11 Triangulação de Delaunay seqüencial de um conjunto de pontos. 76
3.12 Triangulação de Delaunay de um conjunto de pontos em quatro processadores. 77
3.13 Resultados parciais da Triangulação de Delaunay nos quatro processadores. . . 77
4.1 Síntese do processo de geração da malha. 80
4.2 Descrição do formato dos arquivos *.node*, *.poly* e *.ele*. 80
4.3 Exemplos de conteúdo dos arquivos *.node*, *.poly* e *.ele*. 81
4.4 Estrutura de dados da aresta e do triângulo. 81
4.5 PSLG de uma guitarra elétrica. 82
4.6 Triangulação de Delaunay dos vértices do PSLG. Alguns segmentos originais e faces estão ausentes. 82
4.7 Triangulação de Delaunay em conformidade com os segmentos do PSLG. . . . 82
4.8 Triangulação com triângulos removidos de concavidades e buracos. 83
4.9 Malha final (gerada por *OCamlMesh*) composta de 484 triângulos com ângulo mínimo não inferior a 30º. 83
4.10 Códigos da função “FindNeighbors”. 84
4.11 Implementação da estrutura de dados do gerador em linguagem C. 84
4.12 Implementação da estrutura de dados do gerador em linguagem OCaml. 85
4.13 Algumas malhas uniformes construídas pelo gerador *OCamlMesh*. 86
4.14 Algumas malhas gradientes construídas pelo gerador *OCamlMesh*. 87

5.1 Árvore binária totalmente degenerada. 90
5.2 Árvore binária balanceada. 90
5.3 Uma árvore-B de ordem 4 com todas as folhas no nível 2. 91
5.4 Modificações no código da árvore-B para obtenção do limite inferior. 93
5.5 Modificações no código da árvore-B para obtenção do limite superior. 94
5.6 Arquitetura global de uma Multiárvore-B. 95
5.7 Estrutura de dados da Multiárvore-B em OCaml. 95
5.8 Subdivisão de um PSLG em *slabs*. 97
5.9. Algoritmo global do processo de construção da estrutura de pesquisa. 97
5.10 Mapeamentos possíveis de segmentos no espaço binário. 98
5.11. Eventos significativos entre segmento novo (linha tracejada) e segmento vizinho (linha contínua), retratando as possíveis formas de interseção entre eles. 99
5.12 Relação espacial entre o novo segmento e os segmentos vizinhos. 100
5.13 Orientações dos segmentos de alguns triângulos. 101
5.14 Algoritmo *Prepare*. 101
5.15 Algoritmo *Preprocess*. 102
5.16 Algoritmo *Query*. 103
5.17 Tipos de entradas para os experimentos. 104

Lista de Tabelas

2.1 Três critérios para elementos lineares 38
2.2 Complexidade de algoritmos de triangulação. 52
2.3 Eficiências teóricas dos algoritmos de localização de pontos. 62
3.1 Ponteiros explícitos em C traduzidos para OCaml. 69
3.2 Resultados da triangulação por divisão-e-conquista utilizando um PC com processador Pentium III de 1.13 GHz, 128 Mb de memória RAM e 256 Kb de memória Cache, sob o sistema operacional Windows 98. 72
3.3 Resultados de operações matemáticas envolvendo números na representação ponto-flutuante. 73
5.1 Resultados de Testes de Localização de Pontos na Malha do Círculo. 104
5.2 Resultados de Testes de Localização de Pontos na Malha da Chave. 104
5.3 Resultados de Testes de Localização de Pontos na Malha da Letra “A” 105
5.4 Resultados de Testes de Localização de Pontos na Malha da Guitarra elétrica. 105
5.5 Resultados de Testes de Localização de Pontos na Malha do Lago Superior. . 105
5.6 Distribuição de eventos identificados na fase de pré-processamento. 106

Lista de Fórmulas

2.1 A transformação *shear* 60

Lista de Símbolos

ANSI	American National Standards Institute
BST	Binary Search Tree
CSG	Constructive Solid Geometry
DAG	Directed Acyclic Graph
IDE	Integrated Development Environment
INRIA	Institut National de Recherche en Informatique et Automatique
LCF	Logic of Computable Functions
MEF	Método dos Elementos Finitos
ML	Meta-Language
OCaml	Objective Categorical Abstract Machine Language
PLS	Piecewise Linear System
PSLG	Planar Straight Line Graph
RAM	Random Access Memory
SGBD	Sistema de Gerenciamento de Banco de Dados

Terminologia Básica

Aqui são explicados vários termos básicos que são usados no texto desta tese.

Algoritmo é uma seqüência finita e não ambígua de instruções que são necessárias para solucionar um problema. Algoritmos podem ser implementados por programas de computador, mas não representam necessariamente um programa de computador, e sim os passos necessários para realizar uma tarefa.

Complexidade assintótica de um algoritmo determina a velocidade do crescimento de elementos de tempo consumidos por um algoritmo em relação ao número N de elementos de entrada, onde N é maior que uma constante N_0. A função $f_{max}(N)$, que atribui um número máximo de elementos de tempo para cada número N (> 0) de elementos de entrada, é chamada de a complexidade de um algoritmo no pior caso. De forma análoga, a função $f_{min}(N)$, que atribui um número mínimo de elementos de tempo para cada número N, é chamada de a complexidade de um algoritmo no melhor caso; por outro lado, a função $f_{avg}(N)$ é chamada de a complexidade de um algoritmo no caso médio. Normalmente, empregam-se letras gregas para denotar as funções de avaliação da complexidade assintótica de algoritmos. $O(N)$, também conhecida por "Big-Oh", é usada para avaliar o pior caso; $\Omega(N)$ é usada para avaliar o melhor caso e $\theta(N)$ é usada para avaliar caso médio. Na avaliação de um algoritmo, geralmente, a complexidade assintótica para o pior caso é empregada.

OCaml, Objective Caml, também conhecido como O'Caml é uma linguagem de programação funcional da família ML, desenvolvida por INRIA em 1996. Não é uma linguagem puramente funcional, permitindo a existência de valores mutáveis bem como de efeitos colaterais, tipicamente existentes apenas em linguagens imperativas. Esta característica distingue-a de outras linguagens puramente funcionais, como por exemplo Haskell.

Geometria Computacional, em ciência da computação, é o estudo de algoritmos para resolver problemas relacionados à geometria. O objetivo da geometria computacional é desenvolver algoritmos e estruturas de dados eficientes que são aplicados na solução de problemas em termos de objetos geométricos elementares como pontos, retas, segmentos de reta, polígonos, poliedros, superfícies, etc. Exemplos de problemas que se incluem no domínio de estudo da disciplina são: par de pontos mais próximo, fecho convexo, diagrama de

Voronoi, triangulação de Delaunay, entre vários outros. Na geometria computacional as figuras e construções geométricas correspondem a estruturas de dados e algoritmos. Em geral, o interesse é solucionar um problema utilizando o menor numero possível de operações elementares de modo a trazer eficiência no cálculo da solução. A geometria computacional constitui ferramenta fundamental em diversas áreas da computação que necessitam de uma abordagem geométrica, tais como computação gráfica, robótica, sistemas de informação geográfica (SIGs), visão computacional, otimização combinatória, processamento de imagens, teoria dos grafos, desenho de circuitos integrados, desenho e engenharia (CAD/CAM), entre outras.

Triangulação de Delaunay, em matemática e geometria computacional, pode ser definida, para um conjunto de pontos P no plano, como a triangulação $DT(P)$ de P tal que nenhum ponto em P está dentro do círculo-circundante de qualquer triângulo em $DT(P)$. Em outras palavras, essa regra estabelece que o círculo-circundante de um triângulo não deve conter outros pontos além dos pontos do triângulo. A triangulação de Delaunay maximiza o ângulo mínimo e minimiza o ângulo máximo de todos os triângulos na triangulação. Essa técnica foi inventada por Boris Delaunay em 1934. A triangulação de Delaunay é usada com freqüência para construir malhas para o método dos elementos finitos, as quais, para serem precisas e com boa qualidade, devem ser refinadas por algoritmos que obedecem à regra do círculo-circundante.

Grafo, em ciência da computação, é um tipo abstrato de dados que consiste de um conjunto de nós e um conjunto de arestas que estabelecem relações (conexões) entre os nós. Um grafo **G** é definido como segue: **G**=(**V**,**E**), onde **V** é um conjunto finito, não-vazio de vértices e **E** é um conjunto de arestas (ligações entre pares de vértices). Quando as arestas no grafo não tem direção, o grafo é chamado não-direcionado; caso contrário, é chamado direcionado. Na prática, é associada alguma informação a cada nó e a cada aresta.

Grafo Direcionado Acíclico é um grafo direcionado que não possui ciclos.

Árvore balanceada é uma árvore cuja altura da subárvore esquerda de cada nó nunca difere em mais que ±1 da altura de sua subárvore direita.

Capítulo I

Introdução

1.1 Introdução

O problema da localização planar de pontos está presente em muitas aplicações que operam sobre algum domínio geométrico decomposto em regiões. O problema é definido como segue: dada uma subdivisão do plano modelada por um grafo planar de linhas retas (PSLG) G com n vértices e um ponto de consulta Q, determine qual região da subdivisão contém Q.

É um dos mais importantes problemas da geometria computacional e tem sido extensamente investigado nos anos recentes, tendo aplicação em muitas áreas.

A localização planar de pontos desempenha função essencial, sobretudo, em sistemas de informações geográficas. Nesse contexto, dados um mapa e um ponto de consulta especificado por suas coordenadas, objetiva-se encontrar a região do mapa que contém aquele ponto. Note-se que um mapa é nada mais que uma subdivisão do plano em regiões, uma subdivisão planar.

Considere por, exemplo, a mera exibição da localização atual de agentes (como veículos automotores, embarcações, aeronaves, aparelhos telefônicos celulares, pessoas, animais) ou de evento físico (como foco de incêndio, abalo sísmico) em um mapa armazenado eletronicamente. Trata-se de um procedimento relativamente simples. No entanto, em tempo real, localizar e relatar, de modo rápido e automático, o nome da região em que se encontra o agente ou ocorre um evento é uma tarefa custosa.

A modalidade mais elementar do problema de localização planar de pontos é determinar se um ponto é interno a um dado polígono (PSLG). Um exemplo simples de aplicação é o ato de clicar o botão do mouse dentro de uma forma geométrica desenhada sobre uma tela de computador com o objetivo de executar alguma ação associada. Um outro exemplo, é detectar se um sinal de aparelho telefônico celular provém de alguma região específica, por exemplo, a área de um estabelecimento penitenciário sob vigilância e monitoramento. Esse tipo de problema de localização planar de pontos pode ser solucionado, tanto para polígonos

convexos e quanto polígonos não-convexos, com algoritmos geométricos discutidos em (O'ROURKE, 1998). Exclusivamente para polígonos convexos, (WALKER, SNOEYINK; 1999) apresenta um algoritmo baseado em representação de G*eometria Sólida Construtiva* (*Constructive Solid Geometry* – CSG).

Para determinar, por exemplo, se o triângulo *ABC* (Figura 1.1), que tem os vértices A(x_1, y_1), B(x_2, y_2) e C(x_3, y_3), contém o ponto P(x, y), usa-se a fórmula da área do triângulo:

$$\mathrm{AREA}(P_1P_2P_3) = \frac{1}{2}\left\| \begin{matrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{matrix} \right\|, \tag{1.1}$$

onde P_1, P_2 e P_3 são os vértices. Se for usado o valor absoluto do determinante, os vértices podem ser tomados em qualquer ordem. Então, para determinar se o triângulo *ABC* contém o ponto *P*, calcula-se a área do triângulo *ABC* usando a Equação 1.1, definindo-se três novos triângulos, cada um tendo como seus vértices o ponto *P* e dois vértices do triângulo *ABC*.

Dessa operação, resultam três únicos triângulos: *ABP*, *BCP* e *CAP*. Se o triângulo *ABC* contém o ponto *P* (Figura 1.2), então AREA(*ABP*) + AREA(*BCP*) + AREA(*CAP*) = AREA(*ABC*). Se o triângulo *ABC* não contém o ponto *P* (Figura 1.3), então AREA(*ABP*) + AREA(*BCP*) + AREA(*CAP*) > AREA(*ABC*).

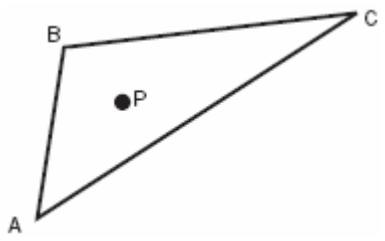


Figura 1.1. Determinação se um ponto *P* está dentro de um triângulo *ABC*.

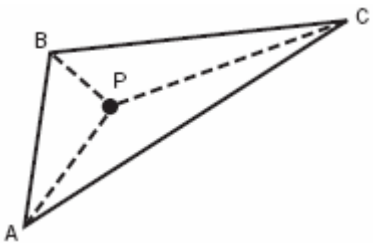


Figura 1.2. Se o ponto *P* estiver dentro do triângulo *ABC*, o somatório da área dos três novos triângulos será igual à área de *ABC*.

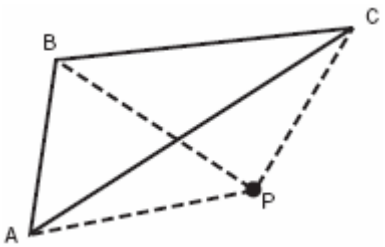


Figura 1.3. Se o ponto *P* estiver fora do triângulo *ABC*, o somatório da área dos três novos triângulo será maior que a área de *ABC*.

Para o problema de determinar em um mapa qual a região que contém um determinado ponto — uma generalização do problema de determinar se um dado polígono contém um certo ponto — existem basicamente dois tipos de algoritmos: *ineficientes* e *ótimos*.

Os *algoritmos ineficientes* são aqueles que empregam a força bruta testando cada região do mapa para verificar qual delas contém o ponto de consulta. O número de comparações necessárias, no pior caso, pode ser igual ao número de regiões do mapa.

Os *algoritmos ótimos* são aqueles baseados em métodos eficientes. A eficiência desses algoritmos é aferida dos pontos de vista teórico e prático. A eficiência teórica refere-se à complexidade do algoritmo no pior caso. A eficiência prática é avaliada pela facilidade de implementação. O uso de estruturas de dados complicadas na busca de eficiência teórica termina acarretando a concepção de um algoritmo pouco prático.

A eficiência teórica dos algoritmos de localização planar é analisada sob três atributos, relativamente ao número (n) de segmentos ou de vértices que compõem a malha:

1. *Tempo de pré-processamento*. O tempo requerido para construir a estrutura de pesquisa — uma estrutura de dados contendo uma subdivisão adicional de um PSLG para agilizar as consultas de pontos. $O(n \log n)$ é o requisito de tempo ideal.

2. *Espaço*. O armazenamento usado para construir e representar a estrutura de pesquisa. $O(n)$ é o requisito de espaço ideal.

3. *Tempo de pesquisa*. O tempo requerido para localizar um ponto na estrutura de pesquisa. $O(\log n)$ é o requisito de tempo ideal para cada pesquisa realizada.

Como a eficiência prática dos algoritmos ótimos é dependente de sua implementação, na próxima seção serão esboçados algoritmos com eficiências teóricas variadas, sem qualquer discussão sobre eficiência do ponto de vista prático.

1.1.1 Algoritmos de Localização Planar de Pontos Existentes

Nesta seção, será traçado o perfil dos algoritmos de Shamos, Lee e Preparata, Preparata, Kirkpatrick (EDAHIRO; KOKUBO; ASANO, 1984) e Berg *et al.* (BERG, 1997).

a) Algoritmo de Shamos

Este algoritmo particiona o plano em linhas horizontais, conhecidas por “slabs”, que atravessam os vértices do PSLG (Figura 1.4). Dessa forma, cada aresta do PSLG é dividida em segmentos por aquelas linhas horizontais. Para cada “slab”, é construída uma árvore binária. Em cada árvore binária, são representados os segmentos da “slab” ordenados pela coordenada x. Dado um ponto de consulta $P(x, y)$, primeiro encontra-se a “slab” que contém

P através do valor de sua coordenada *y*. Depois, a localização de ponto é concluída com outra pesquisa na árvore binária daquela “slab” pela coordenada *x*.

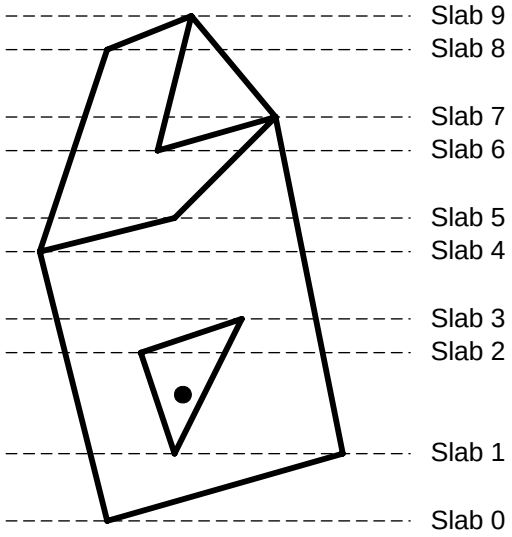


Figura 1.4. Um PSLG particionado em *slabs* horizontais.

No pior caso, o tempo de pesquisa é $O(\log n)$ e, tanto o tempo de pré-processamento quanto o espaço requeridos são $O(n^2)$.

b) Algoritmo de Lee e Preparata

Nesse algoritmo, o PSLG é decomposto em um conjunto de *cadeia monótona* — um caminho do vértice mais inferior, com menor valor para coordenada *y*, até o vértice mais superior, com maior valor para a coordenada *y*. Cada aresta desse caminho é direcionada no sentido do topo e é ordenada da esquerda para a direita. A decomposição do PSLG somente ocorre se o PSLG for regular, isto é, se todo vértice for um ponto inicial de uma aresta e um ponto terminal de outra aresta, exceto no vértice mais inferior e vértice mais superior. Se o PSLG não for regular, ele é regularizado pelo algoritmo “plane-sweep” (O’ROURKE, 1998), como uma etapa de pré-processamento que toma tempo $O(n \log n)$.

O algoritmo localiza um ponto de consulta em $O((\log n)^2)$ e requer espaço $O(n)$.

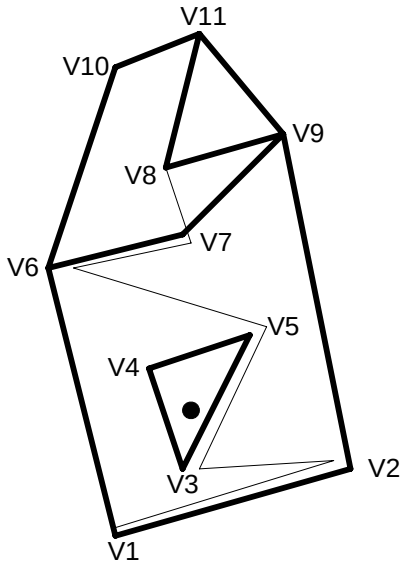


Figura 1.5. Um PSLG decomposto em um conjunto de cadeias monótonas (O procedimento de regularização adicionou novas arestas (v2, v3), (v5, v6) e (v7, v8)) ao PSLG.

c) Algoritmo de Preparata

Esse algoritmo é um melhoramento do algoritmo de Shamos por meio da subdivisão de cada aresta em $O(\log n)$ segmentos. As pesquisas são feitas, simultaneamente, nas direções x e y, em uma árvore que representa a estrutura hierárquica das “slabs”.

O algoritmo apresenta estrutura de dados complicada que representa regiões do PSLG com “slabs” e “trapézios”.

A estrutura de pesquisa é construída em tempo $O(n \log n)$ e com espaço $O(n \log n)$.Cada pesquisa requer tempo $O(\log n)$.

d) Algoritmo de Kirkpatrick

O algoritmo utiliza um grafo acíclico direcionado (*DAG – Directed Acyclic Graph*) com raiz como estrutura de dados adicional e requer que as faces do PSLG sejam triangulares.

Considere a tarefa de localização de pontos em uma subdivisão planar do plano, como o polígono abaixo:

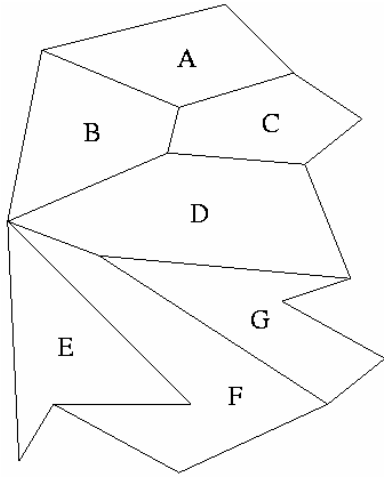


Figura 1.6. Um polígono com faces irregulares.

Como se pode ver, cada subdivisão é identificada com uma letra que será retornada na consulta de pontos.

A solução proposta por Kirkpatrick para responder a cada consulta de ponto em tempo $O(\log n)$, em que n é o número de vértices na subdivisão, segue os passos seguintes, demonstrados sobre o polígono da Figura 1.6:

1. Triangular a subdivisão se ela não contiver somente triângulos, conforme mostrado na Figura 1.7.

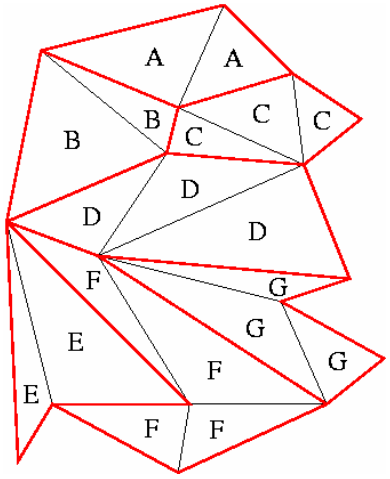


Figura 1.7. Polígono triangulado para construção da estrutura de dados de Kirkpatrick.

2. Construir a estrutura de dados.

2.1. Envolver a triangulação com um triângulo grande o suficiente de modo que nenhuma de suas três arestas intercepte a triangulação. Os novos triângulos formados, representando faces externas à triangulação, são rotulados com “O” como na Figura 1.8.

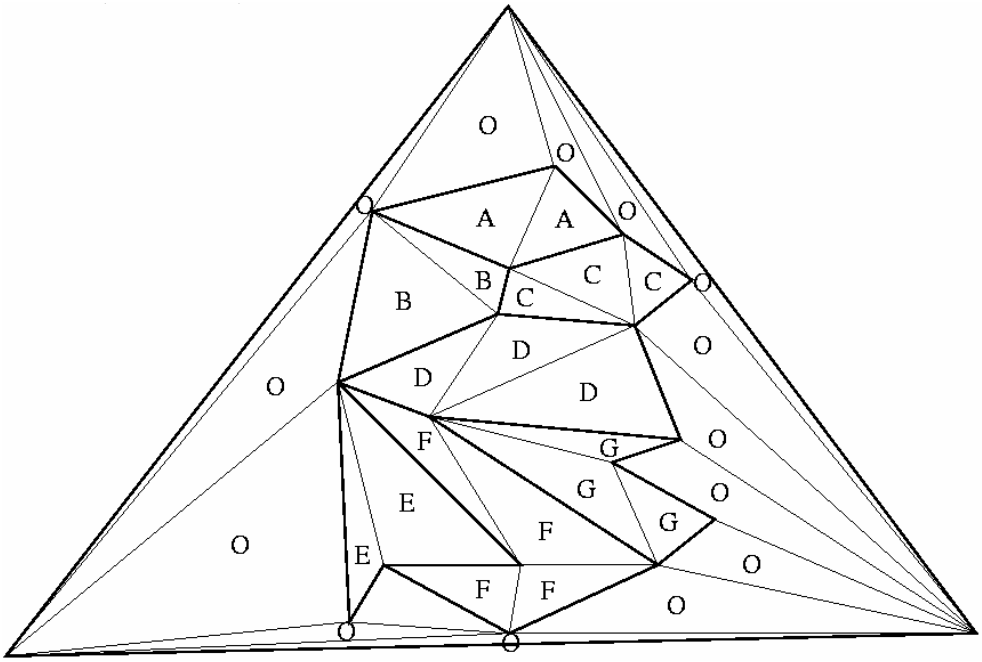


Figura 1.8. Triangulação envolta por um grande triângulo e conectada aos seus vértices.

2.2. Encontrar um conjunto independente de vértices com grau menor ou igual a oito dentro de cada casco convexo, que foram formados pelas faces internas da triangulação como na Figura 1.9.

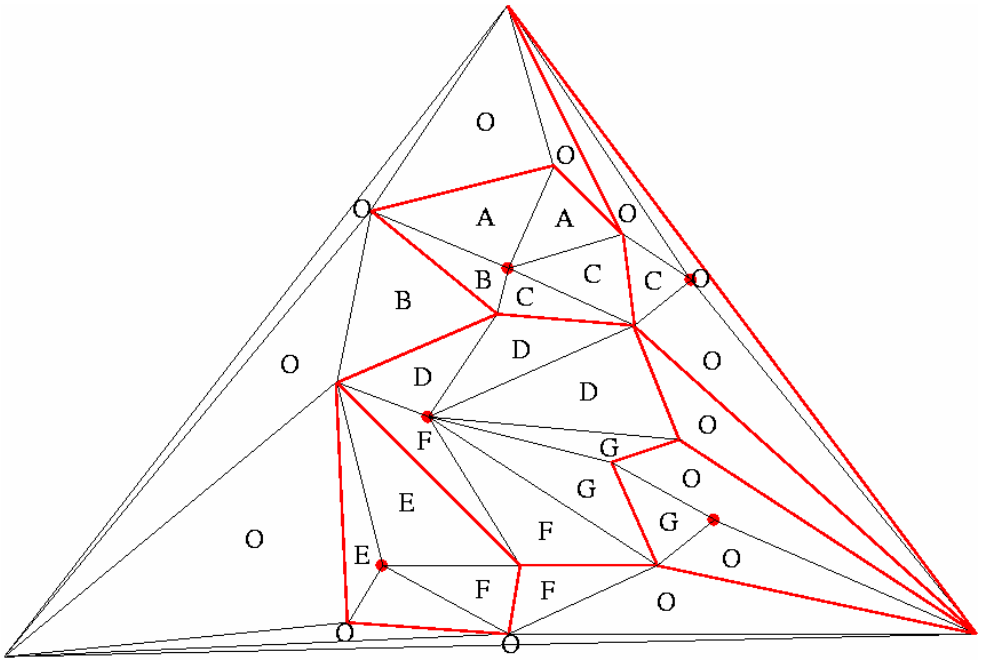


Figura 1.9. Triangulação com conjunto independente de vértices com grau ≤ 8.

2.3. Remover da triangulação o conjunto independente de vértices com grau ≤ 8 (Figura 1.10).

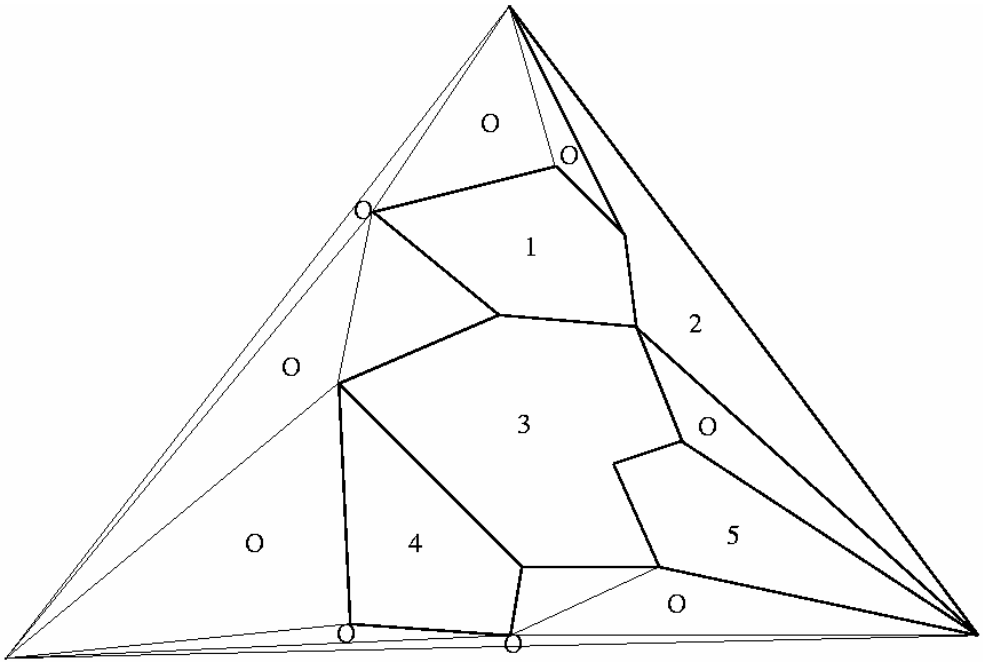


Figura 1.10. Triangulação sem o conjunto independe de vértices com grau ≤ 8.

2.4. Refazer a triangulação, seguindo a mesma regra para rotular as faces como mostrado na Figura 1.11.

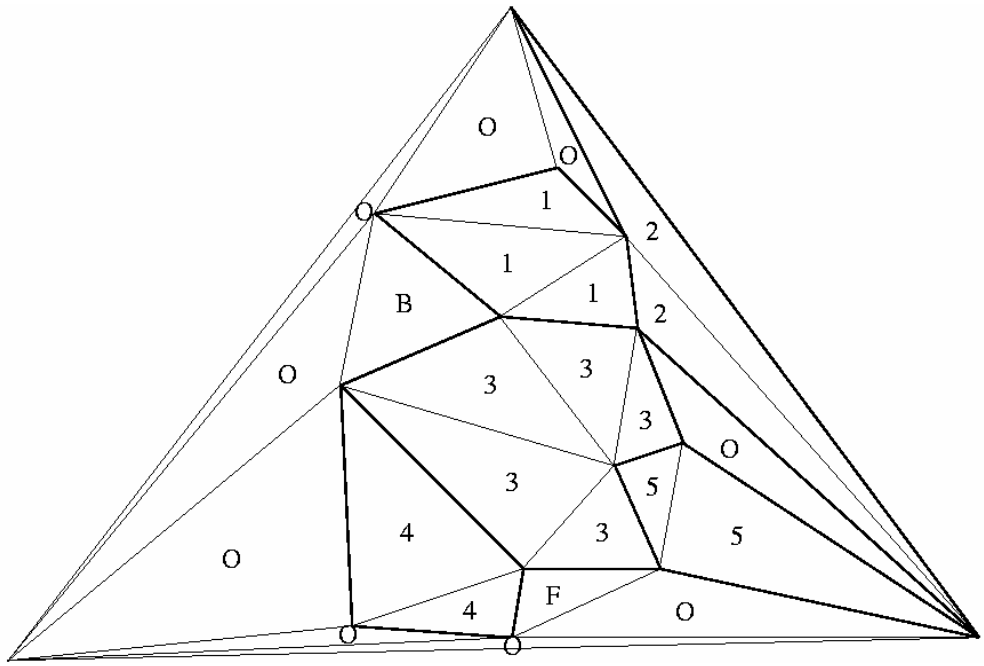


Figura 1.11. Triangulação refeita.

2.5. Repetir os passos anteriores até que restem apenas os vértices do grande triângulo.

O grafo direcionado acíclico (DAG) resultante do pré-processamento do algoritmo de Kirkpatrick é mostrado na Figura 1.12. Cada nó do grafo corresponde a um triângulo. Logo adiante, será mostrado como efetuar procedimento de localização de pontos.

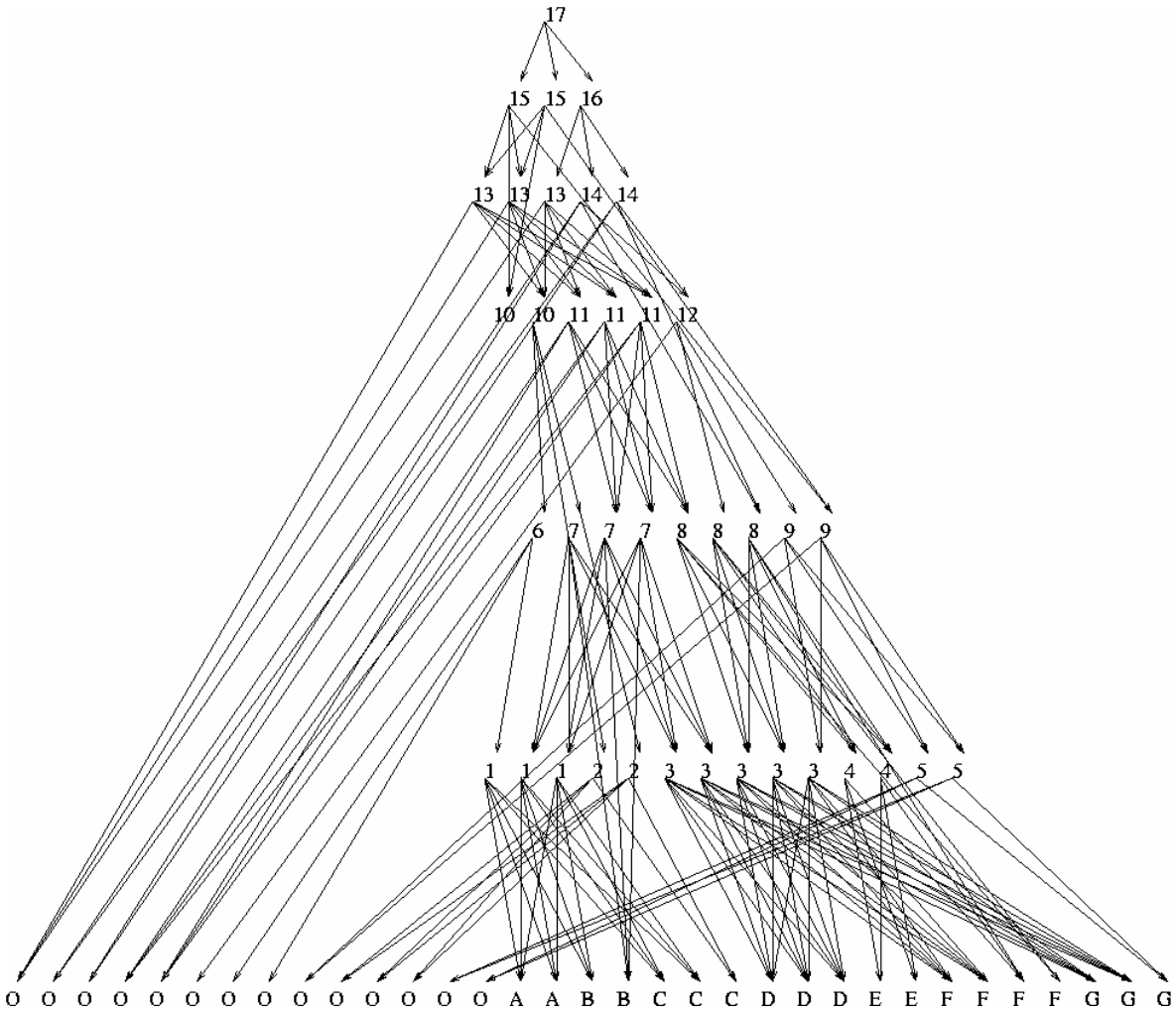


Figura 1.12. Estrutura de pesquisa gerada pelo algoritmo de Kirkpatrick.

A consulta de pontos inicia-se pelo nó-raiz do DAG. Se o triângulo a que se refere o nó visitado não contém o ponto procurado, então o ponto está fora da subdivisão. Caso contrário, checa-se cada um dos nós-filhos e, recursivamente, caminha-se por sua descendência checando até encontrar o nó-folha indicando o triângulo que contém o ponto. Um exemplo de consulta de pontos está exemplificado nas Figuras 1.13 e 1.14 adiante.

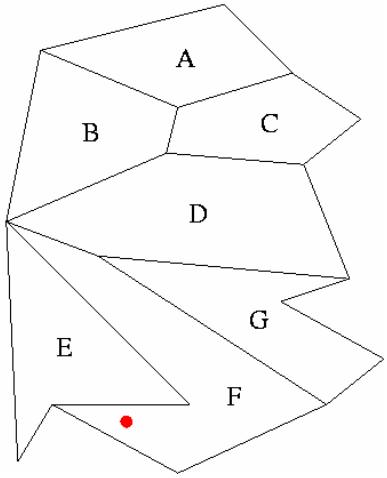


Figura 1.13. Um ponto de consulta sobre uma face *F* da subdivisão original.

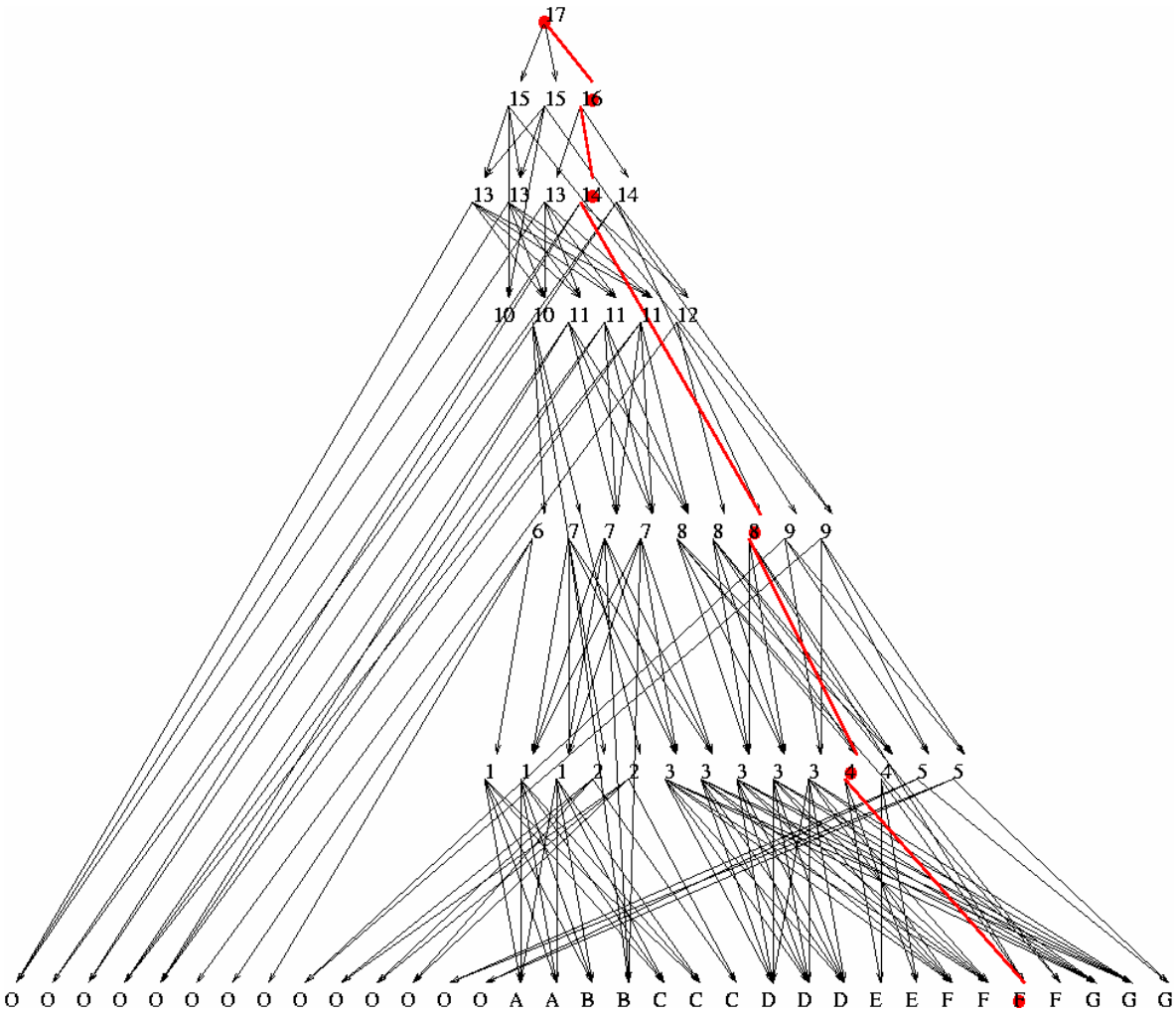


Figura 1.14. O caminho percorrido no DAG para localizar a região que contém o ponto de consulta.

e) Algoritmo apresentado por Berg

Berg *et al.* (BERG, 1997) apresenta um método que cria uma estrutura de busca ao mesmo tempo em que se constrói a triangulação de Delaunay. Seja p o ponto de consulta. Para encontrar o triângulo que contém p, é usada a seguinte abordagem: enquanto se constrói a triangulação de Delaunay, também se constrói a estrutura de localização de pontos D, que é um DAG. As folhas de D correspondem aos triângulos da triangulação atual T, e são mantidos ponteiros entre as folhas de D e a triangulação. Os nós internos de D correspondem aos triângulos que estiveram na triangulação em algum estágio anterior, mas que já foram destruídos. A estrutura de localização de pontos é construída como segue. Inicializa-se D com um único nó-folha, que correspondente ao triângulo inicial $p_1p_2p_3$, também chamado de *supertriângulo* na triangulação de Delaunay incremental.

Suponha que em algum momento, o triângulo $p_ip_jp_k$ da triangulação atual seja dividido em três (ou dois) triângulos. A correspondente mudança em D é adicionar três (ou dois) novas folhas ao grafo D e transformar o nó-folha referente ao triângulo $p_ip_jp_k$ em um nó interno com ponteiros apontando para aquelas três (ou duas) folhas recém-acrescidas. De modo semelhante, quando são substituídos dois triângulos $p_kp_ip_j$ e $p_ip_jp_l$ pelos triângulos $p_kp_ip_l$ e $p_kp_lp_j$, por meio da remoção de uma aresta, são criadas folhas para os dois novos triângulos, e os nós $p_kp_ip_j$ e $p_ip_jp_l$ recebem ponteiros para as duas novas folhas. Como mostrado na Figura 1.15, quando um nó-folha se torna um nó interno, ele recebe, no máximo, três ponteiros partido dele.

Usando a estrutura de pesquisa D, pode-se localizar o próximo ponto p a ser adicionado na triangulação atual. Isso é feito do seguinte modo: começa-se na raiz de D, que corresponde ao triângulo inicial $p_1p_2p_3$. São checados os três filhos da raiz para ver em qual triângulo o ponto p se encontra. Identificado o nó, caminha-se para o seu primeiro filho e novamente é feita a checagem para identificar o triângulo que contém o ponto p, e assim por diante, até que seja alcançado um nó-folha em D. Esse nó-folha corresponde a um triângulo na triangulação atual que contém o ponto p. Uma vez que o grau de saída de qualquer nó é no máximo três, o procedimento para localizar um determinado ponto requer tempo $O(\log n)$ em relação ao número de nós no caminho de busca ou à quantidade de triângulos armazenados em D.

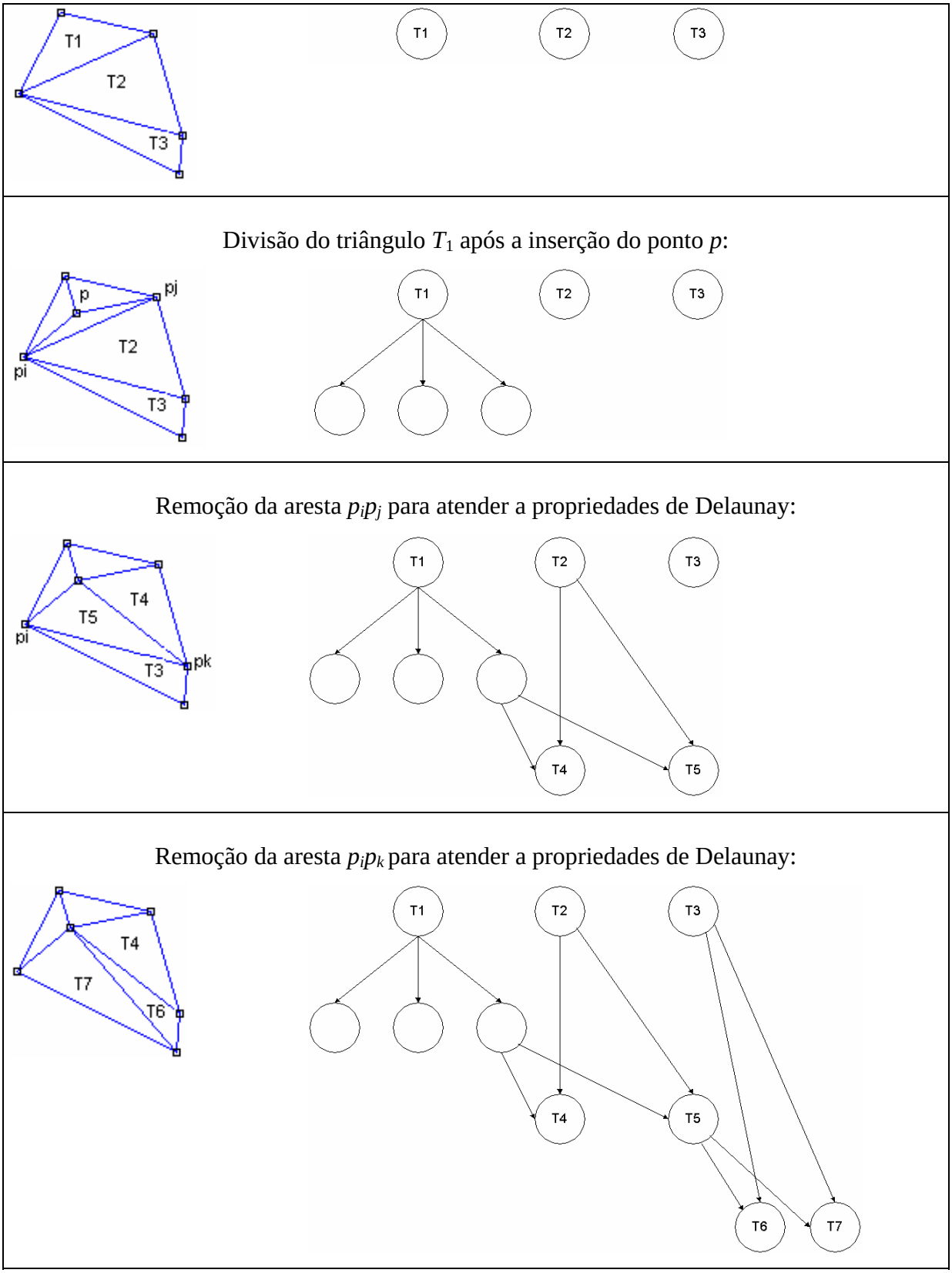


Figura 1.15. O efeito da inserção do ponto p no do triângulo T_1 sobre a estrutura de dados D. Somente são mostradas as partes que sofreram alterações.

Na Figura 1.16 adiante, é ilustrada, em um DAG, a história da construção incremental de uma triangulação de Delaunay. Nela, é demonstrada a complexidade assintótica, no pior caso,

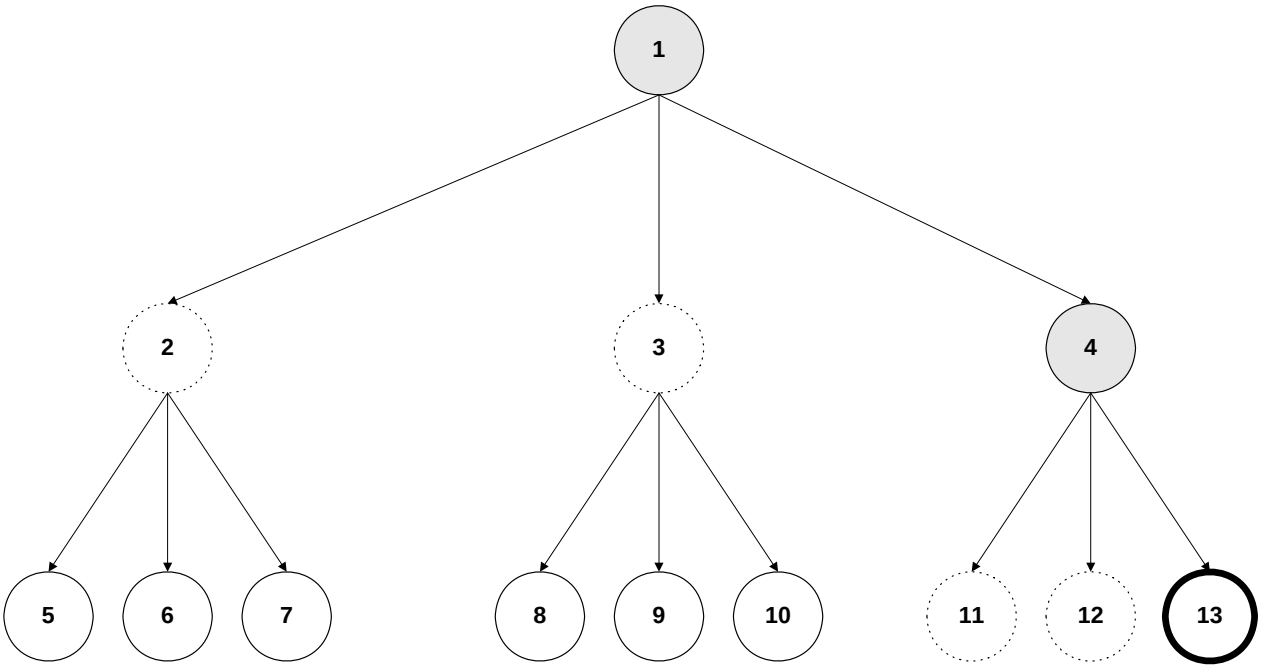
para cada pesquisa realizada, no método apresentado por Berg *et al.* em (BERG, 1997). Assim, considerando-se um total de 13 triângulos, incluindo todos aqueles presentes em algum estágio anterior e também na triangulação atual, obteve-se o seguinte resultado:

Nº de nós (N): 13

Nós testados: 1, 2, 3, 4, 11, 12, 13

Nº de nós testados: 7

Complexidade: (3 * $\log_3$ N) => Constante * $\log_3$ N => ***O*(log N)**



Legenda:

Triângulo testado e em que um dos triângulos-filhos contém o ponto de consulta

Triângulo testado, mas que nem ele nem quaisquer dos triângulos-filhos contêm o ponto de consulta

Triângulo em que se encontra o ponto de consulta

Figura 1.16. Pior caso da localização de pontos no DAG referente à história de uma triangulação de Delaunay incremental.

Comparado ao método da força bruta, o método tratado por *Berg et al.* utiliza estrutura de pesquisa hierárquica que reduz o espaço de busca, mas outros métodos devem ser projetados para efetuar pesquisas em tempo logarítmico.

Alguns algoritmos de localização planar de pontos não foram apresentados: alguns por serem complicados e distante de serem práticos, como é o caso do de Dobkin e Lipton; outros,

por terem semelhanças com os que foram mostrados, como por exemplo, o algoritmo de Lipton e Tarjan que se assemelha ao de Shamos.

1.2 Motivação

A localização planar de pontos é um importante problema da geometria computacional. Em triangulação de Delaunay baseada em abordagem incremental, a localização planar de pontos deve ser dinâmica, ou seja, prover respostas enquanto a malha está sendo construída e constitui, portanto, um dos aspectos críticos à eficiência do processo de geração de malhas (BERG, 1997; O'ROURKE, 1998; MOUNT, 2002).

Os dois procedimentos mais conhecidos para construir a estrutura de pesquisa são "Slabs" e "Mapas Trapezoidais" (BERG, 1997; O'ROURKE, 1998; MOUNT, 2002). Este último possui medidas assintóticas ideais para os três atributos de eficiência, mas apresenta sérias complicações para tornar-se dinâmico, ou seja, permitir alteração na estrutura de pesquisa para retratar eventos de inserção ou remoção (de pontos, de segmentos ou de triângulos) durante a construção da malha. Uma delas é utilizar grafo direcionado acíclico como estrutura de pesquisa e requerer inserção aleatória de segmentos para obter um grafo topologicamente — de altura próximo a $O(\log n)$ — similar a uma árvore balanceada. Já as "Slabs" requerem espaço $O(n^2)$, mas não exigem disposição aleatória, podendo, ser implementadas dinamicamente.

Mucke *et al.* (MUCKE; SAIAS; SHU, 1996), por exemplo, apresentaram um procedimento para encontrar o ponto de consulta, no plano 2D e 3D, sem necessidade de estrutura de pesquisa, simplesmente caminhando através da triangulação, por amostragem randômica. Entretanto, diante dos resultados práticos obtidos, concluíram que, em triangulações com mais de um milhão de pontos, um algoritmo de maior eficiência assintótica, mesmo requerendo pré-processamento e estruturas de dados adicionais, seria a melhor solução para o problema de localização planar de pontos.

Como visto, a formulação de algoritmo de localização planar de pontos que altera dinamicamente a subdivisão, à medida que surgem eventos de inserção ou remoção nos diagramas de Delaunay ou Voronoi (O'ROURKE, 1998), é uma área de pesquisa que se mantém ativa.

Como a eficiência da localização planar de pontos, fundamentalmente, depende da estrutura de dados e algoritmo empregados, nosso problema consiste em *construir um algoritmo dinâmico de localização planar de pontos*.

1.3 Objetivos

O principal objetivo deste trabalho é formular e implementar um algoritmo de localização planar de pontos que altera dinamicamente a subdivisão planar, à medida que surgem eventos de inserção ou remoção nos diagramas de Delaunay ou Voronoi (O'ROURKE, 1998).

O objetivo secundário é discutir os aspectos críticos da geração da malha — Triangulação de Delaunay, sobretudo a baseada em abordagem incremental, em que se insere o problema da Localização Planar de Pontos.

1.4 Contribuições

Este trabalho traz alguns resultados positivos para a geometria computacional, sobretudo, para a geração automática de malhas e localização planar de pontos.

1. *Constatação da robustez da linguagem de programação OCaml para tratar problemas de geração de malhas.* Em experimentos com a triangulação de Delaunay baseada em divisão-e-conquista, foi demonstrada a estabilidade numérica de OCaml para expressar números na representação ponto-flutuante com alta precisão e a instabilidade numérica da linguagem C, que falhou em testes geométricos, devido a erros de arredondamento, e produziu resultados divergentes (MOURA *et al.*, 2005).

2. *Criação da Multiárvore-B Intervalar*. Trata-se de uma estrutura de dados derivada da Árvore-B, incrementada com mecanismo de pesquisa intervalar e arranjada dinamicamente em camadas (MOURA *et al.*, 2006).

3. *Construção de algoritmo dinâmico para o problema da localização planar de pontos.* O algoritmo (MOURA *et al.*, 2006) forma a estrutura de pesquisa para a localização à medida que a malha é construída, admitindo que os segmentos que constituem a malha sejam inseridos em qualquer ordem. Para tanto, a inclusão de cada segmento é modelada como um evento. O método das "Slabs" é utilizado para subdividir o plano. A Multiárvore-B Intervalar é empregada como estrutura de dados.

Tanto a Multiárvore-B Intervalar quanto o Algoritmo Dinâmico de Localização Planar de Pontos poderiam ter sido implementados com qualquer linguagem de programação. Em cada linguagem teriam um desempenho característico. Aqui, eles foram implementados em OCaml, porque foi a linguagem de programação definida desde o início da pesquisa.

1.5 Metodologia

Inicialmente, faz-se uma revisão bibliográfica dos principais problemas relacionados à localização planar de pontos, sobretudo triangulação de Delaunay com a abordagem incremental.

Após esse levantamento, foi estudada a linguagem de programação OCaml (LEROY, 2003), uma linguagem funcional projetada pelo INRIA[1], que, decidiu-se, seria empregada na implementação dos algoritmos.

Feito o estudo sobre OCaml, implementou-se, em OCaml, o algoritmo de Guibas-Stolfi (GUIBAS; STOLFI, 1985), melhorado por Geoff Leach (LEACH, 1992). Os resultados comparativos do código implementado em OCaml com o código implementado originalmente em C permitiram avaliar as vantagens e desvantagens de ambas as linguagens para tratar problemas geométricos que dependem de precisão numérica.

Na seqüência, foi implementado um gerador de malhas de elementos finitos bidimensional. As malhas produzidas pelo gerador foram utilizadas nos testes do algoritmo dinâmico de localização planar de pontos.

1.6 Estrutura da Tese

Nesta tese, não são discutidos aspectos teóricos do método dos elementos finitos, antes são tratados temas puramente relacionados à qualidade da malha e à otimização do seu processo construtivo. A localização planar de pontos constitui o tópico central da pesquisa, sendo a maior contribuição da tese o algoritmo dinâmico baseado em eventos, utilizando o método das “Slabs” como técnica de subdivisão do plano e a Multiárvore-B Intervalar como estrutura de dados. Dessa forma, os demais capítulos da tese estão organizados como se segue:

No Capítulo 2, é enfocada a geração da malha como uma tarefa composta de múltiplas etapas. São relatados os procedimentos computacionalmente críticos, relevando sua importância à qualidade final da malha e ao seu processo de construção. Fala-se da complexidade dos domínios geométricos, muitas vezes compostos de cavidades, fronteiras internas e formas complicadas. Também são discutidas as propriedades ideais de um gerador de malhas, principalmente a de construir uma malha segundo certos critérios de qualidade e a

[1] Institute National de Recherche en Informatique et Automatique (Instituto Nacional de Pesquisa em Automação e Tecnologia da Informação, Rocquencourt, França).

de alguma condição de parada pré-estabelecida. A triangulação de Delaunay — incluindo suas propriedades fundamentais, tipos e complexidades de algoritmos e a finalidade da localização planar de pontos para o algoritmo de triangulação incremental — é outro tópico discutido. Finalmente, são apresentados outros dois tópicos relevantes para a qualidade da malha: a inserção de pontos de Steiner, também chamada de triangulação de Steiner, e duas técnicas de refinamento da malha — melhoria de regiões com triângulos de qualidade insatisfatória.

No Capítulo 3, são apresentados os resultados da implementação da triangulação de Delaunay pelo método de divisão-e-conquista baseada no algoritmo de Guibas-Stolfi, utilizando OCaml. Foi nosso primeiro ensaio do emprego da linguagem OCaml na implementação de algoritmo geométrico. São expostas ainda as características mais relevantes de OCaml e discutidas questões concernentes à tradução de código escrito em C para OCaml.

No Capítulo 4, é apresentado um gerador de malhas escrito em OCaml e exibida uma galeria de objetos geométricos cobertos por malhas uniformes e gradientes. A implementação do gerador de malhas seguiu o mesmo estilo de programação adotado na implementação do algoritmo de triangulação de Delaunay tratado no Capítulo 3.

No Capítulo 5, é apresentado um algoritmo dinâmico de localização planar de pontos, como solução a um dos problemas importantes da geometria computacional. O algoritmo dinâmico apresentado, embora tenha sido motivado pelo método de força bruta de localização planar de pontos empregado no algoritmo do gerador de malhas, é um algoritmo genérico, que pode ser útil em outros contextos, como, por exemplo, aplicações relacionadas a sistemas de informações geográficas.

No Capítulo 6, são apresentadas as conclusões e indicações de trabalhos futuros.

Capítulo II

Aspectos Críticos da Geração da Malha

2.1 Introdução

Os objetos do mundo real cujos fenômenos físicos se pretende simular possuem peculiaridades importantes, como formato irregular, lacunas e estrutura material heterogênea. Para possibilitar a construção automática da malha, esses objetos são modelados por polígonos ou grafos planares de linhas retas — PSLG.

A geração de malhas é um procedimento complexo e custoso que resolve o problema de decompor um domínio geométrico qualquer em partes menores denominadas de elementos. É também uma tarefa demorada, em razão da complexidade dos domínios geométricos e da necessidade de se produzir uma malha que propicie precisão à simulação de fenômenos físicos.

Assim, a dificuldade computacional inerente à construção da malha se justifica, sobretudo, em virtude de ela ocorrer como que proliferando e ajustando elementos geométricos, circunscritos às fronteiras do polígono e sob outras restrições impostas pelas características do objeto real, que lhe foram, de algum modo, transferidas.

2.2 Tipos de Domínios Geométricos

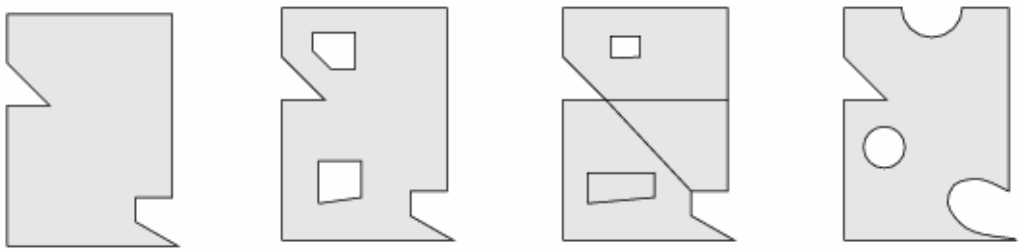
Essa seção foi aqui inclusa por se entender que a complexidade geométrica dos domínios (em duas ou três dimensões) acarreta dificuldades algorítmicas ao processo de construção da malha.

Em duas dimensões, são distinguidos cinco tipos de domínios planares (BERN; PLASSMAN, 2000; BERN; EPPSTEIN, 1992). Na Figura 2.1, são mostrados quatro deles.

- *Conjunto de pontos.* A entrada é um conjunto de pontos no plano. Sem os pontos de Steiner, os vértices da triangulação são exatamente os pontos de entrada, e a fronteira da triangulação é o casco convexo;
- *Polígono simples*. Inclui apenas a fronteira externa e a região interior;
- *Polígono com buracos*. É um polígono simples acrescido de outros polígonos simples sem a região interior. Sua fronteira tem mais que um componente conectado;
- *Domínio múltiplo*. É um polígono com buracos contendo fronteiras internas;
- *Domínio curvo.* É o que permite lados curvos.

Em três dimensões, as entradas, em sua maioria, têm tipos análogos:

- *Poliedro simples*. É topologicamente equivalente a uma esfera;
- *Poliedro geral*. Pode ser multiplamente conectado e ter cavidades, significando que sua fronteira pode ter mais que um componente conectado.
- *Domínios poliédricos múltiplos*. São poliedros gerais com fronteiras internas;
- *Domínios curvos tridimensionais.* São aqueles que têm fronteiras curvas.



(a) (b) (c) (d)

Figura 2.1. Tipos de entradas bidimensionais: (a) polígono simples, (b) polígono com buracos, (c) domínio múltiplo e (d) domínio curvo (BERN; PLASSMAN, 2000).

2.3 Tipos de Malhas

Existem três tipos de malhas: e*struturadas*, *não-estruturadas* e *híbridas*.

Uma *malha estruturada* (Figura 2.2 (a)) em duas dimensões é muitas vezes uma grade quadrada deformada por algumas transformações de coordenadas. Cada vértice da malha, exceto aqueles das bordas, tem uma vizinhança local isomórfica. Em três dimensões, uma malha estruturada é normalmente uma grade cúbica deformada. Não é estritamente necessário armazenar os valores das coordenadas dos nós, pois eles podem ser implicitamente conhecidos pelo número de cada elemento.

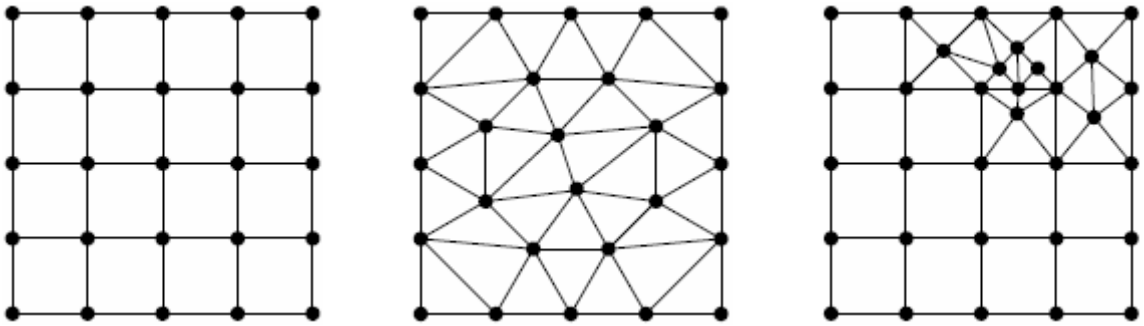
Uma *malha não-estruturada* (Figura 2.2 (b)) é, na maioria dos casos, uma triangulação

com vizinhança local variável.

Uma *malha híbrida* (Figura 2.2 (c)) é aquela resultante da combinação de malhas estruturadas e não-estruturadas.

As malhas estruturadas oferecem certas vantagens e desvantagens sobre as não-estruturadas. Elas são mais simples e também mais convenientes para uso em métodos das diferenças finitas menos complexas. Elas requerem menos memória de computador, pois suas coordenadas podem ser calculadas em vez de explicitamente armazenadas.

A maior desvantagem de uma malha estruturada é a falta de flexibilidade em ajustar-se a um domínio com forma complicada, apesar de inúmeras técnicas para encontrar as transformações de coordenadas apropriadas (BERN; EPPSTEIN, 1992).



(a) Estruturada (b) Não-estruturada (c) Híbrida

Figura 2.2. Tipos de Malhas.

2.4 Propriedades Desejáveis de uma Malha e de Geradores de Malha

Na prática, quando um domínio de entrada e uma condição numérica são dados, a geração de malha segue três passos (TENG, 1999):

1. Converter a geometria de entrada em uma representação[2] padrão, como PSLG;
2. Gerar uma malha construindo seu conjunto de pontos e elementos;
3. Aplicar um algoritmo de melhoramento e refinamento da malha como uma maneira de aprimorar a qualidade da malha construída no Passo 2.

O Passo 3 tem por objetivo melhorar a qualidade da malha e reduzir o erro na aproximação provida pelo método dos elementos finitos. Esse erro evolui com o tamanho do elemento e está relacionado aos ângulos mínimo e máximo. Por essa razão, o tamanho do

[2] Há várias representações para descrever a geometria de um domínio de entrada, como *PSLG*, *PLS (uma extensão natural do PSLG para uma dimensão elevada)*, *CSG*, *d-reps*, *dd-reps*. Veja detalhes em (TENG, 1999).

elemento e dos ângulos mínimo e máximo são uma importante medida de qualidade tanto para geração de malha como para interpolação de superfície.

Investigações (SHEWCHUK, 2002) acerca do espectro de curvas de contorno demonstraram que o condicionamento de matriz de rigidez e a acurácia da interpolação e da simulação dependem do tamanho e forma dos elementos finitos e que medidas de qualidade devem ser estabelecidas.

Infelizmente, as medidas de qualidade para interpolação e condicionamento de matriz são conflitantes (MOORE, 1992). Por exemplo, ângulos pequenos são ruins para condicionamento de matriz, mas não para interpolação.

Na interpolação, há dois tipos de erros: a diferença entre a função interpolada e a função verdadeira e a diferença entre o gradiente da função interpolada e o gradiente da função verdadeira. Erros no gradiente podem ser surpreendentemente importantes, se a aplicação é de renderização, construção de mapas ou simulação, porque eles podem comprometer a acurácia ou criar artefatos visuais não esperados.

Por exemplo, seja f uma função. Seja g uma interpolação linear de f sobre alguma triangulação. A Tabela 2.1 apresenta as medidas de qualidade relacionadas a tamanho e forma do elemento para três critérios diferentes: *erro de interpolação*, *erro de interpolação gradiente* e *condicionamento de matriz de rigidez*.

Tabela 2.1. Três critérios para elementos lineares (SHEWCHUK, 2002).

Critério	Medida de qualidade	Forma dos elementos
Erro de interpolação $\| f - g \| \infty$	- Tamanho muito importante - Forma dos elementos é irrelevante	Qualquer:
Erro de interpolação gradiente $\| \nabla f - \nabla g \| \infty$	- Tamanho importante - Ângulos grandes são ruins; pequenos são bons	ideal ok
Condicionamento do autovalor máximo da matriz de rigidez λ_{Max}	- Ângulos pequenos são ruins; grandes são bons	

Também foi mostrado (SHEWCHUK, 2002) que a convergência do método dos elementos finitos falha quando os ângulos se aproximam de 180º, como o triângulo mostrado na Figura 2.3.

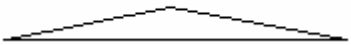


Figura 2.3. Um triângulo de formato inadequado para o método dos elementos finitos.

Os algoritmos de Chew (CHEW, 1989) e Ruppert (RUPPERT, 1995) mostraram-se eficazes em cumprir esses requisitos qualitativos.

Técnicas adaptativas como “circle packing” (BERN, 1994; BERN; MITCHELL; RUPPERT, 1994; BERN; EPPSTEIN, 1997) e “quadtree” (BERN, 2002) também satisfazem essas métricas qualitativas.

“Circle packing” é uma técnica introduzida por Bern, Mitchell e Ruppert (BERN; MITCHELL; RUPPERT, 1994) que preenche o domínio com círculos e depois constrói a malha, acrescentando arestas entre o centro dos círculos e os pontos de tangência nas suas fronteiras.

Chew provou que os algoritmos baseados em “circle packing”, cujos passos são ilustrados na Figura 2.4, calculam o local dos pontos a serem inseridos, chamados “pontos de Steiner”, e chegam a produzir uma malha em que nenhum ângulo é mais agudo que 30º ou mais obtuso que 120º.

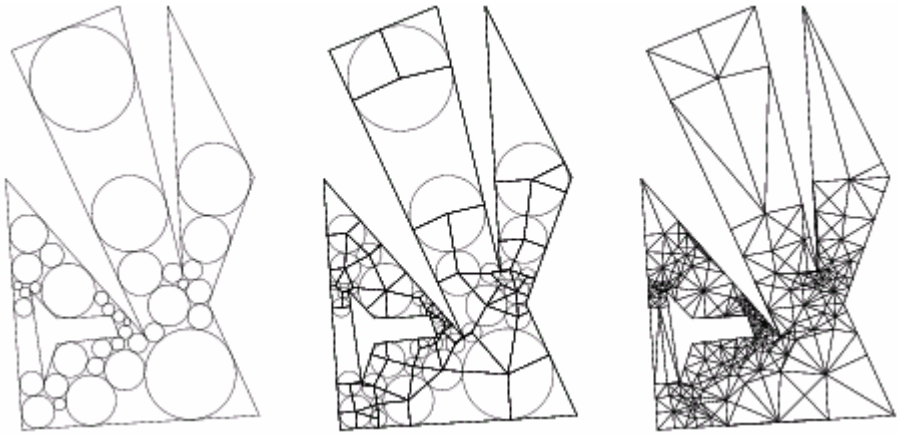


Figura 2.4. Passos da triangulação baseada em “circle packing” (BERN, 1994; BERN; EPPSTEIN, 1997).

Quadtree é uma partição recursiva de uma região do plano em quadrados alinhados aos eixos das coordenadas (Figura 2.5). A técnica é também chamada *malha cartesiana*, porque é produzida por divisão repetitiva de um quadrado em quatro quadrados de ¼ do tamanho. Um quadrado, a *raiz*, cobre a região inteira. Um quadrado pode ser decomposto em quatro quadrados-filhos, formados a partir dos pontos médios de seus segmentos verticais e horizontais. A coleção de quadrados formam então uma árvore, com os quadrados menores posicionados nos seus níveis mais inferiores. *Octrees* são uma generalização das *quadtrees* para três dimensões. Na Figura 2.6, é exibida uma malha tetraédrica derivada de uma *octree*.

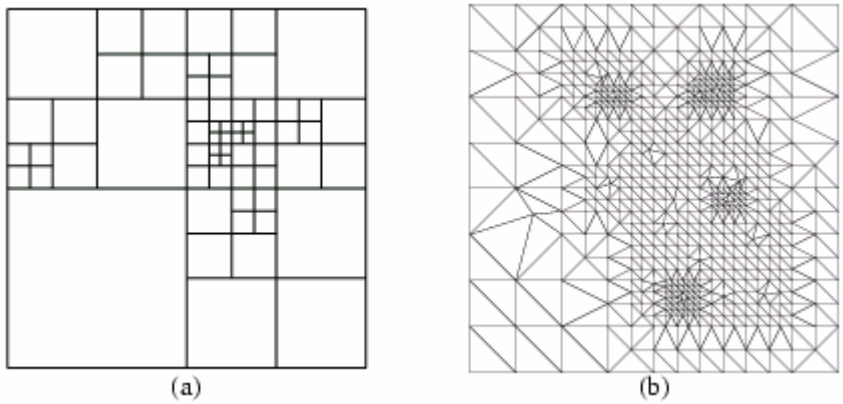


Figura 2.5. (a) Uma *quadtree*. (b) Uma triangulação de um conjunto de pontos baseada em *quadtree* em que nenhum ângulo é menor que 20º (SHEWCHUK, 1997).

Em uma *quadtree* cujos quadrados foram triangulados (BERN; EPPSTEIN, 1992) pode-se garantir que todos os ângulos medem entre 36º e 80º, sendo que, com algum melhoramento adicional, a triangulação pode se tornar equivalente a uma malha de triângulos equiláteros.

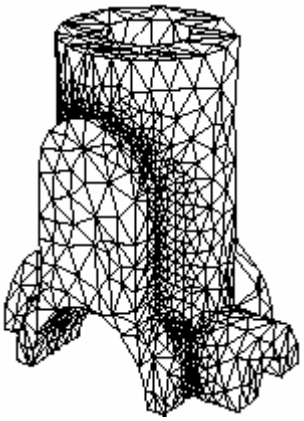


Figura 2.6. A superfície de uma malha tetraédrica derivada de uma *octree* (BERN; PLASSMAN, 2000).

Uma outra preocupação é oferecer tanto controle quanto possível sobre o tamanho dos elementos na malha. Idealmente, esse controle inclui a habilidade de dispor, em gradação, de pequenos a grandes elementos sobre uma distância relativamente curta. A razão para esse requisito é que o tamanho do elemento tem dois efeitos sobre uma simulação de elemento finito. Elementos pequenos, densamente empacotados, oferecem mais acurácia que os maiores, esparsamente empacotados; mas o tempo de computação requerido para resolver um problema é proporcional ao número de elementos. Então, a seleção do tamanho do elemento tem implicações sobre a velocidade e exatidão. Por outro lado, o tamanho do elemento requerido para alcançar uma dada precisão depende do comportamento do fenômeno físico

que está sendo modelado e pode variar por todo o domínio do problema. Por exemplo, uma simulação de fluxo de fluido requer elementos menores em meio à turbulência que em áreas de relativa tranqüilidade; em três dimensões, o elemento ideal em uma parte da malha pode variar em volume por um fator de milhão ou mais do elemento ideal em outra parte da malha. Se elementos de tamanho uniforme são usados por toda a malha, deve-se selecionar um tamanho menor de modo a garantir acurácia suficiente na maior parte da porção de demanda do domínio do problema e por isso possivelmente incorrer em demandas computacionais excessivamente grandes. Para evitar essa cilada, o gerador de malha deve oferecer rápida gradação de tamanhos menores a maiores (SHEWCHUK, 1997).

Um aspecto final é que a malha deve se ajustar às fronteiras do domínio geométrico de qualquer um dos tipos descritos na Seção 2.2. Veja, por exemplo, a malha que cobre a Figura 2.7, cuja fronteira inclui três buracos.

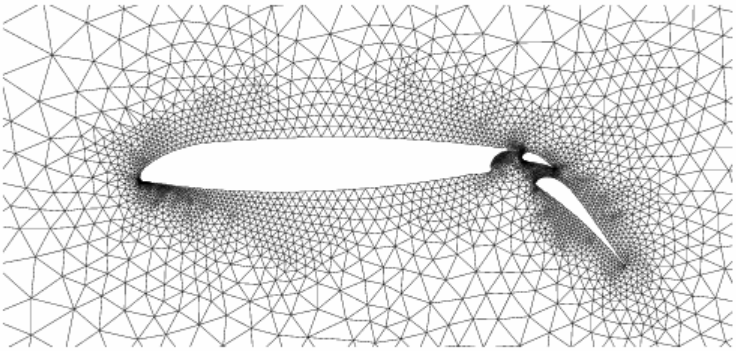


Figura 2.7. Parte da triangulação de uma região com três buracos (BERN; EPPSTEIN; GILBERT, 1994).

Além de servir como instrumento de avaliação de métodos de geração de malha, as métricas de qualidade apontaram o triângulo equilátero como o polígono ideal para representar os elementos da malha triangular bidimensional.

2.5 Triangulação de Delaunay

Nesta seção, será discutida a triangulação de Delaunay em malhas não-estruturadas no plano Euclidiano bidimensional.

A triangulação de Delaunay, semelhante às técnicas de integração ao segmentar a área sob a curva produzida por uma função, particiona a região interna de um polígono. Na integral, os fragmentos são retângulos ou trapézios; na triangulação, esses fragmentos são triângulos.

Para um polígono ser decomposto deverá ter mais de três vértices (Figura 2.8), ou seja, ter no mínimo uma diagonal. Desse modo, o polígono será particionado em triângulos pela adição de uma ou mais diagonais. O'Rourke (O'ROURKE, 1998) apresenta uma propriedade, que quantifica o número de diagonais e triângulos após a triangulação: toda triangulação de um polígono *P* de *n* vértices usa *n*-3 diagonais e consiste de *n*-2 triângulos.

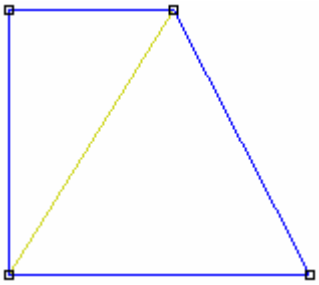


Figura 2.8. Triangulação de um polígono.

Por outro lado, se a entrada para o algoritmo de triangulação for um conjunto de pontos no plano (Figura 2.9), a quantidade de triângulos e arestas é dada como segue (BERG, 1997):

Seja *P* um conjunto de *n* pontos no plano, não todos colineares, e *k* o número de pontos em *P* que estão sobre a fronteira do casco convexo de *P*. Então, em qualquer triangulação de *P* há 2*n*-2-*k* triângulos e 3*n*-3-*k* arestas.

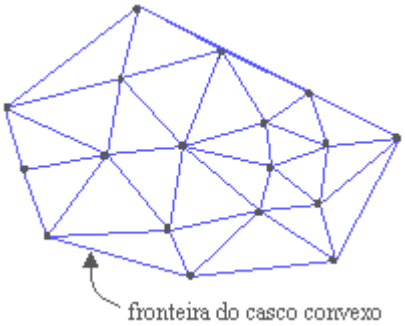


Figura 2.9. A triangulação de Delaunay sobre uma nuvem de pontos.

De fato, essas duas propriedades são interessantes. Mas qual é a importância da triangulação de Delaunay para a geração de malhas?

Primeiro, a maioria dos polígonos que descreve objetos do mundo real tem formato irregular e regiões pertencentes a diferentes domínios de interesse. Nesse contexto, a triangulação de Delaunay, conceitualmente, pode ser vista como uma estratégia de decompor um domínio em triângulos, respeitando suas características geométricas, como um passo inicial do processo de discretização. Desse modo, a triangulação de Delaunay funciona como uma espécie de gabarito para delimitar o espaço de ocupação, o qual, posteriormente, será

decomposto até que sejam atendidos todos os critérios de qualidade referentes à área e medida angular para cada triângulo.

Segundo, a triangulação de Delaunay contribui para a qualidade da malha final, visto que, dado um conjunto de vértices, maximiza o ângulo mínimo entre todas as maneiras possíveis de triangular aquele conjunto (SHEWCHUK, 1997).

Formalmente, uma *triangulação* de um conjunto *V* de vértices é um conjunto *T* de triângulos cujos vértices coletivamente são *V*, cujos interiores não interceptam um ao outro e cuja união é o fecho convexo de *V* e cada triângulo que intercepta *V* o faz somente nos vértices do triângulo.

A *triangulação de Delaunay D* de *V*, introduzida, em 1934, pelo matemático russo Boris Nikolaevich Delone — depois chamado Boris Delaunay[3] —, é um grafo definido como segue. Qualquer círculo no plano é tido como *vazio* se não cerca nenhum vértice de *V*. Vértices são permitidos sobre o círculo. Sejam *u* e *v* dois vértices de *V*. Um *círculo-circundante* da aresta *vw* é qualquer círculo que passa através de *u* e *v*. A aresta *uv* está em *D* se e somente se existe um círculo-circundante de *uv*. Uma aresta que satisfaz essa propriedade é dita ser *Delaunay*. Na Figura 2.10, é ilustrada uma triangulação de Delaunay sobre um conjunto de quatrocentos pontos no plano.

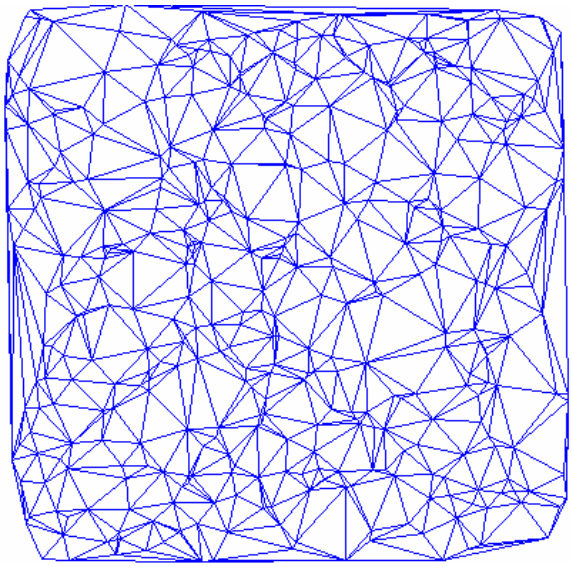


Figura 2.10. A triangulação de Delaunay sobre um conjunto de 400 pontos aleatórios.

Cada aresta que conecta um vértice a seu vizinho mais próximo é Delaunay. Se *w* é o vértice mais próximo a *v*, o menor círculo que passa por *v* e *w* não circunda quaisquer outros vértices.

A definição de um triângulo de Delaunay servirá para garantir que o conjunto de arestas de Delaunay de um conjunto de vértices coletivamente formam uma triangulação. O *círculo-circundante* de um triângulo é o único círculo que passa através de todos os seus três vértices. Um triângulo é dito ser de Delaunay se somente se seu círculo-circundante tem seu interior vazio. Essa definição característica dos triângulos de Delaunay, ilustrada na Figura 2.11, é chamada *propriedade do círculo-circundante vazio* (SHEWCHUK, 1999).

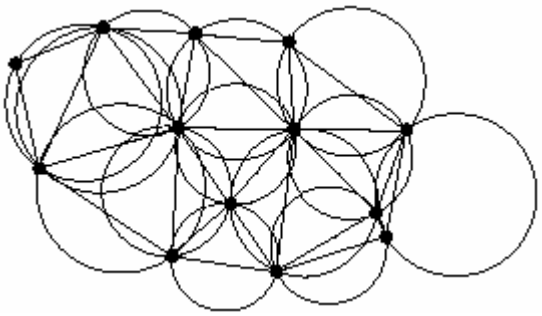


Figura 2.11. Cada triângulo em uma triangulação de Delaunay tem um círculo-circundante vazio (SHEWCHUK, 1999).

Na seção seguinte, será apresentada a definição da principal primitiva geométrica para computar triangulações de Delaunay.

2.5.1 O teste "InCircle"

Suponha a existência de quatro pontos distintos no plano: *A*, *B*, *C* e *D*, conforme a Figura 2.12.

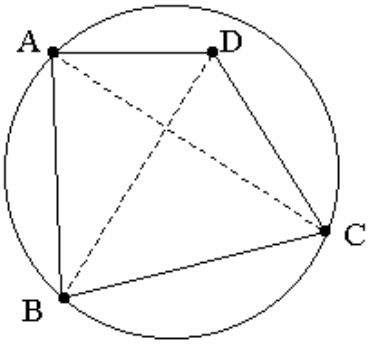


Figura 2.12. O teste "InCircle".

Conforme a propriedade apresentada anteriormente e exemplificada na Figura 2.8, a triangulação do polígono exibido na Figura 2.12 originará dois triângulos por meio da

[3] *Delaunay* é uma transcrição de *Delone* em estilo francês.

inserção de uma única diagonal. Todavia, para completar a triangulação, tem-se que adicionar a diagonal *AC* ou a diagonal *BD*.

O predicado "InCircle" provê a informação essencial que determina a estrutura topológica dos diagramas de Voronoi ou Delaunay.

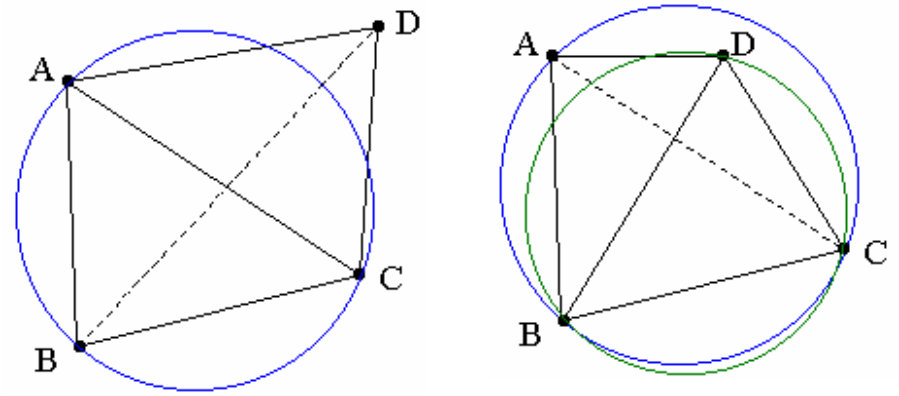
Definição 2.1. O predicado *InCircle(A, B, C, D)* é definido como verdadeiro se e somente se o ponto *D* for interior à região do plano limitada pelo círculo orientado *ABC* e ficar à esquerda dele.

Em particular isso implica que *D* deve estar *dentro* do círculo *ABC* se os pontos *A*, *B* e *C* definem um triângulo orientado em sentido anti-horário e *fora* se os pontos definem um triângulo em sentido horário. Se *A*, *B* e *C* são colineares, interpreta-se a linha como um círculo adicionando-se um ponto no infinito. Se *A*, *B*, *C* e *D* são co-circulares, então esse predicado retorna falso. Este teste equivale a perguntar se $\angle ABC + \angle CDA > \angle BCD + \angle DAB$. A aplicação prática é escolher as duas combinações que determinem dois triângulos cuja soma dos ângulos mínimos seja a máxima. Outra forma equivalente desse teste é dada abaixo, baseada na coordenada dos pontos.

O teste InCircle(A, B, C, D) é também equivalente a

$$\mathrm{D}(A, B, C, D) = \begin{vmatrix} x_A & y_A & x^2_A + y^2_A & 1 \\ x_B & y_B & x^2_B + y^2_B & 1 \\ x_C & y_C & x^2_C + y^2_C & 1 \\ x_D & y_D & x^2_D + y^2_D & 1 \end{vmatrix} > 0$$

Qual a utilidade do teste "InCircle" para a construção de diagramas de Delaunay? Considere-se, por exemplo, o caso de quatro pontos que são vértices de um quadrilátero convexo *ABCD*, como mostrado na Figura 2.13. Os lados *AB*, *BC*, *CD* e *DA* estão no fecho convexo e, portanto, têm que ser inclusos. Para completar a triangulação, tem-se que adicionar a diagonal *AC* ou a diagonal *BD*. Pode-se decidir qual das duas será adicionada, calculando *InCircle*(*A*, *B*, *C*, *D*). Se for falso, então o círculo *ABC* é livre de pontos e *AC* é aresta de Delaunay. Caso contrário, *AC* não é aresta de Delaunay. Na figura abaixo, é ilustrado o uso de "InCircle" para determinar qual das duas arestas é aresta de Delaunay.



InCircle(*A*, *B*, *C*, *D*) = F

InCircle(*A*, *B*, *C*, *D*) = T (externo)
InCircle(*B*, *C*, *D*, *A*) = F (interno)

Figura 2.13. Uso do predicado "InCircle" para decidir qual das duas arestas (*AC* ou *BD*) é aresta de Delaunay.

Essencialmente, todos os algoritmos de triangulação de Delaunay consistem em selecionar iterativamente a partir de um conjunto de vértices um subconjunto de três vértices que satisfaça ao predicado "InCircle" e forme um triângulo. Ressalte-se que, entre todas as maneiras possíveis de triangular aquele conjunto de vértices, os triângulos formados possuem os ângulos mínimos maximizados e, conseqüentemente, os ângulos máximos minimizados.

Mais detalhes sobre o predicado "InCircle" podem ser encontrados em (GUIBAS; STOLFI, 1985).

2.5.2 Algoritmos para construir a Triangulação de Delaunay

Quatro tipos de algoritmos são usados para construir triangulações de Delaunay: divisão-e-conquista (GUIBAS; STOLFI, 1985), incremental (BERG, 1997), "sweepline" (SHEWCHUK, 1997) e com restrições (KALLMANN; BIERI; THALMANN, 2003). Os mais simples são os algoritmos de inserção incremental. Em duas dimensões, existem algoritmos mais rápidos baseados em técnicas de divisão-e-conquista e "sweepline". Para se obter uma informação panorâmica desses e de outros algoritmos de triangulação de Delaunay bidimensionais, deve-se consultar Su (SU, 1994) e Drysdale (SU; DRYSDALE, 1995).

Neste trabalho, serão discutidos apenas os algoritmos baseados em paradigmas de divisão-e-conquista, com restrição e incremental, em virtude de sua popularidade e importância.

a) Divisão-e-Conquista

Em 1975, Shamos e Hoey apresentaram à comunidade científica o primeiro algoritmo baseado em divisão-e-conquista, que requeria tempo O(n log n) para construir diagramas de Voronoi — uma forma dual do diagrama de Delaunay. Mas somente em 1977, a técnica foi pela primeira vez aplicada a problema de casco convexo por Preparata e Hong. Analogamente ao algoritmo "mergesort", sua essência é dividir o problema em duas partes aproximadamente iguais, resolver cada uma delas recursivamente e criar uma solução completa pela junção das duas meias soluções. Quando a recursividade reduz o problema original em pequenos subproblemas, eles normalmente se tornam muito fáceis de ser resolvidos. Esse algoritmo é teoricamente importante, pois tem complexidade assintoticamente ótima, mas é de difícil implementação e, por essa razão, parece não ser usado com tanta freqüência quanto os outros algoritmos mais lentos. O'Rourke, por exemplo, preferiu ilustrar a implementação do algoritmo incremental em (O'ROURKE, 1998). O esboço do algoritmo de divisão-e-conquista é o seguinte:

1. Os pontos são ordenados ao longo do eixo x;
2. Se houver três ou menos pontos, a triangulação de Delaunay é construída diretamente. Caso contrário, os pontos são divididos em dois conjuntos aproximadamente iguais por uma linha perpendicular ao eixo x, o Passo 2 é recursivamente aplicado para construir as triangulações de Delaunay desses conjuntos, e os resultados são agrupados.

O procedimento de combinar a triangulação dos dois subconjuntos é a parte mais complicada e custosa do algoritmo. Uma exposição do algoritmo acompanhada de uma demonstração de sua correção pode ser encontrada em (GUIBAS; STOLFI, 1985). Por brevidade, são omitidos aqui os detalhes, com a sugestão de que se consulte (GUIBAS; STOLFI, 1985). Na prática, a tarefa compreende em tomar das duas subsoluções, a da esquerda e da direita, os vértices dos triângulos localizados próximos a uma linha imaginária perpendicular ao eixo x e refazer sobre tais vértices a triangulação de Delaunay. Nesse processo de reconstrução, há eliminação e acréscimo de arestas (FIGUEIREDO; CARVALHO, 1991). A partir dos vértices mais inferiores, em direção ao topo, toma-se sempre um vértice pertencente ao conjunto da esquerda e um vértice pertencente ao conjunto da direita. O critério a ser atendido é o que determina que existe uma triangulação de Delaunay se somente se houver três vértices sobre o círculo-circundante e nenhum em seu

interior. As arestas que satisfizerem a tal critério são mantidas; as demais são eliminadas e substituídas por outras que atenderem ao critério do círculo-circundante.

Como exemplo, considere a triangulação de Delaunay de um conjunto de pontos, denotada por Del(C). A aplicação do algoritmo de triangulação de Delaunay por divisão-e-conquista resultará em duas triangulações, denotadas por Del(C_1) e Del(C_2), de tamanho aproximadamente igual, localizados, respectivamente, à esquerda e à direita da mencionada linha vertical imaginária (Figura 2.14).

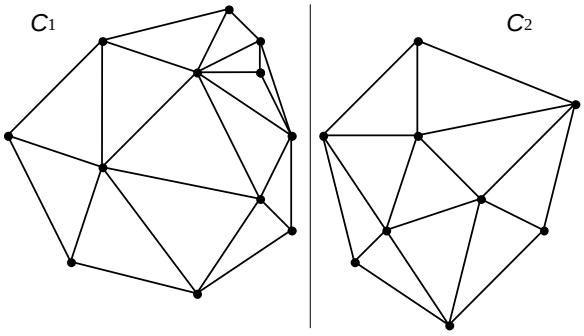


Figura 2.14. Triangulações de Delaunay de C_1 e C_2, separadas por uma linha vertical imaginária.

Na Figura 2.15, é mostrada a construção de Del($C_1 \cup C_2$), a partir de Del(C_1) e Del(C_2). As arestas eliminadas de Del(C_1) e Del(C_2) são representadas em tracejado, enquanto as arestas acrescentadas pelo algoritmo são mostradas em negrito.

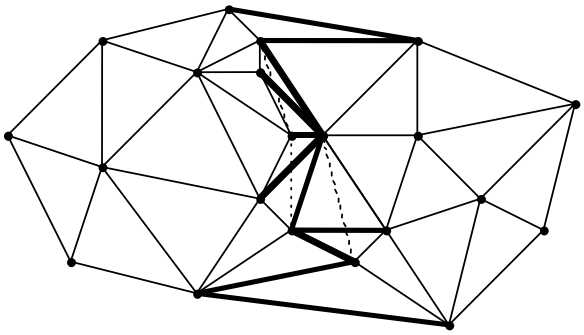


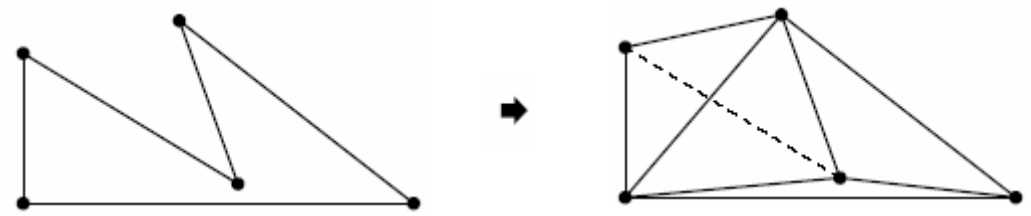
Figura 2.15. Triangulação de Delaunay de $C_1 \cup C_2$.

b) Triangulação de Delaunay Restrita ou com Restrições

A triangulação de Delaunay com restrições é estrutura geométrica fundamental, com aplicações em interpolação, renderização e em geração de malhas, quando a triangulação requerida tiver que se ajustar à forma do objeto que está sendo modelado.

A triangulação com restrições é definida como segue: "*Dado um conjunto C de pontos do plano e um conjunto G de segmentos com extremos em C (tais que dois elementos quaisquer de G não se interceptam a não ser em seus extremos), obter uma triangulação do fecho convexo de C, cujo conjunto de vértices seja C e que inclua todos os segmentos em G*" (FIGUEIREDO; CARVALHO, 1991).

A triangulação de Delaunay apresenta duas deficiências: uma é que ela pode conter triângulos de qualidade pobre; a outra é omitir algumas fronteiras do domínio geométrico. A primeira deficiência é motivada pelas características geométricas do PSLG e é suprida pela inserção de pontos de Steiner, também chamada de triangulação de Steiner. A segunda deficiência é provocada pela tentativa de obtenção de triângulos de qualidade e é reparada pela triangulação de Delaunay com restrições. Na Figura 2.16 (b), é mostrada uma triangulação de um PSLG (a) em que o triângulo da base é de má qualidade e a linha tracejada mostra o segmento que foi omitido. Ambos os problemas podem ser solucionados pela inserção de vértices adicionais, como ilustrado na Figura 2.17. Infelizmente, isso somente é aplicável em duas dimensões.



(a) Um PSLG.

(b) Triangulação do PSLG com triângulos de qualidade pobre, ausência de segmentos originais e presença de segmentos indevidos.

Figura 2.16. Amostra das deficiências da triangulação de Delaunay (SHEWCHUK, 1997).

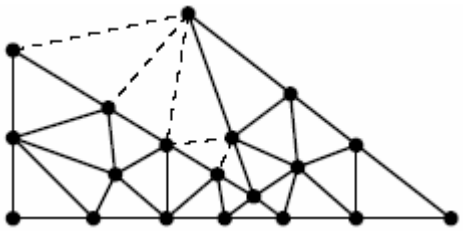


Figura 2.17. Amostra da correção provida pela inserção de pontos de Steiner (SHEWCHUK, 1997).

Chew apresentou um algoritmo de triangulação de Delaunay com restrição, mostrando que ela também pode ser obtida em tempo $O(n \log n)$, usando a técnica de divisão-e-conquista.

A triangulação com restrições de um dado PSLG é similar à triangulação de Delaunay, exceto que nela todo segmento de entrada aparece como uma aresta da triangulação. Contudo, para preservar as características geométricas do PSLG, mantendo todos seus segmentos, algumas arestas podem não atender à propriedade da triangulação de Delaunay, violando a regra do predicado "InCirle", apresentado na Seção 2.5.1. Em outras palavras, uma triangulação de Delaunay com restrição não é necessariamente uma triangulação de Delaunay, como ilustrado na Figura 2.18. Notem-se as diferenças no formato dos triângulos nos dois tipos de triangulações.

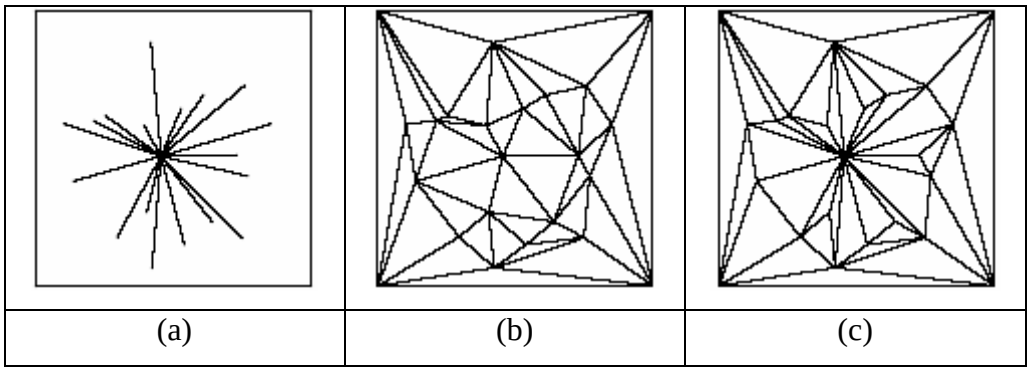


Figura 2.18. (a) Um PLSG. (b) Uma triangulação de Delaunay do PSLG. (c) Uma triangulação de Delaunay com restrições do PSLG (SHEWCHUK, 1997).

c) Incremental

Em algoritmos baseados na abordagem incremental aleatória, a distribuição aleatória de pontos é mais vantajosa que a não-aleatória por ser mais resistente contra distribuições precárias de pontos. O desempenho dos algoritmos incrementais sobre essas distribuições é superior (SU, 1994). Comportamento similar se verifica em algoritmos de pesquisa, em que a seleção randômica oferece boa proteção contra a ocorrência do pior caso (KNUTH, 1998). Por exemplo, uma árvore binária cujos elementos são inseridos em ordem aleatória tende a ser balanceada, ao passo que, se a inserção proceder em ordem ascendente ou descendente, a árvore terá topologia assimétrica, favorecendo a ocorrência do pior caso.

No algoritmo incremental, a triangulação inicia-se com um supertriângulo que envolve todos os pontos de entrada, como mostrado na Figura 2.19.

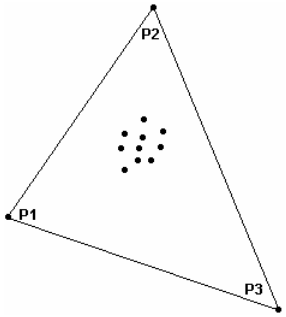


Figura 2.19. Supertriângulo contendo pontos a serem triangulados.

Depois, o algoritmo adiciona os pontos em ordem aleatória e mantém a triangulação de Delaunay do conjunto de pontos atual. Para ilustrar, considere a adição de um ponto p_r. Primeiro, é encontrado o triângulo da triangulação atual que contém p_r. Se acontecer de p_r cair sobre uma aresta e da triangulação, tem-se de adicionar arestas de p_r para os vértices opostos nos triângulos que compartilham e. Na Figura 2.20, são ilustrados esses dois casos. Em um dado momento, tem-se uma triangulação, mas não necessariamente uma triangulação de Delaunay. Isso porque a adição de p_r pode tornar algumas arestas existentes ilegais. Para reparar isso, é chamado um procedimento que substitui as arestas ilegais por arestas legais através de movimentação de arestas. Ao final, são descartados os três vértices do supertriângulo e todas arestas incidentes.

p_r está no interior de um triângulo p_r cai sobre uma aresta

Figura 2.20. Os dois casos em que se adiciona um ponto p_r (BERG, 1997).

2.5.3 Complexidade dos Algoritmos de Triangulação de Delaunay

Aqui, é destacada a ordem de complexidade assintótica e o nível de dificuldade de implementação dos algoritmos de triangulação de Delaunay, uma vez que explanações detalhadas podem ser encontradas em textos de geometria computacional. Na Tabela 2.2, verifica-se que dois desses algoritmos também possuem complexidade $O(n \log n)$, mas em

experimentos realizados por Shewchuk (SHEWCHUK, 1996b), esses algoritmos apresentaram desempenho bem inferior ao do algoritmo baseado em divisão-e-conquista.

Tabela 2.2. Complexidade de algoritmos de triangulação.

Método empregado	Complexidade	Implementação
Força bruta	$O(n^2)$	Trivial
Incremental	$O(n \log n)$	Trivialidade relativa
Sweepline	$O(n \log n)$	Trivialidade relativa
Divisão-e-conquista	$O(n \log n)$	Complexa

2.6 Inserção de Pontos de Steiner

Em uma triangulação, todos os pontos de entrada são cobertos pelos vértices dos triângulos formados pela triangulação de Delaunay; os pontos restantes ou adicionais são os *pontos de Steiner*. Na Figura 2.21, são contrastados os efeitos de uma triangulação com e sem pontos de Steiner.

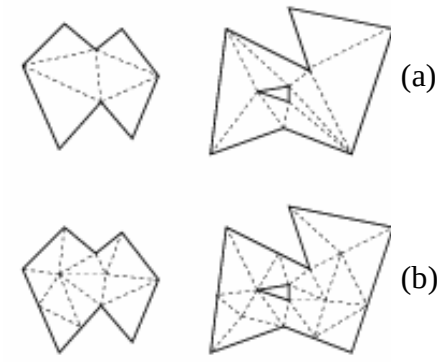


Figura 2.21. Triangulações: (a) sem pontos de Steiner e (b) com pontos de Steiner (BERN; EPPSTEIN, 1992).

O processo de inserção dos pontos de Steiner implica tanto a subdivisão de aresta, quando o ponto é inserido sobre uma delas, quanto sua remoção, quando um conjunto de triângulos é removido para possibilitar o surgimento de novos triângulos.

Por razões tanto práticas quanto teóricas, deve-se preocupar com o número de pontos de Steiner. O ângulo mínimo na triangulação de um conjunto de pontos pode aproximar-se de 60º, mas, na prática, o número de pontos de Steiner requerido para tal pode tornar o tempo de processamento proibitivo.

Em malhas construídas segundo a triangulação de Delaunay, como são inseridos os pontos de Steiner? O critério comum é o estabelecimento de limite para o tamanho das arestas *s*, para a área do triângulo *a* ou para a medida angular do triângulo. Dessa forma, enquanto

houver um triângulo em desconformidade com o critério de qualidade estabelecido, seja com círculo-circundante de raio maior que *s* ou com área maior que *a*, ou ainda com o ângulo mínimo inferior a um dado limite pré-estabelecido, o algoritmo adiciona um ponto de Steiner ao centro do círculo e recomputa a triangulação de Delaunay. Cada triângulo é localizado e testado seqüencialmente.

O algoritmo, apresentado na Figura 2.22, escrito por Bern e Plassmann (BERN; PLASSMAN, 2000), exemplifica como são inseridos os pontos de Steiner.

```
enquanto existir um triângulo t com ângulo menor que α faça
  seja c o centro do círculo circunscrito de t
  se c está dentro do diâmetro do semicírculo da aresta fronteiriça e então
    adicione o ponto médio m de e
  senão
    adicione c
  fim_se
  recompute a triangulação de Delaunay
fim_enquanto
```

Figura 2.22. Algoritmo de inserção de pontos de Steiner.

2.7 Localização Planar de Pontos

Em algoritmos de triangulação de Delaunay incremental ou de paradigma resultante da combinação entre os paradigmas incremental e de divisão-e-conquista, a localização planar de pontos é útil para determinar a que triângulo pertence cada ponto inserido na triangulação.

Os requisitos mencionados obrigam a formulação de uma estrutura que suporte consultas sobre localização de pontos.

2.7.1 Estruturas de Dados para Localização de Pontos

A estrutura de dados definida deve manter a correspondente representação de algo como um PSLG, o qual deve ser pré-processado e armazenado de modo a responder às consultas sobre a região poligonal em que se localiza um determinado ponto.

Na literatura de geometria computacional, são descritas duas estruturas: (*a*) *subdivisão em* "*slabs*" e (*b*) *mapa trapezoidal*.

a) Subdivisão em “Slabs”

Seja *S* uma subdivisão planar com *n* arestas. O problema de localização planar de pontos é armazenar *S* de modo a que possam ser respondidas consultas do seguinte tipo: dado um ponto de consulta *q*, reporte a face *f* de *S* que contém *q*. Se *q* se encontra sobre uma aresta ou coincide com um ponto, o algoritmo de consulta retornará essa informação.

A estrutura de dados necessária para desempenhar consultas sobre localização de pontos é muito simples. Traçam-se linhas verticais através de todos os vértices da subdivisão, como na Figura 2.23. Isso particiona o plano em “slabs”. As coordenadas *x* dos vértices são armazenadas ordenadamente em um “array”. Isso torna possível determinar em tempo de $O(\log n)$ a “slab” que contém um ponto de consulta *q*. No interior de uma “slab”, não há nenhum vértice de *S*. Isso significa que a parte da subdivisão colocada sobre a “slab” tem uma forma especial: todas as arestas que interceptam uma “slab” atravessam-na completamente — elas não têm nenhum ponto extremo na “slab” — e elas não cruzam uma a outra. Isso significa que elas podem ser ordenadas de cima para baixo. Note que toda região na “slab” entre duas arestas consecutivas pertencem a uma única face de *S*. As regiões mais inferior e mais superior da “slab” estão fora dos limites e são parte da face ilimitada de *S*. A estrutura especial das arestas interceptando uma “slab” implica que ela pode ser armazenada ordenadamente num “array”. Rotula-se cada aresta com a face de *S* que está imediatamente acima, dentro da “slab”.

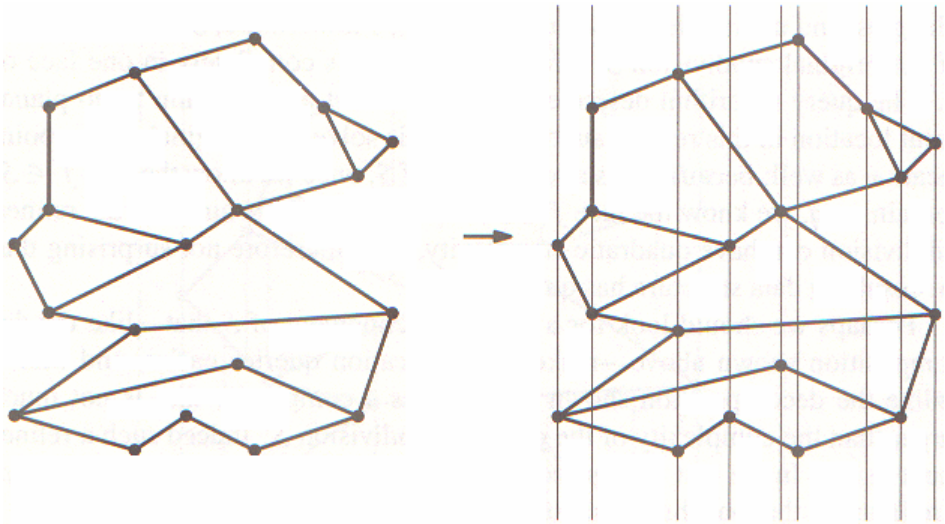


Figura 2.23. Subdivisão em *slabs* (BERG, 1997).

O algoritmo de consulta completo pode então ser descrito como se segue. Primeiro, faz-se uma busca binária com a coordenada *x* do ponto de consulta *q* no "array" que armazena as coordenadas *x* dos vértices da subdivisão. Isso informa a "slab" que contém *q*. Então, efetua-se uma busca binária com a coordenada *y* de *q* naquela "slab". A operação elementar nessa pesquisa binária é: dado um segmento *s* e um ponto *q* tal que a linha vertical através de *q* intercepta *s*, determine se *q* se encontra acima de *s*, abaixo de *s* ou sobre *s*. Isso informa o segmento diretamente abaixo de *q*. O rótulo armazenado com o segmento é a face de *S* contendo *q*. Se for verificado que não há nenhum segmento abaixo de *q*, então *q* se encontra na face ilimitada.

O tempo de consulta para a estrutura de dados é bom: foram efetuadas somente duas pesquisas binárias, a primeira em um "array" de comprimento de no máximo $2n$ (as n arestas da subdivisão têm no máximo $2n$ vértices), e a segunda em um "array" de comprimento máximo n (uma "slab" é cruzada por no máximo n arestas). Portanto, o tempo de consulta é $O(\log n)$.

Quais são os requisitos de armazenamento? Antes de tudo, tem-se um "array" para conter as coordenadas *x* dos vértices, que usa espaço de $O(n)$. Mas também se tem um "array" para cada "slab". Esse "array" armazena as arestas que interceptam a "slab", requerendo, assim, espaço para armazenamento de $O(n)$. Uma vez que existem $O(n)$ "slabs", a quantidade total de espaço de armazenamento é $O(n^2)$. O grafo da Figura 2.24 exemplifica o pior caso da divisão em "slabs".

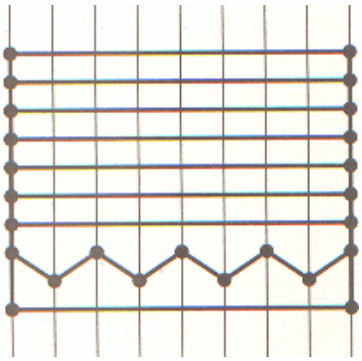


Figura 2.24. Amostra do pior caso da divisão em *slabs* (BERG, 1997).

b) Mapas Trapezoidais

Examinando-se novamente a Figura 2.23, pode-se perceber que os segmentos e as linhas verticais que atravessam os pontos extremos definem uma nova subdivisão, cujas faces são

trapezóides, triângulo e faces ilimitadas semelhantes a trapezóides. Essa nova subdivisão chama-se *mapa trapezoidal*.

Para facilitar a compreensão do tema, foram feitas duas simplificações.

Primeiro, será introduzido um grande retângulo *R* alinhado aos eixos das coordenadas, que contém todo o cenário, isto é, que contém todos os segmentos de *S*. Na verdade, a ausência desse retângulo não é um problema: um ponto de consulta localizado fora dos limites de *R* sempre se encontra na face ilimitada de *S*. O objetivo é restringir a atenção ao que acontece dentro de *R*.

A segunda simplificação é mais difícil de justificar: assume-se que em cada linha vertical não há mais que um segmento com a mesma coordenada *x*. No Subitem b.2 seguinte, são discutidas as providências para tratar os casos que contrariam essa assunção. A conseqüência disso é que não pode haver qualquer segmento vertical. Essa assunção não é realística nem usual: arestas verticais ocorrem freqüentemente em muitas aplicações — como, para exemplificar, em uma malha triangular com inúmeros elementos —, porque a precisão das coordenadas é muitas vezes limitada.

O mapa trapezoidal *T*(*S*) de *S* — também conhecido como *decomposição trapezoidal* de *S* — é obtido por meio de duas extensões verticais partindo de cada ponto extremo *p* de um segmento em *S*, uma extensão em sentido ascendente (extensão vertical superior) e outra em sentido descendente (extensão vertical inferior). As extensões param quando encontram outro segmento de *S* ou a fronteira de *R*. O mapa trapezoidal de *S* é simplesmente a subdivisão induzida por *S*, o retângulo *R* e as extensões verticais superior e inferior. Um exemplo de mapa trapezoidal é mostrado na Figura 2.25.

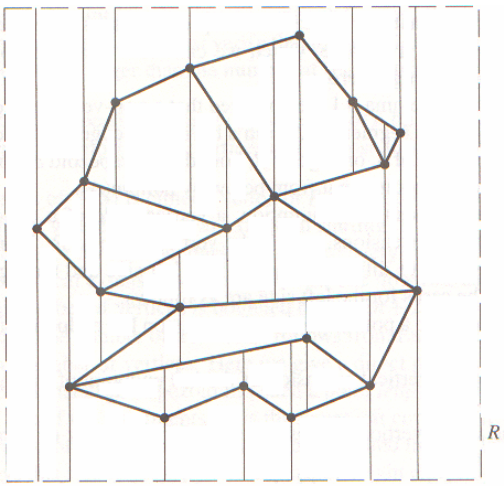


Figura 2.25. Um mapa trapezoidal (BERG, 1997).

Uma face em $T(S)$ é limitada por um número de arestas de $T(S)$. Algumas dessas arestas podem ser adjacentes ou colineares. A quantidade de lados verticais e não-verticais em cada face do trapezóide é estabelecida da seguinte forma: *Cada face em um mapa trapezoidal de um conjunto S de segmentos de linha em posição geral tem um ou dois lados verticais e exatamente dois lados não-verticais* (BERG, 1997).

Observe-se ainda que *f* está toda limitada, o que implica que ela não tem menos que dois lados não-verticais e que deve ter ao menos um lado vertical.

Finalmente, note-se que o mapa trapezoidal é digno de seu nome: cada face é um trapezóide ou um triângulo, que pode ser visto como um trapezóide com uma aresta degenerada de comprimento zero.

Antes de comprovar por que a localização de pontos em um mapa trapezoidal será mais fácil que uma localização de pontos em uma subdivisão em "slabs", será verificado que a complexidade do mapa trapezoidal não é tão maior que o número de segmentos no conjunto que o define. A quantidade máxima de vértices e de trapezóides é medida da seguinte forma: *O mapa trapezoidal* $T(S)$ *de um conjunto S de n segmentos em posição geral contém no máximo* $6n + 4$ *vértices e no máximo* $3n + 1$ *trapezóides* (BERG, 1997).

O mapa trapezoidal pode ser construído incrementalmente. Um trapezóide inicial é adicionado e, em seguida, todos os segmentos da subdivisão poligonal um a um em ordem randômica. À medida que cada segmento é adicionado, o mapa trapezoidal é atualizado.

Para desempenhar a atualização, necessita-se conhecer em qual trapezóide o ponto extremo esquerdo do segmento se encontra. Essa questão é respondida pelo algoritmo de localização planar de pontos. Então, é traçado o segmento de linha da esquerda para a direita, determinando quais trapezóides ele intercepta. Finalmente, retorna-se a esses trapezóides e os fixam para cima. O processo é ilustrado na Figura 2.26.

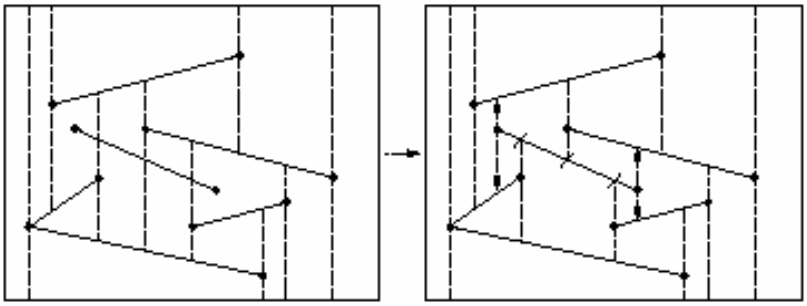


Figura 2.26. Atualização (incremental) de mapa trapezoidal.

b.1) Um Algoritmo Incremental Aleatório

Os algoritmos incrementais aleatórios apresentam uma alternativa atrativa para a localização planar de pontos. Eles constroem o mapa trapezoidal $T(S)$ de um conjunto S de n segmentos. Durante a construção do mapa trapezoidal, é construída também a estrutura de dados D, que pode ser usada para realizar consultas de localização de pontos em $T(S)$.

A estrutura de dados D, chamada *estrutura de pesquisa ou de busca*, é um grafo acíclico direcionado (*DAG – Directed Acyclic Graph*) com uma única raiz. O grafo tem dois tipos de nós: *folhas* e *internos*. Os nós do tipo folha representam cada trapezóide do mapa trapezoidal de S. Seus nós internos têm grau[4] máximo igual a 2. Por esse motivo, sua estrutura é semelhante à estrutura de uma árvore binária. Há dois tipos de nós internos: *nós-x*, que são rotulados com um ponto extremo de algum segmento em S, e *nós-y*, que são rotulados com algum segmento de S.

Uma consulta com um ponto q começa na raiz e procede ao longo de um caminho direcionado rumo a uma das folhas. Essa folha corresponde ao trapezóide $t \in T(S)$ que contém q. Em cada nó no caminho, q tem de ser testado para determinar em qual dos dois nós-filhos prosseguir. Em um *nó-x*, o teste é da seguinte forma: "q está à esquerda ou à direita da linha vertical que atravessa o ponto extremo armazenado neste nó?". Em um *nó-y*, o teste tem a forma: "q está acima ou abaixo do segmento s armazenado aqui?". Os testes em nós internos somente têm dois resultados: *à esquerda* ou *à direita* de um ponto extremo, para um *nó-x*, e *acima* ou *abaixo* para um *nó-y*.

E se ocorrer de um ponto de consulta situar-se exatamente sobre uma linha vertical ou sobre um segmento, qual será a resposta do algoritmo? Na maioria dos textos sobre localização planar de pontos com mapas trapezoidais, não é ao menos aventado esse tipo de situação. Em (BERG, 1997), um ponto de consulta que acarreta tal situação é considerado como pertencente a conjunto de segmentos que não estão em posição geral. No Subitem b.2 adiante, serão tratados especificamente esses casos.

A estrutura de pesquisa D e o mapa trapezoidal T(S), processado por um algoritmo incremental aleatório, estão interligados: um trapezóide $t \in T(S)$ tem um ponteiro para a folha de D correspondente a ele; um nó-folha de D tem um ponteiro para o trapezóide correspondente em $T(S)$. Na Figura 2.27, é exibido o mapa trapezoidal de um conjunto de dois segmentos, s_1 e s_2, e uma estrutura de pesquisa correlata. No *DAG*, os círculos brancos representam os *nós-x*, e os círculos cinzas, os *nós-y*; os quadrados representam os trapezóides.

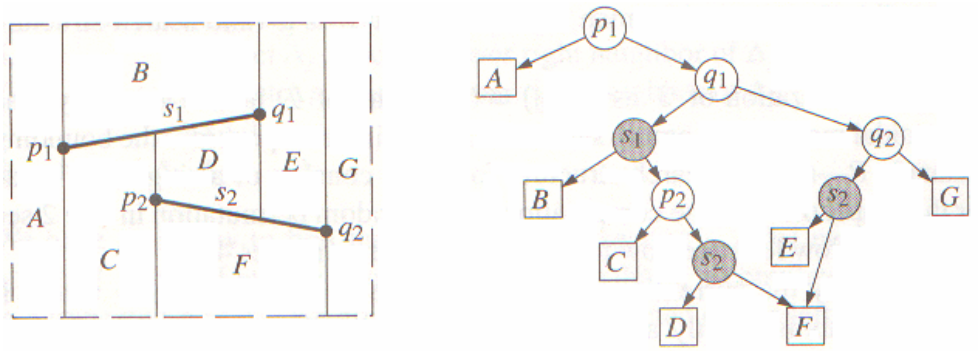


Figura 2.27. Um mapa trapezoidal de dois segmentos e a estrutura de pesquisa correspondente (BERG, 1997).

Nas Figuras 2.32 e 2.33, estão descritos, respectivamente, um algoritmo que computa o mapa trapezoidal e que efetua pesquisa na sua correspondente estrutura de dados acerca da posição em que será inserido um novo segmento. O algoritmo adiciona os elementos um a um e, depois de cada adição, ele atualiza a estrutura de pesquisa e o mapa trapezoidal. Contudo, a ordem em que os segmentos são adicionados influencia a estrutura de busca; algumas ordens levam a uma estrutura de pesquisa com um bom tempo de consulta, enquanto outras produzem uma estrutura de pesquisa com um tempo de consulta ruim. Dito de outra maneira, uma ordem "ruim" pode resultar em uma árvore de altura da ordem de $\Omega(n)$; uma ordem "boa" resultará em uma árvore de altura de $O(\log n)$. E o tempo de pesquisa é proporcional a essa altura. Com a abordagem incremental aleatória, os segmentos são adicionados em ordem randômica, e pode ser provado que a altura esperada da árvore é $O(\log n)$; além do mais, o tempo esperado para construir a estrutura inteira é $O(n \log n)$. Adicionalmente, pode ser também provado que a probabilidade de a altura da árvore exceder a $O(\log n)$ é pequena (O'ROURKE, 1998).

Algoritmo MapaTrapezoidal(S)
Entrada. Um conjunto S de n segmentos de linhas planares.
Saída. O mapa trapezoidal $T(S)$ e uma estrutura de pesquisa D para $T(S)$ em uma caixa delimitadora.

1. Determine uma caixa delimitadora R que contém todos os segmentos de S e inicie a estrutura do mapa trapezoidal T e a estrutura de pesquisa D para ele.
2. Compute uma permutação randômica $s_1, s_2, ..., s_n$ dos elementos de S.
3. **para** $i \leftarrow 1$ **to** n **faça**
4. Encontre o conjunto $t_0, t_1, ..., t_k$ de trapezóides em T adequadamente interceptado por s_i.
5. Remova $t_0, t_1, ..., t_k$ de T e substitua-os pelos novos trapezóides que aparecem por causa da interseção de s_i.
6. Remova as folhas para $t_0, t_1, ..., t_k$ de D e crie folhas para os novos trapezóides. Conecte as novas folhas aos nós internos existentes adicionando alguns novos nós internos.

Figura 2.28. Um algoritmo incremental aleatório (BERG, 1997).

[4] O grau de um vértice é o número de arestas incidentes nesse vértice.

Algoritmo EncontreTrapezoides
Entrada. Um mapa trapezoidal T, uma estrutura de pesquisa D para T e um novo segmento s_i.
Saída. A seqüência t_0, ..., t_k dos trapezóides interceptados por s_i.

1. Seja p e q os pontos extremos esquerdo e direito de s_i.
2. Consulte com p na estrutura de pesquisa D para encontrar t_0.
3. j ← 0;
4. **enquanto** q estiver à direita de *rightp*(t_j) **faça**
5. se *rightp*(t_j) estiver acima de s_i
6. **então** Seja t_{j+1} o vizinho inferior direito de t_j.
7. **senão** Seja t_{j+1} o vizinho superior direito de t_j.
8. j ← j + 1
9. **retorne** t_0, t_1,..., t_j

Figura 2.29. Um algoritmo de pesquisa em mapa trapezoidal (BERG, 1997).

O algoritmo *MapaTrapezoidal* computa o mapa trapezoidal $T(S)$ de um conjunto de n segmentos planares e uma estrutura de busca D para $T(S)$ em tempo esperado de $O(n \log n)$. O tamanho requerido pela estrutura de pesquisa é $O(n)$, e para qualquer ponto de consulta q, o tempo de consulta esperado é $O(\log n)$ (BERG, 1997).

b.2) Lidando com Casos Degenerados

Na explanação do tema sobre mapas trapezoidais e algoritmo incremental aleatório, para simplificar, foram admitidas duas assunções. A primeira é a de que não existem dois pontos extremos distintos com a mesma coordenada x. A segunda é a de que um ponto de consulta nunca se encontra em uma linha vertical de um *nó-x*, nem sobre o segmento de um *nó-y*.

A solução para invalidar a primeira assunção e permitir que haja mais de um (muitos) ponto extremo distinto sobre uma mesma linha vertical é rotacionar ligeiramente o sistema de eixos. Se o ângulo de rotação é suficientemente pequeno, então não haverá mais que um ponto extremo distinto sobre uma única linha vertical. Rotações por meio de ângulos muito pequenos, no entanto, trazem dificuldades numéricas. A melhor abordagem é empregar um mapeamento chamado *transformação shear*, uma espécie de transformação por deformação, descrita na Fórmula 2.1.

$$\varphi: \begin{pmatrix} x \\ y \end{pmatrix} \rightarrow \begin{pmatrix} x + \varepsilon y \\ y \end{pmatrix}$$

Fórmula 2.1. A transformação *shear* (BERG, 1997).

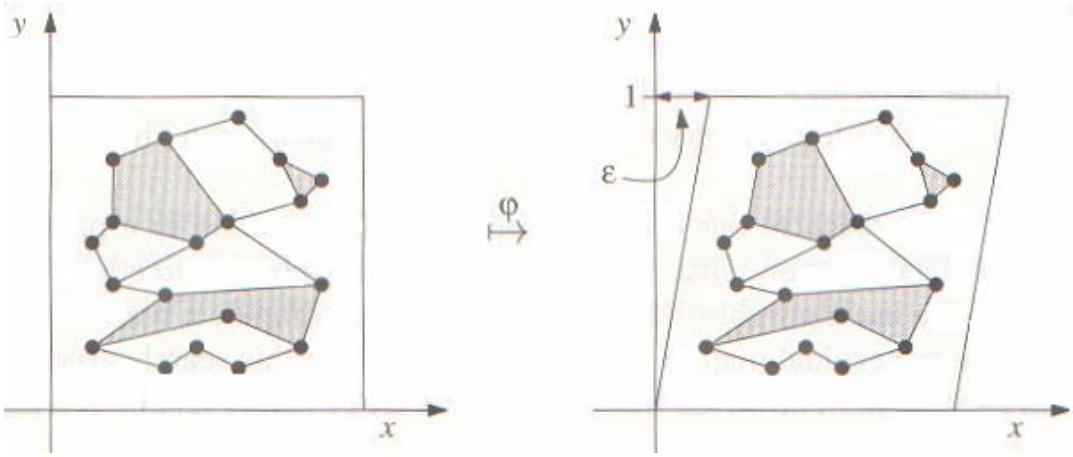


Figura 2.30. Resultado da transformação *shear* (BERG, 1997).

A transformação pode ser aplicada ao longo do eixo *x* mediante algum valor $\varepsilon > 0$. Trata-se, na verdade, de uma espécie perturbação simbólica, em que se aplica uma ordenação lexicográfica pelo coordenada *x*, seguida pela coordenada *y*.

Na Figura 2.30, é ilustrado o efeito da *transformação shear*. Para um dado conjunto *S* de *n* segmentos planares, soluciona-se o problema executando o algoritmo *MapaTrapezoidal* sobre o conjunto $\varphi S := \{\varphi s : s \in S\}$.

O resultado do teste que verificava se um ponto de consulta situa-se sobre um *nó-x*, após a transformação, é: “à direita”, “à esquerda” ou “sobre a linha”. O resultado do teste que verifica se o ponto está sobre um *nó-y* resulta em: “acima”, “abaixo” ou “sobre”.

Particularidades de como a transformação opera estão descritas em (BERG, 1997). O objetivo aqui foi mostrar as estruturas definidas para processar consultas sobre localização planar de pontos, evidenciando tanto suas vantagens quanto suas limitações.

2.7.2 Eficiência dos Algoritmos de Localização Planar

O primeiro algoritmo para o problema de localização de pontos foi proposto por Dobkin e Lipton (EDAHIRO; KOKUBO; ASANO, 1984). Seu algoritmo tem tempo de pesquisa de $O(\log n)$, espaço de $O(n^2)$ e tempo de pré-processamento de $O(n^2 \log n)$. Desde então, vários algoritmos foram propostos.

A eficiência dos algoritmos de localização planar é analisada sob três aspectos:

1. *Tempo de pré-processamento*. O tempo requerido para construir a estrutura de pesquisa a partir de uma representação padrão de um grafo;
2. *Espaço*. O armazenamento usado para construir e representar a estrutura de pesquisa;

3. *Tempo de pesquisa*. O tempo requerido para localizar um ponto na estrutura de pesquisa.

Como denotado na Tabela 2.3, extraída de (EDAHIRO; KOKUBO; ASANO, 1984), à exceção dos algoritmos de localização planar de pontos de Litpton-Tarjan e Kirkpatrick, todos os demais apresentam variações entre si na quantidade de espaço e tempo requeridos. Alguns são mais simples, outros possuem complicações algorítmicas e práticas. Entretanto, em comum, são orientados pelas coordenadas dos vértices e particionam o plano em regiões através de linhas verticais e/ou horizontais e as representam hierarquicamente em estruturas do tipo árvore.

Tabela 2.3. Eficiências teóricas dos algoritmos de localização de pontos.

Algoritmo	**Tempo de pré-processamento**	**Espaço**	**Tempo de pesquisa**
Dobkin-Lipton	$O(n^2 \log n)$	$O(n^2)$	$O(\log n)$
Shamos	$O(n^2)$	$O(n^2)$	$O(\log n)$
Lee-Preparata	$O(n \log n)$	$O(n)$	$O((\log n)^2)$
Lipton-Tarjan	$O(n \log n)$	$O(n)$	$O(\log n)$
Preparata	$O(n \log n)$	$O(n \log n)$	$O(\log n)$
Kirkpatrick	$O(n \log n)$	$O(n)$	$O(\log n)$
Método *Bucket*			
Caso médio	$O(n)$	$O(n)$	$O(1)$
Pior caso	$O(n\sqrt{n})$	$O(n\sqrt{n})$	$O(n)$

Outros métodos diferentes daqueles baseados em mapas trapezoidais foram propostos. Mucke *et al.* (MUCKE; SAIAS; SHU, 1996), por exemplo, apresentaram um procedimento para encontrar o ponto de consulta, no plano 2D e 3D, simplesmente caminhando através da triangulação, após selecionar um “bom ponto de partida” por amostragem randômica. Em uma triangulação de Delaunay de n pontos aleatórios, a medida de eficiência esperada do método é próxima a $O(n^{1/(d+1)})$, onde d é a dimensão planar. Entretanto, diante dos resultados práticos obtidos, concluíram que, em triangulações com quantidade significativa de vértices (por exemplo, acima de um milhão), um algoritmo de eficiência assintótica superior, mesmo requerendo pré-processamento e estruturas de dados adicionais, seria a melhor solução para o problema de localização planar de pontos.

2.8 Refinamento da Malha

Uma das mais úteis técnicas de refinamento ou melhoramento, datada de 1960, é chamada *uniformização Laplaciana*. O nome se deve à sua fórmula de reposicionamento que

pode ser derivada de uma aproximação de diferenças finitas da equação de Laplace. Na uniformização Laplaciana, um vértice *v*, no interior de uma malha, é movido para o centróide (centro de massa) de seus vizinhos, como é exibido na Figura 2.31.

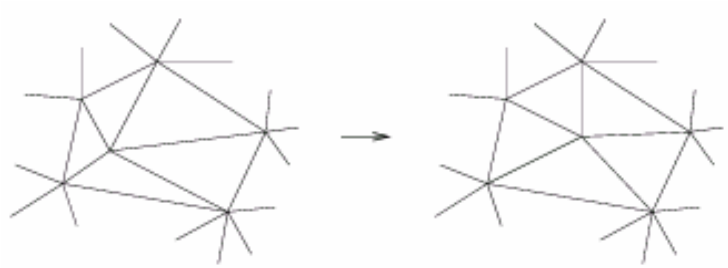


Figura 2.31. Movendo um vértice para o centro de massa de seus vizinhos (BERN; EPPSTEIN, 1992).

A uniformização Laplaciana é aplicada sucessivamente a cada nó interior da malha, por vários ciclos.

Uma segunda técnica, chamada *relaxação da malha*, movimenta as arestas para regularizar graus angulares em triângulos.

O problema de refinar uma dada malha ocorre com bastante freqüência na prática, por exemplo, quando uma computação inicial de elementos finitos revela uma região que requer maior resolução (BERN; EPPSTEIN, 1992).

Nos algoritmos de refinamento de Chew (CHEW, 1989) e de Ruppert (RUPPERT, 1995), a idéia-chave do refinamento da malha é a inserção de um vértice no centróide de um triângulo de "qualidade precária", mantendo-se a propriedade de Delaunay.

Shewchuk, em seu programa *Triangle* (SHEWCHUK, 1996b), implementou uma medida para avaliar a qualidade de um triângulo, chamada *proporção da aresta l ao raio do círculo-circundante r*. O *circuncentro* e o *circunraio* de um triângulo são, respectivamente, o centro e o raio de seu círculo-circundante. O quociente do circunraio de triângulo *r* e o comprimento *l* de sua menor aresta são a métrica que naturalmente é otimizada pelos algoritmos de refinamento.

A proporção da aresta *l* ao raio do círculo-circundante *r* de um triângulo *r/l* está relacionada ao seu menor ângulo θ_{min} por meio da fórmula $r/l = 1/(2\text{sen}\,\theta_{min})$. Se *B* é um limite superior da proporção da aresta ao raio do círculo-circundante de todos os triângulos em uma malha, então não haverá, após o refinamento, qualquer ângulo menor que arc sen de *1/2B*.

O algoritmo de Ruppert (RUPPERT, 1995) emprega um limite de $B = \sqrt{2}$; o segundo algoritmo de refinamento de Chew emprega um limite de $B = 1$. O primeiro algoritmo de refinamento de Chew (CHEW, 1989) divide todo triângulo cujo circunraio é maior que o

comprimento da menor aresta na malha inteira, alcançando assim um limite de $B = 1$, mas força todos os triângulos a terem tamanho uniforme.

A noção de "qualidade precária" também inclui aqueles triângulos com alguma aresta com comprimento muito díspar do valor de *B*. Na Figura 2.32, *v* é o centróide do circuncentro do triângulo *t*. O algoritmo de refinamento constatou que *t* tem uma *proporção da aresta ao raio do círculo-circundante* maior que *B* e refez a triangulação de Delaunay levando em conta o vértice *v*.

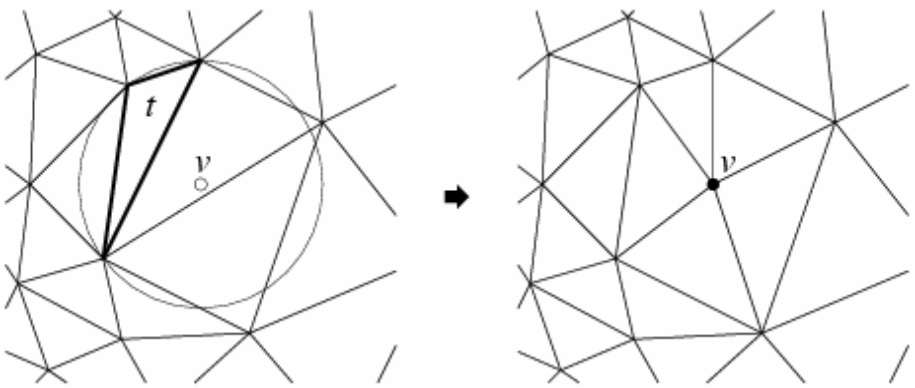


Figura 2.32. Refinamento de uma região da malha com triângulo de qualidade insatisfatória. O vértice *v* é inserido, o triângulo *t* é eliminado, mas a propriedade de Delaunay é mantida. (SHEWCHUK, 1997).

Uma vez que um triângulo de Delaunay não tem qualquer vértice dentro de seu círculo-circundante, uma triangulação de Delaunay é uma estrutura ideal para encontrar pontos que estão distantes de outros vértices. Daí, conclusivamente, pode-se afirmar que uma malha triangular hiperótima é aquela formada por triângulo equiláteros, cujo diagrama de Voronoi correspondente é formado por hexágonos (pois o centróide de cada triângulo da triangulação de Delaunay é um vértice do correspondente diagrama de Voronoi).

Capítulo III

Implementação de Triangulação de Delaunay por Divisão-e-Conquista em OCaml

3.1 Introdução

Com a expectativa de que o compilador OCaml (LEROY, 2003) proporcionasse algum benefício, como em alguns experimentos relatados em (THE COMPUTER LANGUAGE SHOOTOUT BENCHMARKS, 2006), em que o código escrito em OCaml foi mais eficiente, implementou-se o algoritmo de Guibas-Stofi para a triangulação de Delaunay baseado em divisão-e-conquista — o algoritmo mais rápido e eficiente em termos de requisitos de tempo $O(n \log n)$ e espaço computacionais $O(n)$.

Foi selecionado o código implementado em C por Geoff Leach (LEACH, 1992), o qual, com melhorias nos predicados *InCircle* e *Valid*, acelerou-se o algoritmo de Guibas-Stolfi em quatro a cinco vezes. Os resultados são apresentados neste capítulo e em (MOURA *et al.*, 2005).

3.2 O Algoritmo de Divisão-e-Conquista modificado por Dwyer

Foi apresentada, em (DWYER, 1986), uma modificação no algoritmo de Guibas-Stofi para computar a triangulação de Delaunay de n pontos no plano. Usando como entrada pontos distribuídos uniformemente no quadrado unitário, como exemplificado na Figura 3.1, a mudança reduziu o tempo esperado para o pior caso de $O(n \log n)$ para $O(n \log \log n)$. Quanto aos resultados com distribuições não-uniformes, o autor disse que nada se pode afirmar sem um estudo adicional.

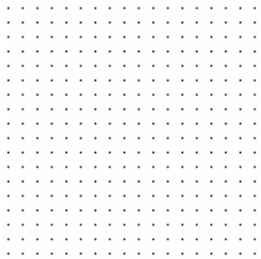


Figura 3.1. Exemplo de distribuição uniforme de pontos no quadrado unitário.

3.3 Melhoramentos efetuados no Algoritmo de Guibas-Stolfi

Leach incorporou alguns melhoramentos ao algoritmo de divisão-e-conquista de Guibas-Stofi. Seus esforços foram dirigidos a testes de mensuração de tempo usando um milhão de pontos distribuídos aleatoriamente no quadrado unitário e com estatística detalhada provida pelo utilitário *gprof* do sistema operacional UNIX. Foram feitos melhoramentos na estrutura de dados. A estrutura de dados original utilizada por Guibas-Stolfi, *quad-edge*, foi substituída pela estrutura de dados *winged-edge*. Os demais melhoramentos enfocaram os predicados "InCircle" e "Valid". O predicado *InCircle* (*A, B, C, D*) define se o ponto *D* está no interior, fora ou em cima do círculo formado pelos pontos *ABC*. O predicado *Valid* (*e*) testa se a aresta *e* está acima ou não de uma determinada aresta. Os melhoramentos nesses predicados aumentaram a velocidade do algoritmo original em quatro a cinco vezes. Informações detalhadas podem ser obtidas em (LEACH, 1992).

3.4 A Linguagem Objective Caml

Objective Caml (OCaml) é uma linguagens de programação funcional projetada e implementada por Gérard Huet, no INRIA, na França, por volta de 1994. Desde então, seu desenvolvimento tem sido continuado como parte do *projeto Cristal*, agora conduzido por Xavier Leroy. É uma descendente distante de Lisp, que incorpora características principais de outras linguagens. OCaml — um acrônimo para "Objective Categorical Abstract Machine Language — é derivada da linguagem ML (*Meta-Language*) clássica projetada, em 1975, na Universidade de Edimburgo, por Robin Milner, para o provador de teoremas LCF (*Logic of Computable Functions*).

As linguagens ML são semifuncionais: o estilo de programação é funcional, mas inclui atribuição de valor a identificadores e efeitos laterais. Na programação funcional, um programa é uma função aplicada a seus argumentos. Ele computa um resultado que é retornado, quando a computação termina, como saída do programa. Desse modo, a combinação de programas acontece por composição de funções (DIVERIO, 1999), ou seja, a saída de um programa torna um argumento de entrada para outro. A seguir, são descritos alguns aspectos relevantes da linguagem:

- *Verificação estática de tipo.* O compilador desempenha checagens de compatibilidade entre os tipos de parâmetros formais e reais em tempo de compilação;

- *Inferência de tipo.* O programador não necessita dar qualquer informação de tipo. Essa inferência ocorre juntamente com a verificação durante a compilação do programa. Na Figura 3.2 adiante, é apresentado o código de uma função que implementa recursivamente a operação de multiplicação de números inteiros. A palavra "val" indica que a função fora avaliada pelo compilador e sua assinatura inferida em termos de dois argumentos inteiros como entrada e um inteiro como saída;
- *Sistema de tipo polimórfico.* É possível escrever programas que trabalham para valores de qualquer tipo, sejam básicos como números inteiros e na representação ponto-flutuante, booleanos, caractere e strings ou sofisticados como tuplas, arranjos, listas, conjuntos, filas, pilhas, etc.;
- *Gerenciamento automático de memória.* A alocação e desalocação de estrutura de dados são controladas de forma implícita e paralela pelo compilador. Tudo isso é levado a efeito sem o emprego de primitivas comuns na linguagem C ("new", "malloc" e "free") e sem a necessidade de paradas no programa enquanto o mecanismo de coleta de lixo ("garbage collection") está sendo executado;
- *Três modos de execução.* No modo *interativo*, o usuário escreve expressões em OCaml que começam com o símbolo (#) e são terminadas com um ponto-e-vírgula duplo (;;). O interpretador avalia a expressão e imprime sua assinatura, como apresentado na Figura 3.2. O modo *compilável para bytecodes* produz um código executável independente da arquitetura da máquina real onde ele roda. O modo *compilável para código nativo* gera um código de alto desempenho, mas apenas para a arquitetura em que foi compilado.

```
        Objective Caml version 3.07
# let rec mult m n =
    if n = 0 then 0
    else m + mult m (n-1);;
val mult : int -> int -> int = <fun>
```

Figura 3.2. Função de multiplicação em OCaml.

3.5 Detalhes da Implementação

A eficiência de algoritmos fundamentalmente recursivos, como, por exemplo, aqueles que implementam métodos de ordenação e busca (*mergesort*, *árvore binária*) deve-se, em

parte, à estrutura empregada para representar os dados. Em linguagens de paradigma imperativo e orientado a objeto, é usual a utilização de ponteiros visando a obter acesso rápido a elementos da estrutura de dados em questão. Todavia, em linguagens funcionais, mesmo aquelas que admitem o estilo de programação imperativo, como OCaml, não é natural o uso de ponteiros, pois tendem a acarretar sérios efeitos colaterais. A recomendação é que o algoritmo do problema a ser resolvido seja concebido abstratamente e seja implementado utilizando estilo funcional. De certo modo, isso implica ajustar todas as instruções e identificadores a operações típicas de *listas*. Para problemas de pouca ou moderada complexidade, a tarefa poderá ser até interessante. Todavia, com problemas de alta complexidade, em geral com especificação envolvendo longos trechos de instruções, a tarefa, além de exaustiva, poderá ter desfecho frustrante.

No caso da reescrita do programa de triangulação por divisão-e-conquista baseada no importante algoritmo de Guibas-Stolfi, insiste-se em manter o estilo de programação imperativo, por algumas razões: (i) assegurar a correspondência morfológica entre os códigos para facilitar a depuração em caso de erros; (ii) esta pesquisa não tem por meta utilizar o estilo de programação funcional; (iii) além de avaliar o desempenho provido por OCaml, é de interesse saber como representar nessa linguagem a estrutura de ponteiros de modo simples, seguro e com a mesma versatilidade que em linguagens imperativas como C e Pascal.

3.5.1 Estrutura de Dados

No programa original, escrito em C, que usa alocação dinâmica de memória, são definidas as estruturas *"point"* e *"edge"*, conforme descritas na Figura 3.3 adiante. O identificador *"p_array"* é uma lista contendo os pontos de entrada para a triangulação; *"free_list_e"* controla a alocação das arestas à medida que são criadas ou removidas. Preferiu-se manter os tipos de dados do programa original, utilizando, por exemplo, o tipo *float* em vez de *double*.

```
struct point {
   float  x, y;
   edge *entry_pt;
};

point *p_array;
```

```
struct edge {
   point   *org;
   point   *dest;
   edge    *onext;
   edge    *oprev;
   edge    *dnext;
   edge    *dprev;
};
edge **free_list_e;
```

Figura 3.3. Estrutura de dados da triangulação em C.

```
type pedge = edge refer pointer refer          (* *edge *)
and ppoint = point refer pointer refer         (* *point *)
and point = {
  mutable x: float;
  mutable y: float;
  mutable entry_pt: pedge;
  mutable num: int
}
and edge = {
  mutable e_org: ppoint;
  mutable e_dest: ppoint;
  mutable e_onext: pedge;
  mutable e_oprev: pedge;
  mutable e_dnext: pedge;
  mutable e_dprev: pedge
};;
let p_array = new_refer (Array.init 1 (fun x -> (new_refer (new_pointer (new_refer ({x=0.0; y=0.0;
entry_pt=new_refer (new_pointer Nil); num=(-1)})))))));;
let free_list_e = Array.init 1 (fun x -> new_refer (new_pointer (new_refer {e_org=new_refer (new_pointer
Nil); e_dest=new_refer (new_pointer Nil); e_onext=new_refer (new_pointer Nil); e_oprev=new_refer
(new_pointer Nil); e_dnext=new_refer (new_pointer Nil); e_dprev=new_refer (new_pointer Nil)})));;
```

Figura 3.4. Estrutura de dados da triangulação em OCaml.

Antes de derivar a estrutura da Figura 3.4 e reescrever o programa em OCaml, foram efetuados inúmeros testes objetivando obter o tipo de dado (OCaml) em que pudessem ser realizadas todas as operações típicas de um ponteiro. A Tabela 3.1 apresenta a correspondência entre ponteiros em C e em OCaml com base em extensão feita aos tipos definidos em (INRIA, 2004).

Tabela 3.1. Ponteiros explícitos em C traduzidos para OCaml.

Instruções em C	Instruções em OCaml
int k;	let k = ref 0
int m;	and m = ref 0
int *ptr;	and ptr = ref (new_pointer (ref 0))
int *ptr2;	and ptr2 = ref (new_pointer (ref 0))
int *p;	and p = ref (new_pointer (ref 0))
int **q;	and q = ref (new_pointer (ref (new_pointer (ref 0)))) in
ptr = &k;	ptr := new_pointer k;
ptr2 = &k;	ptr2 := new_pointer k;
k = 1;	k := 1;
m = 2;	m := 2;
ptr2 = &m;	ptr2 := new_pointer m;
*ptr = 7;	!^(!ptr) := 7;
ptr2 = ptr;	ptr2 := !ptr;
*ptr2 = 8;	!^(!ptr) := 8;
q = &p;	q := new_pointer p;
**q = 100;	(!^(!(!^(!q)))) := 100;

A versão em OCaml do programa de triangulação nos termos dos exemplos apresentados na Tabela 3.1 forneceu os mesmos resultados que a versão em C, fazendo-se ressalva a alguns casos de falta de robustez numérica, em que o programa escrito na linguagem funcional foi mais estável. Contudo, a codificação fora custosa, como se depreende do código-fonte da função *delete_edge* escrita em C (Figura 3.5) e em OCaml (para acomodar a constante *Nil*, a representação de ponteiros foi alterada de forma tal que os operadores “!” e “:=” foram respectivamente substituídos por “!!” e “^=”), conforme Figura 3.6.

```
/* Remove an edge.*/
void delete_edge(edge *e)
{
  point *u, *v;
  /* Cache origin and destination. */
  u = Org(e);
  v = Dest(e);
        ...
  if (Org(e->dprev) == v)
    e->dprev->onext = e->dnext;
  else
    e->dprev->dnext = e->dnext;
  free_edge(e);
}
```

Figura 3.5. Trecho de código da função *delete_edge* em C.

```
(* Remove an edge. *)
let delete_edge e =
  let u = new_refer (new_pointer(new_refer ({x=0.0; y=0.0; entry_pt=new_refer (new_pointer Nil);
num=(-1)}) ))
  and v = new_refer (new_pointer(new_refer ({x=0.0; y=0.0; entry_pt=new_refer (new_pointer Nil);
num=(-1)}) )) in
  (* Cache origin and destination. *)
  u ^= !!(org e);
  v ^= !!(dest e);
        ...
if ( !!(org (!!(!^(!!e)).e_dprev)) ) = !!v then
  (!!(!^(!! ( (!!(!^(!!e)).e_dprev) )))).e_onext ^= !!(!!(!^(!!e)).e_dnext)
else
  (!!(!^(!! ( (!!(!^(!!e)).e_dprev) )))).e_dnext ^= !!(!!(!^(!!e)).e_dnext);
  free_edge e;;
(*val delete_edge : pedge -> unit = <fun>*)
```

Figura 3.6. Trecho de código da função *delete_edge* em OCaml baseado em proposta do INRIA (INRIA, 2004).

Portanto, em virtude da prolixidade notacional do programa em OCaml, as estruturas de *point* e *edge* exibidas anteriormente na Figura 3.4 foram modificadas para a forma exibida na Figura 3.7, e alterado o código de todas as funções, como exemplificado na Figura 3.8.

```
type t_point = {
  mutable x: float;
  mutable y: float;
  mutable entry_pt: int;
  mutable num: int
}
and t_edge = {
  mutable org: int;
  mutable dest: int;
  mutable onext: int;
  mutable oprev: int;
  mutable dnext: int;
  mutable dprev: int;
  mutable e_num: int
};;
let p_array = ref [||];;
let e_array = ref [||];;
let free_list_e = ref [||];;
```

Figura 3.7. Estrutura de dados (simplificada) da triangulação em OCaml.

```
(* Remove an edge. *)
let delete_edge e =
  let u, v = ref 0, ref 0 in
  (* Cache origin and destination. *)
  u := org e;
  v := dest e;
       ...
  if (org(!e_array.(e).dprev) = !v) then
     !e_array.(!e_array.(e).dprev).onext <- !e_array.(e).dnext
  else
     !e_array.(!e_array.(e).dprev).dnext <- !e_array.(e).dnext;
  free_edge e;;
(*val delete_edge : int -> unit = <fun>*)
```

Figura 3.8. Trecho de código (simplificado) da função *delete_edge* em OCaml.

3.6 Resultados dos Experimentos

A Tabela 3.2 apresenta resultados dos experimentos realizados. Na implementação, foi utilizado o compilador *ocamlopt* constante de OCaml versão 3.07 e o IDE da Borland C++ Builder 1.0, a versão que estava disponível. Os dados dos polígonos e nuvens de pontos foram lidos de arquivos-texto e depois ordenados pelo eixo *x* usando o conhecido método *mergesort*.

Vale ressaltar que todos os resultados não devem ser analisados sob o mesmo aspecto. No caso do conteúdo dos arquivos *b10000.node*, *b100000.node* e *b1000000.node*, é importante considerar aspectos referentes a tempo de processamento. A versão escrita em OCaml foi menos eficiente em todos os três casos (dez mil, cem mil e um milhão de pontos). Porém, no caso do conteúdo de *torre.poly*, *circulo.poly* e *boca.poly* já é interessante avaliar aspectos relacionados à exatidão. Por exemplo, na triangulação dos pontos contidos em *circulo.poly* e *boca.poly*, os resultados foram divergentes. Rastreando os códigos-fontes (em C e OCaml), foi constatado que, na função *merge*, os valores para o identificador *cot_r_cand*, em um mesmo instante da iteração, estavam diferentes, como demonstrado na Tabela 3.3 adiante. Ocorreu que a versão escrita em C falhou no cálculo dos produtos escalar e vetorial e, usando a precisão-padrão de seis casas decimais, tornou o valor de *cot_r_cand* inferior ao de *cot_l_cand*. Posteriormente, em uma instrução do tipo *if-then-else* que avaliava se o valor de *cot_r_cand* era menor que o de *cot_l_cand*, o programa escrito em C seguiu um caminho diferente do programa escrito em OCaml. Fatos semelhantes se repetiram em outras iterações. Em razão disso, houve divergência[5] na triangulação do *círculo* e da *boca*, conforme visualmente apresentado na Figura 3.9.

[5] A aferição da qualidade da triangulação foi fundamentada em resultados de experimentos correspondentes produzidos pelo programa Triangle (SHEWCHUK, 1996b).

Tabela 3.2. Resultados da triangulação por divisão-e-conquista utilizando um PC com processador Pentium III de 1.13 GHz, 128 Mb de memória RAM e 256 Kb de memória Cache, sob o sistema operacional Windows 98.

Dados de entrada		**Tempos de processamento (em segundos)**					
Nome do arquivo	**Número de pontos**	**Ordenação**		**Triangulação**		**Plotagem**	
		C	**OCaml**	**C**	**OCaml**	**C**	**OCaml**
b10000.node	10.000	0.06	0.00	**0.05**	**0.16**	0.50	0.99
b100000. node	100.000	0.11	0.39	**1.10**	**2.58**	5.11	9.45
b1000000. node	1.000.000	2.14	7.58	**16.26**	**98.92**	56.63	113.36

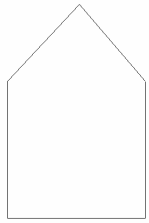
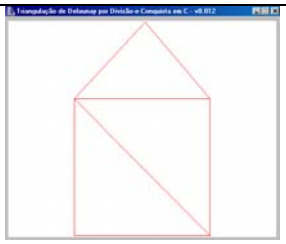
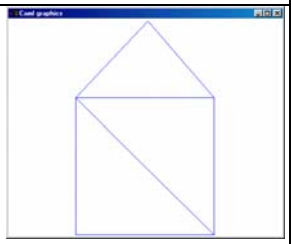
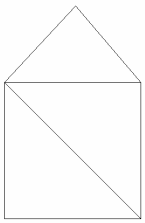
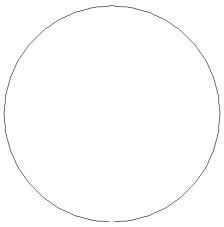
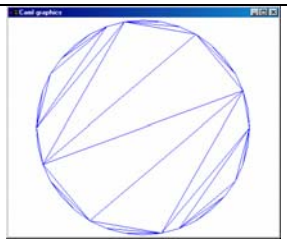
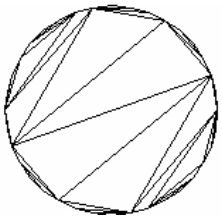
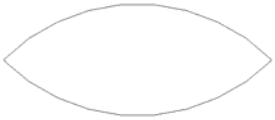
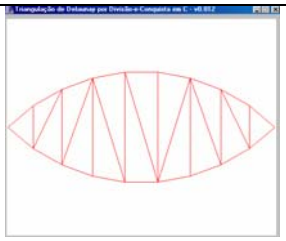
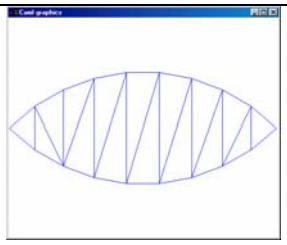
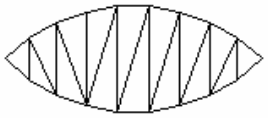
Desenho do PSLG	**Versão em C**	**Versão em OCaml**	**Programa Triangle**
			
torre.poly			
			
circulo.poly			
			
boca.poly			

Figura 3.9. Resultados visuais de triangulações de PSLG.

3.6.1 Amostra da Robustez Numérica de OCaml

A atividade de achar algoritmos que produzem melhor precisão com menos cálculo é uma área de pesquisa da matemática atual denominada de *análise numérica*. Entretanto, a Tabela 3.3 expressa resultados de operações matemáticas, implementadas em C e em OCaml, sem o

incremento de qualquer técnica para melhorar a precisão. Os seis últimos casos ilustram uma das ocorrências causadoras da disparidade ocorrida durante a execução do procedimento *merge* das versões em C e em OCaml do programa de triangulação.

Tabela 3.3. Resultados de operações matemáticas envolvendo números na representação ponto-flutuante.

Operação matemática	Resultado em C	Resultado em Ocaml
$\sqrt{2}$	1,4142135381698608	1,4142135623730951 **(*)**
$(\sqrt{2})^2$	1,9999999315429164	2,0000000000000004 **(*)**
Pi=acos(-1)	3,1415926535897931	3,1415926535897931
e = (1 + (1/x)) ^ x, para x=1000000	2,7182804690957534	2,7182804690957534
E ^ 0,01	1,0100501620330897	1,0100501620330897
D_p_l_cand = dot_product_2v (0,0000000000000000, -0,1204265073161450, 0,0811250971963144, -0,1672641040204717)	0,0201430507004261	0,0201430318465498 **(*)**
D_p_r_cand = dot_product_2v (-0,0811250971963144, -0,1672641040204719, 0,0000000000000000, -0,2141017007247985)	0,0358115434646606	0,0358115291409926 **(*)**
c_p_l_cand = cross_product_2v (0,0000000000000000, -0,1204265073161450, 0,0811250971963144, -0,1672641040204717)	0,0097696194425225	0,0097696121110349 **(*)**
c_p_r_cand = cross_product_2v (-0,0811250971963144, -0,1672641040204719, 0,0000000000000000, -0,2141017007247985)	0,0173690374940634	0,0173690212811955 **(*)**
cot_r_cand = d_p_r_cand / c_p_r_cand	2,0618035793304443	2,0618046671267440
cot_l_cand = d_p_l_cand / c_p_l_cand	2,0618050098419189	2,0618046671267414

(*) Resultado mais preciso

3.7 Avaliação da Possibilidade de Paralelização

A disponibilidade de máquinas com múltiplos processadores e com grande quantidade de memória pode ser um aliado para acelerar a triangulação e a geração de malhas. Com tal motivação, muitos esforços têm sido orientados para o desenvolvimento de ambientes, geradores de malhas e trianguladores utilizando os recursos do processamento paralelo. Em princípio, a essência desses esforços é converter algoritmos seqüenciais já existentes em algoritmos paralelos.

Na geração de malhas, o processamento paralelo é recomendado para melhorar a eficiência tanto da fase de processamento quanto da fase de pré-processamento. No último caso, aplica-se, em específico, à triangulação e ao refinamento da Malha.

Ressalte-se que a eficiência do processamento paralelo depende da capacidade computacional da máquina, incluindo a quantidade de processadores, que é em potência de 2; depende também de um ambiente para criar e gerenciar o processamento paralelo e de um algoritmo e programa que farão a decomposição do problema em partes, que serão processadas separadamente.

De modo geral, o processamento paralelo, é descrito como segue:

1. Decomposição do domínio. Consiste em decompor o domínio geométrico em *p* partes iguais (subdomínios), correspondentes ao número *p* de processadores disponíveis.

2. Atribuir cada subdomínio a um processador. Cada processador produzirá uma subsolução.

3. Reunir as subsoluções. É um passo complexo que exige coordenação, também conhecido como "Merge", que reúne cada subsolução gerada por cada processador e produz a solução final.

Uma estratégia de paralelização típica é descrita na Figura 3.10. Na ilustração, uma malha de elementos finitos é usada como entrada. Depois, essa malha é particionada em um número de subdomínios que serão distribuídos entre os processadores e cada um dos processadores fará a computação do subdomínio separadamente. Ocorre comunicação entre os processadores. Os ciclos de comunicação e de computação se encerram quando a solução obtida atende aos requisitos de qualidade desejados. Os custos com computação e comunicação e também a forma como os subdomínios foram definidos são exemplos de fatores que podem impactar sobre o desempenho do processamento paralelo.

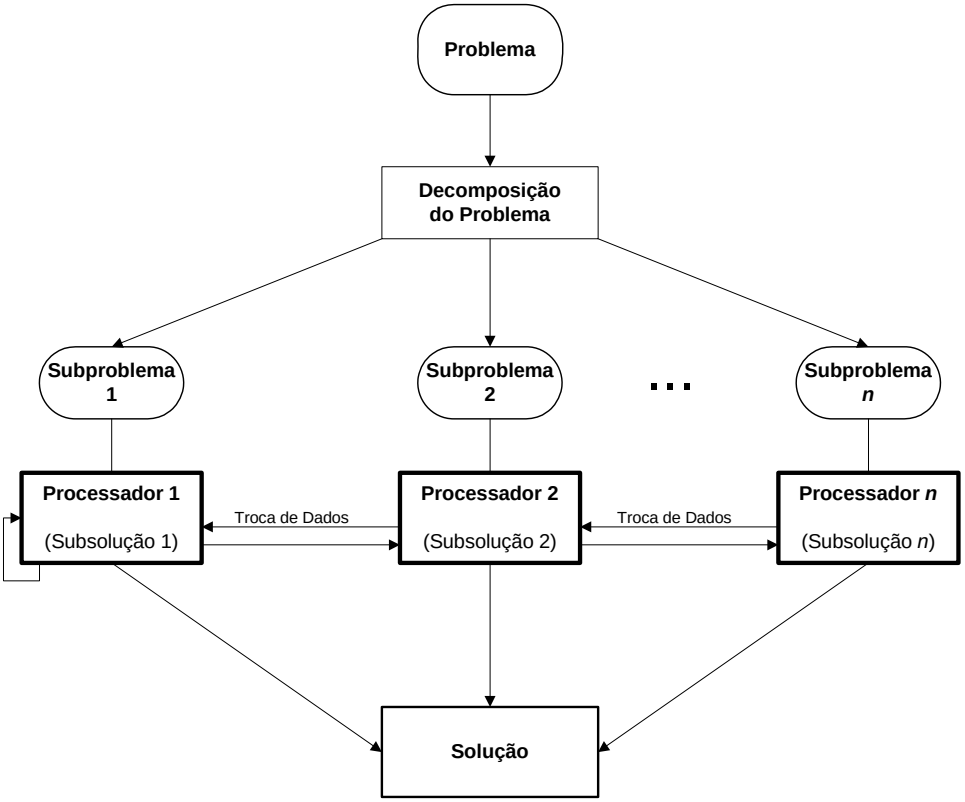


Figura 3.10. Uma estratégia típica de paralelização.

A maneira de particionar um domínio e de distribuir suas partes entre os processadores de um computador constitui uma poderosa ferramenta que pode ajudar a acelerar a solução do problema. As técnicas de partição podem ser classificadas em duas categorias:

- *Técnicas estáticas*: A partição é realizada somente uma vez e antes da execução do procedimento de solução do problema.
- *Técnicas dinâmicas*: Usam algoritmos de balanceamento de cargas, que periodicamente determinam uma nova partição do problema.

Em se tratando da triangulação de Delaunay, sua maior desvantagem é a quantidade de tempo requerido para obter a triangulação sobre um conjunto de pontos. Esse tempo pode ser reduzido com o uso de mais de um processador. Vários algoritmos têm sido propostos (LEE, PARK, PARK; 2001), (CHEN, CHUANG, WU; 2002), (HARDWICK, 1997), (CHERNIKOV, CHRISOCHOIDES; 2006), (MERRIAM; 1992). Seguem adiante as estratégias gerais empregadas comumente em algoritmos paralelos de triangulação de Delaunay.

Vários algoritmos paralelos para a triangulação de Delaunay têm sido propostos, cada qual com métodos específicos de triangulação e de partição do domínio geométrico ou do conjunto de pontos a ser triangulado. Por exemplo, em (CHEN, CHUANG, WU; 2002), é apresentado um algoritmo paralelo para triangulação de Delaunay por divisão-e-conquista; em (CHERNIKOV, CHRISOCHOIDES; 2006), é apresentado um algoritmo paralelo para triangulação de Delaunay restrita; em (LEE, PARK, PARK; 2001), é proposto um algoritmo paralelo melhorado para a triangulação de Delaunay utilizando a abordagem de construção incremental; em (MERRIAM; 1992), são discutidos aspectos teóricos e práticos referentes à implementação de um algoritmo paralelo para triangulação de Delaunay baseado na abordagem *avanço de fronteira*.

Quanto à eficiência obtida pelo processamento paralelo, os algoritmos que operam sobre domínios geométricos 2D apresentam maior eficiência que aqueles que operam sobre domínios 3D. Detalhes podem ser obtidos em (KOHOUT; 2004) e (HARDWICK, 1997).

Em um algoritmo paralelo de triangulação de Delaunay, os pontos de entrada são subdivididos de acordo com suas coordenadas em k áreas retangulares (onde k é o número de processadores). Cada processador responderá exclusivamente pela triangulação dos pontos que estiverem na área que lhe foi designada. Os pontos são distribuídos a cada processadores em quantidades quase iguais. Para desempenhar sua tarefa, o processador precisa conter também os pontos que se encontram em áreas adjacentes à área atribuída a este processador.

Existem no máximo quatro áreas adjacentes. Os pontos localizados em áreas adjacentes, próximo à fronteira delas com a área de cada processador, servem como uma “interface”. Uma mesma interface pode aparecer em mais de um processador. Dessa forma, é possível identificar os triângulos construídos redundantemente por mais de um processador.

É necessária uma etapa que reúna os resultados parciais gerados por cada processador. A etapa de junção tem de levar em conta que cada processador gera muitos triângulos idênticos, que se sobrepõem. Assim, cada triângulo redundante deve ser considerado apenas no processador em que aparecer primeiro.

Todo procedimento do algoritmo, exceto o da etapa que reúne os resultados parciais, pode ser processado paralelamente.

Na Figura 3.11, está ilustrado o cenário de uma triangulação de Delaunay de um conjunto de pontos em máquina com um único processador. Na Figura 3.12, é apresentado o cenário dessa mesma triangulação em uma máquina com quatro processadores. O exemplo apresentado pode ser visto como o pior caso do processamento paralelo utilizando quatro processadores. Houve muito desperdício de esforço computacional construindo triângulos redundantes. Na Figura 3.13, é ilustrado como os resultados parciais são computados em cada um dos processadores. Os triângulos em cor cinza são aqueles que foram construídos redundantemente em mais de um processador.

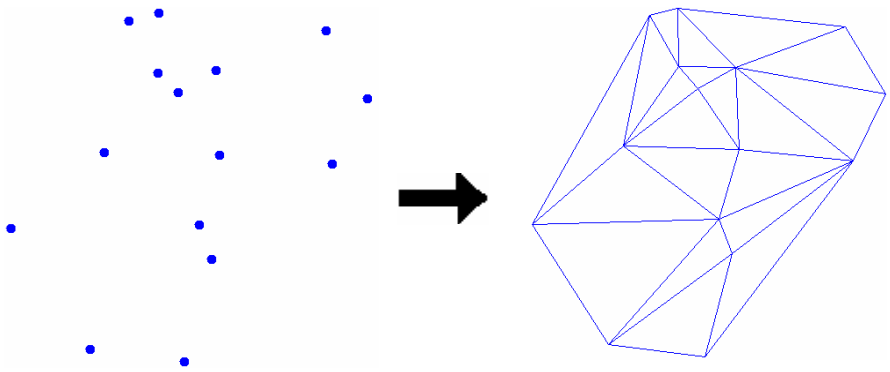


Figura 3.11. Triangulação de Delaunay seqüencial de um conjunto de pontos.

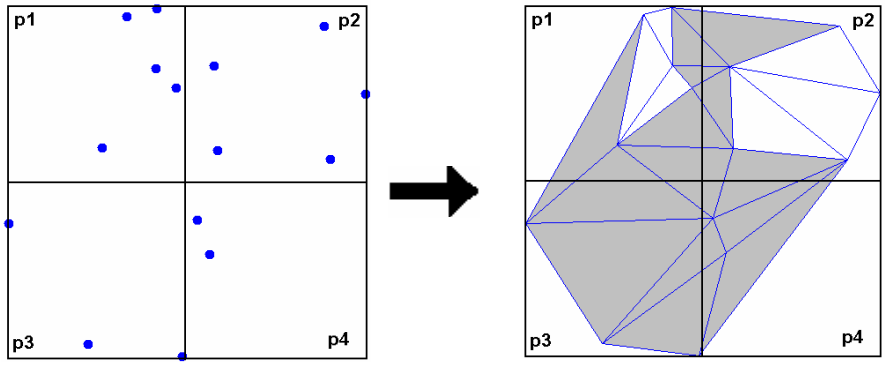


Figura 3.12. Triangulação de Delaunay de um conjunto de pontos em quatro processadores.

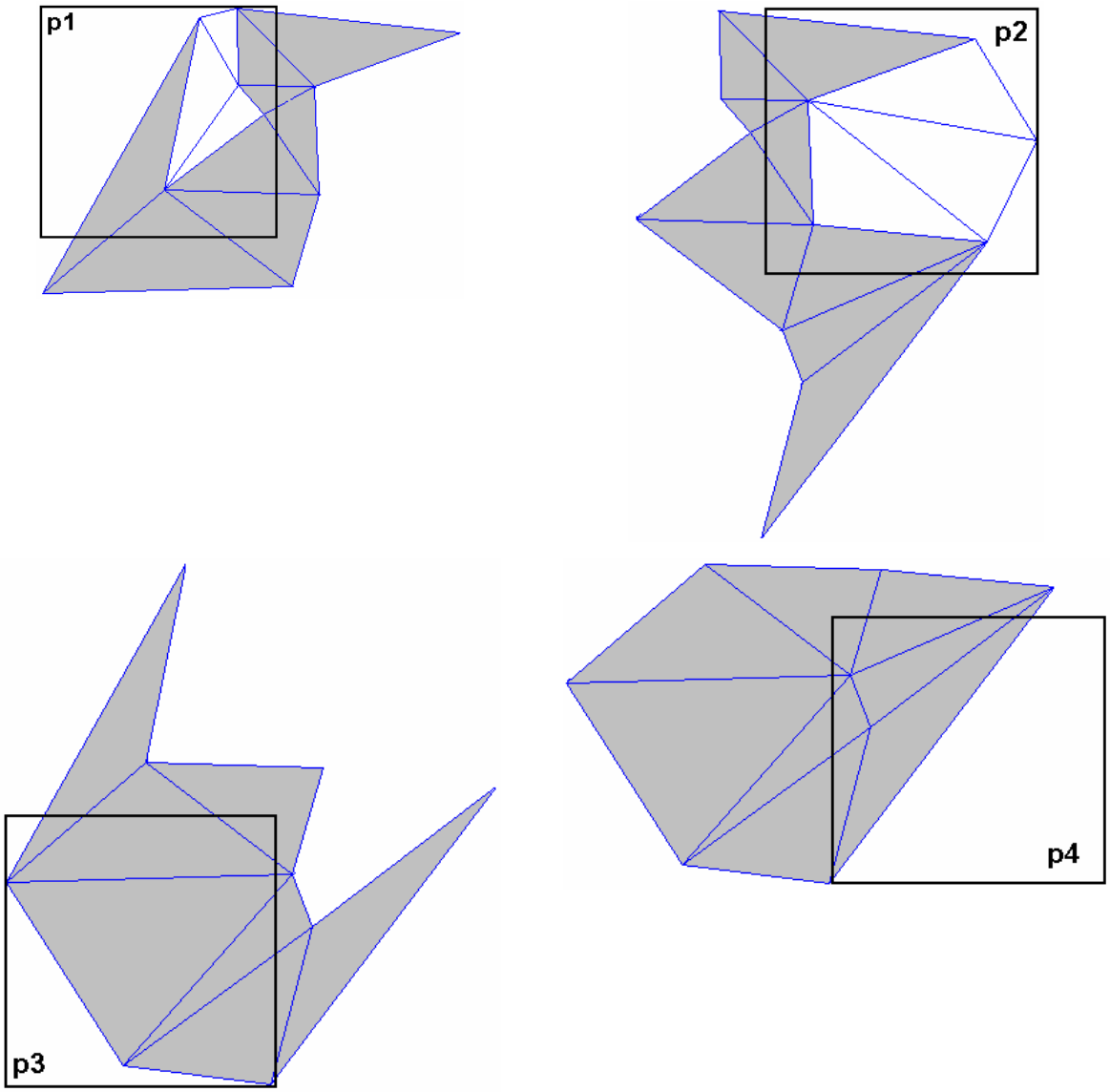


Figura 3.13. Resultados parciais da Triangulação de Delaunay nos quatro processadores.

Capítulo IV

Gerador de Malhas escrito em OCaml

4.1 Introdução

Estão disponíveis na *web*, como parte do site *Meshing Research Corner* (MESHING SOFTWARE SURVEY, 2005), mantido pela Carnegie Mellon University, informações sobre dezenas de geradores de malhas. Os produtos de *software* apresentam características variáveis, incluindo a forma dos elementos finitos (triangular, quadrilátera, tetraédrica, hexaédrica, estruturada), o tipo de algoritmo utilizado (*Avanço de Fronteira*, *Sweeping*, etc.) e o tipo de melhoramento e refinamento da malha adotados, por exemplo. Embora, para cerca de um quarto dos produtos haja indicação de disponibilidade do código-fonte, na realidade, nem sempre estão disponíveis para ser copiados, havendo recusa a pedidos de fornecimento por parte de seus autores.

Foram examinados e executados os códigos de *EasyMesh* (NICENO, 2005), *Geompack* (JOE, 1991), *GMSH* (REMACLE, 2005), *Javamesh* (LIN, 1997), *Mesh2D* (KARAMETE, 2005), *Triangle* (SHEWCHUK, 1996b) e apenas examinado o código de *TMG* (PAOLINI, 2004). Não foi examinada a biblioteca CGAL, pois foram considerados somente o conteúdo publicado no mencionado site. Relativamente à qualidade da malha, apesar de comprovada superioridade de *Triangle*, *GMSH* e *Javamesh*, foi decidido não utilizar quaisquer deles como referência para a implementação em OCaml. Os dois primeiros apresentavam elevada complexidade algorítmica; e todos os três requeriam bastante tempo para conversão em virtude das técnicas de programação complexas que foram empregadas. Finalmente, foi escolhido o código-fonte do gerador de malhas chamado *Mesh*, construído por Labelle (LABELLE; SHEWCHUK, 2003), na U. C. em Berkeley, o qual reúne três características favoráveis: malha de boa qualidade, algoritmo com complexidade relativa e estilo de programação simples. Foi denominado de *OCamlMesh* o gerador escrito em OCaml.

Triangle é um rápido e robusto programa gerador de malhas e triangulador de Delaunay escrito em C. O programa de Labelle também é escrito em C, mas não é tão célere quanto *Triangle*, porque emprega método da força bruta para implementar dois procedimentos

críticos: *triangulação de Delaunay* e *localização planar de pontos*. Entretanto, essas desvantagens implicaram usar de preferência seu código para conversão, pois propiciariam maiores oportunidades de pesquisa, experimentos e descobertas.

4.2 Características do Gerador

O programa possui cerca de 2.000 linhas de código e foi projetado para plataformas compatíveis com o UNIX, mas pode ser compilado em qualquer sistema operacional que suporte o padrão ANSI C. Ele toma como entrada um PSLG, que, por definição, é apenas uma lista de vértices e segmentos. A técnica empregada consiste em adicionar pontos de Steiner e, conseqüentemente, arestas para dividir a região interior em triângulos bem-formados.

Na Figura 4.1, é ilustrado o processo de geração de malhas. Na Figura 4.1 (b), vê-se um resultado da execução do programa na linha de comando do sistema operacional. Para rodar o programa, sua a sintaxe é a seguinte:

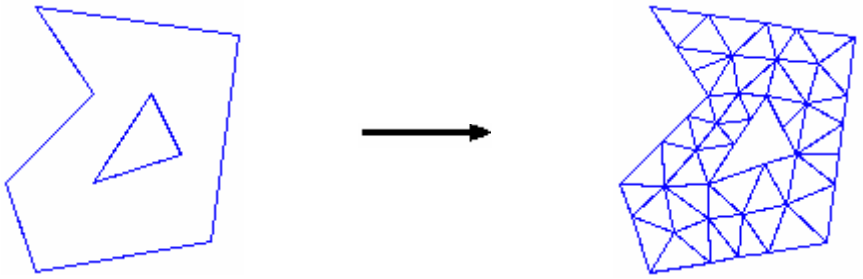
```
ocamlmesh arq_entrada num_vertices angulo_minimo [arq_saida]
```

O argumento `arq_entrada` deve ser um arquivo com extensão **.node** ou extensão **.poly**. Um arquivo com extensão **.node** contém um conjunto de pontos no plano, não sendo todos colineares. Um arquivo com extensão **.poly** representa um PSLG. Um arquivo **.poly** pode também conter informação sobre buracos e concavidades, bem como atributos sobre a região e restrições sobre as áreas de triângulos.

O argumento `num_vertices` indica o número total de vértices da malha, ou seja, os vértices originais do PSLG acrescidos dos pontos de Steiner inseridos. Por exemplo, o grafo da Figura 4.1 (a) contém nove vértices. A malha da Figura 4.1 (b) contém exatamente 40 vértices. Portanto, para a construção da malha, por força do argumento `40`, foram introduzidos 31 pontos de Steiner no PSLG.

O argumento `angulo_minimo` indica a medida do ângulo mínimo de cada triângulo. Quando `angulo_minimo` for maior que 30º, o valor para o argumento `num_vertices` deve ser (-1), e a malha será gradiente. Nesse caso, o gerador somente tentará obter triângulos bem-formados, mas não do mesmo tamanho. Isso é útil quando uma aplicação requer triângulos menores somente próximo à fronteira.

O argumento `arq_saida` é opcional e se refere a um arquivo com extensão **.ele**, o qual contém dados geométricos dos elementos finitos que formam a malha.



(a) PSLG sem malha (b) PSLG com malha (resultado da execução `ocamlmesh simple.poly 40 30.0`)

Figura 4.1. Síntese do processo de geração da malha.

Nas Figuras 4.2 e 4.3 adiante, são descritos, respectivamente, o formato e exemplos do conteúdo dos arquivos com extensões **.node**, **.poly** e **.ele**.

Estrutura do arquivo **.node** Primeira linha: <# de vértices> <dimensão (deve ser 2)> <# de atributos> <# de marcadores de fronteira (0 ou 1)> Linhas restantes: <vértice #> <x> <y> [atributos] [marcador de fronteira]
Estrutura do arquivo **.poly** Primeira linha: <# de vértices> <dimensão (deve ser 2)> <# de atributos> <# de marcadores de fronteira (0 ou 1)> Linhas seguintes: <vértice #> <x> <y> [atributos] [marcador de fronteira] Linha um: <# de segmentos> <# de marcadores de fronteira (0 ou 1)> Linhas seguintes: <segmento #> <ponto inicial> <ponto final> [marcador de fronteira] Linha um: <# de buracos> Linhas seguintes: <buraco #> <x> <y> Linha opcional: <# de atributos de região e/ou restrições sobre área> Linhas opcionais seguintes: <região #> <x> <y> <atributo> <área máxima >
Estrutura do arquivo **.ele** Primeira linha: <# de triângulos> <nós por triângulo > <# de atributos> Linhas restantes: <triângulo #> <nó> <nó> <nó> ... [atributos]

Figura 4.2. Descrição do formato dos arquivos *.node*, *.poly* e *.ele*.

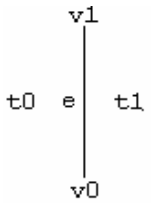
9.node	*simple.poly*	*square.poly*	*square.ele*
9 2 0 0 1 0 1 0 2 1 1 0 3 1 0 0 4 0 0 0 5 0 1 1 6 1 1 1 7 1 0 1 8 0 0 1 9 0.3 0.4 0.5	9 2 0 0 1 1 0 2 7 1 3 8 8 4 1 9 5 3 6 6 0 3 7 3 3 8 5 6 9 6 4 9 0 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6 6 6 1 7 7 8 8 8 9 9 9 7 1 1 4 4	4 2 0 0 1 0 0 2 1 0 3 1 1 4 0 1 4 0 1 1 2 2 2 3 3 3 4 4 4 1 0	22 3 0 1 10 1 14 2 1 10 15 3 3 11 16 4 11 3 17 5 2 12 18 6 13 4 19 7 2 8 12 8 9 4 13 9 6 5 10 10 5 7 10 11 9 5 11 12 5 8 11 13 8 5 12 14 5 6 12 15 7 5 13 16 5 9 13 17 6 10 14 18 10 7 15 19 11 8 16 20 9 11 17 21 12 6 18 22 7 13 19

Figura 4.3. Exemplos de conteúdo dos arquivos *.node*, *.poly* e *.ele*.

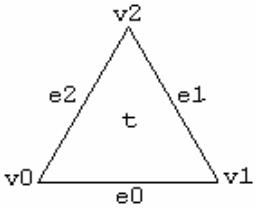
4.3 Estrutura de Dados

Uma estrutura é usada para representar cada aresta e outra é usada para representar cada triângulo (Figura 4.4). Cada registro que representa uma aresta contém dois ponteiros para vértices e dois ponteiros para triângulos. Cada registro que representa um triângulo contém três ponteiros para vértices e três ponteiros para arestas. Como ocorreu na implementação do algoritmo de triangulação de Delaunay por divisão-e-conquista, também foram mantidos os tipos de dados originais do gerador de malhas.

```
struct Edge {
  struct Vertex *vertex[2];
  struct Triangle *triangle[2];
  boolean frozen;
}
```




```
struct Triangle {
  struct Vertex *vertex[3];
  struct Edge *edge[3];
  float minAngle;
  float quality;
  boolean inside;
}
```



(a) Estrutura da aresta (b) Estrutura do triângulo

Figura 4.4. Estrutura de dados da aresta e do triângulo.

4.4. Notas sobre o Algoritmo

Nas Figuras 4.5 até 4.9, é ilustrado o processo de geração de malha a partir de um PSLG que representa uma guitarra elétrica.

O primeiro estágio do algoritmo foi encontrar a triangulação de Delaunay dos vértices de entrada como na Figura 4.5. Em geral, alguns dos segmentos de entrada e faces não aparecem na triangulação; o segundo estágio é recuperá-los. Na Figura 4.6, é ilustrado o estágio de recuperação de segmentos e faces. O terceiro estágio do algoritmo é remover triângulos de concavidades e buracos (Figura 4.7). O quarto estágio do algoritmo é aplicar um algoritmo de refinamento, como descrito na Seção 2.8. Na Figura 4.8, é mostrada a malha final, que não tem qualquer triângulo com ângulo inferior a 30º.

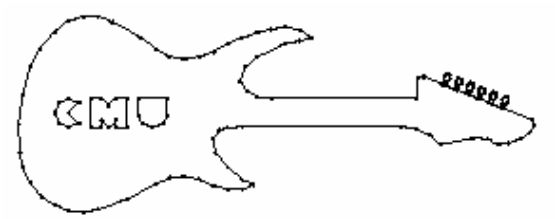


Figura 4.5. PSLG de uma guitarra elétrica.

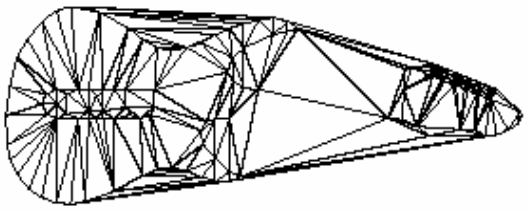


Figura 4.6. Triangulação de Delaunay dos vértices do PSLG. Alguns segmentos originais e faces estão ausentes.

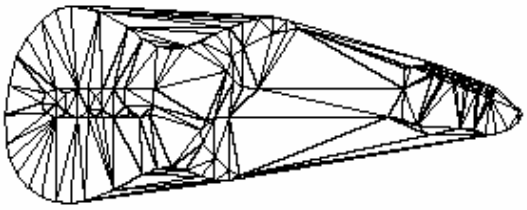


Figura 4.7. Triangulação de Delaunay em conformidade com os segmentos do PSLG.

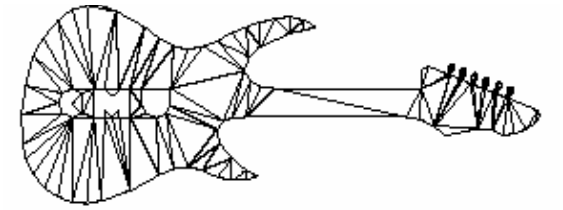


Figura 4.8. Triangulação com triângulos removidos de concavidades e buracos.

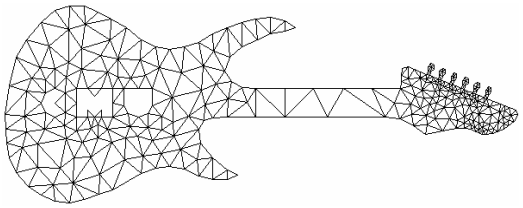


Figura 4.9. Malha final (gerada por *OCamlMesh*) composta de 484 triângulos com ângulo mínimo não inferior a 30º.

A maior desvantagem do algoritmo é ser lento para construir a malha com muitos (acima de dezena de milhar) pontos de Steiner, porquanto usa algoritmo de triangulação de Delaunay de complexidade $O(n^2)$ e algoritmo de localização planar de pontos de complexidade $O(n)$.

4.5. Detalhes da Implementação em OCaml

A primeira providência para a conversão de código escrito em C para OCaml constituiu-se em se adotar uma estrutura alternativa para representar os ponteiros e todas suas operações.

A definição de tipos para representar ponteiros proposta em (INRIA, 2004) possui notação complicada e trabalhosa para implementar ponteiros, sobretudo suas operações típicas, quando seu escopo de apontamento é extenso. Por exemplo: ponteiros duplos, ponteiros triplos, etc.

A definição de tipos para retratar ponteiros proposta em (MOURA *et al.*, 2005) tem notação simples e de fácil implementação. Seu princípio básico é simular em um vetor as posições de memória reservada para cada ponteiro. Por essa razão, foi utilizada também na reescrita do código de *Mesh* em OCaml. A função “FindNeighbors”, por exemplo, em sua assinatura requer ponteiros duplos. Na Figura 4.10, é mostrada sua implementação em C e em OCaml.

```c
void FindNeighbors(struct Edge *e, struct Triangle *t,
struct Edge **e0, struct Edge **e1, struct Vertex **v)
{
  if (e == t->edge[0]) {
    *e0 = t->edge[1];
    *e1 = t->edge[2];
    *v = t->vertex[2];
  }
  else if (e == t->edge[1]) {
    *e0 = t->edge[2];
    *e1 = t->edge[0];
    *v = t->vertex[0];
  }
  else if (e == t->edge[2]) {
    *e0 = t->edge[0];
    *e1 = t->edge[1];
    *v = t->vertex[1];
  }
  else
    FatalError("edge 'e' is not part of triangle 't'");
```

(a) Implementação em linguagem C

```ocaml
let findNeighbors e t e0 e1 v =
  if e = !triangleArray.(t).tr_edge.(0) then begin
    e0 := !triangleArray.(t).tr_edge.(1);
    e1 := !triangleArray.(t).tr_edge.(2);
    v := !triangleArray.(t).tr_vertex.(2)
  end
  else if e = !triangleArray.(t).tr_edge.(1) then begin
    e0 := !triangleArray.(t).tr_edge.(2);
    e1 := !triangleArray.(t).tr_edge.(0);
    v := !triangleArray.(t).tr_vertex.(0)
  end
  else if e = !triangleArray.(t).tr_edge.(2) then begin
    e0 := !triangleArray.(t).tr_edge.(0);
    e1 := !triangleArray.(t).tr_edge.(1);
    v := !triangleArray.(t).tr_vertex.(1)
  end
  else
    Printf.printf "FatalError: edge 'e' is not part of
triangle 't'";;
(*Assinatura da função: val findNeighbors : int ->
int -> int ref -> int ref -> int ref -> unit =  <fun>*)
```

(b) Implementação em linguagem OCaml

Figura 4.10. Códigos da função “FindNeighbors”.

Nas Figuras 4.11 e 4.12, é descrita a estrutura de dados do gerador implementada, respectivamente, em linguagens C e OCaml.

```c
struct Vertex {
  float x;
  float y;
};

struct Edge {
  struct Vertex *vertex[2];
  struct Triangle *triangle[2];
  boolean frozen;                /* is the edge frozen? (not allowed to flip) */
};

struct Triangle {
  struct Vertex *vertex[3];
  struct Edge *edge[3];
  float minAngle;                /* store this expensive-to-compute value */
  float quality;                 /* quality of the triangle */
  boolean inside;                /* is the triangle "inside" the mesh? */
};

struct Segment {
  struct Vertex *endpoint[2];
};

struct Vertex *vertexArray;
struct Edge *edgeArray;
struct Triangle *triangleArray;
struct Segment *segmentArray;
struct Vertex *holeArray;
```

Figura 4.11. Implementação da estrutura de dados do gerador em linguagem C.

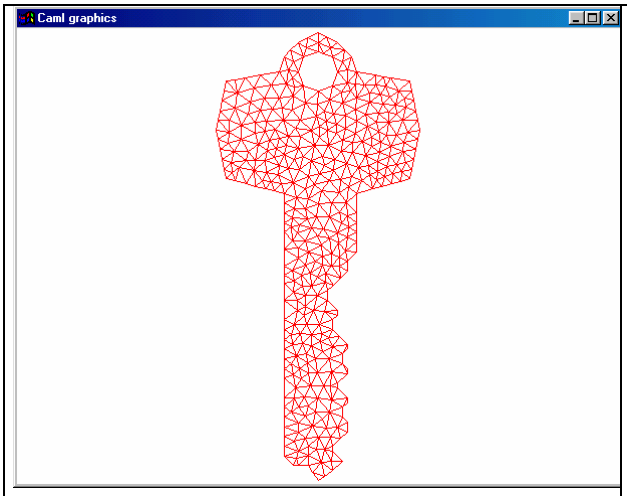
```
type vertex = {
  mutable ve_x: float;
  mutable ve_y: float;
  mutable ve_num: int
}
and edge = {
  mutable ed_vertex : int array;
  mutable ed_triangle : int array;
  mutable ed_frozen : bool;        (* is the edge frozen? (not allowed to flip) *)
  mutable ed_num: int
}
and triangle = {
  mutable tr_vertex : int array;
  mutable tr_edge : int array;
  mutable tr_min_angle : float;   (* store this expensive-to-compute value *)
  mutable tr_quality : float;      (* quality of the triangle *)
  mutable tr_inside : bool;        (* is the triangle "inside" the mesh? *)
  mutable tr_num: int
}
and segment = {
  mutable se_endpoint: int array;
  mutable se_num: int
};;

let vertexArray = ref [||];;        (* Array de vértices *)
let edgeArray = ref [||];;          (* Array de ponteiros para vértices *)
let triangleArray = ref [||];;      (* Array de triângulos *)
let segmentArray = ref [||];;       (* Array de ponteiros para vértices *)
let holeArray = ref [||];;          (* Array de vértices *)
```

Figura 4.12. Implementação da estrutura de dados do gerador em linguagem OCaml.

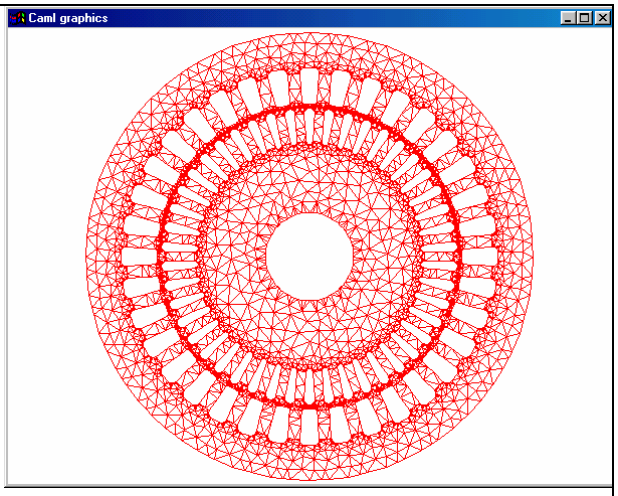
Na próxima seção, é apresentada uma amostra de malhas produzidas pelo gerador *OCamlMesh*.

4.6. Resultados da Triangulação de Domínios Geométricos

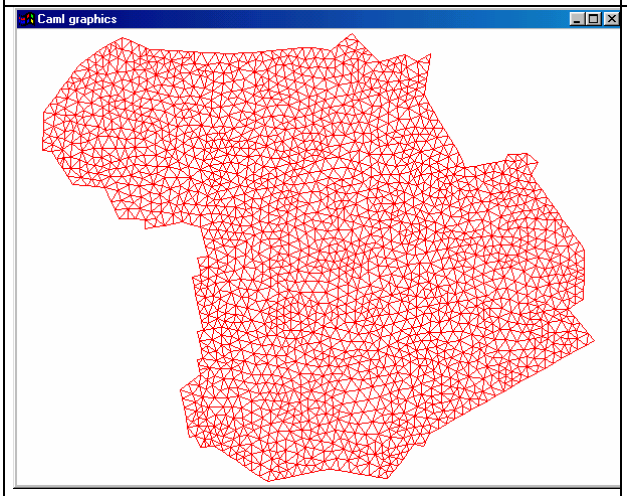
Na Figura 4.13, são apresentados alguns casos de triangulação de domínios irregulares com buracos e concavidades somente para mostrar a uniformidade da malha obtida e a habilidade do gerador para lidar com domínios irregulares. As primitivas geométricas “moveto” e “lineto” da biblioteca “Graphics”, que é inclusa na distribuição da linguagem OCaml, foram usadas para plotar a malha.



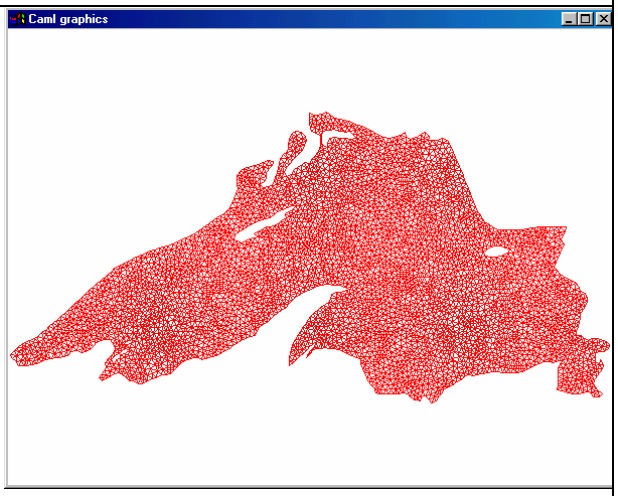
(a) Malha uniforme sobre o PSLG *key.poly*: 675 triângulos



(b) Malha uniforme sobre o PSLG *motortrifas.poly*, com buracos no eixo e nas ranhuras: 23.180 triângulos



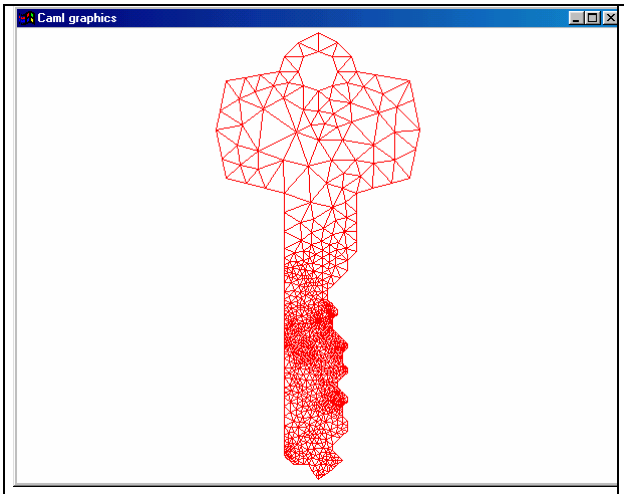
(c) Malha uniforme sobre o PSLG *africa.poly*: 3.790 triângulos



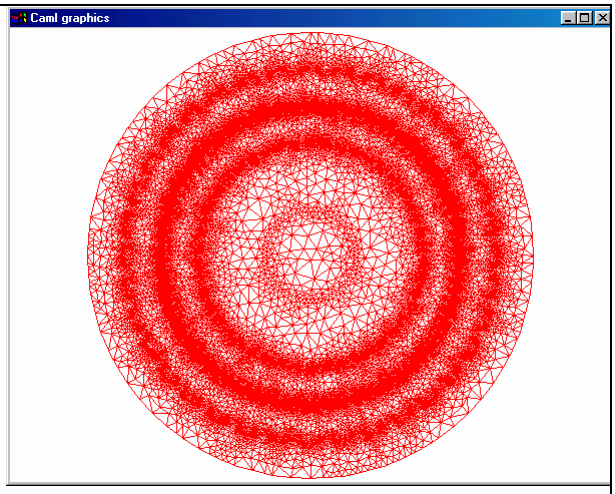
(d) Malha uniforme sobre o PSLG *superior.poly*: 9.423 triângulos

Figura 4.13. Algumas malhas uniformes construídas pelo gerador *OCamlMesh*.

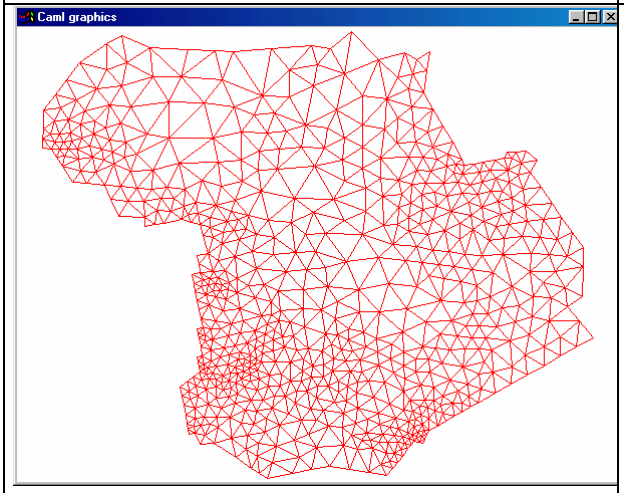
Na Figura 4.14, são apresentadas malhas gradientes, cada qual com variados tamanhos de triângulos em face da medida adotada para o ângulo mínimo do triângulo.



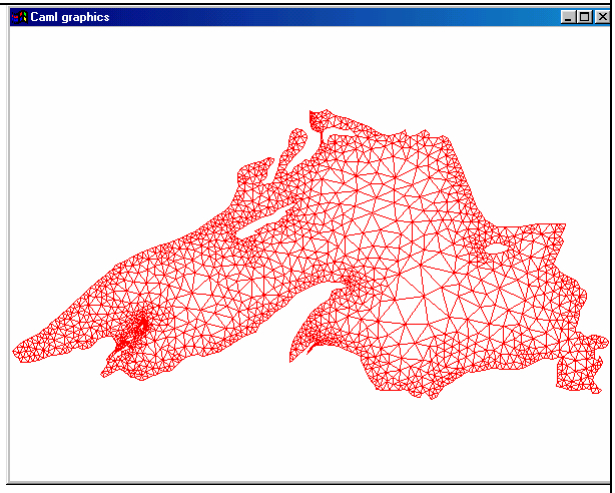
(a) Malha gradiente sobre o PSLG *key.poly*: 1.679 triângulos com ângulo mínimo de 34º



(b) Malha gradiente sobre o PSLG *motortrifas.poly*: 182.877 triângulos com ângulo mínimo de 30º



(c) Malha gradiente sobre o PSLG *africa.poly*: 1.276 triângulos com ângulo mínimo de 33º



(d) Malha gradiente sobre o PSLG *superior.poly*: 3.059 triângulos com ângulo mínimo de 32º

Figura 4.14. Algumas malhas gradientes construídas pelo gerador *OCamlMesh*.

Capítulo V

Localização Planar de Pontos Dinâmica usando Multiárvore-B Intervalar

5.1 Introdução

Um mapa é uma representação do espaço que divide o plano em um número finito de regiões. Um mapa pode ser modelado por um PSLG.

Dados um mapa e uma pergunta sobre um ponto *p*, especificado por suas coordenadas, o problema de localização planar de pontos consiste em encontrar a região no mapa que contém *p*.

Algumas técnicas de localização planar de pontos existentes requerem uma estrutura de pesquisa adicional. A estrutura de pesquisa adicional representa uma subdivisão auxiliar do mapa para facilitar a obtenção de resultados das pesquisas.

Os algoritmos de localização planar de pontos podem ser dinâmicos ou estático. Um algoritmo dinâmico é aquele que altera a estrutura de pesquisa à medida que ocorrem eventos de inserção ou de deleção na subdivisão planar. Ou seja, um algoritmo em que a estrutura de pesquisa é construída juntamente com o mapa que ela representa. Um algoritmo estático é que não suporta modificações na subdivisão planar, como o algoritmo dinâmico.

Entretanto, é necessário recursos adicionais para construir a estrutura de pesquisa. Tais recursos incluem tempo e espaço. Conforme mostrado no Capítulo 1, existem medidas ideais para o tempo requerido para construir a estrutura de pesquisa e para o espaço necessário para armazená-la. Por essa razão, a formulação de um algoritmo dinâmico de localização planar de pontos é uma área de pesquisa que se mantém ativa.

A localização planar de pontos dinâmica tem utilidade em qualquer aplicação em que a subdivisão planar sofre mudanças ou evolui, seja de modo contínuo ou não. Um exemplo é a triangulação de Delaunay incremental, já discutida no Capítulo 2.

Neste capítulo, é apresentado um algoritmo dinâmico de localização planar de pontos. No algoritmo, a técnica das “Slabs” é empregada para construir a estrutura de pesquisa, e a Multiárvore-B Intervalar é utilizada como estrutura de armazenamento dos dados.

A Multiárvore-B Intervalar é uma estrutura de dados derivada da Árvore-B, uma árvore balanceada, aparelhada com mecanismo de pesquisa intervalar e organizada dinamicamente em níveis ou camadas.

5.2 Árvores Balanceadas

Os algoritmos baseados em árvore de pesquisa binária (BST – *Binary Search Tree*) trabalham bem para uma ampla variedade de aplicações, mas, na prática, apresentam problema de desempenho no pior caso, sobretudo, se o usuário do algoritmo não tomar cuidado em relação a alguns tipos de entrada. Por exemplo, arquivos com registros já ordenados, arquivos com chaves duplicadas, arquivos com alternância de chaves grandes e pequenas: todos podem conduzir a tempos de construção de ordem quadrática e tempos de pesquisa de ordem linear.

O caso ideal é manter as árvores perfeitamente balanceadas. Essas estruturas garantem que todas as pesquisas podem ser concluídas em menos que $\log n + 1$ comparações, mas pode ser custoso mantê-las para operações de inserção e remoção.

Uma abordagem para produzir melhor balanço em BST é periodicamente rebalanceá-las explicitamente. De fato, pode-se balancear completamente a maioria das BST em tempo linear. Tal rebalanceamento melhorará o desempenho, mas sem garantias contra o desempenho de ordem quadrática no pior caso.

Por outro lado, o tempo de inserção para uma seqüência de chaves durante operações de rebalanceamento pode tornar-se proporcional ao quadrado do tamanho da seqüência; por outro lado, pode-se não pretender rebalancear freqüentemente árvores de dimensões elevadas porque cada operação de rebalanceamento custa ao menos um tempo proporcional, de ordem linear, ao tamanho da árvore.

Então, o problema de garantir bom desempenho em pesquisas em estruturas do tipo BST é solucionado por meio de três abordagens algorítmicas: randomização, amortização ou otimização (SEDGEWICK, 1998).

Um algoritmo aleatório introduz decisão randômica para reduzir dramaticamente a chance de um cenário de pior caso, sem se importar em que ordem os itens são apresentados para entrada. Um exemplo típico desse arranjo é o elemento aleatório usado no procedimento de partição do algoritmo de ordenação “quicksort”. Outros exemplos são as árvores de pesquisa binária aleatórias, que usam cerca de $2n \log n$ comparações para ser construída com

n itens, independentemente de sua ordem de entrada, e cerca de 2 log *n* comparações para efetuar pesquisas.

Uma abordagem baseada em amortização consiste em realizar trabalho extra em um momento para evitar trabalho adicional mais tarde. Dessa forma, ela é capaz de garantir limites superiores ao custo médio por operações, que é o custo total de todas as operações dividido pelo número de operações. Um exemplo é a árvore *Splay* (SLEATOR; TARJAN, 1985).

A abordagem da otimização visa a garantir bom desempenho para cada operação. Os métodos desenvolvidos segundo essa abordagem requerem que seja mantida nas árvores alguma informação estrutural. Um exemplo são as árvores-B.

Uma vantagem em usar estrutura de pesquisa baseada em árvore-B é que ela não se degenera, como exemplificado na Figura 5.1. Ela se mantém sempre balanceada (Figura 5.2), a um custo mínimo, independentemente da ordem de entrada dos nós. Uma outra vantagem é não ter-se que utilizar inserção aleatória de segmentos para, probabilisticamente, obter uma estrutura de pesquisa mais similar possível à de uma árvore balanceada, como ocorre com algoritmos de localização planar baseados em mapas trapezoidais (BERG, 1997). Em vez de árvore, eles empregam grafo direcionado acíclico como estrutura de pesquisa.

Sabe-se que a árvore-B requer uma grande quantidade de espaço para armazenamento e é mais utilizada para busca em memória externa. Entretanto, como a árvore-B havia apresentado melhores resultados que a árvore *Splay*, que teve desempenho quase linear em experimentos de inserção, deleção e pesquisa, confirmando o que tinha sido declarado por Sedgewick em (SEDGEWICK, 1998), preferiu-se utilizar primeiro a árvore-B, como estrutura básica do algoritmo dinâmico de localização planar de pontos. Outros tipos de árvores balanceadas podem ser utilizados, como é sugerido na indicação de trabalhos futuros no Capítulo 6.

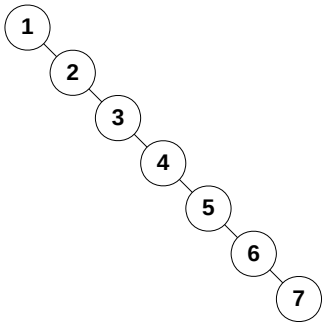


Figura 5.1. Árvore binária totalmente degenerada.

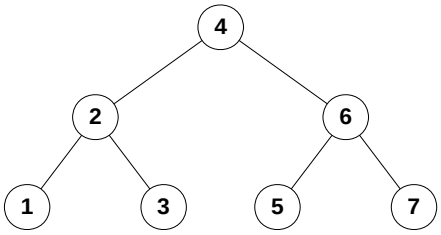


Figura 5.2. Árvore binária balanceada.

5.2.1 A Árvore-B

A árvore-B, introduzida, em 1972, por Bayer e McCreight, é um tipo de árvore balanceada multidirecional largamente utilizada em sistemas gerenciadores de bases de dados (SGBD). Sua estrutura permite que registros sejam inseridos, removidos e recuperados com a garantia de desempenho satisfatório no pior caso. Uma árvore-B de ordem *m* e com *n* nós tem altura O(log *n*) e satisfaz as propriedades adiante (KNUTH, 1998):

1. Cada nó tem no máximo *m* filhos.
2. Cada nó, exceto o da raiz e as folhas da árvore, tem ao menos *m*/2 filhos.
3. A raiz é uma folha da árvore ou tem ao menos 2 filhos.
4. Todas as folhas aparecem no mesmo nível.
5. Um nó não-folha com *k* filhos contém *k*-1 chaves.

Diferentemente de uma árvore binária, cada nó de uma árvore-B pode ter um número variável de chaves e filhos. Uma árvore de pesquisa multidirecional de ordem *m* é uma árvore geral na qual cada nó tem *m* ou menos subárvores e contém uma chave a menos que a quantidade de suas subárvores. Na Figura 5.3, em que é mostrada uma árvore-B de ordem 4, cada nó, exceto os folhas, tem entre [4/2] e 4 filhos, contendo uma ou três chaves. Ao nó-raiz é permitido conter de uma a três chaves; nesse exemplo, ele tem apenas uma. Todas as folhas estão no nível 2. As chaves aparecem em ordem crescente da esquerda para a direita.

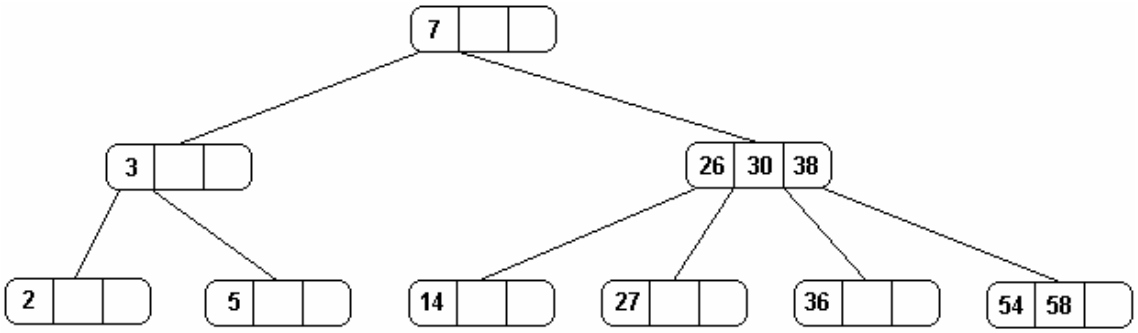


Figura 5.3. Uma árvore-B de ordem 4 com todas as folhas no nível 2.

Suponha uma árvore-B de ordem *m* contendo *N* chaves e que *N* + 1 folhas apareçam no nível *l*. Então, o número de máximo de níveis, correspondente à altura da árvore é dado por 1 + $\log_m N$.

Isso significa, por exemplo, que, se *N* = 1.999.998 e *m* = 199, então *l* é no máximo 3, que, no caso de pesquisa, equivale também ao número máximo de comparações para encontrar um determinado elemento.

Em virtude da popularidade da árvore-B e da variedade de trabalhos escritos tratando de sua história, estrutura, desempenho e aplicações, serão enfocados os resultados de seu emprego na resolução do problema de localização planar de pontos.

5.2.2 A Árvore-B Intervalar

Em uma árvore-B, assim como em uma árvore binária, uma operação de busca é aquela que aceita um argumento *a* e tenta encontrar o registro cuja chave seja *a*. É possível que a busca de um determinado elemento na árvore não tenha sucesso, ou seja, que talvez não exista nenhum nó da árvore cuja chave tenha o mesmo valor que o argumento usado na operação de busca. Nesse caso, o algoritmo que implementa a operação de busca pode retornar um ponteiro nulo ou um registro nulo.

Essa regra, na prática, é incompatível com as peculiaridades da localização planar de pontos, pois, na maioria das vezes, valor do ponto de consulta não corresponde exatamente ao valor de nenhum dos vértices dos segmentos armazenados na estrutura de dados usada como estrutura de pesquisa. Nas estruturas de pesquisas utilizadas para representar subdivisões fundamentadas em “slabs” ou mapas trapezoidais, quando isso acontece, é oportuno conhecer os nós com valor imediatamente anterior e posterior àquele consultado. Por essa razão, a árvore-B teve seu algoritmo de busca aprimorado com um mecanismo de pesquisa intervalar. Um mecanismo de pesquisa que, em lugar de um sinalizador de insucesso, retorne, por referência, se houver, os nós com valor imediatamente anterior e posterior. A esse tipo de árvore-B foi dado o nome de árvore-B intervalar.

Assim, na árvore-B intervalar, em operações de busca, quando não se encontra um nó contendo a chave de mesmo valor que o argumento da busca, em lugar de um ponteiro ou registro nulos, retorna-se, se houver, o registro com valor imediatamente anterior ou, ainda, retorna-se o registro com valor imediatamente superior ao do argumento.

Nas Figuras 5.4 e 5.5, são mostradas, em estilo negrito, modificações, respectivamente, nos códigos-fontes das funções “binsearch” e “search” para transformar a árvore-B em árvore-B intervalar. As modificações efetuadas no código da função “binsearch” têm a finalidade reter o registro-candidato ao limite inferior, ou seja, o valor imediatamente inferior àquele procurado. As modificações feitas no código da função “search” têm a finalidade de reter o registro-candidato ao limite superior, ou seja, o valor imediatamente superior àquele procurado. As variáveis que não foram definidas no cabeçalho das duas funções são de escopo global.

```
1.  let binsearch x a n p =
2.      let i, left, right = ref 0, ref 0, ref 0 in
3.      if !num_comp = 1 then begin
4.          p_lower := p;
5.          i_lower := !i;
6.          lower_bound := a.(0)
7.      end
8.      else if x.k_coord > a.(0).k_coord && !lower_bound.k_coord <> a.(0).k_coord then begin
9.          p_lower := p;
10.         i_lower := 0;
11.         lower_bound := a.(0)
12.     end;
13.     if x.k_coord <= a.(0).k_coord then
14.         0
15.     else begin
16.         if x.k_coord > a.(n-1).k_coord && !lower_bound.k_coord <> a.(n-1).k_coord then begin
17.             p_lower := p;
18.             i_lower := n-1;
19.             lower_bound := a.(n-1)
20.         end;
21.         if x.k_coord > a.(n-1).k_coord then
22.             n
23.         else begin
24.             left := 0;
25.             right := (n-1);
26.             while (!right - !left) > 1 do
27.                 i := (!right + !left)/2;
28.                 if x.k_coord > a.(!i).k_coord && !lower_bound.k_coord <> a.(!i).k_coord then begin
29.                     p_lower := p;
30.                     i_lower := !i;
31.                     lower_bound := a.(!i)
32.                 end;
33.                 if x.k_coord <= a.(!i).k_coord then
34.                     right := !i
35.                 else
36.                     left := !i
37.             done;
38.             !right
39.         end;
40.     end;;
```

Figura 5.4. Modificações no código da árvore-B para obtenção do limite inferior.

O procedimento para obter do limite inferior é mais complexo que o procedimento para obter o limite superior. Essa diferença na complexidade é herdada das características algorítmicas das funções “search” e “binsearch”. Por exemplo, na linha 12 do código da função “search”, mostrado na Figura 5.5, é descrita a condição de parada do código original, sem levar em conta o conjunto de instruções para captura do limite superior. Se a condição de parada não é atendida, ou seja, se se chegou ao último elemento do nó sem se ter encontrado uma chave com valor igual ao valor do elemento procurado, então deve-se reter, como itens do limite superior: o valor do elemento visitado e os dois índices referentes, respectivamente,

à posição da elemento (chave) e do nó no array que representa a árvore-B. Já a retenção do limite inferior é mais complicada, como mostram as linhas 3 a 12, 16 a 20 e 28 a 32 do código da função “binsearch”, mostrado na Figura 5.4.

```
1.  let search x id_btree =
2.      let p, i, n = ref 0, ref 0, ref 0
3.      and k = ref (Array.init (2 * _M) (fun x -> {k_coord=(Array.init 2 (fun x -> 0.0)); k_indice=0}))
4.      and result = ref {k_coord=(Array.init 2 (fun x -> (-1.0))); k_indice=(-1)}
5.      and break = ref false in
6.      p := !root;
7.      while (!p <> (-1)) && (!break = false) do
8.          i := 0;
9.          k := !multi_page_array.(id_btree).mbt_btree.(!p).bt_key;
10.         n := !multi_page_array.(id_btree).mbt_btree.(!p).bt_n;
11.         i := binsearch x !k !n !p;
12.         if (!i < !n) && (x.k_coord = !k.(!i).k_coord) then begin
13.             found !p !i id_btree;
14.             result := !multi_page_array.(id_btree).mbt_btree.(!p).bt_key.(!i);
15.             break := true
16.         end
17.         else begin
18.             if (!i < !n && !num_comp = 1) ||
19.               (!i < !n && x.k_coord < !k.(!i).k_coord && !upper_bound.k_coord <> !k.(!i).k_coord) then begin
20.                 p_upper := !p;
21.                 i_upper := !i;
22.                 upper_bound := !k.(!i)
23.             end
24.         end;
25.         if not !break then
26.             p := !multi_page_array.(id_btree).mbt_btree.(!p).bt_branch.(!i)
27.     done;
28.     !result;;
```

Figura 5.5. Modificações no código da árvore-B para obtenção do limite superior.

5.2.3 A Multiárvore-B Intervalar

A Multiárvore-B Intervalar é um arranjo de múltiplas árvores-B intervalares em camadas. O custo de cada operação na Multiárvore-B Intervalar é $O(\log n)$. Na Figura 5.6, está representada uma Multiárvore-B de ordem 4. A árvore da camada zero é a árvore-B superior. No exemplo, considerou-se apenas um nó, mas pode haver inúmeros, dependendo da quantidade de “slabs”, ou seja, de coordenadas *x* distintas na malha. As setas partindo da árvore-B superior referem-se aos ponteiros para a árvore-B inferior em que se encontram os segmentos de cada “slab”. Na proposta deste trabalho, foi definida uma Multiárvore-B de ordem 5, por ter apresentado desempenho superior. Na Figura 5.7, pode-se perceber que a árvore-B está contida na estrutura de dados da Multiárvore-B.

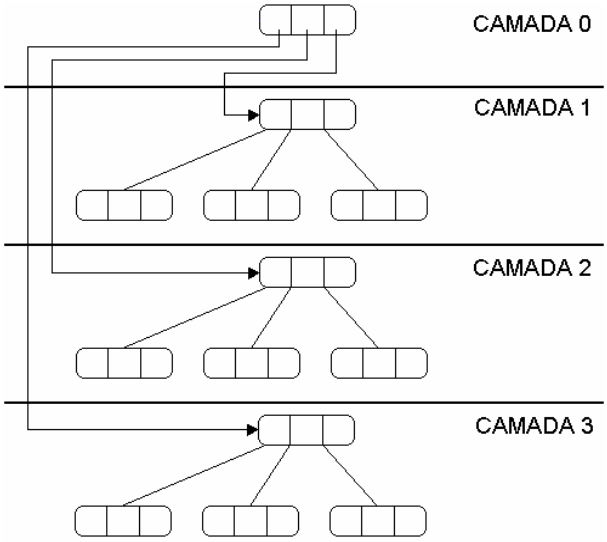


Figura 5.6. Arquitetura global de uma Multiárvore-B.

```
type t_key = {
  mutable k_coord: float array;
  mutable k_indice: int
}
and page_btree = {
  mutable bt_n: int;
  mutable bt_key: t_key array;
  mutable bt_branch: int array
}
and page_multibtree = {
  mutable mbt_root: int;
  mutable mbt_btree: page_btree array;
  mutable mbt_newp: int;
  mutable mbt_num_p: int;
  mutable mbt_elem_count: int
};;
```

Figura 5.7. Estrutura de dados da Multiárvore-B em OCaml.

O tipo *t_key* define a estrutura de cada nó da árvore-B. Na árvore-B superior, *k_coord* contém os limites inferior e superior das coordenadas *x* dos segmentos que definem cada "slab", e *k_indice* aponta para a camada em que está a árvore-B inferior correlata. Na árvore-B inferior, *k_coord* contém as coordenadas do segmento com as coordenadas do vértice inicial invertidas, e *k_indice* refere-se ao identificador do segmento na malha.

O tipo *page_btree* define a estrutura da árvore-B como descrita em (KNUTH, 1998).

O tipo *page_multibtree* define a estrutura da Multiárvore-B. O campo *mbt_btree* é um "array" de árvores-B. Os quatro últimos campos constituem os parâmetros de cada árvore-B: *mbt_root* indica o nó que está na raiz da árvore; *mbt_newp* é uma espécie de ponteiro para o nó recém-criado; *mbt_num_p* é um índice seqüencial dos nós criados e *mbt_elem_count* é um contador de elementos válidos, isto é, o somatório de valores constantes de *page_btree.bt_n*.

5.3 Localização Planar com Multiárvore-B Intervalar

Como mostrado na Figura 5.6, a Multiárvore-B Intervalar é uma estrutura de dados de composta de múltiplas camadas. Nela, são armazenadas informações sobre os vértices inicial e final dos segmentos de um PSLG. Essa organização em múltiplas camadas é importante para o problema da localização planar de pontos.

A camada zero contém uma Árvore-B Intervalar que armazena, em ordem crescente, os valores de coordenada *x* dos vértices do PLSG. Tal armazenamento permite organizar os segmentos do PSLG em intervalos. Nas Árvores-B contidas nas demais camadas, estão armazenados, em ordem crescente, pela coordenadas *y*, os segmentos (ou pedaços de segmentos) do PSLG. Cada registro da Árvore-B Intervalar da camada zero, existe um ponteiro para uma Árvore-B Intervalar que contém segmentos com vértices inicial e final coincidentes com o limite inferior e limite superior do intervalo.

A localização da região do PSLG em que se encontra um determinado ponto é realizada em duas etapas: primeiro pesquisa-se coordenada *x* do ponto, na camada zero da Multiárvore-B Intervalar para definir a Árvore-B inferior em que será feita a segunda pesquisa. Depois, pesquisa-se pela coordenada y do ponto na Árvore-B inferior apontada para definir a posição do ponto relativa aos segmentos e finalmente a face do PSLG, que corresponde à região em que se encontra o ponto de consulta. Ambas as pesquisas são intervalares. Ou seja, se em um PLSG não existir vértice de mesmo valor que o ponto de consulta, deve-se retornar, como resultado da pesquisa, o intervalo em que está o ponto de consulta: tanto em relação ao eixo *x* quanto em relação ao eixo *y* dos segmentos do PSLG.

5.3.1 Construção Incremental da Estrutura de Pesquisa baseada em Eventos

Nesta fase, a maior preocupação foi validar o método baseado em eventos apresentado adiante. Para isso, o algoritmo deveria aceitar, incrementalmente, como entrada, cada um dos segmentos de um PSLG qualquer e, dinamicamente, construir uma estrutura de pesquisa utilizando o método das “Slabs”. O algoritmo seria considerado correto se produzisse a mesma estrutura de pesquisa para cada nova ordem de entrada dos segmentos e indicasse corretamente a região contendo cada um dos pontos previamente estabelecidos.

Considere o PSLG da Figura 5.8. Ele é particionado em *l* linhas verticais, que atravessam cada um de seus sete vértices, formando, portanto, *l*-1 “slabs”. Os limites inferior e superior

que definem cada “slab” são armazenados na árvore-B superior, localizada na camada zero. Para cada “slab”, é definida uma árvore-B inferior a partir da camada um. Cada árvore-B inferior armazena os vértices inicial e final dos segmentos, com as coordenadas do vértice inicial invertidas. Quando as coordenadas *x* do novo segmento não constam da árvore-B superior, é criada uma nova árvore-B inferior. Na Figura 5.9, é sintetizado o processo de construção da estrutura de pesquisa para a localização planar de pontos.

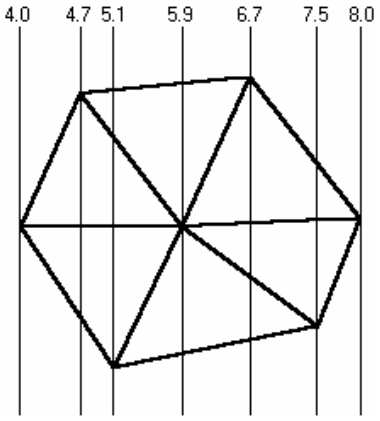


Figura 5.8. Subdivisão de um PSLG em *slabs*.

1. Para cada novo triângulo *T* formado pela triangulação de Delaunay faça
2. Armazene os segmentos de *T* em *S*
3. Altere a ordem dos vértices de cada segmento em *S*, tal que o vértice inicial seja menor que vértice final
4. Para i de 1 até 3 faça
5. Verifique a orientação do segmento S_i em relação ao centróide de *T* e armazene em S_i.*LeftFace* e em S_i.*RightFace*, conforme o caso, a posição do centróide de *T* relativa ao segmento S_i.
6. Insira o segmento S_i na fila *F*
7. Enquanto a fila *F* não estiver vazia faça
8. Atualize valores de *x_max*, *y_max*, *coord_min* e *coord_max*
9. Se o registro {*coord_min.x*, *coord_max.x*}não estiver na Árvore-B Superior então
10. Inclua registro {coord_min.x, coord_max.x} na Árvore-B Superior
11. Inclua-o na Árvore-B Inferior (ocorrência do Evento 1, conforme Figura 5.11)
12. Atualize dados de controle da Multiárvore-B Intervalar
13. Senão
14. Verifique o tipo de evento ocorrido (conforme Figura 5.11)
15. Execute os procedimentos referentes ao evento ocorrido
16. Atualize dados de controle da Multiárvore-B Intervalar

Figura 5.9. Algoritmo global do processo de construção da estrutura de pesquisa.

O algoritmo constrói a estrutura de pesquisa para a localização planar de pontos à medida que a malha é construída e admite que os triângulos e os segmentos sejam inseridos em qualquer ordem. No caso do grafo da Figura 5.8, para a inserção dos seis triângulos há 6! seqüências diferentes; no caso dos segmentos, há 12! diferentes seqüências de inserção. Para contemplar todas as possibilidades, foram apurados 14 tipos de eventos que, depois de normalizados, foram reduzidos a 6 tipos, conforme mostrado na Figura 5.11.

O algoritmo não registra a história da construção incremental de uma triangulação de Delaunay, como ocorre com o algoritmo dinâmico apresentado por Berg *et al.* (BERG, 1997), e discutido no Capítulo 1. Quando um triângulo deixa de existir, os dados sobre seus segmentos são removidos da Multiárvore-B Intervalar.

O critério para definição do conjunto de eventos da Figura 5.11 foi denominado de mapeamento do espaço binário. Considerada a possibilidade do segmento a ser inserido na estrutura de pesquisa interceptar um outro segmento ou de ser interceptado. Na Figura 5.10, são mostradas as possíveis maneiras de segmento se situar em relação a outro segmento. A ocupação dessas regiões determinam o modo como um segmento interceptar outro segmento. O caso 5 representa um segmento espúrio, contendo uma lacuna, e, por isso, é descartado. O caso 0 representa a inexistência de segmento vizinho ou ausência de interseção com algum segmento vizinho. No contexto da Figura 5.11, o caso 0 é tratado como o Evento 1, em que dois segmentos não se interceptam.

Casos	**Regiões do Segmento**		
	Direita	**Central**	**Esquerda**
0			
1			
2			
3			
4			
5			
6			
7			

Figura 5.10. Mapeamentos possíveis de segmentos no espaço binário.

Assim, conhecidas as possíveis relações entre o novo segmento e, no máximo, dois segmentos vizinhos não simultâneos, à esquerda ou à direita, foi obtido o conteúdo da Figura 5.11. O código de evento que aparece na primeira coluna, é aquele que ele tinha antes de um procedimento de normalização que excluiu os casos redundantes.

Evento	Descrição do Evento	Amostra do Evento
1	new_segment_left_endpoint < right_neighbor_left_endpoint && new_segment_right_endpoint < right_neighbor_left_endpoint	
3	new_segment_left_endpoint < right_neighbor_left_endpoint && new_segment_right_endpoint < right_neighbor_right_endpoint	
6	new_segment_left_endpoint = left_neighbor_left_endpoint && new_segment_right_endpoint = left_neighbor_right_endpoint	
7	new_segment_left_endpoint = left_neighbor_left_endpoint && new_segment_right_endpoint > left_neighbor_right_endpoint	
8	new_segment_left_endpoint = left_neighbor_left_endpoint && new_segment_right_endpoint < left_neighbor_right_endpoint	
9	new_segment_left_endpoint > left_neighbor_left_endpoint && new_segment_right_endpoint > left_neighbor_right_endpoint	

Figura 5.11. Eventos significativos entre segmento novo (linha tracejada) e segmento vizinho (linha contínua), retratando as possíveis formas de interseção entre eles.

Quando um segmento intercepta outro segmento, ocorre um corte vertical sobre a coordenada *x* do ponto de interseção. Segue adiante a interpretação para cada um dos seis eventos.

O Evento 1 retrata o caso em que o novo segmento não corta nenhum segmento e é o primeiro a ocupar uma “slab”.

O Evento 3 retrata o caso em que o novo segmento, com base em seu ponto extremo direito, corta outro(s) segmento(s) e é cortado pelo ponto extremo esquerdo de outro segmento.

O Evento 6 retrata o caso em que o novo segmento não corta nenhum outro segmento, mas ocupa uma “slab” em que se encontra(m) outro(s) segmento(s) com as mesmas medidas dele para as coordenadas *x*.

O Evento 7 retrata o caso em que o novo segmento é cortado pelo ponto extremo direito de outro segmento.

O Evento 8 retrata o caso em que o novo segmento corta, com base em seu ponto extremo direito, outro(s) segmento(s), mas não é cortado por nenhum segmento.

O Evento 9 retrata o caso em que o novo segmento corta, com base em seu ponto extremo esquerdo, outro(s) segmento(s) e é cortado pelo ponto extremo direito de outro segmento.

Finalmente, foi avaliada a ocorrência de o novo segmento ter dois segmentos vizinhos simultâneos. Isso implicou uma situação mais complexa: a existência de nove regiões distintas, três para cada um dos segmentos, o vizinho-esquerdo, o novo segmento e o vizinho-direito. Um procedimento automático de normalização, que também foi escrito em OCaml,

excluiu os casos espúrios — aqueles em cuja representação binária existe a seqüência “101”, como o caso 5 da Figura 5.10 — e reduziu a amostra de 512 (2^9) casos (2 estados a 9 regiões) para 45 casos (Figura 5.12). Tais casos são, na verdade, manifestações dos seis tipos de eventos que aparecem na Figura 5.11. A região sombreada que aparece em cada linha da Figura 5.12 representa cada uma das possibilidades de ocupação de espaço pelo novo segmento. Os números 0 e 1 definem a posição do novo segmento no espaço binário, significando, respectivamente, região não-ocupada e região ocupada.

Casos	Espaço ocupado pelo segmento vizinho esquerdo			Espaço intermediário			Espaço ocupado pelo segmento vizinho direito			Evento correspondente conforme Figura 5.11
1	0	0	0	0	0	0	0	0	1	Evento 9
2	0	0	0	0	0	0	0	1	0	Evento 9
3	0	0	0	0	0	0	0	1	1	Evento 9
4	0	0	0	0	0	0	1	0	0	Evento 8
5	0	0	0	0	0	0	1	1	0	Evento 8
6	0	0	0	0	0	0	1	1	1	Evento 6
7	0	0	0	0	0	1	0	0	0	Evento 1
8	0	0	0	0	0	1	1	0	0	Evento 3
9	0	0	0	0	0	1	1	1	0	Evento 3
10	0	0	0	0	0	1	1	1	1	Evento 3
11	0	0	0	0	1	0	0	0	0	Evento 1
12	0	0	0	0	1	1	0	0	0	Evento 1
13	0	0	0	0	1	1	1	0	0	Evento 3
14	0	0	0	0	1	1	1	1	0	Evento 3
15	0	0	0	0	1	1	1	1	1	Evento 3
16	0	0	0	1	0	0	0	0	0	Evento 1
17	0	0	0	1	1	0	0	0	0	Evento 1
18	0	0	0	1	1	1	0	0	0	Evento 1
19	0	0	0	1	1	1	1	0	0	Evento 3
20	0	0	0	1	1	1	1	1	0	Evento 3
21	0	0	0	1	1	1	1	1	1	Evento 3
22	0	0	1	0	0	0	0	0	0	Evento 9
23	0	0	1	1	0	0	0	0	0	Evento 9
24	0	0	1	1	1	0	0	0	0	Evento 9
25	0	0	1	1	1	1	0	0	0	Evento 9
26	0	0	1	1	1	1	1	0	0	Evento 3
27	0	0	1	1	1	1	1	1	0	Evento 3
28	0	0	1	1	1	1	1	1	1	Evento 3
29	0	1	0	0	0	0	0	0	0	Evento 9
30	0	1	1	0	0	0	0	0	0	Evento 9
31	0	1	1	1	0	0	0	0	0	Evento 9
32	0	1	1	1	1	0	0	0	0	Evento 9
33	0	1	1	1	1	1	0	0	0	Evento 9
34	0	1	1	1	1	1	1	0	0	Evento 3
35	0	1	1	1	1	1	1	1	0	Evento 3
36	0	1	1	1	1	1	1	1	1	Evento 3
37	1	0	0	0	0	0	0	0	0	Evento 8
38	1	1	0	0	0	0	0	0	0	Evento 8
39	1	1	1	0	0	0	0	0	0	Evento 6
40	1	1	1	1	0	0	0	0	0	Evento 7
41	1	1	1	1	1	0	0	0	0	Evento 7
42	1	1	1	1	1	1	0	0	0	Evento 7
43	1	1	1	1	1	1	1	0	0	Evento 7
44	1	1	1	1	1	1	1	1	0	Evento 7
45	1	1	1	1	1	1	1	1	1	Evento 7

Figura 5.12. Relação espacial entre o novo segmento e os segmentos vizinhos.

Como o grafo que compõe a malha é planar, não há sobreposição do novo segmento a quaisquer de seus vizinhos. Quando isso aparentemente é sugerido pela Figura 5.12, interprete-se que possuem coordenadas *x* em comum, mas estão ladeados assimetricamente ou simetricamente, como no exemplo da Figura 5.11.

5.3.2 Pré-processamento

Nesta fase, constrói-se a estrutura de pesquisa para a malha que estiver sendo processada. A divisão da malha em "slabs" é levada a cabo, incrementalmente, à medida que cada novo triângulo é criado.

Para a identificação exata da face sobre a qual está o ponto consultado, os vértices de cada novo segmento foram reposicionados de forma a que a coordenada *x* do vértice inicial seja menor ou igual à coordenada *x* do vértice final, como indicado pelas setas nos exemplos da Figura 5.13. Esse modelo de orientação também é importante para identificar corretamente o evento, ou seja, a vizinhança e o tipo de relação que o novo segmento estabelece com outros segmentos. Desse modo, a subdivisão planar obtida tem o mesmo formato que a subdivisão resultante da aplicação do método de "slabs" sobre uma malha estática, já construída.

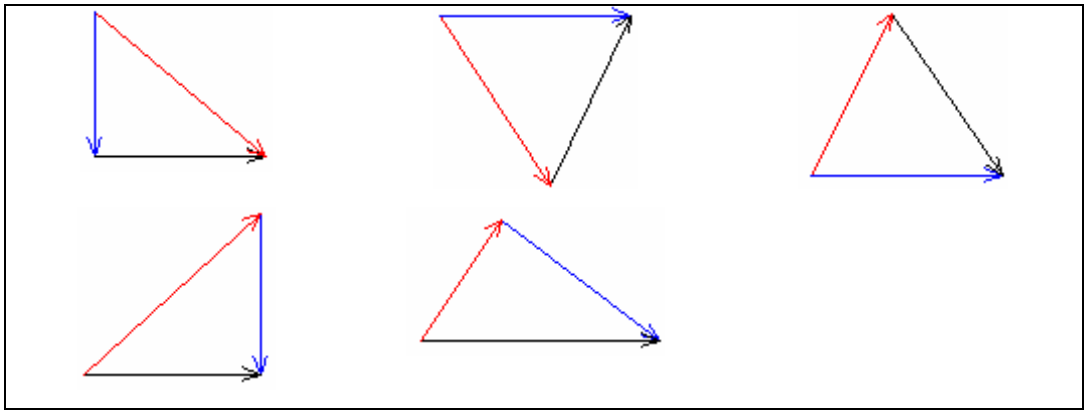


Figura 5.13. Orientações dos segmentos de alguns triângulos.

Os algoritmos são descritos em pseudocódigo, a partir da Figura 5.14. Em *Preprocess* (Figura 5.15) e *Query* (Figura 5.16), *k_indice* = -1 implica que não houve retorno de valor válido em pesquisa realizada na Multiárvore-B.

```
Algorithm Prepare
Input. A triangle T
Output. The triangle T with sorted segments
1.  Sort the endpoints of segments by x-coordinate, using the y-coordinate as a secondary key
2.  Sort all segments S of T by x-coordinate
3.  c ← centroid of the triangle T
4.  for i = 1 to 3 do
5.      cp ← cross_product(S[i].k_coord[0], S[i] .k_coord [1], S[i] .k_coord [2], S[i] .k_coord [3], c)
6.      if cp <= 0 then LeftFace[S[i].k_indice] ← T
7.      else RightFace[S[i].k_indice] ← T
8.      if LeftFace[S[i].k_indice] = -1 and RightFace[S[i].k_indice] = -1 then Preprocess(S[i].k_coord)
```

Figura 5.14. Algoritmo *Prepare*.

```
Algorithm Preprocess
Input. A segment S
Output. The interval multibtree as search structure for point location problem
1.  Insert S in the queue a_segment /* In event functions which there is intersection subsegments are inserted in queue too */
2.  while queue a_segment not empty do
3.      Update y_min and y_max coordinates /* y_min and y_max are used in event functions, as Event1, etc. */
4.      coord_min ← min_coordinate(a_segment)
5.      coord_max ← max_coordinate(a_segment)
6.      Find the node {coord_min.x, coord_max.x} in Upper Btree
7.      if exact value found then Event6()
8.      else if lower_bound.k_indice <> -1 then
9.          if coord_min.x = lower_bound.k_coord[0] and coord_max.x = lower_bound.k_coord[1] then Event7()
10.         else if coord_min.x = lower_bound.k_coord[0] and coord_max.x < lower_bound.k_coord[1] then Event8()
11.         else if coord_min.x = lower_bound.k_coord[1] and coord_max.x > lower_bound.k_coord[1] or
12.                 coord_min.x > lower_bound.k_coord[1] and coord_max.x > lower_bound.k_coord[1] then Event1()
13.         else if coord_min.x > lower_bound.k_coord[0] and coord_max.x > lower_bound.k_coord[1] or
14.                 coord_min.x > lower_bound.k_coord[0] and coord_max.x = lower_bound.k_coord[1] or
15.                 coord_min.x > lower_bound.k_coord[0] and coord_max.x < lower_bound.k_coord[1] then Event9()
16.     else if lower_bound.k_indice <> -1 and upper_bound.k_indice <> -1 then
17.         if lower_bound.tk_coord = upper_bound.tk_coord then
18.             if coord_min.x < upper_bound.k_coord[0] and coord_max.x < upper_bound.k_coord[0] or
19.                coord_min.x < upper_bound.k_coord[0] and coord_max.x = upper_bound.k_coord[0] then Event1()
20.                coord_min.x < upper_bound.k_coord[0] and coord_max.x < upper_bound.k_coord[1] or
21.                coord_min.x < upper_bound.k_coord[0] and coord_max.x = upper_bound.k_coord[1] or
22.                coord_min.x < upper_bound.k_coord[0] and coord_max.x > upper_bound.k_coord[1] then Event3()
23.             else if coord_min.x = upper_bound.k_coord[0] and coord_max.x < upper_bound.k_coord[1] then Event8()
24.         else if lower_bound.tk_coord < upper_bound.tk_coord then
25.             if coord_min.x >= lower_bound.k_coord[1] and coord_max.x <= upper_bound.k_coord[0] then Event1()
26.             else if coord_min.x = lower_bound.k_coord[0] and coord_max.x < lower_bound.k_coord[1] or
27.                 coord_min.x = upper_bound.k_coord[0] and coord_max.x < upper_bound.k_coord[1] then Event8()
28.             else /* The tests continue according to Figure 6.11 */
29.             ...
30.     else Event1()
```

Figura 5.15. Algoritmo *Preprocess*.

5.3.3 Consulta de Pontos

Dado um ponto de consulta *Q*, efetuam-se duas pesquisas na Multiárvore-B Intervalar. A primeira pesquisa, tendo como argumento a coordenada *x* de *Q*, ocorre na Árvore-B Superior e seleciona uma das árvores-B inferiores. A segunda pesquisa procura, na Árvore-B Inferior selecionada, segmento com coordenada *y* igual ou mais próxima à coordenada *y* de *Q*. Como todas as árvores-B são de ordem 5, o número de comparações para cada pesquisa é $O(\log_5 n)$. Ao final, um teste de orientação, fundamentado em produto misto, define o segmento de referência para identificação da face do PSLG em que se encontra o ponto de consulta *Q*. O algoritmo em pseudocódigo é descrito a seguir na Figura 5.16.

```
Algorithm Query Q
Input. A query point Q
Output. The face number which the point Q lie in or returns –1 if point lies outside the mesh
1.  Get parameters of the Upper Btree
2.  if Upper Btree not empty then
3.      Find node in format of {Q[0], 0.0}in Upper Btree
4.      if lower_bound.k_indice <> -1 and upper_bound.k_indice <> -1 then
5.          if lower_bound.tk_coord = upper_bound.tk_coord then slab ← upper_bound.k_indice
6.          else if lower_bound.k_coord < upper_bound.k_coord then
7.              if Q[0] > lower_bound.k_coord[0] and Q[0] < lower_bound.k_coord[1] then slab ← lower_bound.k_indice
8.              else if upper_bound.k_coord[0] <> upper_bound.k_coord[1] then slab ← upper_bound.k_indice
9.              else slab ← lower_bound.k_indice
10.         else if lower_bound.k_coord > upper_bound.k_coord then slab ← upper_bound.k_indice
11.     else if lower_bound.k_indice <> -1 then slab ← upper_bound.k_indice
12.     Get parameters of the Lower Btree
13.     Find node in format of {Q[1], 0.0, 0.0, 0.0}in Lower Btree
14.     Segment.k_indice ← -1
15.     if exact value found then Segment ← exact value
16.     else if lower_bound.k_indice <> -1 and upper_bound.k_indice <> -1 then
17.         if lower_bound.k_coord = upper_bound.k_coord then
18.             Segment ← upper_bound
19.         else if lower_bound.k_coord < upper_bound.k_coord then
20.             cp_lower_bound ← cross_product ({lower_bound.k_coord[1], lower_bound.k_coord[0]},
                                                {lower_bound.k_coord[2], lower_bound.k_coord[3]}, Q)
21.             cp_upper_bound ← cross_product ({upper_bound.k_coord[1], upper_bound.k_coord[0]},
                                                {upper_bound.k_coord[2], upper_bound.k_coord[3]}, Q)
22.             if cp_lower_bound <= 0.0 && cp_upper_bound > 0.0 then Segment ← lower_bound
23.             else if cp_lower_bound <= 0.0 then Segment ← lower_bound
24.             else Segment ← upper_bound
25.         else if lower_bound.k_coord > upper_bound.k_coord then Segment ← upper_bound
26.     else if lower_bound.k_indice <> -1 then Segment ← lower_bound
27.     else if upper_bound.k_indice then Segment ( upper_bound
28.     cp ( cross_product({Segment.k_coord[1], Segment.k_coord[0]}, {Segment.k_coord[2], Segment.k_coord[3]}, Q)
29.     if cp  <= 0.0 then
30.         if RightFace[Segment.k_indice] <> -1 then
31.             return RightFace[Segment.k_indice]
32.         else return LeftFace[Segment.k_indice]
33.     else if LeftFace[Segment.k_indice] <> -1 then
34.         return LeftFace[Segment.k_indice]
35.     else return RightFace[Segment.k_indice]
36. else return (-1)
```

Figura 5.16. Algoritmo Query.

5.4 Experimentos Computacionais

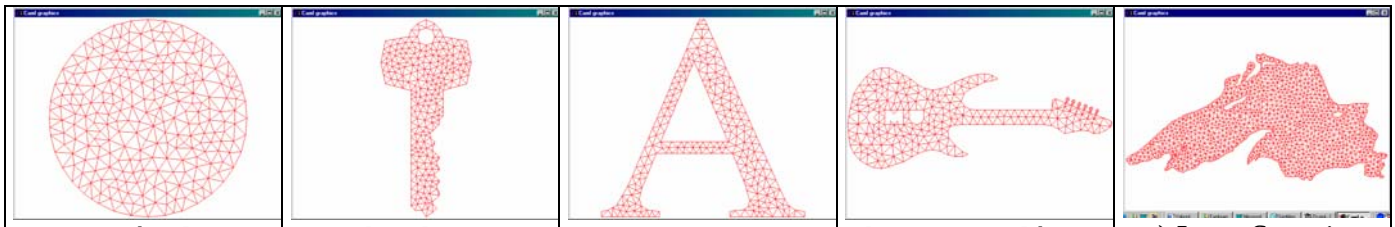
Os algoritmos foram implementados em OCaml, e os experimentos foram executados no modo interativo da linguagem, utilizando um PC com processador Pentium 4 de 2.8 GHz, 496 Mb de memória RAM e 256 Kb de memória Cache, sob o sistema operacional Windows.

O algoritmo de localização planar de pontos aceita, como entrada, qualquer tipo de malha. Preferiram-se malhas não-estruturadas construídas pelo gerador de malhas escrito em OCaml, a partir do algoritmo de Labelle (LABELLE; SHEWCHUK, 2003).

Nos experimentos, foram utilizados os domínios geométricos da Figura 5.17, cada um com malha de 250, 500, 1.000, 2.000 e 4.000 vértices, após a inserção dos pontos de Steiner.

Em todos os testes, o ponto de consulta foi o centróide de cada triângulo da malha, o qual foi previamente calculado e armazenado (Figura 5.14, linha 3) em um “array”. A exatidão dos resultados de localização do ponto de consulta, tanto com o uso de força bruta quanto com o de Multiárvore-B Intervalar, foi comprovada por um procedimento computacional. Nesse procedimento, para cada valor de centróide do “array”, efetuavam-se alternadamente consultas seqüencial e binária e se verificava se os resultados eram equivalentes. Não houve qualquer divergência. Utilizou-se uma Multiárvore-B de ordem 5, por ter-se mostrado mais eficiente.

Os resultados são apresentados nas Tabelas 5.1 a 5.5. Os custos com pré-processamento e espaço de armazenamento não se aplicam ao método da força bruta, pois, na prática, eles não são requeridos: a localização planar de pontos acontece diretamente na malha.



a) Círculo | b) Chave | c) Letra “A” | d) Guitarra elétrica | e) Lago Superior

Figura 5.17. Tipos de entradas para os experimentos.

Tabela 5.1. Resultados de Testes de Localização de Pontos na Malha do Círculo.

Informações da malha		**Tempo (em segundos)**			**Espaço (em número de elementos)**		
Vértices (originais + P. Steiner)	Triângulos	Pré-proces-samento	Pesquisa Força Bruta	Pesquisa M. Árvore-B Intervalar	M. Árvore-B Intervalar Superior	M. Árvore-B Intervalar Inferior	Total
250	448	0.60	**0.21**	**0.00**	246	7.788	8.034
500	924	2.36	**0.99**	**0.00**	496	22.382	22.878
1.000	1.901	11.15	**4.84**	**0.05**	996	62.924	63.920
2.000	3.833	63.61	**23.12**	**0.11**	1.984	175.670	177.654
4.000	7.796	540.69	**105.51**	**0.33**	3.984	507.390	511.374

Tabela 5.2. Resultados de Testes de Localização de Pontos na Malha da Chave.

Informações da malha		**Tempo (em segundos)**			**Espaço (em número de elementos)**		
Vértices (originais + P. Steiner)	Triângulos	Pré-proces-samento	Pesquisa Força Bruta	Pesquisa M. Árvore-B Intervalar	M. Árvore-B Intervalar Superior	M. Árvore-B Intervalar Inferior	Total
250	406	0.72	**0.22**	**0.00**	190	8.079	8.269
500	861	5.16	**0.88**	**0.05**	418	26.350	26.768
1.000	1.802	33.17	**4.39**	**0.06**	889	81.251	82.140
2.000	3.721	231.67	**20.81**	**0.11**	1.835	238.778	240.613
4.000	7.620	2667.57	**97.66**	**0.33**	3.795	701.639	705.434

Tabela 5.3. Resultados de Testes de Localização de Pontos na Malha da Letra “A”.

Informações da malha		**Tempo (em segundos)**			**Espaço (em número de elementos)**		
Vértices (originais + P. Steiner)	Triângulos	Pré-proces-samento	Pesquisa Força Bruta	Pesquisa M. Árvore-B Intervalar	M. Árvore-B Intervalar Superior	M. Árvore-B Intervalar Inferior	Total
250	365	0.27	**0.16**	**0.00**	245	4.851	5.096
500	797	1.15	**0.71**	**0.00**	494	14.470	14.964
1.000	1.690	4.56	**3.84**	**0.05**	982	40.090	41.072
2.000	3.581	21.03	**19.72**	**0.11**	1.969	115.033	117.002
4.000	7.347	124.29	**92.77**	**0.27**	3.946	326.398	330.344

Tabela 5.4. Resultados de Testes de Localização de Pontos na Malha da Guitarra elétrica.

Informações da malha		**Tempo (em segundos)**			**Espaço (em número de elementos)**		
Vértices (originais + P. Steiner)	Triângulos	Pré-proces-samento	Pesquisa Força Bruta	Pesquisa M. Árvore-B Intervalar	M. Árvore-B Intervalar Superior	M. Árvore-B Intervalar Inferior	Total
250	332	0.11	**0.11**	**0.00**	179	2.615	2.794
500	793	0.66	**0.77**	**0.00**	419	10.132	10.551
1.000	1.695	2.91	**4.29**	**0.05**	897	31.545	32.442
2.000	3.587	17.69	**19.88**	**0.11**	1.872	95.559	97.431
4.000	7.437	79.64	**98.70**	**0.33**	3.832	284.033	287.865

Tabela 5.5. Resultados de Testes de Localização de Pontos na Malha do Lago Superior.

Informações da malha		**Tempo (em segundos)**			**Espaço (em número de elementos)**		
Vértices (originais + P. Steiner)	Triângulos	Pré-proces-samento	Pesquisa Força Bruta	Pesquisa M. Árvore-B Intervalar	M. Árvore-B Intervalar Superior	M. Árvore-B Intervalar Inferior	Total
250	***	***	***	***	***	***	***
500	***	***	***	***	***	***	***
1.000	1.465	3.57	**2.74**	**0.05**	910	30.389	31.299
2.000	3.458	19.61	**19.72**	**0.11**	1.901	102.154	104.055
4.000	7.440	119.30	**98.42**	**0.33**	3.900	312.150	316.050

*** Testes não executados em razão de o total de vértices da malha ser inferior à quantidade (518) de vértices originais que formam a fronteira do PSLG.

Foram realizados testes com número maior de pontos, e o tempo de pesquisa na Multiárvore-B Intervalar pareceu ter crescimento linear. Por isso, considera-se importante substituir futuramente seu núcleo por outros tipos de árvores balanceadas, conforme sugerido no Capítulo 6.

O procedimento mais exaustivo do algoritmo de localização planar de pontos apresentado é o que trata a situação de o novo segmento cortar múltiplos segmentos já armazenados na árvore-B inferior. Nesse caso, executam-se operações de deletar, cortar verticalmente e reincluir cada um deles na árvore-B inferior.

Sobre os resultados, os experimentos com os PSLG da *Chave* e *Círculo* foram os que apresentaram o pior desempenho na fase de pré-processamento. Isso se deu em virtude de sua altura considerável. Em PSLG com tal característica, o novo segmento pode cortar

verticalmente uma grande quantidade de segmentos contidos na árvore-B inferior. Isso demonstra que o desempenho da fase de pré-processamento é influenciado pela geometria do domínio que estiver sendo computado. Dessa forma, a introdução de um teste simples no algoritmo para girar a figura que tiver a altura maior que a largura pode melhorar o desempenho do pré-processamento.

Na Tabela 5.6, é apresentada a distribuição dos eventos durante o pré-processamento de cada PSLG. O *Círculo* e a *Chave* produziram o maior número de eventos, ou seja, seus segmentos sofreram mais cortes que os segmentos dos demais domínios geométricos. O *Lago Superior* com 4.000 vértices que, mesmo tendo quase a mesma quantidade de eventos que a *Chave* com 2.000 vértices, apresentou melhor desempenho, comparável ao da *Guitarra Elétrica* e da *Letra* "A".

Tabela 5.6. Distribuição de eventos identificados na fase de pré-processamento.

PSLG	Vértices	Evento 1	Evento 3	Evento 6	Evento 7	Evento 8	Evento 9	Total
Círculo	2.000	1.944	40	59.362	457.528	1.246	669	520.789
Chave	2.000	1.892	35	78.402	837.508	1.156	603	919.596
Letra “A”	2.000	1.927	52	40.612	237.972	1.222	639	282.424
Guitarra elétrica	2.000	1.826	82	32.271	167.316	1.134	612	203.241
Lago Superior	2.000	1.858	60	36.415	194.467	1.154	611	234.565
Lago Superior	4.000	3.830	87	109.037	800.700	2.447	1.286	917.387

Capítulo VI

Conclusões

No Capítulo 2, foi destacado o problema da localização planar de pontos, um problema aberto à introdução de técnicas e estruturas de dados avançadas, para o qual será apresentada neste trabalho uma solução fundamentada em uma árvore balanceada intervalar.

Como mostrado no Capítulo 3, a instabilidade numérica do programa em C de (LEACH, 1992) para expressar com alta precisão números na representação ponto-flutuante provocou divergências nos resultados da triangulação do *círculo* e da *boca*. Isso ocorreu porque, com freqüência, algoritmos que tomam decisão baseada em testes geométricos falham quando os testes produzem respostas falsas provocadas por erro de arredondamento. Uma solução para esse problema de robustez é usar técnicas de aritmética exata como as descritas em (SHEWCHUK, 1996a). Por outro lado, elas são muito morosas e podem reduzir a velocidade da aplicação em até dez vezes, se não forem usadas técnicas adaptativas, propostas em (SHEWCHUK, 1997). Contudo, mesmo sem estar aparelhada com técnicas de *computação robusta*, a versão em OCaml mostrou-se numericamente mais estável, posto que menos eficiente que a versão em C.

Como se optou por não utilizar o estilo funcional na implementação de códigos em OCaml. Restou uma questão a ser respondida: que benefícios o estilo de programação funcional trará? Melhorará o tempo de resposta, o qual não foi bom, tanto na implementação do algoritmo de Triangulação de Delaunay quanto na implementação do gerador de malhas?

O gerador de malhas *Mesh*, desenvolvido por Labelle (LABELLE; SHEWCHUK, 2003) produz malhas de boa qualidade. Entretanto, o tempo que despende para computar a triangulação de Delaunay e localização planar de pontos torna-o proibitivo para processar malhas com elevada quantidade de elementos. Com a melhoria de sua complexidade temporal para $O(n \log n)$, *Mesh* poderá ser efetivamente utilizado em pré-processadores de elementos finitos, incluindo aplicações as mais diversas, como: diagnóstico médico, geologia, física, engenharias elétrica, civil, mecânica e química.

A implementação de *Triangle*, hoje largamente empregado em simulações numéricas em face de sua velocidade e precisão, não seria tão proveitosa quanto o foi a de *Mesh*. Por outro

lado, a prolixidade do código de *Triangle* induziu-nos a conduzir a pesquisa como descrito adiante.

Primeiro, foi implementado o algoritmo de triangulação de Delaunay por divisão-e-conquista de Guibas-Stolfi e descobriu-se que OCaml é numericamente mais robusta que C, sendo, portanto, recomendada para tratar problemas geométricos que requerem exatidão.

Segundo, a experiência obtida na implementação do algoritmo de Guibas-Stolfi foi determinante para o sucesso da codificação de *OCamlMesh*.

Terceiro, a descoberta de fraquezas potenciais de *Mesh* fez com que neste trabalho fosse tratado o problema de localização planar de pontos usando a *Multiárvore-B Intervalar* como estrutura de pesquisa. A partir do capítulo seguinte, discute-se sobre técnicas de localização planar de pontos utilizando a mencionada árvore balanceada.

A Multiárvore-B Intervalar demonstrou ser uma estrutura de pesquisa relativamente eficiente para o problema da localização planar de pontos. O baixo desempenho do pré-processamento não decorre de qualquer peculiaridade arquitetural ou algorítmica da estrutura de pesquisa, mas do método de subdivisão utilizado, baseado em “slabs”.

Em experimentos adicionais, constatou-se que a inserção de segmentos nas correspondentes “slabs”, sem explicitamente cortá-los, reduz, quase quadraticamente, o tempo de pré-processamento. Para tanto, o cálculo da interseção passa a ser feito na consulta de pontos, durante o percurso nas camadas inferiores da Multiárvore-B Intervalar, com pouco impacto sobre o tempo de pesquisa.

Como trabalho futuro, acredita-se que a substituição do método de subdivisão em “slabs” por outro método de menor ordem assintótica proverá ainda melhores resultados quanto aos requisitos de espaço e tempo de pré-processamento.

No desenvolvimento do trabalho, foi interessante perscrutar a infra-estrutura da malha, porque chamou a atenção para o problema da localização planar de pontos dinâmica. A Multiárvore-B Intervalar, um dos subprodutos da solução apresentada para o mencionado problema, teve bom desempenho nas consultas de localização planar de pontos. Entretanto, seria interessante implementar essa estrutura de árvores em camadas utilizando outros tipos árvores balanceadas, como: *árvore 2-3-4*, *árvore Red-black*, *árvore Kd-tree* e *árvore AVL*.

6.1 Trabalhos Futuros

No texto da tese, algumas questões deixaram de ser tratadas. Algumas que surgiram em um estágio avançado da pesquisa; outras que não figuravam como objeto principal de interesse. Cada uma das questões segue adiante como sugestões para futuros trabalhos.

A primeira das questões é o emprego do estilo de programação funcional para toda implementação em OCaml, que poderá tornar o código mais eficiente independentemente do algoritmo que estiver sendo implementado.

Uma questão importante é a efetuação de experimentos de localização planar de pontos comparando o desempenho da Multiárvore-B Intervalar com outros tipos de árvores balanceadas, como *árvore 2-3-4*, *árvore Red-black*, *árvore Kd-tree* e *árvore AVL*, todas organizadas em camadas e incrementadas com mecanismo de pesquisa intervalar.

Outra questão bastante significativa é o desenvolvimento de técnica de subdivisão do espaço de ordem assintótica melhor que a do método das “Slabs” para minimizar o tempo de pré-processamento no algoritmo dinâmico de localização planar de pontos.

Outra questão, cuja avaliação foi positiva, é a paralelização dos procedimentos de triangulação de Delaunay e da geração de malhas para melhoria de desempenho.

Uma questão final é a construção de ferramenta de MEF totalmente em OCaml, incluindo, a exemplo do programa escrito por David Meeker (MEEKER, 2003), recursos e editor gráficos para modelar domínios geométricos bidimensionais e também tridimensionais, bem como cobertura às etapas de processamento e pós-processamento.

Referências Bibliográficas

BERG, MARK DE *et al.* **Computational Geometry: Algorithms and Applications**. 2. ed. Berlin: Springer, 1997.367 p.

BERN, MARSHAL. Adaptive Mesh Generation. *Error Estimation and Adaptive Discretization Methods in Computational Fluid Dynamics*, T. Barth; H. Deconinck (editores), Springer-Verlag, Heidelberg, 2002, páginas 1-46.

BERN, MARSHAL. DEPARTMENT OF INFORMATION AND COMPUTER SCIENCE, UNIVERSITY OF CALIFORNIA **Faster Circle Packing with Application to Nonobtuse Triangulation.** Irvine, 1994. 7p. Relatório Técnico 94-33.

BERN, MARSHAL; EPPSTEIN, DAVID. Mesh Generation and Optimal Triangulation. Computing in Euclidean Geometry, Ding-Zhun Du; Frank Hwang, editores, Lecture Notes Series on Computing. **World Scientific**. Singapore, v. 1, p. 23-90, 1992.

BERN, MARSHAL; EPPSTEIN, DAVID. Quadrilateral Meshing by Circle Packing. In: SIXTH INTERNATIONAL MESHING ROUNDTABLE, 1997, Park City, p. 7-19.

BERN, MARSHAL; EPPSTEIN, DAVID; GILBERT, JOHN R. Provably Good Mesh Generation. **Journal of Computer and System Sciences.** [s.l.], vol. 48, n. 3, p. 384-409, jun. 1994.

BERN, MARSHAL; MITCHELL, SCOTT; RUPPERT, J. Linear-size Nonobtuse Triangulation of Polygons. In: PROC. 10TH SYMP. COMP. GEOM, 1994. **Anais da ACM**, 1994. p. 221-230.

BERN, MARSHAL; PLASSMAN, P. **Mesh Generation**. Handbook of Computational Geometry. 1 ed. [s.l.]: North-Holland, 2000.38 p.

CHEN, MIN-BIN; CHUANG, TYNG-RUEY; WU, JAN-JAN. "A Parallel Divide-and-Conquer Scheme for Delaunay Triangulation. In: 9TH INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED SYSTEMS, págs. 571-576, **IEEE Computer Society** Press, Chungli, Taiwan, Dezembro 2002.

CHERNIKOV, ANDREY; CHRISOCHOIDES; NIKOS. Parallel 2D Constrained Delaunay Mesh Generation. **ACM Transactions on Mathematical Software**, em revisão, 2006.

CHEW, L. PAUL. DEPARTAMENT OF COMPUTER SCIENCE, CORNELL UNIVERSITY. **Guaranteed-Quality Triangular Meshes.** Ithaca, 1989. 18p. Relatório Técnico 89-983.

DIVERIO, TIARAJÚ ASMUZ. **Teoria da Computação: Máquinas Universais e Computabilidade.** 1. ed. Porto Alegre: Editora Sagra Luzzatto, 1999.205 p.

DWYER, REX A. A Simple Divide-and-Conquer Algorithm for Computing Delaunay Triangulations in $O(n \log \log n)$ Expected Time. In: SYMPOSIUM ON COMPUTATIONAL GEOMETRY, 1986. p. 276-284.

EDAHIRO, MASATO; KOKUBO, IWAO; ASANO, TAKAO. A New Point Location Algorithm and Its Practical Efficiency – Comparison with Existing Algorithms. **ACM Transactions on Graphics.** [s.l.], v. 3, n. 2, p. 86-109, abr. 1984.

FIGUEIREDO, L. H. de; CARVALHO, P. C. P. Introdução à Geometria Computacional. In: XVII COLÓQUIO BRASILEIRO DE MATEMÁTICA, IMPA, 1991.

GUIBAS, LEONIDAS J.; STOLFI, JORGE. Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams. **ACM Transactions on Graphics.** [s.l.], vol. 4, n. 2, p. 74-123, abr. 1985.

HARDWICK, JONATHAN C. Implementation and Evaluation of an Efficient 2d Parallel Delaunay Triangulation Algorithm. In: PROCEEDINGS OF THE 9TH ANNUAL ACM SYMPOSIUM ON PARALLEL ALGORITHMS AND ARCHITECTURES, **ACM**, Vol 15, Junho 1997.

INRIA. Pointers in OCaml. INRIA, 2000, Disponível em: <http://caml.inria.fr/resources/doc/guides/pointers.en.html>. Acesso em: 5 abr. 2004.

JOE, B. GEOMPACK – A Software Package for the Generation of Meshes Using Geometric Algorithms. **Advances in Engineering Software.** [s.l.], v. 56, n. 13, p. 325-331, 1991.

KALLMANN, M.; BIERI, H.; THALMANN, D. Fully Dynamic Constrained Delaunay Triangulations, In: GEOMETRIC MODELLING FOR SCIENTIFIC VISUALIZATION, Heidelberg, Germany, G. Brunnett, B. Hamann, H. Mueller (Editores), ISBN 3-540-40116-4, **Springer-Verlag**, Heidelberg, Germany, pp. 241-257, 2003.

KARAMETE, B. KAAN. Mesh2d, Scientific Computational Research Center, SCOREC. Disponível em: <http://home.nycap.rr.com/kaan/>. Acesso em: 8 jul. 2005.

KNUTH, DONALD ERVIN. **The Art of Computer Programming: Sorting and Searching.** 2. ed. USA: Addison Wesley, 1998.780 p.

KOHOUT, JOSEF. DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING, UNIVERSITY OF WEST BOHEMIA IN PILSEN. **Delaunay Triangulation in Parallel and Distributed Environment.** Pilsen, Czech Republic, 2004, 101p. Relatório Técnico DCSE/TR-2004-02.

KOHOUT, JOSEF. **Delaunay Triangulation in Parallel and Distributed Environment**, 2004, 73 f. Tese (Doutorado em Ciência da Computação) – University of West Bohemia in Pilsen, Department of Computer Science and Engineering, Univerzitni 8, 30614 Pilsen, Czech Republic, Março, 2004.

LABELLE, FRANCOIS; SHEWCHUK, J. R. Anisotropic Voronoi Diagrams and Guaranteed-Quality Anisotropic Mesh Generation. In: PROCEEDINGS OF THE 19TH ACM SYMPOSIUM ON COMPUTATIONAL GEOMETRY, 8-10 de Junho, 2003, San Diego, CA, USA. **Anais da ACM**, 2003.

LEACH, GEOFF. Department of Computer Science, Royal Melbourne Institute of Technology. **Improving Worst-Case Optimal Delaunay Triangulation Algorithms**. Melbourne, 1992. 7p. **Manuscrito não publicado.**

LEE, S.; PARK, C.; PARK, C. An improved parallel algorithm for delaunay triangulation on distributed memory parallel computers. **Parallel Processing Letters**, 11:341--352, 2001.

LEROY, Xavier. The Objective Caml System. INRIA. 1996. Disponível em <http://pauillac.inria.fr/ocaml/>. Acesso em: 13 mar. 2003.

LIN, HSUAN-CHENG. **Javamesh - A Two Dimensional Triangular Mesh Generator for Finite Elements**, 1997, 158 f. Dissertação (Mestrado em Engenharia Civil) – University of Pittsburgh, Pittsburgh.

MEEKER, DAVID. Finite Element Method Magnetics. Disponível em: <http://femm.foster-miller.net/>. Acesso em: 12 mar. 2003.

MERRIAM, MARSHAL L. NASA, AMES RESEARCH CENTER, **Parallel Implementation of an Algorithm for Delaunay Triangulation**, Moffett Field, California, 1992. 14p. Memorando Técnico 103951.

MESHING SOFTWARE SURVEY. Meshing Research Corner. Carnegie Mellon University. Disponível em: <www.andrew.cmu.edu/user/sowen/software/triangle.html>. Acesso em: 5 jul. 2005.

MOORE, DOUGLAS WILLIAM. **Simplical Mesh Generation with Applications**, 1992, 116 f. Tese (Doutorado em Ciência da Computação) – Faculty of the Graduate School of Cornell University, Ithaca.

MOUNT, DAVID M. DEPARTAMENT OF COMPUTER SCIENCE, UNIVERSITY OF MARYLAND. **Computational Geometry.** Maryland, 2002. 122p. Lecture Notes.

MOURA, ANDRÉ LUIZ; CAMACHO, JOSÉ ROBERTO; GUIMARÃES JR., SEBASTIÃO CAMARGO; SALERNO, CARLOS HENRIQUE. A functional language to implement the divide-and-conquer Delaunay triangulation algorithm. **Applied Mathematics and Computation**, Elsevier, The Netherlands, v. 168, n. 1, p. 178-191, 2005.

MOURA, ANDRÉ LUIZ; CAMACHO, JOSÉ ROBERTO; GUIMARÃES JR, SEBASTIÃO CAMARGO. Localização Planar de Pontos Dinâmica baseada em Eventos sobre Slabs. In: PROCEEDINGS DO 7º CBMAG – CONGRESSO BRASILEIRO DE ELETROMAGNETISMO, 7-10 de Agosto de 2006, Belo Horizonte. Anais do Momag2006, Belo Horizonte, UFMG, 2006.

MUCKE, E; SAIAS, I.; SHU, B. Fast Randomized Point Location without Preprocessing in two- and three-dimensional Delaunay Triangulations. In: PROC. 12TH ACM SYMPOSIUM ON COMPUTATIONAL GEOMETRY, 1996. p. 274-283.

NICENO, BOJAN. **EasyMesh.** University of Trieste, D.I.N.M.A. Disponível em <http://www-dinma.univ.trieste.it/~nirftc/research/easymesh/>. Acesso em: 17 jul. 2005.

O'ROURKE, JOSEPH. **Computational Geometry in C.** 2. ed. USA: Cambridge University Press, 1998.376 p.

PAOLINI, MAURIZIO. TMG (Triangular Mesh Generator). Istituto di Analisi Numerica (CNR) of Pavia, Dipartimento di Matematica, University of Milano. Disponível em: http://www.dmf.bs.unicatt.it/~paolini/tmg/. Acesso em 4 abr. 2004.

REMACLE, JEAN-FRANÇOIS. GMSH. Ecole Polytechnique de Montreal & University of Liege. Liege. Disponível em: <http://www.meca.polymtl.ca/~remacle/Mesh.html>. Acesso em: 6 jul. 2005.

RUPPERT, J. A Delaunay Refinement Algorithm for Quality 2Dimensional Mesh Generation. **Journal of Algorithms.** [s.l.], vol. 18, n. 3, p. 548-585, mai. 1995.

SEDGEWICK, ROBERT. **Algorithms in C.** 3. ed. [s.l.]: Addison-Wesley Publishing Company, Inc., 1998.

SHEWCHUK, J. R. Robust Adaptive Floating-Point Geometric Predicates. In: PROCEEDING OF THE TWELFTH ANNUAL SYMPOSIUM ON COMPUTATION GEOMETRY, Maio de 1996. **Anais da Association for Computing Machinery**, 1996.

SHEWCHUK, J. R. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulation. In: FIRST WORKSHOP ON APPLIED COMPUTATIONAL GEOMETRY, Maio de 1996, Philadelphia, Pennsylvania. **Anais da ACM**, 1996. p. 124-133.

SHEWCHUK, J. R. **Delaunay Refinement Mesh Generation**, 1997, 207 f. Tese (Doutorado em Ciência da Computação) – School of Computer Science, Computer Science Department, Carnegie Mellon University, Pittsburgh.

SHEWCHUK, J. R. DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE, UNIVERSITY OF CALIFORNIA AT BERKELEY. **Delaunay Mesh Generation**. Berkeley, 1999, 115p. Lecture Notes.

SHEWCHUK, J. R. DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES, UNIVERSITY OF CALIFORNIA AT BERKELEY. **What is a Good Linear Finite Element? Interpolation, Conditioning, Anisotropy, and Quality Measures.** Berkeley, 2002. 66p. Relatório Técnico.

SLEATOR, DANIEL DOMINIC; TARJAN, ROBERT ENDRE. Self-Adjusting Binary Search Trees. **Journal of the Association for Computing Machinery.** [s.l.], v. 32, n. 3, p. 652-686, jul. 1985.

SU, PETER. **Efficient Parallel Algorithms for Closest Point Problems**, 1994, 138 f. Tese (Doutorado em Ciência da Computação) – Dartmouth College, Hanover.

SU, PETER; DRYSDALE, ROBERT L. SCOT. A Comparison of Sequential Delaunay Triangulation Algorithms. In: PROCEEDINGS OF THE ELEVENTH ANNUAL SYMPOSIUM ON COMPUTATIONAL GEOMETRY, junho de 1995, Vancouver, British Columbia, Canada. **Anais da Association for Computing Machinery**, 1995. p. 61-70.

TENG, SHANG-HUA. Unstructured Mesh Generation: Theory, Practice, and Perspective. **International Journal of Computational Geometry & Applications**. [s.l.], World Scientific Publishing Company, jan. 1999.

THE COMPUTER LANGUAGE SHOOTOUT BENCHMARKS, 2006. Disponível em: <http://shootout.alioth.debian.org/>. Acesso em: 25 abr. 2006.

WALKER, ROBERT J.; SNOEYNK, JACK. DEPARTMENT OF COMPUTER SCIENCE, UNIVERSITY OF BRITISH COLUMBIA. **Practical Point-in-Polygon Tests Using CSG Representations of Polygons**. Vancouver, Canada, 1999. 10p. Relatório Técnico TR-99-12.