GeoLab – Um Ambiente para Desenvolvimento de Algoritmos em Geometria Computacional

Este exemplar corresponde à redação final da tese devidamente corrigida e defendida pelo Sr. Welson Régis Jacometti e aprovada pela Comissão Julgadora.

Campinas, 18 de Dezembro de 1992

Prof. Dr. Pedro Jussieu de Rezende

Departamento de Ciência da Computação
Instituto de Matemática, Estatística e Ciência da Computação
Universidade Estadual de Campinas

GeoLab – Um Ambiente para Desenvolvimento de Algoritmos em Geometria Computacional

Welson Régis Jacometti[1]
Novembro, 1992

Dissertação submetida ao Departamento de Ciência da Computação
Universidade Estadual de Campinas, como requisito parcial para
a obtenção do título de Mestre em Ciência da Computação

[1]Bolsista do Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) no período de Março de 1990 a Fevereiro de 1992 sob nº 130767/90 5; bolsista da Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP) no período de Março a Julho de 1992 sob nº 91/4497-6.

Prefácio

Esta tese discute o projeto e a implementação de um ambiente de programação voltado para o desenvolvimento de algoritmos e estruturas de dados em Geometria Computacional chamado **GeoLab**.

Este ambiente provê o suporte necessário para a implementação de algoritmos geométricos através de uma biblioteca de objetos geométricos básicos e de um conjunto significativo de algoritmos fundamentais como algoritmos para construção de envoltórias convexas, diagramas de Voronoi, árvores espalhadas mínimas, etc. Tanto o conjunto de objetos básicos como o de algoritmos fundamentais podem ser ampliados através de mecanismos do ambiente que permitem que novas construções sejam incorporadas dinamicamente.

O ambiente dispõe ainda de ferramentas que permitem a inclusão dinâmica de novos modos de operação junto à sua interface, complementando a gama de facilidades para o desenho de novos objetos (ou variações para os objetos já existentes) e a implementação de algoritmos geométricos dinâmicos ou que realizam preprocessamento.

Objetos e algoritmos geométricos são mapeados em classes em C++ que especificam protocolos para sua manipulação pelo ambiente. Novas entidades interagem homogeneamente com o ambiente satisfazendo estes protocolos.

Algoritmos geométricos podem ser animados. Animação *per se* é conseguida graças à introdução de código adicional no corpo dos algoritmos, através do qual o ambiente permite controlar a velocidade e o nível de detalhes de uma animação. Existe também um outro tipo de animação, chamado *dynamic move*, conseguida através de um dos modos funcionais do **GeoLab** e que consiste na execução repetitiva de um algoritmo enquanto sua entrada sofre modificações.

Agradecimentos

Esta tese teve a colaboração direta e indireta de várias pessoas, às quais gostaria de externar meu sincero agradecimento.

A Pedro Jussieu de Rezende, pela fantástica orientação (e, admito, pela cobrança). Aos companheiros de **GeoLab**, César Nivaldo Gon e Laerte Ferreira Morgado, aos quais devo não só agradecimentos mas também desculpas pelas dores de cabeça que causei, cada vez que era obrigado a mudar alguma coisa no projeto. Meus agradecimentos também a Eduardo Aguiar Patrocínio e Rackel Valadares Amorim.

Ao amigo Daniel Tavares Correia Xavier, por ter me socorrido durante o aprendizado de C++ e pelas discussões sobre o XView.

Meus sinceros agradecimentos também a Ana Cláudia Biazetti, Alexandre Prado Teles e Carlos Alberto Furuti, por colaborarem indiretamente com meu trabalho através da manutenção de algumas ferramentas das quais eu dependia.

A Antônio Carlos Raposo, por ter me levado a praticar pára-quedismo e a conhecer todas aquelas pessoas fantásticas da escola Azul do Vento. As horas de lazer que lá passei em muito me ajudaram.

A meus pais, Pierina Arnosti Jacometti e Antônio Jacometti, pelo apoio que só eles poderiam ter me dado.

A minha esposa, Gláucia Cristiana de Mattias Jacometti, pelo apoio incondicional, incentivo, paciência e amor que me dedicou nestes anos todos. Esta tese não teria se concretizado sem sua ajuda.

Ao amigo Cláudio Sérgio da Rós Carvalho, que nos deixou abreviadamente, com muitas saudades.

Finalmente, gostaria de agradecer a ajuda financeira recebida ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) e à Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP). O governo brasileiro deveria, para o bem do povo brasileiro, olhar com mais cuidado para estas e outras instituições de fomento à pesquisa.

Este trabalho é dedicado a
Gláucia Cristiana de Mattias Jacometti.

Conteúdo

Prefácio iii

Agradecimentos v

1 Introdução 1
1.1 Geometria Computacional e Ambientes de Programação 1
1.2 Trabalhos Relacionados 2
1.2.1 LineTool 2
1.2.2 WOGC 3
1.2.3 XYZ Geobench 3
1.2.4 LEDA 3
1.3 Requisitos para um Ambiente de Desenvolvimento 4
1.4 Estrutura da Tese 5

2 Apresentando o GeoLab 7
2.1 Interface Gráfica 8
2.1.1 O Editor e os Modos de Operação 9
2.1.2 Ferramentas para Tratamento de Modelos Geométricos 11
2.1.2.1 Ferramentas Genéricas e Extensões aos Modos de Operação 11
2.1.2.2 Geradores de Entrada 11
2.1.2.3 Gerenciadores de Algoritmos Geométricos 12
2.1.3 Composição de Algoritmos 16
2.2 Suporte para Programação 18
2.2.1 Programação Orientada a Objetos 18
2.2.2 Abstrações Voltadas para Programação 19
2.2.2.1 Sistema de Visualização 19
2.2.2.2 Objeto Geométrico 20
2.2.2.3 Algoritmo 23

3 Programando através do GeoLab 27
3.1 Algoritmos Geométricos sob o **GeoLab** 27
3.1.1 Relacionamento com Objetos Geométricos 28
3.1.2 Bibliotecas de Algoritmos Geométricos 30
3.1.3 Ligação de Algoritmos Geométricos 30
3.1.3.1 O Arquivo *.geolab-menu* 30

3.1.3.2 Protocolo de Comunicação para Algoritmos Geométricos 32
3.1.4 Consumo e Produção de Objetos Geométricos 33
3.1.4.1 Obtendo Entradas para Algoritmos Geométricos 34
3.1.4.2 Conversões entre Representações 35
3.1.4.3 A Saída de um Algoritmo Geométrico 37
3.2 Incorporando Novos Objetos Geométricos 37
3.2.1 O Papel da Dupla Hierarquia 37
3.2.2 Armazenamento e Recuperação de Novos Objetos 37
3.2.2.1 Instanciação de Objetos Geométricos Externos 37
3.2.2.2 O Papel dos Conversores entre Representações 38
3.3 Modos de Criação Interativa e Modos Funcionais 38
3.3.1 Encadeamento de Tratadores de Eventos 39
3.3.2 Modos de Criação Interativa de Objetos Geométricos 42
3.3.3 Modos Funcionais 43
3.3.4 Recomendações 43

4 Algoritmos Geométricos 45
4.1 Força Bruta *versus* Algoritmos Ótimos 48
4.2 Robustez em Implementações 50
4.3 Algoritmos para Conjuntos de Pontos 50
4.3.1 Envoltórias Convexas 50
4.3.1.1 Varredura de Graham 52
4.3.1.2 Marcha de Jarvis 52
4.3.1.3 Cascas Convexas 56
4.3.2 Problemas de Proximidade 57
4.3.2.1 Diâmetro de um Conjunto de Pontos 57
4.3.2.2 Cálculo do Par mais Próximo 58
4.3.2.3 Todos os Vizinhos mais Próximos 60
4.3.2.4 Árvore Espalhada Euclidiana Mínima (AEEM) 61
4.3.3 A Construção de Diagramas de Voronoi 64
4.3.3.1 O algoritmo de Guibas e Stolfi 66
4.3.3.2 Diagrama de Voronoi de Vizinho mais Distante 67
4.3.3.3 Aproximações para Objetos Genéricos 72
4.3.4 Círculos Vazios e Envolventes 73
4.3.4.1 Menor Círculo Envolvente – MCE 73
4.3.4.2 Maior Círculo Vazio - MCV 77
4.4 Polígonos 77
4.4.1 *Winding Number* 77
4.4.2 Núcleo de um Polígono Simples 78
4.4.2.1 Conceitos e Nomeclatura 79
4.4.2.2 O Algoritmo 80
4.4.2.3 Implementação 83
4.4.3 Pares Antipodais 84
4.4.4 Envoltória Convexa de Polígonos Simples 85

4.4.4.1 Notações e Conceitos 85
4.4.4.2 O Algoritmo 87
4.4.4.3 Implementação 89
4.5 Suporte para Criação de Ilustrações 91

5 Conclusão **93**
5.1 Ferramentas Utilizadas na Construção do **GeoLab** 93
5.1.1 Ambiente Hospedeiro 93
5.1.2 Orientação a Objetos 94
5.2 Trabalhos Futuros 94
5.2.1 Composição de Algoritmos 94
5.2.2 Bibliotecas de Estruturas de Dados e Algoritmos de Uso Geral 94
5.2.3 Geradores de Entrada 95
5.2.4 Tolerância a Falhas e Coleta de Lixo 95
5.2.5 Geometria Espacial, Projetiva e Suporte para Diferentes Métricas 95

A Interface Gráfica **99**
A.1 Editor e Modos de Operação 99
A.1.1 Desenho de Objetos Geométricos 100
A.1.2 Seleção de Objetos 101
A.1.3 Translação e Cópia de Objetos 102
A.1.4 Redimensionamento de Objetos 102
A.1.5 *Zoom* 103
A.1.6 *Scroll* 104
A.1.7 Animação por Movimentação Dinâmica 104
A.1.8 Modos de Construção Interativa e Modos Funcionais Externos 105
A.1.9 Acesso a Ferramentas via Seqüências de Teclas 105
A.2 Manipulação de Arquivos 106
A.3 Manipulação da Área de Visualização 108
A.3.1 Alças 108
A.3.2 Posição Relativa de Objetos 108
A.3.3 Parametrização de *Zoom* 108
A.3.4 Painel de Controle para Animação de Algoritmos 109
A.4 Atributos de Objetos Geométricos 110
A.5 Geradores de Entrada 111
A.5.1 Geradores de Pontos Randômicos 111
A.5.2 Cobertura para Objetos Geométricos 111
A.5.3 Geradores de Polígonos Randômicos 113
A.5.4 Geradores Internos *versus* Geradores Externos 114
A.6 Algoritmos Geométricos 114
A.6.1 Seleção de Objetos e Opções Disponíveis 116
A.6.2 Configuração para Algoritmos Externos 116
A.6.3 Controle do Tempo de Execução de um Algoritmo 117
A.6.4 Controle de Memória Alocada por um Algoritmo 118

A.7 Composição de Algoritmos 118

B Interface de Programação **120**

B.1 Algoritmos Geométricos 121
B.1.1 Hierarquia de Algoritmos e Protocolo de Comunicação 121
B.1.2 Aspectos de Programação de Animações de Algoritmos 122
B.1.3 Exemplo Prático 124
B.1.4 Biblioteca de Algoritmos Geométricos 130
B.2 Objetos Geométricos 131
B.2.1 Hierarquia de Objetos 131
B.2.2 Suporte à Programação 132
B.2.2.1 Protocolos para Objetos Geométricos 132
B.2.2.2 Introdução de Novos Objetos Geométricos 138
B.3 Modos de Criação Interativa e Modos Funcionais Externos 138
B.3.1 Tratamento de Eventos 138
B.3.2 Interação com o Sistema de Visualização 139
B.3.3 Modos de Construção Interativa 140
B.3.3.1 Manipulação da Lista de Objetos Geométricos 140
B.3.3.2 Recomendações 142
B.3.4 Modos Funcionais 142
B.3.4.1 Sugestões para Algoritmos Geométricos com Preprocessamento 142
B.3.4.2 Recomendações 142
B.4 O Arquivo *.geolab-menu* 142
B.4.1 Variáveis de Ambiente 144
B.4.1.1 Diferentes Ambientes de Trabalho 144
B.4.1.2 Localização de Algoritmos Geométricos no Sistema de Arquivos . . . 144
B.5 Criação de *Shared Libraries* 144
B.5.1 Shared Libraries de Uso Geral 145
B.5.2 Limitações Impostas pelo Sistema 145
B.6 Manutenção do Ambiente 145
B.6.1 Documentação Básica e *Header Files* 149
B.6.1.1 `World.doc` 149
B.6.1.2 `Obj.h` 159
B.6.1.3 `Algorithm.h` 171

Bibliografia **175**

Lista de Tabelas

2.1 Protocolo de mensagens para objetos geométricos "puros" 23
2.2 Protocolo de mensagens para objetos geométricos "gráficos" 24
2.3 Resumo de primitivas geométricas para objetos do tipo Ponto em duas dimensões . . . 25
2.4 Protocolo de mensagens para Algoritmos Geométricos 25

A.1 Variações possíveis para os modos de criação interativa de objetos básicos 101
A.2 Variações possíveis no modo de seleção de objetos 102
A.3 Variações possíveis no modo de translação e cópia de objetos 102
A.4 Variações possíveis no modo de *Zoom* 103
A.5 Variações possíveis no modo de *Scroll* 104
A.6 Variações possíveis na manipulação de conjuntos de entrada para o modo de movimentação dinâmica 105
A.7 Seqüências de teclas para acesso direto a ferramentas da interface 106

B.1 Protocolo de mensagens para Algoritmos Geométricos 122
B.2 Resumo de primitivas geométricas para objetos do tipo Ponto em duas dimensões . . . 133
B.3 Protocolo de mensagens para objetos geométricos "puros" 134
B.4 Protocolo de mensagens para objetos geométricos "gráficos" 136

Lista de Figuras

2.1 Estrutura do **GeoLab** 7
2.2 Interface Gráfica do Ambiente 8
2.3 Modos de Operação do Editor 9
2.4 Botões de acesso às ferramentas, algumas sub-janelas e menus 11
2.5 Coberturas de pontos e pontos randômicos internos a objetos 12
2.6 Um polígono simples com 200 vértices 13
2.7 Um diagrama de Voronoi sofrendo *Dynamic Move* 13
2.8 Animação de um passo genérico no cálculo da triangulação de Delaunay 14
2.9 Mais detalhes na animação da figura anterior 14
2.10 Painel de controle de animação 15
2.11 Gráfico de desempenho para algoritmos que calculam Menor Círculo Envolvente . . . 16
2.12 Cálculo do diâmetro de um conjunto de pontos através de composição de algoritmos . 17
2.13 Parte da dupla hierarquia de objeto geométricos do **GeoLab** 20
2.14 A relação entre a dupla hierarquia de objetos e os algoritmos geométricos – buscando soluções que permitam a criação de bibliotecas genéricas 21

3.1 A posição de um ponto com relação a uma reta direcionada 28
3.2 Fragmento de algoritmo geométrico com destaque para parâmetros inicializados e código para animação isolado (compilação condicional) 31
3.3 Arquivo *.geolab-menu* e o correspondente menu de opções 32
3.4 Exemplo de implementação do protocolo para algoritmos geométricos 34
3.5 Listas de objetos manipulados pelo **GeoLab** 35
3.6 Esquema conceitual dos conversores entre diferentes representações de um mesmo objeto geométrico 36
3.7 Fluxo de controle em uma aplicação construída sobre o XView 40
3.8 Fluxo de controle no **GeoLab** – extensões ao XView 41
3.9 Um desenhador de subdivisões planares como modo de criação interativa externo . . . 42
3.10 Implementação do localizador de pontos através de modos funcionais 43

4.1 Implementação de um algoritmo força bruta para o problema do par de pontos mais próximos 49
4.2 Implementação de um algoritmo ótimo para o problema do par mais próximo (através da triangulação de Delaunay) 49
4.3 Envoltória convexa de um conjunto de pontos 51

4.4 Mapeamento de valores sobre a parábola $y = x^2$ para realização da redução do problema da ordenação ao problema da construção da envoltória convexa 51
4.5 Alguns passos na construção da envoltória convexa através da Varredura de Graham . 52
4.6 Implementação da Varredura de Graham 53
4.7 Passos da execução da Marcha de Jarvis 54
4.8 Determinação da inclinação relativa entre duas retas em função do posicionamento dos pontos que as determinam 54
4.9 Implementação da Marcha de Jarvis 55
4.10 Comparação entre a Varredura de Graham e a Marcha de Jarvis 56
4.11 Cascas convexas de um conjunto de pontos 56
4.12 Comparação entre dois algoritmos não ótimos para construção de cascas convexas . . . 57
4.13 Mapeamento dos conjuntos A e B sobre um círculo unitário C 58
4.14 Implementação de um algoritmo ótimo para o cálculo do $diam(S)$ 58
4.15 Busca do par mais próximo através de um algoritmo de divisão-e-conquista 59
4.16 Número máximo de candidatos constante para cada ponto em F_1 60
4.17 Comparação entre três implementações para o problema do par mais próximo 61
4.18 Implementação de um algoritmo ótimo para o problema de todos os vizinhos mais próximos que utiliza a triangulação de Delaunay 62
4.19 Par mais próximo e todos os vizinhos mais próximos para um conjunto de pontos . . . 62
4.20 Árvore Espalhada Euclidiana Mínima de um conjunto de pontos no plano 63
4.21 Implementação do algoritmo para Árvore Espalhada Euclidiana Mínima 65
4.22 Diagrama de Voronoi (sólido) e a triangulação de Delaunay (tracejado) de um conjunto de pontos 66
4.23 Unidade básica de informação de uma *QuadEdge* 67
4.24 Relações de adjacência para arestas de uma *QuadEdge* 67
4.25 Funcionamento dos operadores *MakeEdge* e *Splice* 68
4.26 Implementação da estrutura *QuadEdge* 69
4.27 Diagrama de Voronoi de vizinho mais distante (e a triangulação dual) de um conjunto de pontos 70
4.28 Simulação de um diagrama de Voronoi para um conjunto de segmentos 72
4.29 Simulação de um diagrama de Voronoi para um conjunto de segmentos, círculos, elipses e retângulos 73
4.30 Menor Círculo Envolvente de um conjunto de pontos 74
4.31 Contra exemplo para a afirmação de que o diâmetro de um conjunto de pontos determina uma aresta no dual do diagrama de Voronoi de vizinho mais distante 74
4.32 Implementação do algoritmo de Shamos para o problema MCE 75
4.33 Implementação do algoritmo de Bhattacharya e Toussaint para o problema MCE . . . 76
4.34 Maior Círculo Vazio de um conjunto de pontos 77
4.35 Implementação do algoritmo para o cálculo do MCV de um conjunto de pontos 78
4.36 Implementação do algoritmo para o cálculo do *winding number* 79
4.37 Núcleo de um polígono simples 79
4.38 Ilustração da definição de F_i e L_i 80
4.39 Inicialização de K_1 81
4.40 Passo geral quando v_i é côncavo 82

4.41 Passo geral quando v_i é convexo 83
4.42 Pares antipodais de um polígono convexo 84
4.43 Mapeamento de arestas em vetores e vértices em setores para a proposição do algoritmo de Shamos 85
4.44 Implementação do algoritmo de Shamos para o cálculo dos pares antipodais de um polígono convexo 86
4.45 Ilustração das definições de *lóbulo*, *alça* e *corpo*. 87
4.46 Exemplo de configuração 88
4.47 Tratamento do primeiro caso – v_{i_j} não está estritamente à esquerda de $l(C)$ 88
4.48 v_j à esquerda de $l(C)$. (a) v_j na região R_0; (b) v_j na região R_1; (c) v_j na região R_2. . 89
4.49 Implementação do algoritmo de Lee para o cálculo da envoltória convexa de polígonos simples 90
4.50 Exemplo de ilustração produzida pelo **GeoLab** 91

5.1 **GeoLab** – Um ambiente de desenvolvimento de algoritmos em Geometria Computacional 97

A.1 Visão geral da interface gráfica do **GeoLab** destacando-se o editor gráfico e os modos de operação 100
A.2 Objetos selecionados diferenciam-se dos outros através das alças 102
A.3 Diferentes níveis de aproximação de objetos obtidos através do modo de *Zoom* 103
A.4 Quadros de animação via movimentação dinâmica do algoritmo que constrói o diagrama de Voronoi de um conjunto de pontos 104
A.5 Exemplo de ilustração produzida no **GeoLab** através de modos de construção interativa incorporados dinamicamente 106
A.6 Janela de aviso registrando a ocorrência de um erro fatal no **GeoLab** 107
A.7 (a) Menu de opções para a manipulação das alças dos objetos; (b) Exemplo de visões com e sem alças 108
A.8 Menu de opções para alteração da posição relativa entre objetos 109
A.9 Painel de controle para as opções do modo de *Zoom* 109
A.10 Painel de controle para animação de algoritmos 110
A.11 Menu de opções e janela de configuração para a edição dos atributos de objetos geométricos 111
A.12 Pontos aleatoriamente gerados no interior de polígonos através do gerador de pontos randômicos 112
A.13 Painel de configuração dos parâmetros do gerador de pontos randômicos 112
A.14 Diagrama de Voronoi aproximado para um conjunto de segmentos obtido através de coberturas randômicas sobre os segmentos 113
A.15 Painel de configuração dos parâmetros do gerador de coberturas randômicas 113
A.16 Painel de configuração dos parâmetros do gerador de polígonos randômicos 114
A.17 Arquivo de configuração e o menu de algoritmos correspondente, construído pelo **GeoLab** 115
A.18 Janela de configuração para algoritmos geométricos externos 117
A.19 Janela de monitoramento de tempo de execução de algoritmos geométricos externos . 117
A.20 Janela de monitoração de memória alocada por algoritmos externos 118
A.21 Painel de controle para composição de algoritmos 119

A.22 Passos da execução de uma composição para o problema do cálculo do diâmetro de um conjunto de pontos no plano 119

B.1 Transmissão de dados entre ambiente e algoritmos geométricos externos – bibliotecas "independentes" em função da existência de uma dupla hierarquia de objetos 131

Capítulo 1

Introdução

Esta tese discute o projeto e a implementação de um ambiente de programação voltado para o desenvolvimento de algoritmos e estruturas de dados em Geometria Computacional. Dentre os principais recursos deste ambiente, o qual chamamos **GeoLab**, destacam-se:

- Um amplo conjunto de estruturas de dados (objetos geométricos) e ferramentas (algoritmos geométricos) específicos para problemas em Geometria Computacional;
- Uma interface gráfica ampla e flexível para manipulação de modelos geométricos;
- Mecanismos para programação que possibilitam o crescimento incremental do ambiente, através de bibliotecas compartilhadas, favorecendo o reaproveitamento dos resultados desenvolvidos através do **GeoLab** em outros projetos;
- Facilidades para animação de algoritmos; e
- Suporte para composição de meta-algoritmos (macros) que, no futuro, servirá de base para a geração automática de código em C++ para algoritmos programados interativamente.

1.1 Geometria Computacional e Ambientes de Programação

Ambientes de programação voltados para problemas de domínios específicos tem sido a forma de se organizar, coerentemente, o conjunto de ferramentas necessárias nestes domínios. Assim o é também em Geometria Computacional.

Desde o aparecimento do termo Geometria Computacional, cunhado por Shamos [Sha78] para definir o estudo dos aspectos construtivos, computacionais e combinatórios que surgem em problemas geométricos, o desenvolvimento de aplicações que necessitam dos resultados obtidos em GC tem esbarrado em uma profunda falta de suporte para sua implementação.

De acordo com Forrest [For87a], a pesquisa realizada sobre problemas com conotação geométrica pode ser dividida em duas grandes áreas: Geometria Computacional, nos termos propostos por Shamos (acentuando-se suas características combinatoriais) e Modelagem Geométrica. Esta última, introduzida por ele próprio, seria a definição, análise e síntese de informações geométricas através de um sistema computacional, onde a simples aplicação de técnicas geométricas clássicas seria totalmente inapropriada.

Ainda segundo Forrest, tem-se procurado preencher a distância que separa estes dois ramos de pesquisa por acreditar-se que eles representam dois aspectos de um mesmo tipo de problemas: o teórico e o aplicado. No primeiro (Geometria Computacional), numerosos progressos ocorreram desde que o assunto passou a ser considerado, e publicações proliferam em grande quantidade e qualidade. No segundo, progressos práticos incontestavelmente ocorreram, mas com muito menos ênfase no que diz respeito a publicações. A razão para tal está fortemente relacionada com o fato de que aplicações práticas por vezes correspondem a produtos comerciais onde existe o interesse inverso ao do ambiente acadêmico no tocante à divulgação do conhecimento.

No campo teórico, as últimas duas décadas foram marcadas por uma enorme profusão de resultados que constituem hoje uma base robusta e consistente para que novos problemas, cada vez mais complexos, sejam estudados. A origem destes problemas, ainda que esta ciência realimente-se constantemente, são principalmente as áreas do conhecimento aplicado, notadamente engenharia, computação gráfica, robótica, etc. Paradoxalmente, existem inúmeros algoritmos propostos em Geometria Computacional que jamais foram implementados. Quando implementados (ou mesmo quando apenas se propõe uma implementação), pouca atenção é dada a itens de extrema relevância nas aplicações práticas como constantes multiplicativas (especialmente para a comparação entre algoritmos diferentes para um mesmo problema), estruturas de dados, robustez numérica, restrições e casos degenerados, o que dificulta (e mesmo impossibilita) seu aproveitamento.

Procurando contribuir para que estas discrepâncias sejam amenizadas, desenvolvemos um ambiente que reúne condições para a aplicação dos resultados teóricos em Geometria Computacional – o **GeoLab**. Mais que isso, introduzimos neste ambiente suporte não só para a pesquisa e ensino (e.g., através de mecanismos para animação de algoritmos) mas também para a aplicação prática dos resultados nele desenvolvidos (e.g., através da construção de bibliotecas de algoritmos).

A idéia de se desenvolver tal ambiente foi por nós trabalhada procurando efetivamente cumprir todos os requisitos necessários em um ambiente de desenvolvimento. Flexibilidade, eficiência e possibilidade de crescimento organizado e incremental não só são características desejáveis, mas presentes no **GeoLab**.

1.2 Trabalhos Relacionados

Esforços recentes têm sido realizados para a produção de bibliotecas e ambientes de desenvolvimento de algoritmos geométricos. Alguns destes esforços resultaram em trabalhos de incontestável qualidade que, de certa forma, influenciaram ou mesmo participaram do **GeoLab**. As ênfases adotadas nestes trabalhos, entretanto, diferem entre si e também com a ênfase que procuramos adotar em nosso projeto, conforme mostramos na comparação apresentada na seção 1.3.

1.2.1 LineTool

O trabalho desenvolvido por Ericson e Yap, denominado LineTool [EY88], consiste de um ambiente para a especificação de modelos geométricos através de expressões algébricas com a finalidade de permitir a inspeção exata destes modelos sob regras de restrição e dependência entre objetos. O propósito do trabalho, dentro do contexto de Geometria Computacional, seria permitir aos pesquisadores propor e analisar conjecturas dentro dos aspectos teóricos relacionados com o desenvolvimento, prova e exemplificação de resultados (i.e., algoritmos, teoremas, etc).

O LineTool é composto basicamente por um editor gráfico manipulado através de uma linguagem textual onde objetos gráficos e numéricos como pontos, retas, círculos, ângulos e distâncias podem ser especificados e então avaliados em função de restrições quanto a questões como consistência geométrica, propriedades invariantes, etc. Existe ainda a possibilidade de se utilizar o editor para a produção de ilustrações em trabalhos científicos. A linguagem utilizada para a construção do LineTool é um dialeto de Lisp, sendo que os autores mencionam uma migração para Maple.

Os termos essencialmente matemáticos adotados pelo LineTool o caracterizam, conforme os autores comentam, como um meio termo entre um provador de teoremas e um ambiente de desenvolvimento de algoritmos geométricos.

1.2.2 WOGC

O *Workbench On Computational Geometry* [Kni90] é um ambiente de desenvolvimento de algoritmos geométricos desenvolvido por Alan Knight, na Carleton University, Canadá. Trata-se de um sistema razoavelmente grande, escrito em Smalltalk para plataformas do tipo Macintosh, que implementa um conjunto de objetos e algoritmos geométricos como suporte para programação geométrica. Vários algoritmos geométricos, inclusive o intrincado algoritmo para a triangulação de polígonos simples em tempo $O(n \log\log n)$ de Tarjan e Van Wyk [TW88], foram implementados através do WOCG.

Knight explorou o paradigma de orientação a objetos para a construção do WOCG. Algoritmos geométricos são implementados como mensagens pertencentes ao protocolo do objeto sobre o qual operam (i.e., existem objetos como ponto, segmento, conjunto de pontos, conjunto de segmentos, etc). Novos algoritmos são incorporados ao ambiente através da inclusão de código ao mesmo, o que só não implica na necessidade de recompilação do ambiente por estar sendo utilizada uma linguagem interpretada. Modelos geométricos são, a exemplo dos demais ambientes do gênero, manipulados através de um editor gráfico. Existem recursos para animação e depuração de algoritmos.

1.2.3 XYZ Geobench

O ambiente desenvolvido por Peter Schorn [Sch91] possui grande similaridade com o WOCG. Foi desenvolvido em Pascal orientado a objetos em máquinas do tipo Macintosh e possui um numeroso conjunto de algoritmos geométricos, mapeados também como mensagens nos protocolos dos objetos geométricos.

A diferença básica com relação ao WOCG é que no XYZ-Geobench procurou-se explorar, como ênfase principal, conceitos relacionados à robustez numérica na implementação de algoritmos geométricos. De fato, os algoritmos implementados através do XYZ-Geobench são robustos graças ao aprimoramento de algumas técnicas de tratamento numérico de operações de ponto flutuante em algoritmos geométricos.

Assim como o WOCG, modelos geométricos são manipulados através de um editor gráfico, existindo também recursos para animação de algoritmos.

1.2.4 LEDA

O projeto levado a cabo por Mehlhorn e Näher [MN89], tem como objetivo a produção de bibliotecas de estruturas de dados e algoritmos eficientes e reutilizáveis. Embora longe de ser um ambiente de desenvolvimento de software, o LEDA (*Library of Efficient Data Types and Algorithms*) objetiva servir como ferramenta na produção rápida e confiável de algoritmos e aplicações em geral.

Não sendo o único neste sentido, o projeto LEDA apresenta consistência e versatilidade nas suas construções, escritas em C++, criando tipos polimórficos para várias estruturas de dados, e de fácil utilização (software desenvolvido pela *GNU Free Software Foundation* [GNU92] e também por Gorlen, Orlow e Plexico [GOP90] são outros exemplos de bibliotecas de algoritmos e estruturas de dados genéricos em C++).

1.3 Requisitos para um Ambiente de Desenvolvimento

Nosso ambiente de desenvolvimento de algoritmos geométricos, **GeoLab**, relaciona-se com os outros projetos mencionados no sentido de objetivar auxiliar no processo de produção de software (no caso, especificamente em Geometria Computacional) provendo estruturas e algoritmos fundamentais como suporte para tal (que formam a biblioteca de capacitação geométrica do ambiente). Além disso, esta biblioteca fundamental é ainda complementada através da elaboração de um sistema computacional específico para a manipulação de objetos de natureza geométrica que provê mecanismos para incorporação de ferramentas de forma ilimitada e, especialmente, dinâmica.

Os requisitos considerados no projeto do **GeoLab** são divididos em duas categorias: Suporte de Programação (i.e., bibliotecas de estruturas de dados e algoritmos geométricos) e Suporte Interativo (i.e., ambiente de manipulação gráfica, integração de algoritmos ao ambiente, ferramentas para visualização de objetos geométricos, etc).

O suporte de programação inclui os seguintes requisitos:

- Tipos abstratos de dados para a representação de objetos geométricos tais como pontos, segmentos, polígonos, subdivisões planares, etc, assim como suporte para a criação e introdução de novos tipos;
- Estruturas de dados e algoritmos geométricos básicos assim como suporte para a criação e incorporação de novos componentes básicos;
- Algoritmos e estruturas de dados de uso geral (e.g., algoritmos de ordenação, listas, filas, pilhas, etc);
- Ferramentas para a produção de bibliotecas reutilizáveis; e
- Flexibilidade para intercambiar representações, modelos numéricos, algoritmos e estruturas de dados.

Não menos importante, o ambiente deve assegurar a robustez dos algoritmos implementados utilizando suas primitivas. Este requisito foi abordado de forma secundária neste trabalho.

Quanto aos requisitos do suporte interativo, destacamos:

- Um ambiente gráfico e interativo para a manipulação de modelos geométricos;
- Recursos para incorporação de novos algoritmos e objetos geométricos;
- Suporte para animação de algoritmos;
- Suporte para análise e depuração de algoritmos;

- Ferramentas para manipulação de memória secundária; e
- Facilidades para a construção de entradas para testes de pior caso, caso médio e também casos degenerados de algoritmos geométricos.

Considerando estes requisitos, o **GeoLab** diferencia-se dos demais projetos mencionados. O LEDA passa a ser apenas um dos componentes do ambiente (e de fato parcialmente o é) e tanto o WOCG quanto o XYZ-Geobench deixam muito a desejar no tocante ao suporte para a introdução de novos algoritmos e objetos geométricos, assim como na produção de bibliotecas realmente reutilizáveis. Em ambos os casos, diferentemente do **GeoLab**, exige-se uma interferência direta sobre o ambiente, exigindo-se inclusive recompilação constante no caso do XYZ-Geobench. Este, por outro lado, reúne características de robustez numérica ainda não abordadas no **GeoLab**.

1.4 Estrutura da Tese

Esta tese está organizada da seguinte forma: o capítulo 2 inicia a descrição do **GeoLab** a partir dos requisitos apresentados neste capítulo. O capítulo 3 descreve os quatro tipos de entidades externas introduzíveis dinamicamente no ambiente. O capítulo 4 descreve os atuais algoritmos geométricos existentes, que juntamente com os objetos básicos providos pelo ambiente constituem o suporte geométrico do **GeoLab**. Ao final, após a conclusão, existem dois apêndices que versam sobre a utilização do **GeoLab**, constituindo seu Guia de Referência.

Capítulo 2

Apresentando o GeoLab

No capítulo anterior foram apresentados os requisitos básicos a serem supridos por um ambiente de desenvolvimento de algoritmos geométricos. Não menos importante é a forma com que estes requisitos são abordados (i.e., implementados). De fato, nós acreditamos que a organização coerente dos requisitos previamente apresentados e sua implementação são fatores fundamentais ao sucesso do trabalho proposto.

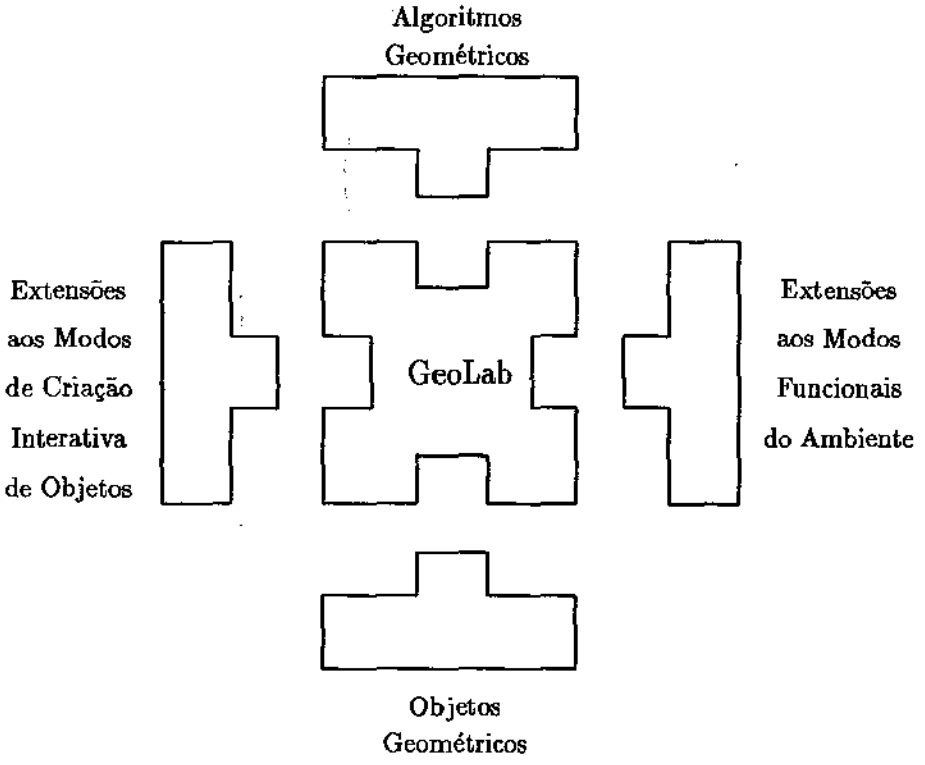

Figura 2.1: Estrutura do **GeoLab**

A figura 2.1 ilustra a estrutura do **GeoLab** em termos de implementação. Este capítulo descreve o componente central, referenciado aqui como **Módulo Básico**, que agrupa os recursos mínimos que

caracterizam o ambiente de desenvolvimento de algoritmos. Nele também são definidos os mecanismos que tornam o ambiente passível de ser incrementado, através da criação de novos recursos (módulos periféricos na figura), e que serão abordados no capítulo seguinte.

2.1 Interface Gráfica

Knight [Kni90] observa que um ambiente de computação geométrica deve basear-se na representação gráfica dos elementos geométricos sobre os quais opera. A representação textual pode ser útil em sistemas onde grandes computações são realizadas e os dados são gerados a partir de dispositivos automáticos como satélites, sensores ou robôs, mas não auxilia (tanto quanto a representação gráfica) um usuário a criar exemplos, testes, ilustrar o funcionamento ou verificar os resultados de um algoritmo. Particularmente, no que diz respeito a ilustração de algoritmos (e.g., animação), Brown também observa que interfaces gráficas são de fundamental importância na obtenção de resultados visuais satisfatórios [Bro87]. Estas observações fundamentam nosso esforço em desenvolver uma interface gráfica bastante ampla e flexível, visando otimizar em vários pontos o uso do ambiente.

Existem dois níveis de interface com o **GeoLab**. Diferentes necessidades no uso do ambiente definem qual nível deve ser considerado. Um destes níveis corresponde à sua interface gráfica, o outro diz respeito a sua interface de programação, esta última abordada na seção 2.2.

É digno de nota que a estes dois níveis de comunicação correspondem duas classes distintas de usuários. Usando nomeclatura similar à proposta por Amorim [Amo92], estas classes são identificadas como *usuário-final* e *usuário-programador*. A primeira relaciona os usuários que fazem qualquer uso do **GeoLab** que não envolva trabalho a nível de implementação (como simplesmente a visualização ou teste de algoritmos, animação ou construção de modelos geométricos). Para esta classe de usuários a interface gráfica é a única forma de interação com o ambiente.

A figura 2.2 ilustra a interface gráfica do **GeoLab** destacando seus componentes.

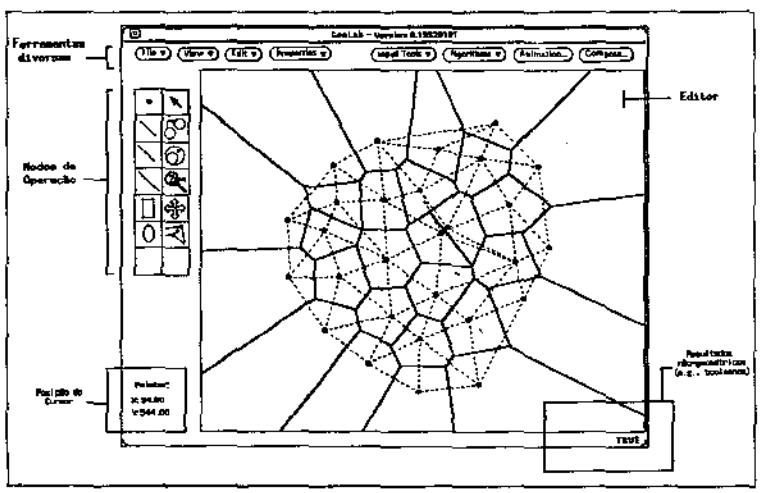

Figura 2.2: Interface Gráfica do Ambiente

2.1.1 O Editor e os Modos de Operação

O editor de objetos geométricos constitui a forma de acesso a modelos geométricos sendo trabalhados por parte dos usuários, independentemente de sua representação interna. Estes modelos podem ser criados, alterados, destruídos ou ainda serem fornecidos como entrada para algoritmos geométricos.

A atuação sobre o editor depende de dois conceitos simples, o de *modo de operação* e o de *objeto selecionado.* Diferentes modos de operação alteram a forma com que o editor interpreta as ações executadas pelo usuário (através do *mouse* ou teclado). O conceito de *objeto selecionado* é utilizado para delimitar o campo de atuação tanto dos modos de operação quanto das demais ferramentas disponíveis no ambiente. A seleção de objetos é feita através de um dos modos de operação e vários objetos podem ser ou estar selecionados em um determinado momento.

Os modos de operação subdividem-se em duas classes (vide figura 2.3):

- Modos de criação interativa de objetos;
- Modos funcionais (e.g., seleção, *Zoom*)

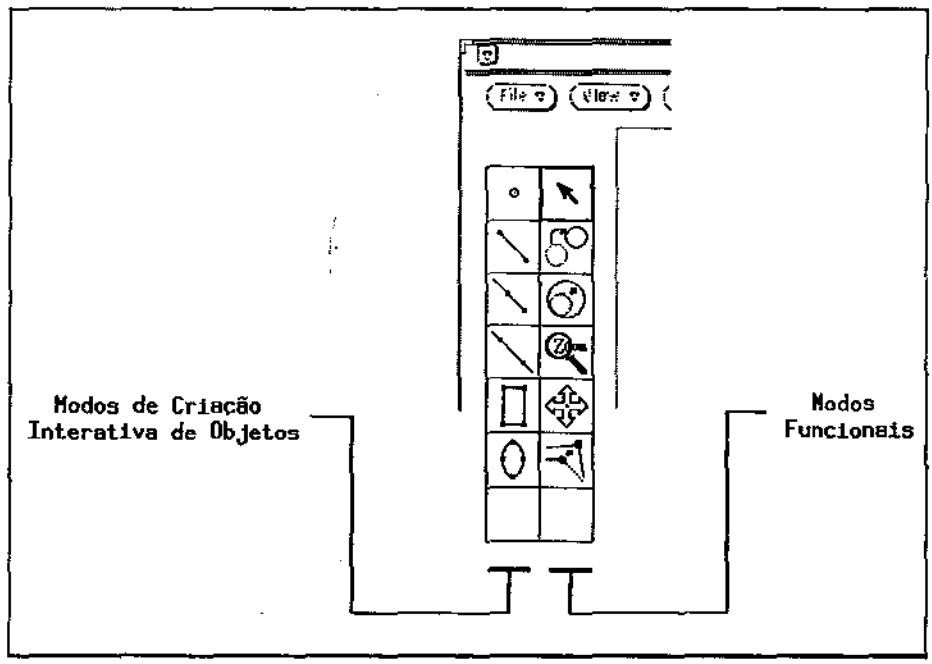

Figura 2.3: Modos de Operação do Editor

Os modos de criação interativa compreendem alguns objetos básicos internos ao **GeoLab**[1]. Estes objetos são:

- Pontos
- Segmentos

[1] O conceito de objetos internos/externos será introduzido no capítulo seguinte.

- Raios
- Retas
- Retângulos e Quadrados
- Elipses e Círculos

Outros objetos internos existem (e.g., polígonos), porém não existem modos de criação interativa internos associados a eles.

Ao final da coluna correspondente aos modos de criação interativa existe um botão polivalente (intencionalmente deixado em branco – vide figura 2.3) que é utilizado para a introdução de novos modos de criação interativa, podendo ser multiplexado seja para objetos "novos" (e.g., polígonos, subdivisões planares) ou simplesmente variações para os objetos que já constam das opções (e.g., construir um círculo por três pontos). Isto constitui um dos possíveis modos de expansão do **GeoLab** (realizado pelo usuário-programador) e sobre o qual discorreremos no capítulo 3.

Os modos funcionais compreendem:

- Seleção de objetos no editor;
- Translação ou cópia de objetos;
- Redimensionamento de objetos existentes;
- Ampliação ou redução de objetos (*Zoom*);
- Movimentação da área de visualização do editor (*Scroll*); e
- Aplicação dinâmica de algoritmos sobre entradas.[2]

Da mesma forma que os modos de criação interativa, também os modos funcionais são passíveis de expansão (veja seção 3.3).

Um conjunto mínimo inicial básico mas suficiente de recursos para operação por usuários do ambiente é portanto parte integrante dele. A partir do momento em que novos objetos gráficos necessitem ser criados interativamente, ou novos modos funcionais acrescentados, existem recursos que permitem sua inclusão no ambiente de forma simples e, mais importante, ilimitada. É também digno de nota que o acréscimo de novos objetos e novos modos funcionais é feito de forma dinâmica ao **GeoLab** (i.e., através da ligação em tempo de execução de novos módulos), o que lhe dá uma característica extremamente desejável em qualquer sistema computacional: flexibilidade.

A falta de detalhes nos assuntos desta seção é proposital. Detalhes são dados nos capítulos que se seguem e nos apêndices ao final da tese.

[2]Este é um importante mecanismo para animação de algoritmos, descrito na seção 2.1.2.3.

2.1.2 Ferramentas para Tratamento de Modelos Geométricos

Em adição ao editor e aos modos de operação, o **GeoLab** provê inúmeras ferramentas para a manipulação de objetos gráficos. Estas ferramentas estão divididas em três grupos básicos de atuação junto ao editor:

- Ferramentas genéricas;
- Geradores de entrada; e
- Gerenciadores de algoritmos geométricos.

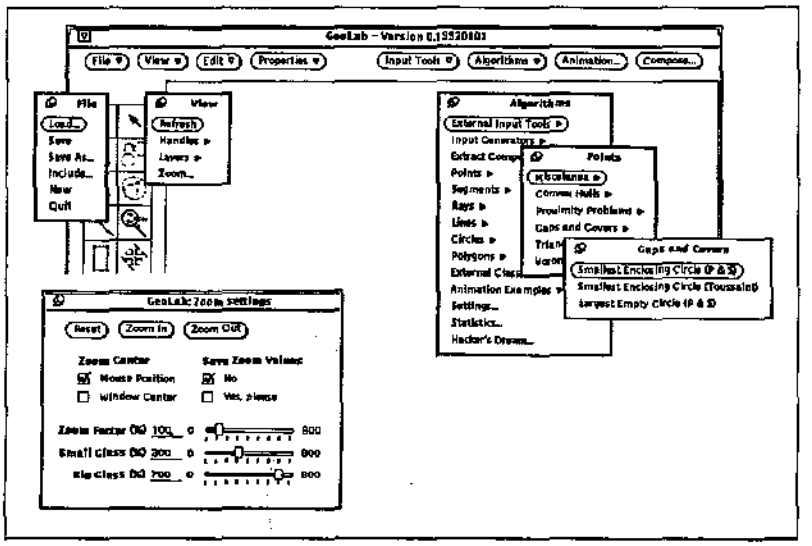

Figura 2.4: Botões de acesso às ferramentas, algumas sub-janelas e menus

2.1.2.1 Ferramentas Genéricas e Extensões aos Modos de Operação

Correspondem a recursos como armazenamento e recuperação de objetos em memória secundária, tratamento da área de visualização do editor (e.g., *Zoom*) e edição de atributos dos objetos gráficos (e.g., cor, tipo de traço). Algumas destas ferramentas complementam e/ou parametrizam os modos funcionais do editor (e.g., fator para ampliação de objetos no modo de *Zoom*).

2.1.2.2 Geradores de Entrada

Uma das dificuldades que surge na experimentação de algoritmos geométricos é a geração de modelos geométricos. Enquanto o editor dispõe de recursos para a criação ou edição de modelos, fica a cargo do usuário o trabalho de construção do modelo, o que nem sempre é agradável ou possível. Isto é particularmente verdade quando o que se deseja são grandes conjuntos de objetos dispostos randomicamente, ou dispostos segundo condições especiais (e.g., pontos co-circulares).

Foram incorporadas ferramentas para suprir algumas das necessidades quanto à geração automática de modelos geométricos. É possível, por exemplo, gerar grandes quantidades de pontos distribuídos

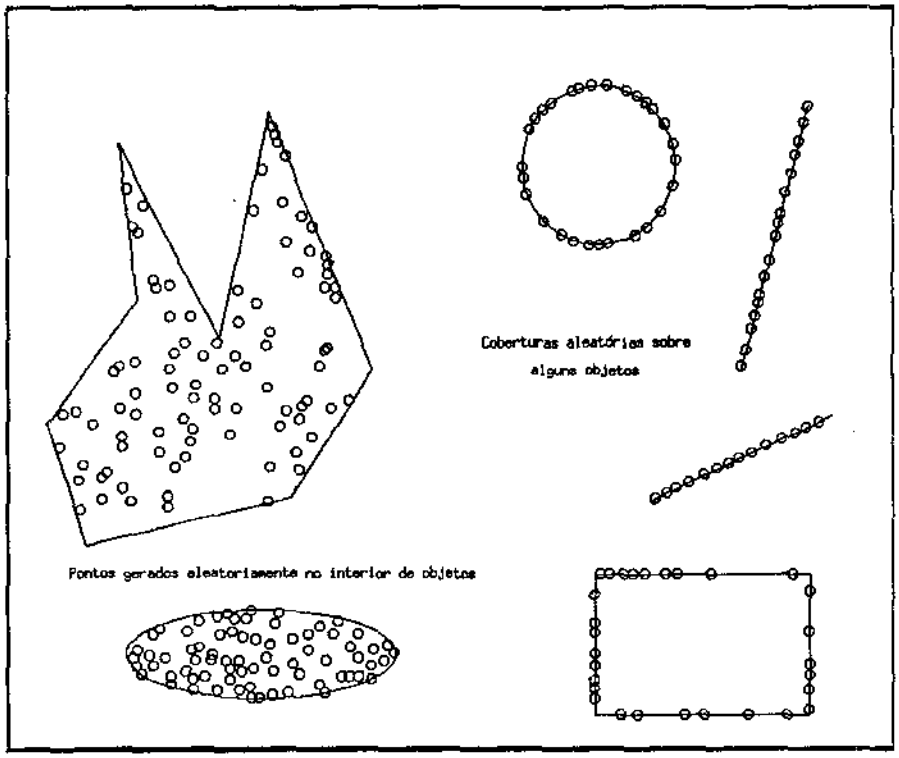

Figura 2.5: Coberturas de pontos e pontos randômicos internos a objetos

randomicamente sobre o plano de visualização, sobre a fronteira de qualquer objeto selecionado ou no interior de objetos selecionados (vide figura 2.5).

Outra possibilidade é utilizar conjuntos de pontos para a geração de polígonos randômicos, com variações de implementação para polígonos genéricos, simples, em forma de estrela ou convexos (figura 2.6).

Outros geradores de entradas podem ainda ser incorporados através dos mesmos mecanismos utilizados para a introdução de algoritmos geométricos externos (vide seção 3.1) e com isto facilitar ainda mais o trabalho de criação de casos de entradas para algoritmos geométricos.

2.1.2.3 Gerenciadores de Algoritmos Geométricos

- **Animação de Algoritmos**

  Animação de algoritmos é uma das maiores facilidades disponíveis no (e inerentes ao) **GeoLab**. As aplicações para este tipo de recurso visual são muito variadas: desde a depuração até o estudo e apresentação de um algoritmo.

  Em geral, animar um algoritmo não é uma tarefa simples [Bro87, Amo92, Bro92], basicamente devido aos problemas que surgem ao se propor uma representação gráfica para elementos que não possuem tal natureza. Representar graficamente de forma adequada várias estruturas de dados (e.g., árvores, listas, grafos) e o próprio comportamento de um algoritmo genérico são os desafios a serem vencidos por um animador. No entanto, em se tratando de um ambiente de manipulação de objetos geométricos, cuja representação pictórica é inerente, torna-se mais fácil e por vezes adequado realizar uma animação.

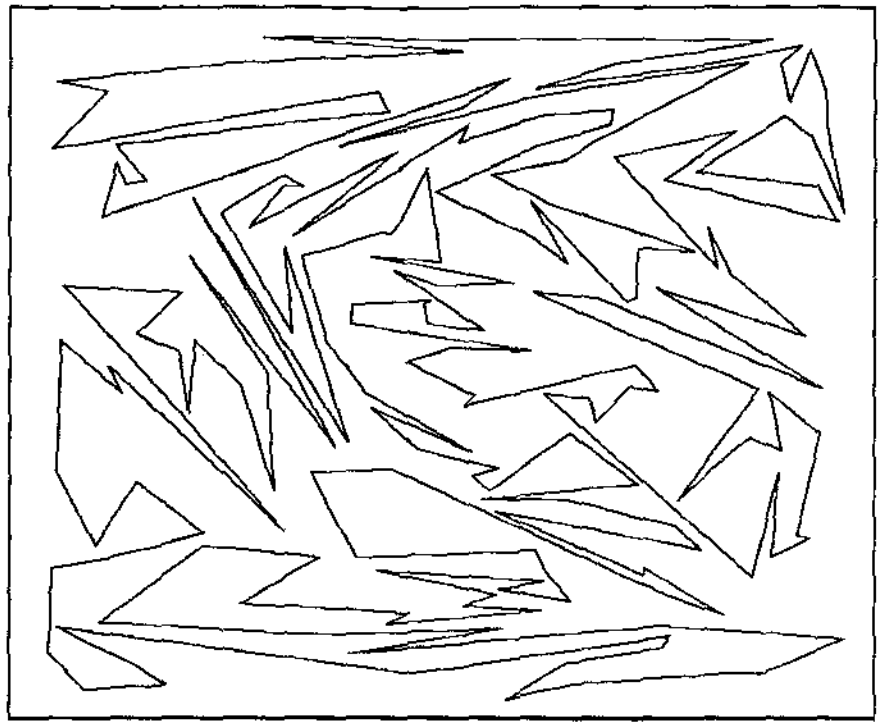

Figura 2.6: Um polígono simples com 200 vértices

Nós habilitamos o **GeoLab** a fazer dois tipos de animação. O primeiro tipo dispensa qualquer esforço extra por parte do usuário-programador. Trata-se de um recurso que denominamos *Dynamic Move* (um dos modos funcionais disponíveis) e cuja tarefa é executar continuamente um algoritmo enquanto os objetos que constituem sua entrada são alterados (i.e., movidos ou redimensionados). Isto proporciona uma visão dinâmica do funcionamento do algoritmo, permitindo um melhor entendimento de seu comportamento (e resultado) e favorecendo a criação de testes de casos extremos (e.g., provocando a colinearidade de três pontos).

A figura 2.7 é uma tentativa de capturar o efeito deste tipo de animação[3]. Nós apresentamos um diagrama de Voronoi [PS85] com 5 pontos, um dos quais movendo-se horizontalmente, da esquerda para a direita (implementação do algoritmo proposto por Guibas e Stolfi em [GS85]).

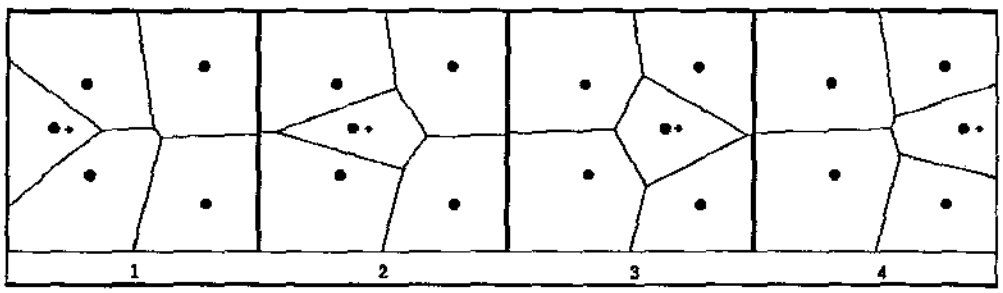

Figura 2.7: Um diagrama de Voronoi sofrendo *Dynamic Move*

[3] Um *video-tape* seria sem dúvida mais apropriado.

Animação via *Dynamic Move* é apropriada para pequenos conjuntos de entrada e também algoritmos "rápidos", onde a velocidade de apresentação dos quadros(resultados intermediários de chamadas ao algoritmo) é suficientemente grande para dar a noção de movimento.

Outro tipo de animação possível no **GeoLab** é a representação do funcionamento interno de um algoritmo. Neste caso, faz-se necessário o acréscimo de código específico para animação nos algoritmos geométricos (trabalho feito pelo usuário-programador). O ambiente pode então controlar itens como velocidade de animação e pontos de parada. Além disso, no caso comum em que um algoritmo é construído utilizando-se de outros como blocos de construção, pode-se controlar qual a "profundidade" de animação desejada. Apesar de não ser um ambiente exclusivamente voltado para animação, o recurso de animação por profundidade permite ao usuário-programador estruturar animações de forma bastante poderosa, variando a quantidade de informações apresentadas a cada nível e proporcionando animações tanto a nível de apresentação (onde apenas passos específicos de um algoritmo são mostrados) quanto a nível de depuração (onde os mais ínfimos detalhes podem ser mostrados), como podemos ver nas figuras 2.8 e 2.9[4].

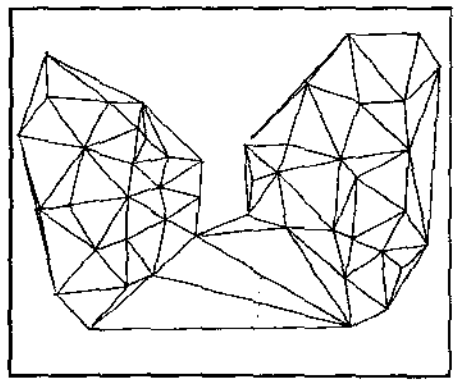

Figura 2.8: Animação de um passo genérico no cálculo da triangulação de Delaunay

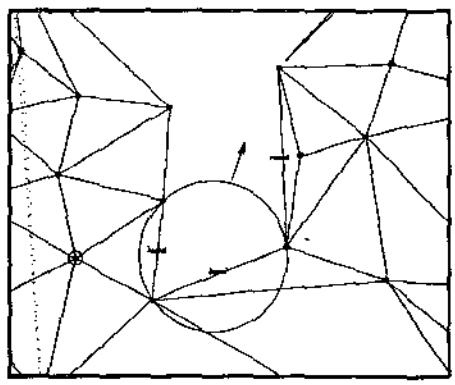

Figura 2.9: Mais detalhes na animação da figura anterior

[4] O leitor interessado pode observar que escolhemos dois quadros de uma longa animação correspondente a duas figuras que ilustram o funcionamento do algoritmo no artigo onde foi originalmente proposto [GS85].

Os mecanismos que permitem estes recursos estão na verdade divididos tanto na interface gráfica do ambiente quanto na sua interface programática. Para o usuário-final é suficiente o conhecimento apenas da aplicação da ferramenta (figura 2.10), enquanto o usuário-programador deve conhecer também quais os recursos de programação que o sistema oferece para que tornar possível animar um algoritmo.

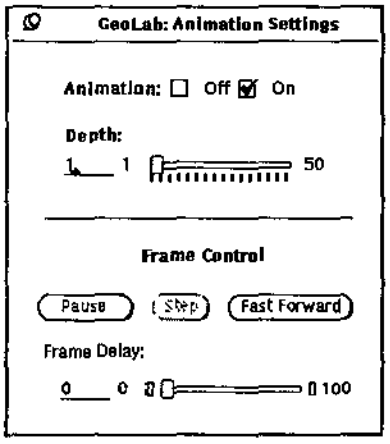

Figura 2.10: Painel de controle de animação

- **Análise de Algoritmos (*Benchmark* e Depuração)**

  No tocante a análise de algoritmos, o **GeoLab** permite o acompanhamento dos tempos de execução (i.e., tempo real e tempo virtual, este último descontando-se o tempo gasto pelo sistema operacional em operações como escalonamento, paginação, etc) e da memória utilizada por um algoritmo.

  O acompanhamento dos tempos de execução é particularmente útil na análise das constantes multiplicativas associadas às complexidades de algoritmos e também na comparação de algoritmos diferentes para um mesmo problema. Foi através deste recurso que constatamos, por exemplo, que para pontos distribuídos uniformemente no plano, o algoritmo proposto por Bhattacharya e Toussaint [BT85] para o cálculo do menor círculo envolvente é mais eficiente (em termos de constante multiplicativa) que o proposto por Shamos [Sha78], apesar da aparente maior simplicidade do primeiro com relação ao segundo[5].

  Algoritmo de Shamos Cálculo do Menor Círculo Envolvente

  1. Calcula diagrama de Voronoi de vizinho mais distante para os pontos da entrada. $O(n \log n)$.
  2. Seleciona a maior aresta do dual do diagrama e verifica se todos os pontos situam-se dentro do círculo que tem como diâmetro a aresta selecionada. Se sim, termina. $O(n)$.
  3. Seleciona o menor dos círculos determinados pelas faces do dual do diagrama (que é uma triangulação). $O(n)$.

[5] As considerações sobre este e outros algoritmos analisados são feitas em detalhes no capítulo 4. Os dados obtidos para a construção do gráfico da figura 2.11 e para os demais gráficos apresentados no capítulo 4 foram obtidos em uma Sun SparcStation 2

Algoritmo de Bhattacharya e Toussaint

1. Calcula o diâmetro do conjunto de pontos (isto pode ser feito calculando-se a envoltória convexa dos pontos e em seguida os pares antipodais, selecionando-se o menor). $O(n \log n)$.
2. Verifica se todos os pontos situam-se dentro do círculo cujo diâmetro foi calculado no passo anterior. Se sim, termina. $O(n)$.
3. Calcula o diagrama de Voronoi de vizinho mais distante (só os pontos na envoltória, calculada no passo 1, necessitam ser considerados). $O(n \log n)$.
4. Seleciona o menor dos círculos determinados pelas faces do dual do diagrama (que é uma triangulação). $O(n)$.

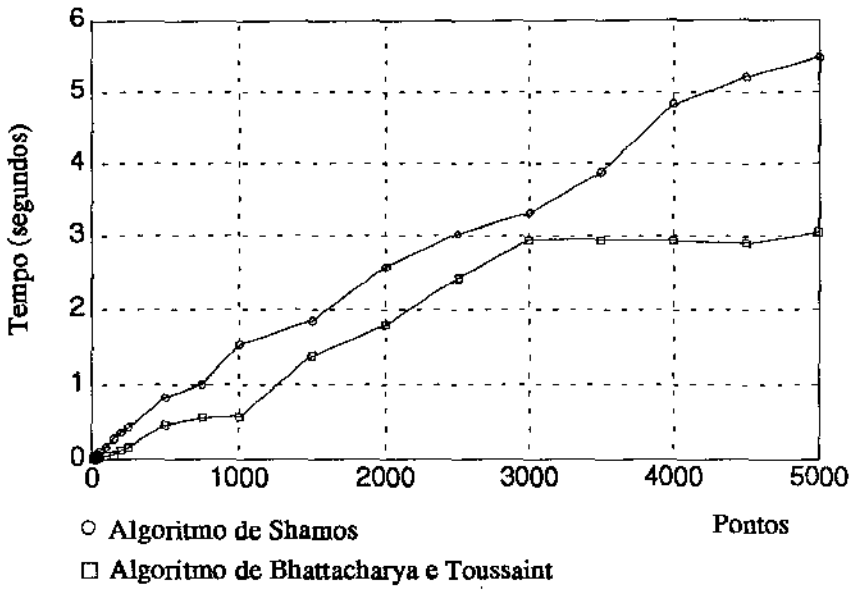

Figura 2.11: Gráfico de desempenho para algoritmos que calculam Menor Círculo Envolvente

A monitoração do uso de memória do sistema tem como objetivo principal auxiliar no processo de depuração de algoritmos. Um tipo de falha de programação normalmente obscura (não desalocar memória associada a apontadores) torna-se perceptível ao longo de uma sessão com o ambiente.

2.1.3 Composição de Algoritmos

Agrupar ferramentas simples para a construção de computações mais complexas é uma idéia básica, utilizada em todas as áreas da Ciência da Computação. No **GeoLab**, cuidamos de possibilitar o agrupamento de ferramentas tanto a nível de programação (com a criação de bibliotecas compartilhadas - veja capítulo 3) quanto a nível de interação com sua interface gráfica. Chamamos este recurso de Composição de Algoritmos, cuja finalidade é permitir a criação de *macros*, constituídas de chamadas aos algoritmos disponíveis no menu de opções. A utilidade deste tipo de construção revela-se particularmente interessante durante o processo de pesquisa que antecede a criação de novos algoritmos geométricos. Como exemplo, considere a existência dos seguintes algoritmos:

- *StarShaped* - dado um conjunto de pontos, constrói um polígono em forma de estrela.
- *StarHull* - calcula a envoltória convexa dos vértices de um polígono em forma de estrela.
- *PolDiameter* - calcula o diâmetro de um polígono convexo.

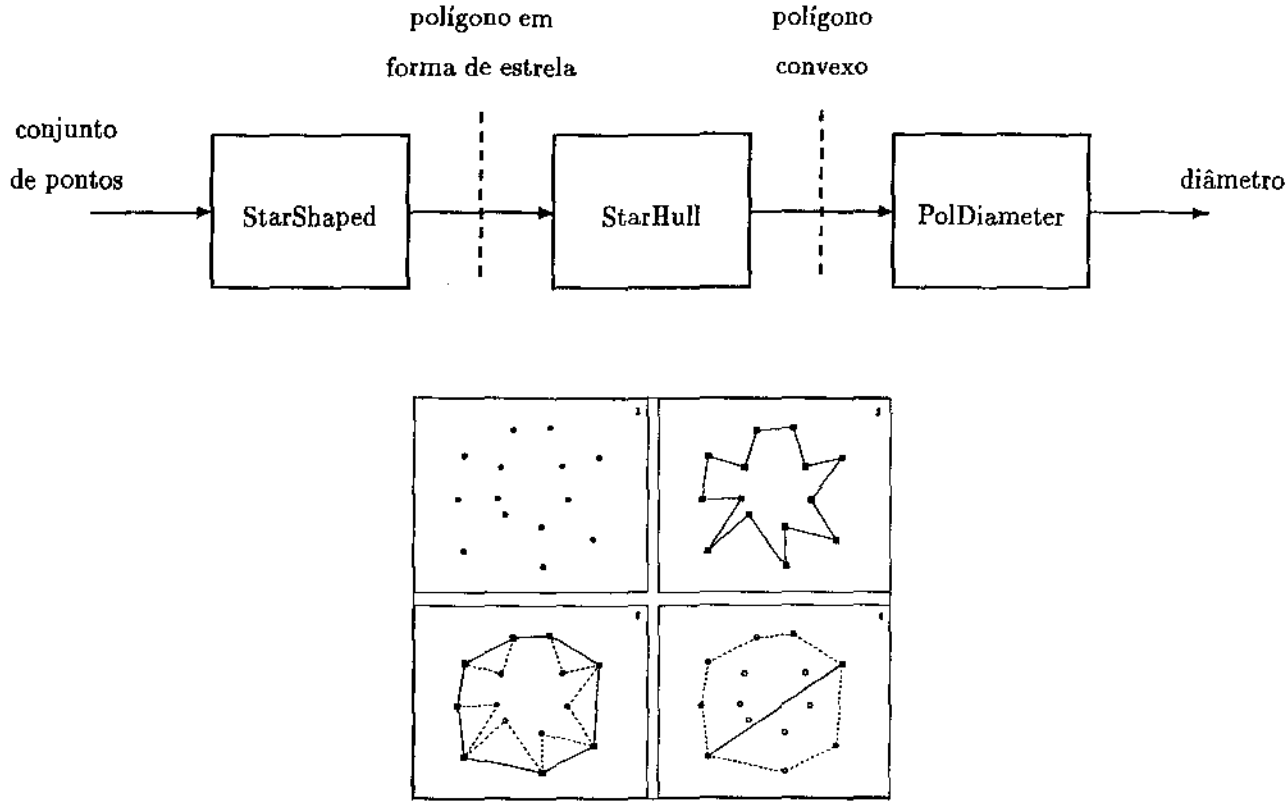

Figura 2.12: Cálculo do diâmetro de um conjunto de pontos através de composição de algoritmos

Com o mecanismo de composição de algoritmos, é possível criar uma composição (de fato, um algoritmo) para o cálculo do diâmetro de um conjunto de pontos (veja seção 4.3.2.1) encadeando-se os algoritmos citados de tal forma que a saída de um dos passos torne-se entrada para o passo seguinte (figura 2.12).

A idéia de compor algoritmos tal como nós desenvolvemos é muito semelhante à apresentada por Zloof em [Zlo75]. Brown [Bro87] explorou conceitos análogos para a criação de filmes de animação de algoritmos (chamados *Scripts*). A diferença principal entre o conceito de *script* e o de algoritmo composto é que, neste último, diferentes conjuntos de entrada podem ser manipulados.

Embora útil, o corrente módulo de composição de algoritmos pode ser ainda explorado em vários aspectos, sendo este um dos trabalhos futuros a ser realizado sobre o **GeoLab**. Citamos como exemplos de possíveis extensões, a incorporação de estruturas condicionais e repetitivas em composições e a geração automática de código (C++) para algoritmos compostos.

2.2 Suporte para Programação

Juntamente com a interface gráfica, o ambiente para desenvolvimento de algoritmos geométricos dispõe de uma série de ferramentas voltadas para auxiliar no processo de construção de algoritmos geométricos e tipos abstratos de dados (para representação de objetos geométricos e não-geométricos). Nossa abordagem consiste em oferecer um conjunto amplo e extensível de classes em C++ [Str87], organizadas através de hierarquias.

Antes de descrever estas hierarquias, é conveniente mencionar algumas características da programação orientada a objetos, pois a aplicação deste paradigma tem importância fundamental dentro do contexto de desenvolvimento de algoritmos através do **GeoLab**.

2.2.1 Programação Orientada a Objetos

Orientação a objetos é um paradigma de programação onde dados e funções (chamadas métodos ou mensagens) são agrupados de forma a definir **classes**. Idealmente, dados que compõem uma classe são acessados única e exclusivamente pelos métodos definidos pela classe. A isto corresponde o conceito de **tipo abstrato de dados**, i.e., uma estrutura de dados manipulável através de funções especificamente designadas e que escondem a representação interna utilizada para a organização das informações (**encapsulamento, abstração**). Um **objeto** é definido como uma *instância*[6] de uma determinada classe [Xav92].

Uma classe pode ser derivada de outra (neste caso uma **subclasse**), *herdando* características e possivelmente alterando e/ou acrescentando dados e métodos à sua representação (**especialização**). Métodos não redefinidos não necessitam ser reimplementados (**reaproveitamento de código**).

Classes diferentes podem definir e implementar métodos e operadores[7] com nomes idênticos, associando a eles semânticas próprias em função de suas representações internas (**sobrecarga**).

Uma classe representando um tipo abstrato de dados (e.g., lista duplamente encadeada) pode ser construída de tal forma que o *tipo* de informações que manipula possa ser definido através de parâmetros no ato de sua *instanciação* (e.g., `dlist(int) dli;` - `dli` é um objeto do tipo lista duplamente encadeada de inteiros). A isto chamamos **polimorfismo paramétrico**.

Derivações entre classes introduzem o conceito de **hierarquia**. A classe principal de uma hierarquia (ou super-classe) define um conjunto de dados e métodos comuns a todas as suas subclasses. Uma subclasse pode desempenhar qualquer papel dentro de uma aplicação onde um elemento do tipo de sua super-classe é esperado. Por este motivo, métodos redefinidos em uma derivação podem ter referências que só são resolvidas em tempo de execução, e a isso chamamos **ligação tardia**.

Existem várias linguagens que permitem a implementação de sistemas computacionais utilizando-se o paradigma de orientação a objetos. Segundo [Xav92], estas linguagens classificam-se basicamente em: "puras" (i.e., construídas unicamente voltadas para o paradigma) como Smalltalk, e "derivadas" (i.e., provenientes da adição de recursos a linguagens anteriormente não orientadas a objetos) como CLOS (Common Lisp Object System), Objective-C, Object Pascal e C++.

A razão para utilizarmos uma linguagem que permite orientação a objetos segue do mapeamento lógico perfeito entre os elementos manipulados pelo **GeoLab** e o conceito de objetos. Primariamente, o ambiente deve ser capaz de prover de forma simples e eficiente uma grande quantidade de tipos

[6] Os termos *instância* e *instanciação* são neologismos largamente utilizados neste texto.

[7] Operadores nada mais são que métodos que já possuem algum significado especial na linguagem utilizada e que podem ter seu significado redefinido para classes definidas pelo programador (e.g., "==" em C++)

abstratos de dados (e.g., objetos geométricos). Vários destes tipos são logicamente deriváveis (e.g., um círculo é uma especialização de elipse), o que torna desejável mecanismos de herança (principalmente no tocante à questão de reaproveitamento de código, item importante na construção de grandes sistemas computacionais [Fur91]).

Além disso, escolhemos abordar a construção de algoritmos geométricos também hierarquicamente, por termos identificado previamente [Jac91] que algoritmos em Geometria Computacional podem ser organizados em categorias (classes) que compartilham as mesmas características (e.g., algoritmos que se utilizam do paradigma de varredura planar[8]) e portanto podem ser adequadamente mapeados em hierarquias, facilitando sua implementação e utilização.

2.2.2 Abstrações Voltadas para Programação

Existem três tipos de abstrações utilizadas e fornecidas pelo **GeoLab** com a finalidade de organizar e facilitar o trabalho de implementação de objetos e algoritmos geométricos:

- Sistema de Visualização;
- Objeto Geométrico; e
- Algoritmo.

2.2.2.1 Sistema de Visualização

A implementação de sistemas que requerem a apresentação de informações em forma gráfica normalmente exigem uma interação direta com recursos específicos da máquina hospedeira. Apesar de existirem conjuntos de ferramentas (e.g., *XLib* [Jon89]) que criam uma camada de abstração sobre estes recursos, muitas vezes tal abstração é insuficiente, ou por ser demasiadamente genérica, ou por impor restrições indesejáveis quanto a sua funcionalidade (e.g., a maioria dos *toolkits* exige coordenadas inteiras para o desenho de elementos gráficos, o que absolutamente não ocorre no **GeoLab**). Além disso, o uso indiscriminado destas ferramentas reduz consideravelmente a portabilidade da aplicação em questão.

Para solucionar estes problemas e criar uma abstração mais efetiva para as necessidades de nosso ambiente, desenvolvemos um sistema de visualização chamado *World* que define todas as ferramentas para a representação gráfica de objetos geométricos. O uso deste sistema proporciona:

- Uso de coordenadas reais (i.e., não somente inteiras) no desenho de elementos gráficos, inclusive com a possibilidade de definição da orientação do plano de coordenadas (i.e., origem do sistema de coordenadas com relação à janela gráfica, faixa de valores nos eixos coordenados, etc);
- Manipulação de elementos gráficos não suportados por *toolkits* tradicionais (e.g., raios, círculos de tamanho ilimitado, pontos no infinito, etc.);
- Controle das dimensões e posicionamento do plano virtual de visualização, permitindo que operações de *Zoom* e *Scroll* sejam realizados facilmente;
- Tratamento de eventos externos através de filtros (veja seções 3.3 e 3.3), o que chamamos *encadeamento de tratadores de eventos*);

[8] *Plane-Sweep.*

- Integração de um pacote de gerenciamento de janelas (i.e., XView [Hel90]) e uma linguagem orientada a objetos[9]; e
- Isolamento de código dependente de máquina no que diz respeito a entrada e saída gráfica, facilitando a portabilidade do ambiente.

2.2.2.2 Objeto Geométrico

A abordagem utilizada para a implementação dos inúmeros objetos geométricos manipuláveis pelo **GeoLab** consistiu da criação de uma dupla hierarquia de classes (veja figura 2.13). Como veremos a seguir, isto tem importância fundamental em questões como criação de bibliotecas genéricas, uniformidade de tratamento de objetos pelo editor e economia de espaço em representações derivadas e/ou compostas.

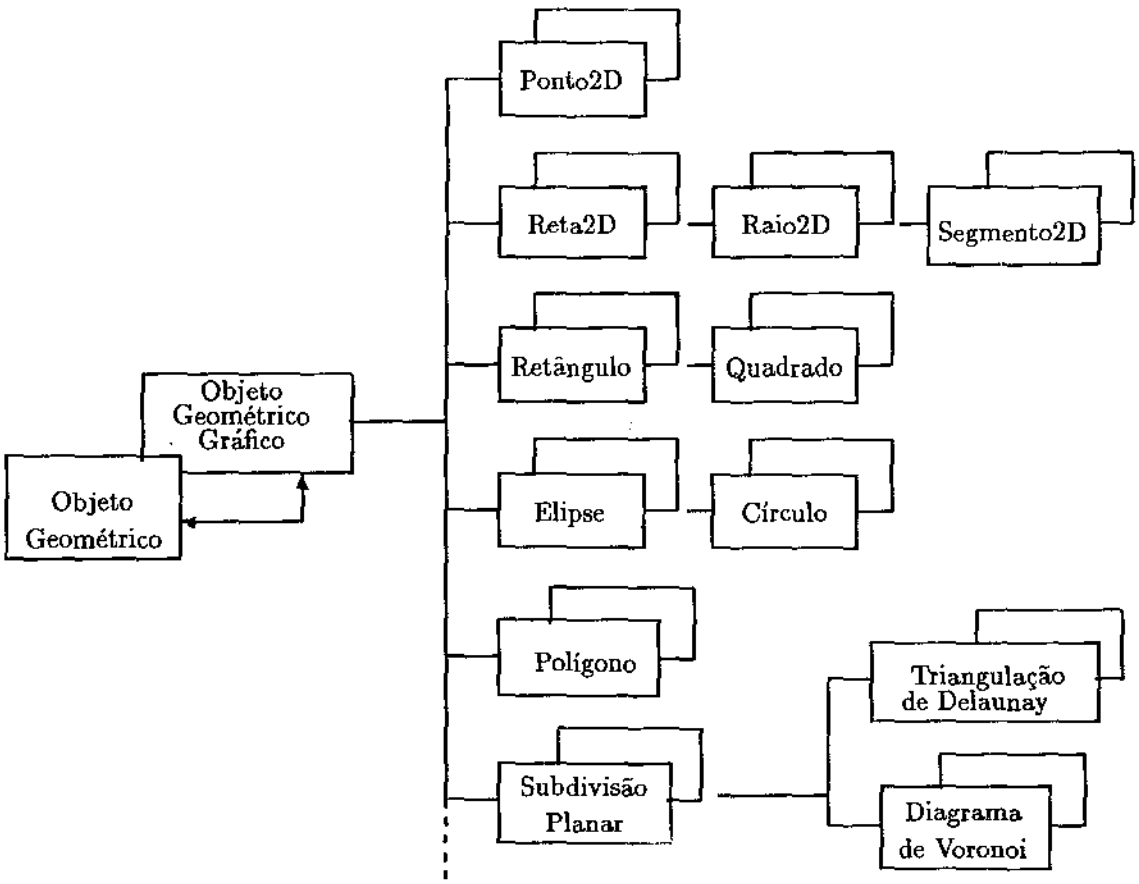

Figura 2.13: Parte da dupla hierarquia de objeto geométricos do **GeoLab**

- **Porque uma dupla hierarquia?**

  A função de uma dupla hierarquia de objetos geométricos é separar dados e métodos essenciais (e.g., o par de coordenadas que definem um ponto) de dados e métodos utilizados pelo **GeoLab** (e.g., cor, tipo de traço, rótulo).

[9] De fato, uma das observações decorrentes deste projeto é a falta de suporte ao paradigma de orientação a objetos disponível na versão corrente de um gerenciador de janelas supostamente de uso geral.

Para cada objeto geométrico "puro" (e.g., `Point2D`) existe um objeto geométrico "gráfico" associado. Enquanto objetos "puros" implementam dados e métodos específicos com relação à sua geometria, objetos "gráficos" implementam o conjunto de dados e métodos que definem o protocolo de interação utilizado pelo **GeoLab** para manipulá-los. Esta complementação de funcionalidade permite a criação de objetos compostos (e.g., um segmento é dado por dois pontos) sem a indesejável duplicação de informações que ocorreria se não houvesse a dupla hierarquia (i.e., um ponto "puro" traz consigo apenas informações sobre suas coordenadas, ao passo que um ponto gráfico possui informações extras como cor. Um segmento "puro" é dado por dois pontos "puros" ao passo que um segmento "gráfico" tem, além de dois pontos "puros", uma só informação sobre cor). A economia de memória e código para manipulação de tais objetos é uma decorrência imediata desta abordagem. Outra decorrência favorável desta organização é o estabelecimento de um protocolo através do qual o **GeoLab** comunica-se com os mais variados tipos de objetos de maneira uniforme. A criação de um novo tipo de objeto geométrico requer apenas a implementação deste protocolo para que este possa ser manipulado pelas várias ferramentas que compõem o módulo básico (veja seção 3.2).

Esta abordagem tem ainda uma outra característica positiva, que é a de tornar possível a implementação de algoritmos geométricos quase independentes do ambiente, já que estes restringem-se à manipulação de objetos "puros", conforme o esquema apresentado na figura 2.14.

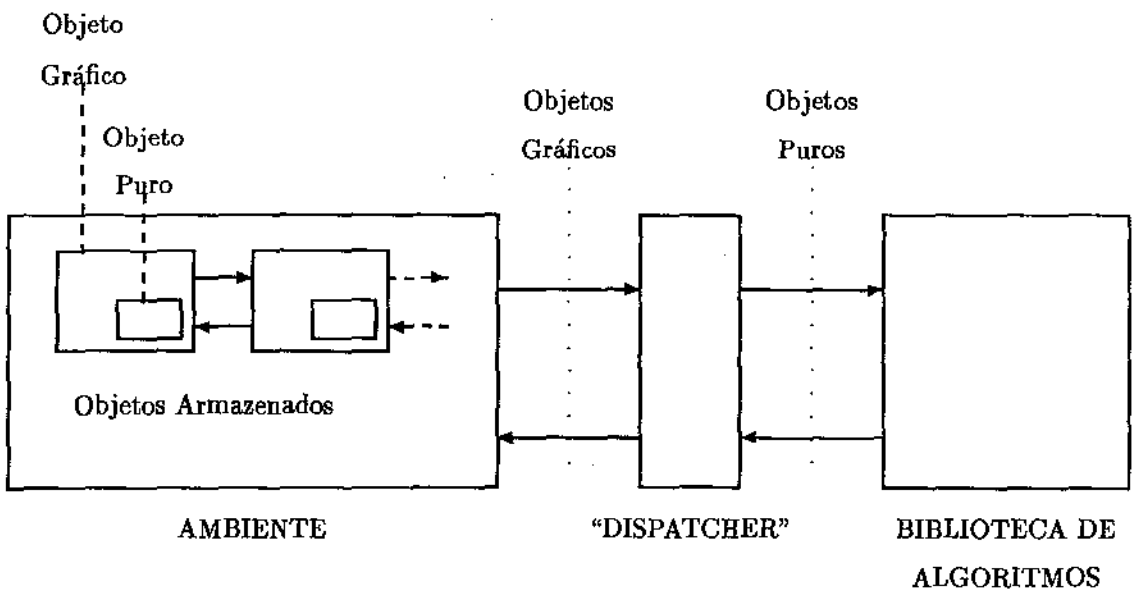

Figura 2.14: A relação entre a dupla hierarquia de objetos e os algoritmos geométricos – buscando soluções que permitam a criação de bibliotecas genéricas

Na figura, identifica-se a presença de um mediador entre o ambiente e a biblioteca de algoritmos geométricos. Sua tarefa é essencialmente esconder a existência do ambiente através da dupla hierarquia de algoritmos. A presença deste intermediador excede de certa forma o poder de expressão do paradigma de orientação a objetos (i.e., construtores virtuais). Detalhes sobre sua implementação são dados nos apêndices.

- **Especialização *versus* Caracterização Dinâmica**

  Do ponto de vista de um programador, a hierarquia apresentada na figura 2.13 imediatamente sugere várias centenas de linhas de código, além de um exagerado número de classes, mas não é exatamente isso que acontece no **GeoLab**.

  Em primeiro lugar, o grande número de objetos geométricos distintos requer, obviamente, um esforço razoável de implementação. Apesar disto, parte deste esforço é por nós evitado através da adoção de um mecanismo que chamamos **caracterização dinâmica de objetos**. Este mecanismo consiste em capacitar os objetos da hierarquia a informarem seus tipos e subtipos geométricos (e.g., *polígono* e *simples*, respectivamente) e, em muitos casos, isto é suficiente para evitar a derivação de classes (que envolve trabalho de implementação). Citamos como exemplo o caso da classe `Polígono`. Este tipo poderia ser especializado em um grande número de subclasses (e.g., simples, em forma de estrela, monotônico, convexo), tal como é feito em ambiente similares ao **GeoLab**, trabalho este que pode ser evitado com sucesso com o mecanismo de caracterização dinâmica. Em segundo lugar, isto por vezes torna-se *necessário*, pois a linguagem adotada não permite *migração de tipos* (i.e., se um polígono simples fosse uma subclasse e não uma característica de polígono, nada mudaria esta condição para uma dada instância de polígono simples durante a execução do sistema). Em muitos casos, a simulação de migração é útil (notadamente para algoritmos não construtivos, como o que verifica a simplicidade de um polígono) e a caracterização dinâmica de objetos geométricos a torna possível.

  Caracterização dinâmica e outros métodos introduzem um segundo protocolo, a nível de objetos "puros", voltados não só para a manipulação destes pelo ambiente, mas também para facilitar e reduzir o esforço de implementação em muitos casos. Apresentamos a seguir estes dois protocolos nas tabelas 2.1 e 2.2.

- **Suporte Geométrico**

  Operações geométricas básicas completam o conjunto de métodos disponíveis nos objetos geométricos. Em função dos variados tipos de objetos existentes, não existe um padrão fixo para estes métodos. Ao invés disso, procuramos cobrir uma grande quantidade de operações para cada objeto, tornando-os suficientemente completos a fim de facilitar a implementação de algoritmos geométricos diversos. Como veremos no próximo capítulo, ainda que algumas ferramentas necessitem ser implementadas, sempre é possível acrescentá-las externamente – via ligação dinâmica (i.e., sem a necessidade de recompilação do ambiente).

  Alguns objetos geométricos, notadamente aqueles utilizados mais freqüentemente em algoritmos (e.g., pontos, segmentos e polígonos), mereceram especial atenção. Para estes objetos existe uma quantidade significativamente maior de primitivas. A tabela 2.3 ilustra algumas primitivas para objetos do tipo `Ponto` em duas dimensões.

  Nós ressaltamos aqui a utilização de **Coordenadas Homogêneas** [NS79] como um dos itens que introduziram grande flexibilidade e robustez na implementação de objetos geométricos. Vários problemas de implementação (e.g., a representação e o tratamento geométrico para pontos no infinito) que surgem em algoritmos geométricos são uniformemente resolvidos com o uso desta técnica. Detalhes são postergados para os apêndices.

Método	*Tarefa*
type	retornar o tipo do objeto (e.g., `Segment2D_t`)
subtype	retornar (ou ajustar) o subtipo do objeto (e.g., `Vertical_t`)
new_GraphicObj	construtor virtual simulado
reproduct	auto-duplicador
to_base	converter a estrutura em uma representação base (HalfEdge)
from_base	extrair informações a partir de uma representação base
load	recuperar as informações sobre o objeto
save	escrever (de forma recuperável) as informações do objeto
print	produzir uma descrição textual do objeto
translate	transladar o objeto para uma dada posição
rotate	rotacionar o objeto
scale	ampliar ou reduzir o objeto
draw	desenhar o objeto em um dispositivo gráfico
draw_handles	desenhar as alças do objeto
select_test	verificar se o objeto está sendo apontado pelo *mouse* ou se está totalmente contido dentro de um retângulo de seleção
distance_to_point	calcular a menor distância de um ponto ao objeto
obj_volatile	retornar (ou ajustar) a condição de objeto *volátil* (veja seção 3.1.4.3)

Tabela 2.1: Protocolo de mensagens para objetos geométricos "puros"

2.2.2.3 Algoritmo

Algoritmo Geométrico é a última, e menos comum, abstração provida pelo **GeoLab**. Sua introdução não foi meramente estética, mas para atender a necessidades tanto no que diz respeito à integração de algoritmos ao ambiente quanto à modelagem de problemas em Geometria Computacional.

É papel desta abstração estabelecer um conjunto de métodos que são utilizados pelo ambiente para a sua comunicação com a biblioteca de algoritmos geométricos. Estes métodos implementam facilidades que auxiliam o ambiente a realizar tarefas como alimentação de entrada e saída, instanciação de tipos externos (vide seção 3.1) e mesmo o controle do menu de algoritmos geométricos disponíveis. O conjunto é propositalmente pequeno (vide tabela 2.4).

A abstração de algoritmos em classes permite a construção de hierarquias de algoritmos e portanto o mapeamento de problemas com características comuns (e.g., problemas que se utilizam de **varredura planar**). Isto serve como fator de organização e, em muitos casos, facilita e permite a reutilização de código em implementações de algoritmos.

O próprio ambiente utiliza-se desta abstração para duas classes de problemas especiais, que são **Métodos de Criação Interativa de Objetos Geométricos** e **Modos Funcionais**. Estas classes têm como finalidade permitir a adição de novos recursos ao **GeoLab** (estabelece-se então uma relação com os modos funcionais multiplexados comentados na seção 2.1.1).

É interessante notar que estas classes de problemas facilitam também a construção de algoritmos não *single-shot*. Um exemplo é a construção de um localizador de pontos sobre uma subdivisão planar. Este algoritmo pode ser introduzido no ambiente como um modo funcional (que ao ser acionado realiza o preprocessamento necessário sobre a subdivisão) e responde então aos acionamentos do *mouse*.

Método	*Tarefa*
int_start	iniciar a criação interativa do objeto
int_change	alterar o último valor registrado na criação interativa
int_end	finalizar a criação interativa do objeto
resize_start	iniciar o redimensionamento do objeto
resize_change	alterar o último valor registrado no redimensionamento
resize_end	finalizar o redimensionamento do objeto
select	selecionar o objeto
unselect	desfazer uma seleção
selected	verificar se um objeto está selecionado
glue	fixar as coordenadas do objeto
unglue	desfazer uma operação de glue
glued	verificar se um objeto está fixado
anchor	tornar um objeto não-removível[10]
unanchor	desfazer uma operação de anchor
anchored	verificar se um objeto é não-removível
set_handle	tornar as alças visíveis/invisíveis
set_color	alterar os atributos de cor do objeto gráfico
set_line_attr	alterar o tipo de traço usado para desenhar o objeto gráfico
set_draw_mode	ajustar modo de desenho para o objeto (e.g., cópia de bits, ou-exclusivo)
reproduct	auto-duplicador
load	recuperar as informações sobre o objeto
save	escrever (de forma recuperável) as informações do objeto
type	retornar o tipo do objeto
subtype	retornar (ou ajustar) o subtipo do objeto
get_external	identificar o algoritmo e o módulo onde tal objeto foi produzido para o caso de objetos externos
outros	todos os métodos existentes para objetos puros são também acessíveis pelos objetos gráficos através da interceptação de chamadas (note que não se trata de herança, e sim de uma dupla hierarquia)

Tabela 2.2: Protocolo de mensagens para objetos geométricos "gráficos"

Método	*Tarefa*
lessX	verificar se a coordenada X é menor que a de um dado ponto
lessY	verificar se a coordenada Y é menor que a de um dado ponto
equalX	verificar se dois pontos tem a mesma coordenada X
equalY	verificar se dois pontos tem a mesma coordenada Y
equal	verificar se dois pontos são iguais
lexicoXless	comparar lexicograficamente as coordenadas X Y de dois pontos
lexicoYless	comparar lexicograficamente as coordenadas Y X de dois pontos
circular_less	comparar circularmente (sentido anti-horário) dois pontos
left_turn	verificar se o ponto está a esquerda de uma reta direcionada dada por dois pontos
right_turn	verificar se o ponto está a direita de uma reta direcionada dada por dois pontos
colinear	verificar se três pontos são colineares
which_side	calcular a posição de um ponto com relação a uma reta direcionada
area	calcular a área de um triângulo (dados três pontos)
inCircle	calcular a posição de um ponto com relação a um círculo dado por três pontos
e_distance	calcular a distância euclidiana entre dois pontos
e_distance_sqr	calcular o quadrado da distância euclidiana entre dois pontos
angle	calcular o ângulo formado pela reta dada por dois pontos e o eixo X
gravitycentre	calcular o centro de gravidade de um triângulo (dados três pontos)
incentre	calcular o incentro de um triângulo
baricentre	calcular o baricentro de um triângulo
middle	calcular o ponto médio de um segmento dado por dois pontos

Tabela 2.3: Resumo de primitivas geométricas para objetos do tipo Ponto em duas dimensões

Método	*Tarefa*
is_able_to_handle	verifica se o algoritmo é aplicável a um dado conjunto de entrada
animation	verificar se o algoritmo possui código para animação
binder	extrair os dados de entrada passados pelo **GeoLab** e organizá-los da forma que for adequada para o algoritmo
unbinder	desalocar memória e realizar tarefas de *clean-up* em modos de construção interativa e modos funcionais externos
instantiation	produzir instâncias de objetos geométricos externos ao **GeoLab** e que são criados pelo algoritmo

Tabela 2.4: Protocolo de mensagens para Algoritmos Geométricos

Capítulo 3

Programando através do GeoLab

Este capítulo descreve as técnicas e conceitos incorporados de forma a permitir o crescimento incremental de nosso ambiente de desenvolvimento de algoritmos geométricos. Os quatro blocos externos apresentados na figura 2.1 representam as fronteiras em expansão do **GeoLab**, e incluem, além de algoritmos e objetos geométricos, modos para criação interativa de objetos e modos funcionais.

Com esta abordagem, tentamos criar uma separação entre o que é o ambiente propriamente dito e o que são as componentes em desenvolvimento. O impacto imediato, que consideramos de extrema importância, é a possibilidade de que vários usuários trabalhem concorrentemente e independentemente no desenvolvimento de módulos externos, modelando o conjunto de ferramentas que utilizam de acordo com suas necessidades particulares, tal como ocorre em ambientes de programação em geral (e.g., UNIX).

Nós nos baseamos em recursos poderosos disponíveis nas máquinas hospedeiras para chegar a este resultado. A adição de módulos é feita através de um mecanismo chamado *ligação dinâmica*[1], que possibilita a ligação em tempo de execução de *bibliotecas compartilhadas*[2]. Estas bibliotecas correspondem, por sua vez, a esforços de programação reaproveitáveis, e constituem-se na parcela exportável do **GeoLab**.

As seções a seguir descrevem os quatro tipos de módulos externos, abordando, a nível de conceitos, sua criação, ligação e exportação pelo ambiente. Detalhes de programação são reservados ao "Guia de Referência" do **GeoLab**, apresentado nos apêndices.

3.1 Algoritmos Geométricos sob o GeoLab

Chamamos *algoritmo geométrico* entidades que realizam processamento sobre objetos geométricos e produzem como resultado *objetos geométricos*. A princípio, o **GeoLab** não conhece nenhum algoritmo geométrico, pois estes são externos ao ambiente (i.e., o ambiente sobrevive sem eles). De fato, o ambiente conhece apenas um protocolo (vide tabela 2.4) através do qual comunica-se com algoritmos indicados pelo usuário.

[1] *Dynamic link.*
[2] *Shared libraries.*

3.1.1 Relacionamento com Objetos Geométricos

Da forma como caracterizamos algoritmos geométricos, excluímos algoritmos implementados como métodos dos objetos pertencentes à hierarquia de objetos geométricos. A razão para isto é: separar operações primitivas de aplicações[3].

A distinção entre operações primitivas e aplicações (algoritmos geométricos) é uma tarefa que cabe tanto ao projetista do ambiente quanto a seus usuários, já que a estes também é permitido incluir operações primitivas no ambiente. Para determinar se um dado algoritmo deve ser implementado como um método de um particular objeto ou um *algoritmo geométrico* externo utilizamos a dependência de representação como fator de separação, i.e., algoritmos que dependem fortemente da representação interna de um particular objeto geométrico devem ser implementados como métodos deste objeto. Os demais algoritmos encaixam-se na categoria de aplicações (i.e., *algoritmos geométricos* externos).

Um exemplo da aplicação desta distinção é o caso do algoritmo que decide se um dado ponto p está à direita de uma reta direcionada r, determinada por dois pontos a e b (figura 3.1).

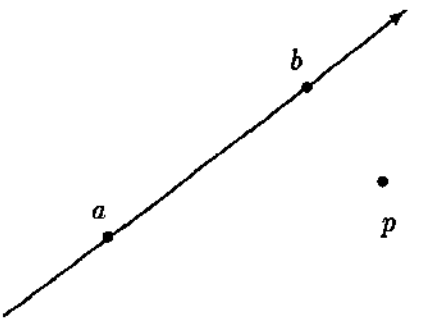

Figura 3.1: A posição de um ponto com relação a uma reta direcionada

O algoritmo clássico para a resolução deste problema consiste em calcular a área sinalada do triângulo formado por p, a e b e em seguida analisar seu sinal [PS85]. Este cálculo é feito em função do determinante:

$$\mathcal{A}(p,a,b) = \begin{vmatrix} p.x & p.y & 1 \\ a.x & a.y & 1 \\ b.x & b.y & 1 \end{vmatrix}$$

Caso o resultado seja negativo dizemos que p está à direita de $\overrightarrow{ab}$; se positivo, que está à esquerda e se igual a 0, que os pontos p, a e b são colineares.

A implementação desta operação não implica em dependência de representação para objetos do tipo Ponto se suas coordenadas puderem ser obtidas externamente (idealmente via métodos). Assim sendo, tal operação poderia ser implementada como um algoritmo geométrico externo.

Por outro lado, nossa implementação de objetos do tipo Ponto utiliza-se de **coordenadas homogêneas** [NS79]. Nesta representação, um ponto em duas dimensões é dado por uma tripla de

[3] Leitores acostumados à programação no ambiente UNIX [Bac86, KP84] devem lembrar-se da filosofia de implementação deste poderoso sistema operacional: a separação entre um núcleo conciso e consistente (i.e., *kernel* acessado através de *system calls*) e o conjunto de comandos e aplicações implementados sobre este núcleo.

valores, X, Y e W. Sua coordenada x é definida como sendo X/W e a coordenada y como Y/W. Quando W é igual a zero interpretamos o ponto como estando no infinito na direção (X,Y).

Esta representação viabiliza uma implementação alternativa para a operação em questão, baseada no produto entre os vetores que representam os pontos.

1. Seja $p = \begin{bmatrix} p_1 & p_2 & p_3 \end{bmatrix}$, $a = \begin{bmatrix} a_1 & a_2 & a_3 \end{bmatrix}$, e $b = \begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix}$.

2. A reta direcionada r determinada por a e b é dada por:

$$r = \begin{bmatrix} b_3a_2 - b_2a_3 \\ b_1a_3 - b_3a_1 \\ b_2a_1 - b_1a_2 \end{bmatrix} \text{ ou } r = a \times b \text{ (produto)}$$

3. A posição do ponto p com relação a r é então dada pelo produto interno entre p e r tal que:

$$p \cdot r \begin{cases} = 0 & p \text{ está sobre a reta } r \\ < 0 & p \text{ está a direita de } r \\ > 0 & p \text{ está a esquerda de } r \end{cases}$$

Como esta alternativa depende fortemente da representação interna do objeto Ponto, sua implementação é mais adequada como um método deste objeto.

A abordagem de algoritmos segundo a metodologia apresentada caracteriza uma das diferenças básicas entre o **GeoLab** e outros projetos relacionados, como o XYZ Geobench[4] [Kni90, Sch91]. Nestes projetos, a implementação de algoritmos geométricos como métodos das classes manipuladas é a única organização utilizada. Observamos abaixo que isto nem sempre é o mais adequado, aproveitando para ressaltar as características positivas de nossa abordagem:

- Organização Semântica

  Limitar algoritmos a serem métodos das classes que eles manipulam pode causar confusões, pois pode não ficar claro quais métodos devem pertencer a que classes no caso destes métodos manipularem mais de uma classe.

- Independência de Representações

  Se um algoritmo é independente de uma representação (i.e., manipula um tipo abstrato), implementá-lo como um método de uma particular classe pode restringir sua utilização com diferentes representações do mesmo objeto que possuam um mesmo repertório de métodos.

- Restrições quanto à Caracterização Dinâmica

  Como discutido anteriormente (seção 2.2.2.2), nem sempre é necessário ou desejável introduzir novos objetos geométricos através da derivação de classes. Em algumas situações a caracterização dinâmica pode ser mais interessante e, nestes casos, a implementação de algoritmos como métodos destas classes teria que considerar a introdução de código extra para tornar tais métodos sensíveis ao mecanismo de caracterização dinâmica.

[4] eXperimental geometrY Zürich Geobench.

3.1.2 Bibliotecas de Algoritmos Geométricos

Criar algoritmos geométricos através do **GeoLab** consiste em implementar tais algoritmos como bibliotecas compartilhadas, externas ao ambiente. Isto imediatamente sugere que, se bem estruturadas, tais bibliotecas tornam-se utilizáveis não somente pelo **GeoLab** mas também em outras aplicações. Estruturar bem um algoritmo significa implementá-lo de forma auto-contida, sem incluir recursos dependentes de uma particular aplicação que não possam ser desconsiderados por outra.

Nosso ambiente favorece a criação de bibliotecas bem estruturadas através de dois mecanismos[5] que possibilitam isolar código dependente do ambiente (e.g., código para animação) de código específico do algoritmo geométrico, que são:

- Parâmetros Inicializados

  Parâmetros que definem valores para variáveis dependentes do ambiente (e.g., profundidade de animação) são automaticamente inicializados de forma a tornarem-se transparentes (e sem uso) para outras aplicações.

- Compilação Condicional

  Código dependente do ambiente é isolado de código específico do algoritmo através de compilação condicional para prevenir o caso da biblioteca vir a ser recompilada para ser utilizada por outra aplicação. Neste caso o código dependente é automaticamente eliminado.

Como exemplo, apresentamos um fragmento de código (figura 3.2) que é parte da implementação do algoritmo para o cálculo da envoltória convexa de um conjunto de pontos no plano.

Virtualmente todos os algoritmos geométricos implementados através do **GeoLab** tornam-se utilizáveis por outras aplicações desde que o usuário-programador observe a aplicação destes mecanismos. Objetos geométricos e estruturas de dados (e.g., listas, árvores) utilizados pelo algoritmo obviamente também devem ser incorporados a estas aplicações.

3.1.3 Ligação de Algoritmos Geométricos

Como mencionamos anteriormente, cabe ao usuário-final identificar quais algoritmos devem ser ligados ao ambiente em tempo de execução. Uma vez ligados, estes algoritmos são manipulados através de um protocolo.

A indicação de quais algoritmos devem ser ligados é feita textualmente, através de arquivos de configuração. Diferentes usuários podem ter diferentes arquivos de configuração de acordo com os algoritmos que lhes interessam. Isto é particularmente interessante pois permite o trabalho independente sobre diferentes conjuntos de problemas por vários usuários ao mesmo tempo.

Abaixo descrevemos a estrutura do arquivo de configuração e em seguida abordamos o protocolo de comunicação para algoritmos.

3.1.3.1 O Arquivo *.geolab-menu*

Cada algoritmo ligado dinamicamente ao **GeoLab** requer uma linha em um arquivo lido pelo ambiente no início de sua execução, contendo o nome do algoritmo e o módulo onde este se encontra. Estas

[5]Estes mecanismos são providos pela linguagem utilizada, e seu uso faz parte das várias recomendações feitas ao usuário-programador do **GeoLab** (veja o apêndice B).

```
list(Point2D_ptr) graham(list(Point2D_ptr) & points,
                         World *w = 0, int anim_depth = 0, int ref = 0) {

    Point2D_ptr vi, vi1, vi2;
    if (!points.empty()) {

      // ... ommited code ...
      // circular sorting of the input points around
      // the one with smallest  Y coordinate

      vi = points.first();
      do {
        vi1 = points.cyclic_succ(vi);
        vi2 = points.cyclic_succ(vi1);

#ifdef _GEOLAB_ANIMATION
        ANIMATION_BEGIN(w, anim_depth, ref)
          mark_points(w, vi, vi1, vi2);
          frame_stop_mark(w);
        ANIMATION_END(w, anim_depth, ref)
#endif

        side = points[vi]→which_side(*points[vi1], *points[vi2]);
        if (side == scomp || (side == Colinear && vi2 ≠ vi)) {
          points.del_item(vi1);
          if (vi ≠ points.first())
            vi = points.pred(vi);
        }
        else
          vi = points.succ(vi);
      } while (vi1 ≠ points.first());
    }
    return points;
}
```

Figura 3.2: Fragmento de algoritmo geométrico com destaque para parâmetros inicializados e código para animação isolado (compilação condicional)

linhas de descrição podem ser estruturadas com a utilização de uma pequena linguagem de controle que oferece recursos para a introdução de informações estruturais para a construção do menu de opções de algoritmos geométricos do ambiente (figura 3.3).

```
/* Isto é um comentário */

TITLE "Algorithms"                          /* Título do menu principal */

"Option #1" "link_function_name1" "module_name1"
"Option #2" "link_function_name2" "module_name2"

  MENU "Submenu"                            /* Título de um submenu */
    "Option #3" "link_function_name3" "module_name3"
    "Option #4" "link_function_name4" "module_name4"
  END
```

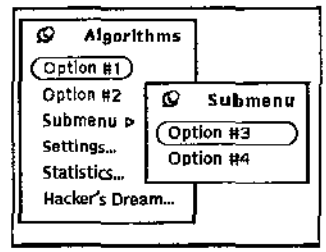

Figura 3.3: Arquivo *.geolab-menu* e o correspondente menu de opções

Como pode ser visto na figura 3.3, cada opção no menu de algoritmos indica o nome de uma *função de ligação* (*link function*) e o nome da *biblioteca compartilhada* a ser ligada (*dynamic module-name*). A função de ligação é invocada tão logo a biblioteca tenha sido ligada, e seu resultado é a produção de uma instância do algoritmo sendo ligado. De posse desta instância, o ambiente pode então recorrer ao protocolo de comunicação para manipular o algoritmo.

3.1.3.2 Protocolo de Comunicação para Algoritmos Geométricos

Algoritmos são mapeados em uma hierarquia de classes e acessados através de um protocolo conforme a descrição abaixo[6]:

- Disponibilidade de Animação – *animation*

  Um algoritmo geométrico deve responder ao ambiente se possui animação (existência de código específico para animação). O ambiente utiliza esta informação para orientar seus usuários a respeito de quais algoritmos podem ser animados (através de sua interface gráfica).

- Habilidade de Tratar Objetos Geométricos – *is_able_to_handle*

  Um algoritmo deve também saber informar ao ambiente se um dado conjunto de objetos geométricos é aceitável como entrada. Isto é utilizado para habilitar/desabilitar opções no menu de algoritmos em função dos objetos que estão selecionados no editor.

[6]Veja tabela 2.4 para um resumo desta descrição.

- Instanciação de Objetos Geométricos – *instantiation*

  Algoritmos que criam objetos geométricos externos ao ambiente devem prover ao ambiente instâncias destes objetos através deste método. Isto é necessário pois o ambiente por si só não tem condições de gerar estas instâncias quando está, por exemplo, carregando objetos externos de memória secundária. Assim, o ambiente recorre aos algoritmos geométricos que produziram tais objetos para realizar esta tarefa. Nós detalhamos este mecanismo na seção 3.2.

- Invocação de Algoritmos Geométricos – *binder*

  A execução do algoritmo propriamente dita é feita através de uma interface (veja figura 2.14) especialmente implementada para extrair objetos a serem processados pelo algoritmo da lista de objetos selecionados do **GeoLab**. Esta interface divide funções com o ambiente e é responsável por organizar os objetos de entrada de forma adequada para o algoritmo em questão (e.g., montar um vetor de pontos para o algoritmo que constrói o diagrama de Voronoi).

- *Clean-up* em modos externos – *unbinder*

  Modos externos complexos, notadamente modos funcionais que controlam a execução de algoritmos que realizam preprocessamento, podem utilizar-se do método *unbinder* para realizar tarefas como desalocação de memória. *unbinder* é acionado automaticamente pelo ambiente quando o modo externo é desativado ou quando um novo modo é instalado.

A figura 3.4 mostra uma implementação do protocolo de comunicação para algoritmos geométricos para o problema do cálculo do par mais próximo em um conjunto de pontos[7].

3.1.4 Consumo e Produção de Objetos Geométricos

A característica principal do mapeamento de algoritmos geométricos em classes (e o conseqüente estabelecimento de um protocolo entre ambiente e algoritmos) é a divisão de tarefas entre ambiente e usuário-programador. Cada algoritmo possui necessidades próprias que são difíceis (ou mesmo impossíveis) de prever, e um tratamento mais rígido por parte do ambiente resultaria em restrições que afetariam mais cedo ou mais tarde o objetivo maior, que é a generalidade.

Como se pode ver, grande parte de implementação do protocolo consiste de programação trivial (como *animation*, que só requer uma linha de código). O único método que apresenta maior dificuldade é o de ligação (transferência de dados) entre ambiente e algoritmo (*binder*). Este método tem como função receber dados do ambiente e organizar os resultados produzidos pelo algoritmo, transferindo-os então para o ambiente.

A seguir são feitas considerações com relação ao método de ligação, explorando três possíveis fases de sua execução.

[7] O algoritmo propriamente dito é mostrado no próximo capítulo.

```
class ClosestPair : public ProximityProblems {
    bool animation() { return true; }

    bool is_able_to_handle(const Selected &SO, const Graphic &GO) {
        if (objects_type(SO, GO) ≠ Point2D_t)
          return false;
        else
          return true;
    }

    Obj_ptr instantiation(ObjType t) {
        if (t == UndirectedEdge_t)
          return new UndirectedEdge();
        else
          // error
    }

    Result binder(const Selected &SO, const Graphic &GO, World *w, int anim_depth, int ref) {
        Result result;
        UndirectedEdge_ptr cpair;

        if (!SO.empty()) {
          if ((cpair = closest_pair(extract_points(SO, GO), w, anim_depth, ref)) ≠ 0)
            result.push(cpair);
        }

        return result;
    }
};
```

Figura 3.4: Exemplo de implementação do protocolo para algoritmos geométricos

3.1.4.1 Obtendo Entradas para Algoritmos Geométricos

Entradas para algoritmos geométricos são obtidas através das estruturas recebidas como parâmetros pelo método de ligação (*binder*). São duas listas que representam, respectivamente (figura 3.5):

1. Os objetos geométricos no editor; e

2. Os objetos selecionados no momento.

A entrada para um algoritmo é obtida percorrendo-se a lista de objetos selecionados e organizando os objetos geométricos correspondentes da forma que for mais adequada para o dado algoritmo. A lista de objetos geométricos mantém os objetos na forma "gráfica". O método de ligação deve ser responsável por extrair os objetos puros (contidos nos objetos gráficos) para então passá-los para o algoritmo.

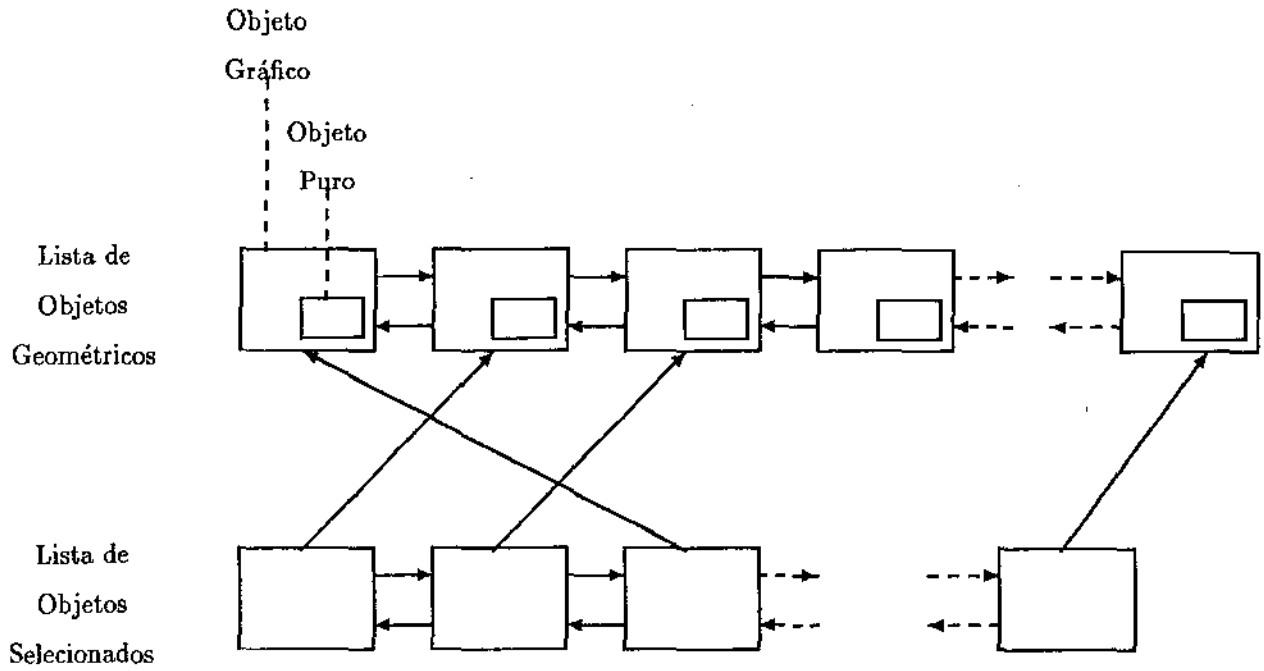

Figura 3.5: Listas de objetos manipulados pelo **GeoLab**

Quando a entrada sofre alterações em um algoritmo (e.g., pontos são eliminados na construção de uma envoltória convexa), deve-se copiar os objetos que serão afetados, pois as listas pertencentes ao **GeoLab** não podem ser alteradas (são protegidas por recursos da linguagem).

A razão para deixar que tais estruturas sejam percorríveis por algoritmos externos é evitar o tempo e memória utilizados para a coleta (duplicação) dos objetos geométricos pelo ambiente, sua transmissão para o algoritmo sendo chamado e sua subseqüente destruição. Em muitos casos (como no caso de modos de construção interativa – veja seção 3.3) isto não é sequer necessário.

3.1.4.2 Conversões entre Representações

Uma das dificuldades do usuário-programador na elaboração do método de ligação é a conversão entre representações para objetos geométricos.

Grande parte dos algoritmos existentes em Geometria Computacional faz uso de estruturas de dados criadas especificamente para eles, seja para tornar a implementação possível, seja para assegurar eficiência. Conversões entre representações tornam-se inevitáveis a partir do momento em que consideramos o resultado de um dado algoritmo não como um produto final, mas como um novo objeto que pode tornar-se alvo de processamento por outros algoritmos.

Um exemplo clássico são as várias estruturas para a representação da topologia de modelos geométricos que surgiram após a proposta inicial feita por Baumgart [Bau75]:

- Doubly Connected Edge List – DCEL [PS85]
- Winged-Edge [MS82]
- Half-Edge [Män88]

- Quad-Edge [GS85]
- Radial-Edge [Wei86]

As diferenças significativas entre estas estruturas torna difícil sua utilização de forma intercambiável em algoritmos geométricos, embora representem uma mesma classe de modelos geométricos (com maior ou menor abrangência).

Outros projetos relacionados [Kni90, Sch91] abordaram este assunto de forma bastante ineficiente: para cada uma de n representações utilizadas de um mesmo objeto geométrico são construídos pelo menos n conversores. Assim, para vários tipos de objetos com múltiplas representações, teremos um número proibitivo de conversores.

No **GeoLab**, este assunto foi abordado procurando-se minimizar o número de conversores entre diferentes representações através da introdução de uma **estrutura base**. Cada objeto deve ser capaz de converter-se na estrutura base e também de reconstruir-se a partir dela (figura 3.6). Isto limita em 2 o número de conversores para objetos com múltiplas representações (i.e., dois conversores para cada representação) e, como veremos na seção 3.2.2.2, torna os conversores úteis em outros aspectos da implementação de objetos geométricos.

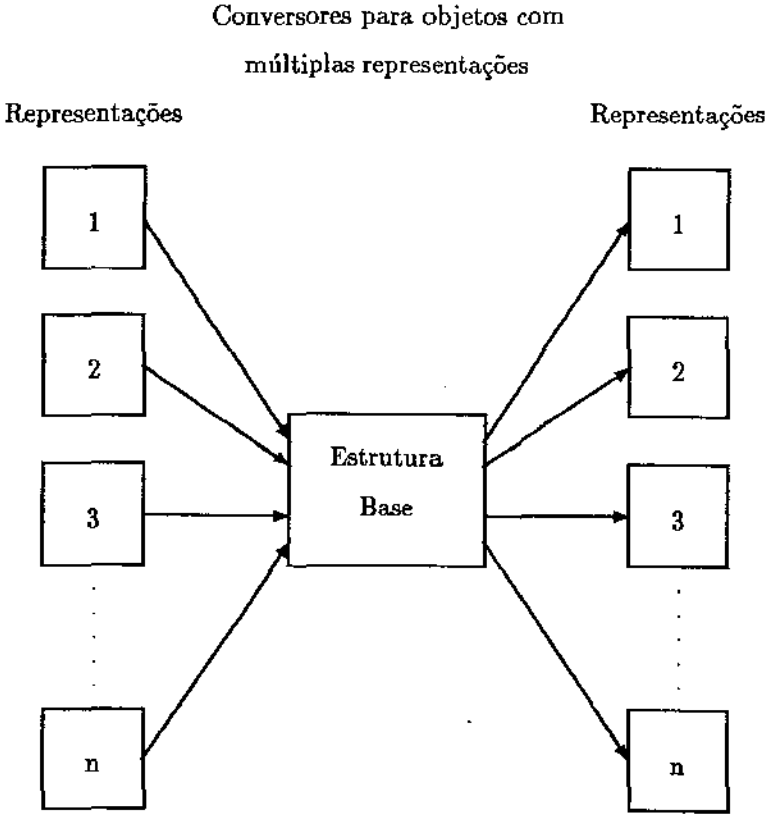

Figura 3.6: Esquema conceitual dos conversores entre diferentes representações de um mesmo objeto geométrico

A estrutura de dados base escolhida para representação padrão é conhecida por *HalfEdge* e descrita em detalhes em [Män88].

3.1.4.3 A Saída de um Algoritmo Geométrico

Por razões de homogeneidade, todo resultado de processamento por algoritmos geométricos é encarado como "objeto geométrico". A quantidade e o tipo destes resultados pode variar caso a caso, e o ambiente não faz restrições neste sentido. Objetos de saída são organizados em uma lista de resultados que é devolvida ao ambiente e este, por sua vez, encarrega-se de realizar o encapsulamento de objetos "puros" em objetos "gráficos" e de mostrá-los ao usuário.

Algoritmos que produzem dados não-geométricos (como o valor da área de um polígono) também encaixam-se neste modelo pois foram implementadas classes especiais para sua manipulação. Inteiros, reais, booleanos e mensagens são todos mapeados em "objetos geométricos" cuja função é transportá-los e representá-los graficamente no ambiente. Este recurso, aliado à possibilidade de caracterizar um objeto como sendo *volátil* (i.e., um objeto que é apenas mostrado graficamente e então descartado) torna possível a comunicação de vários tipos de informações entre os algoritmos e o ambiente que não apenas dados geométricos (e.g., mensagens de erro, valores numéricos).

3.2 Incorporando Novos Objetos Geométricos

Conforme apresentado anteriormente, objetos geométricos são organizados em uma dupla hierarquia de classes em C++, e um conjunto razoável de tais objetos já é interno ao ambiente (e.g., ponto, segmento, retângulo, ...). A existência deste conjunto atende a necessidades prementes como os modos básicos para a criação interativa de objetos geométricos e a capacitação geométrica inicial do ambiente.

Existem duas formas de se incorporar novos objetos geométricos no ambiente. A primeira é sua produção por algoritmos geométricos e a segunda á sua criação através de modos de construção interativa (veja seção 3.3).

3.2.1 O Papel da Dupla Hierarquia

Novos objetos geométricos são introduzidos no ambiente ampliando-se a dupla hierarquia de objetos (i.e., objetos puros e gráficos). Estas hierarquias definem os protocolos através dos quais o ambiente comunica-se com os objetos. Tendo tais protocolos implementados, um novo objeto passa a ser manipulável pelo ambiente através de sua interface de forma homogênea.

Novos tipos ampliam a dupla hierarquia de objetos, seja pela adição de novas classes, seja pela especialização de objetos já existentes. Esta última possibilidade permite que sejam criados algoritmos geométricos que operem sobre diferentes representações de uma mesma entidade. Para tanto, basta definir o conjunto necessário de métodos em uma classe abstrata e então implementá-los nas especializações. Este é um importante mecanismo, proporcionado pelo uso de uma linguagem com suporte para orientação a objetos, para a implementação e teste de algoritmos com independência de representações.

3.2.2 Armazenamento e Recuperação de Novos Objetos

3.2.2.1 Instanciação de Objetos Geométricos Externos

Um interessante problema de programação surge com a necessidade de armazenar e recuperar objetos geométricos externos em memória secundária.

Da forma como escolhemos incorporar este recurso ao ambiente, fica a cargo de cada objeto escrever ou ler seu conteúdo em um descritor de arquivo. Para tal, torna-se necessário ao ambiente simplesmente ter uma instância do objeto para então efetuar a chamada ao método correspondente.

Salvar os objetos no editor é uma tarefa trivial para o ambiente. Para tanto, basta solicitar a todos os objetos na lista de objetos gráficos que escrevam seus dados em um descritor de arquivos. Recuperar estas informações requer um pouco mais de trabalho.

Quando encontra um objeto geométrico a ser carregado (existe um campo no arquivo de dados que identifica o tipo do objeto que se segue), o ambiente necessita obter uma instância do tipo correspondente para então solicitar a esta instância que carregue seus dados. Para objetos que já fazem parte do ambiente isto não constitui problema. Entretanto, no caso de objetos externos, o ambiente não é capaz de criar por si só uma instância do objeto (que não era conhecido no momento de sua compilação). Ao descobrir esta impossibilidade, o que se faz então é identificar a origem do objeto (i.e., o algoritmo geométrico que o produziu) através de informações armazenadas juntamente com ele. Feito isso, o ambiente solicita ao algoritmo a criação de uma instância do objeto para então proceder à sua carga.

Mesmo um objeto originário de modos de criação interativa externos (seção 3.3) é tratado desta forma pois, como veremos, modos de criação interativa correspondem a tipos especiais de algoritmos geométricos.

3.2.2.2 O Papel dos Conversores entre Representações

Conversores entre representações podem facilitar grandemente o trabalho de implementação de novos algoritmos geométricos, principalmente no que diz respeito ao seu armazenamento e recuperação em memória secundária.

Uma vez que a representação básica dispõe de métodos para salvamento e recuperação de suas informações, a implementação de conversores nos novos objetos evita a necessidade de se implementar métodos próprios para salvamento e recuperação[8]. Para salvar-se, basta então converter-se para a estrutura base e ordenar que esta conversão escreva seu conteúdo no descritor de arquivo. Para recuperar a informação salva executa-se os passos na ordem inversa (i.e., carrega-se a estrutura base e a converte de volta para o objeto original).

3.3 Modos de Criação Interativa e Modos Funcionais

A habilidade de incluir novos algoritmos e objetos geométricos aliada à abstração de um sistema de visualização (*World*) possibilitaram a definição de duas classes especiais de algoritmos geométricos, cuja função é tornar também os modos de interação do editor gráfico passíveis de expansão por ferramentas externas. Estas classes correspondem a *Modos de Criação Interativa de Objetos* e *Modos Funcionais*.

Conforme apresentado na seção 2.1.1, novos modos de interação são instalados em botões polivalentes especialmente colocados na interface gráfica do ambiente, e sua utilização se dá de forma idêntica aos modos internos (veja apêndice A). O usuário-programador do ambiente beneficia-se deste

[8] Na verdade, esta abordagem não elimina trabalho de implementação, apenas o transfere para um lugar onde pode ser mais útil, evitando também o envolvimento de manipulação de memória secundária por parte do usuário-programador ao inserir novos objetos geométricos.

recurso não só através da criação de novas ferramentas para manipulação de modelos geométricos, mas também para a modelagem e construção de diversos algoritmos geométricos que envolvem interação.

A funcionalidade disponível para a criação de novos modos depende fortemente do modelo de manipulação de eventos (i.e., ações realizadas pelo usuário sobre os dispositivos de interação como *mouse* e teclado) que passamos a discutir em seguida.

3.3.1 Encadeamento de Tratadores de Eventos

A interação com aplicações implementadas sobre ambientes de janelas utiliza-se de um mecanismo chamado *tratamento de eventos* como forma de interpretação de ações executadas por seus usuários [Xav92]. Chamamos *eventos* às ações de comunicação com a aplicação através dos dispositivos conectados à maquina hospedeira (e.g., uma tecla pressionada, movimentação do *mouse*, etc.).

Aplicações implementadas sobre o XView apoiam-se na existência de um gerente de janelas que recebe eventos e dispara funções devidamente cadastradas para sua manipulação (estas funções recebem o nome de *callback functions*). O cadastramento destas funções é realizado na fase de inicialização das componentes da interface de uma aplicação e a entidade responsável por manter tais informações e acioná-las é chamada de *notifier*. Mesmo programas que não utilizam o XView simulam seu funcionamento substituindo o *notifier* por um *loop* de eventos cuja função é essencialmente a mesma. A figura 3.7 apresenta a estrutura típica de aplicações nestes moldes [Hel90].

Para permitir que novos modos interativos pudessem ser adicionados dinamicamente ao **GeoLab**, tornou-se necessário estender o modelo tradicional de tratamento de eventos[9]. A extensão foi por nós chamada de *encadeamento de tratadores de eventos* e é realizada pelo sistema de visualização. No modelo estendido, possivelmente mais de um tratador de eventos pode ser cadastrado para um mesmo elemento da interface, e este cadastramento pode ser realizado e alterado a qualquer momento durante a execução da aplicação.

A ocorrência de eventos faz com que os tratadores cadastrados sejam acionados em seqüência até serem esgotados, ou até que um dos tratadores comunique que o evento não deva ser propagado para os tratadores subseqüentes (neste caso, dizemos que o evento foi *consumido*). Isto permite a descentralização no tratamento de eventos, o que facilita sua implementação basicamente por permitir o reaproveitamento de tratadores em situações diversas (tratadores passam a ser vistos como *filtros*, desempenhando pequenas tarefas, cooperando para a realização de tarefas mais complexas através do encadeamento[10]). Nós ilustramos a extensão criada através da figura 3.8 .

No caso do **GeoLab**, vários tratadores são de uso geral, facilitando o trabalho de implementação de ferramentas e homogeneizando o comportamento da interface. Alguns destes tratadores são:

1. Compressor de eventos – Tratador responsável por comprimir eventos de movimentação do *mouse*.

2. Restritor de Movimentos – Responsável por auxiliar o desenho de objetos sobre eixos verticais ou horizontais.

3. Desenhador de objetos simples – Objetos como pontos, segmentos, círculos são manipulados por um mesmo tratador de eventos.

[9] É digno de nota que o XView não suporta o conceito de objetos na criação de tratadores de eventos (i.e., métodos) e este problema também é solucionado por esta extensão.

[10] Note-se mais uma vez aqui a influência da filosofia do sistema operacional UNIX no desenvolvimento deste projeto.

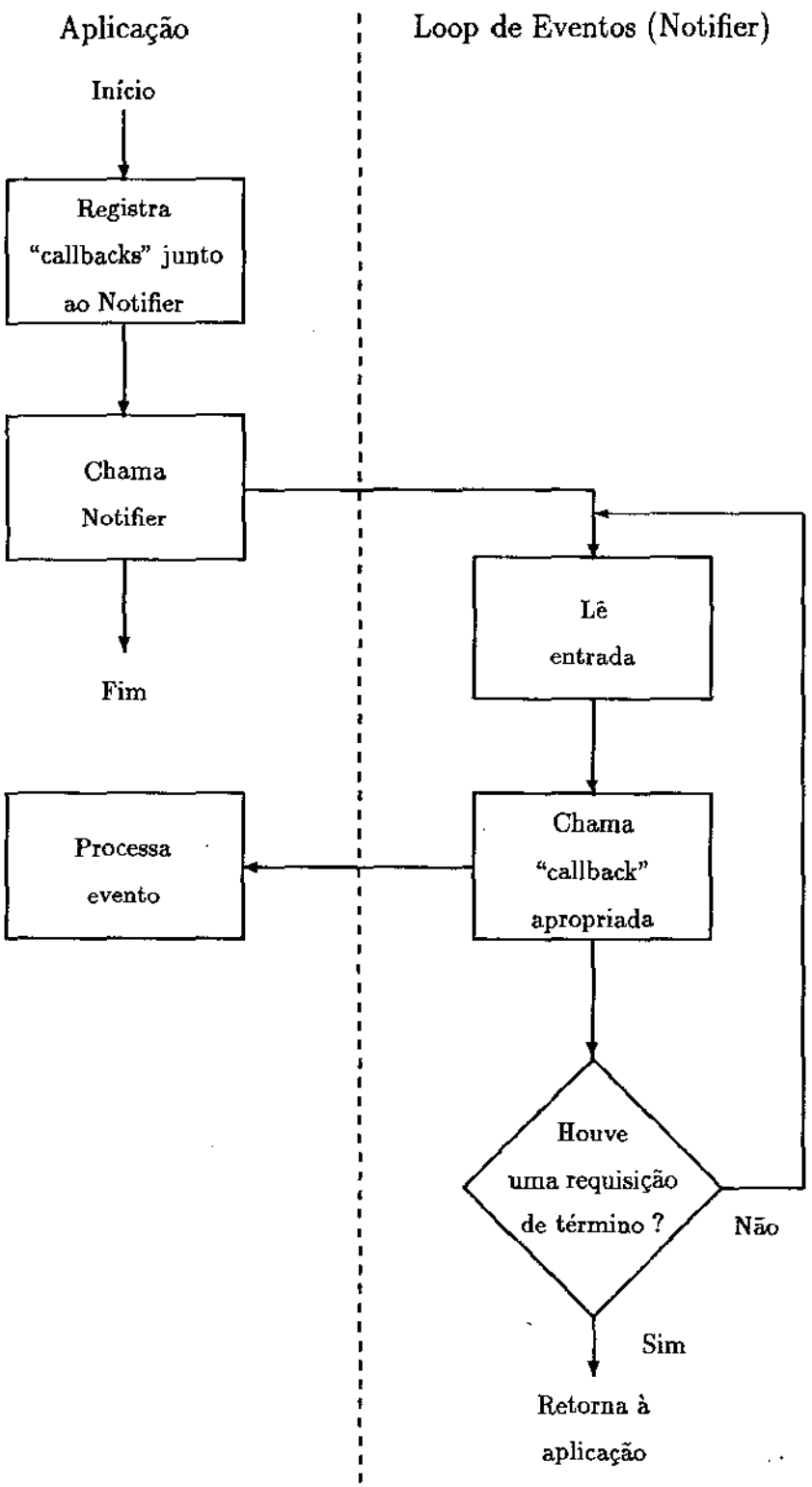

Figura 3.7: Fluxo de controle em uma aplicação construída sobre o XView

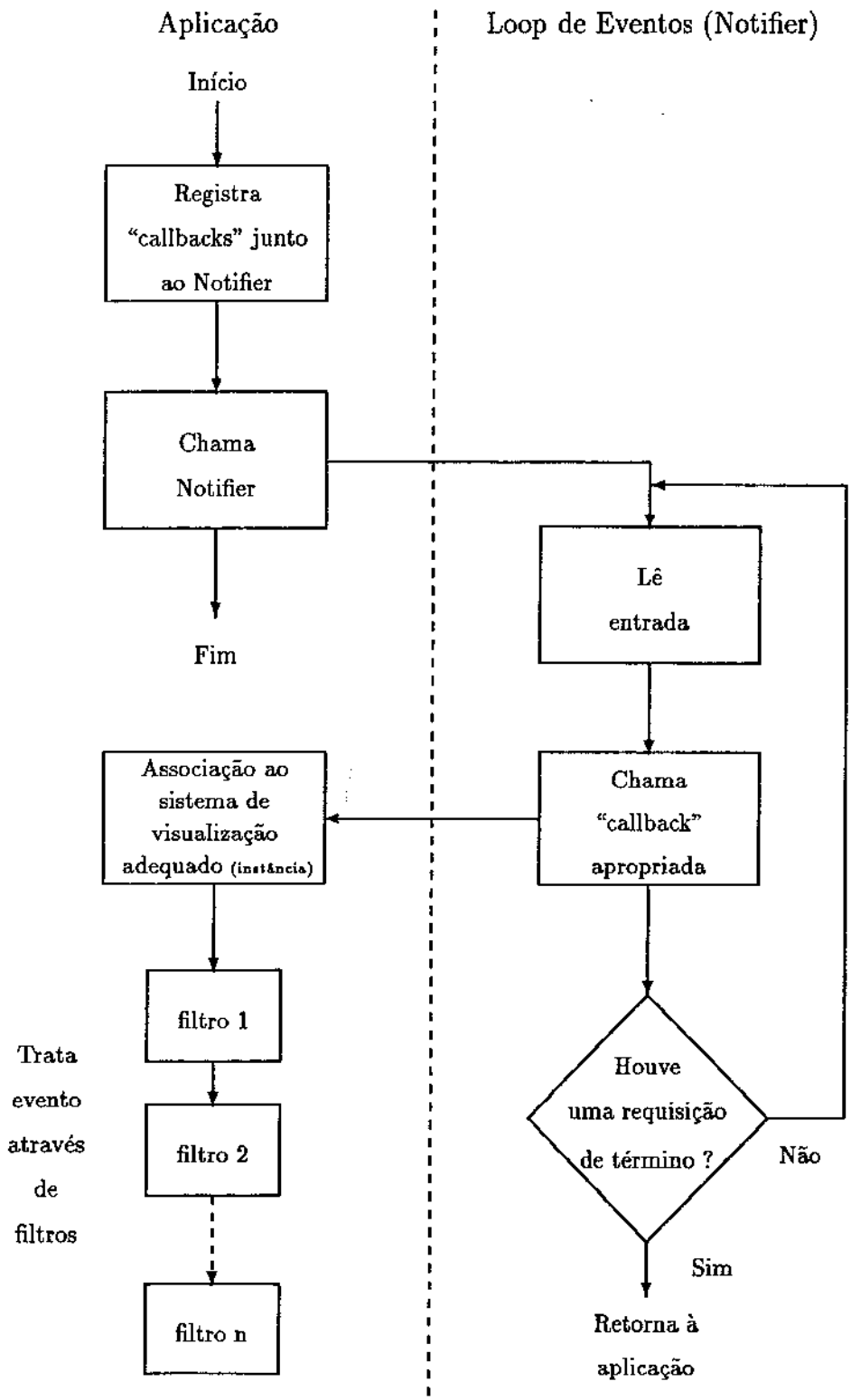

Figura 3.8: Fluxo de controle no **GeoLab** – extensões ao XView

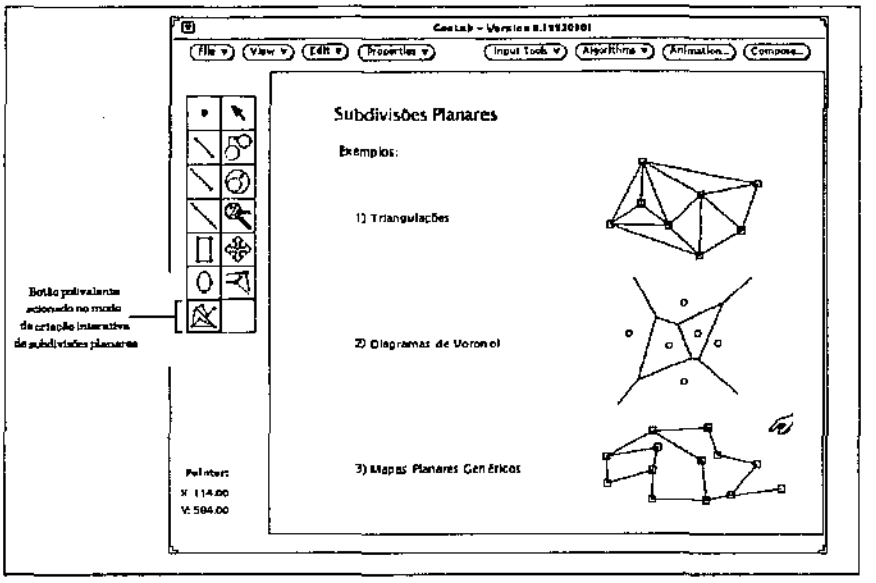

Figura 3.9: Um desenhador de subdivisões planares como modo de criação interativa externo

4. Controlador de área de visualização – Operações como *Zoom* e *Scroll* possuem tratadores próprios, mas que podem ser reutilizados caso deseje-se acrescentar opções a estas funções.

3.3.2 Modos de Criação Interativa de Objetos Geométricos

A introdução de novos modos de criação interativa é realizada através de algoritmos geométricos que têm por função registrar tratadores de eventos no **GeoLab**. Quando invocados, algoritmos deste tipo mudam o comportamento do botão polivalente disponível na interface. Ao ser acionado, este botão por sua vez instala junto ao **GeoLab** os tratadores registrados e o controle de entrada de dados passa a seguir o fluxo determinado pelo algoritmo. O ambiente provê as ferramentas necessárias para a manipulação de sua lista de objetos de tal forma que novos objetos construídos interativamente possam ser inseridos no editor, manipulados através das ferramentas disponíveis na interface e também servir de entrada para outros algoritmos geométricos.

A figura 3.9 mostra a construção de uma subdivisão planar através de um modo de criação interativa externo, ligado ao ambiente dinamicamente. Ressaltamos ainda a existência de modos para a introdução de texto e figuras (*pixmaps*), voltados para a produção de ilustrações em trabalhos de pesquisa e ensino em Geometria Computacional (as figuras mostradas no capítulo 4 e nos apêndices foram confeccionadas utilizando-se estes modos).

Modos de criação interativa podem ainda manipular a lista de objetos geométricos mantida pelo **GeoLab** de forma a complementar sua funcionalidade. Como exemplo, citamos a construção de desenhadores de objetos complexos (e.g., subdivisões planares, grafos). Utilizando-se o conceito de objeto selecionado, torna-se possível implementar modos de criação interativa bastante poderosos, que permitam a construção destes objetos de forma organizada e incremental (i.e., dado que existam duas subdivisões planares no editor, ao selecionar uma e ativar o modo de criação de subdivisões planares pode-se ajustar o funcionamento da ferramenta de forma a atuar sobre a subdivisão selecionada, permitindo que seja expandida). A flexibilidade na utilização desta ferramenta é resultado direto da filosofia que procuramos seguir para todas as ferramentas de programação proporcionadas pelo

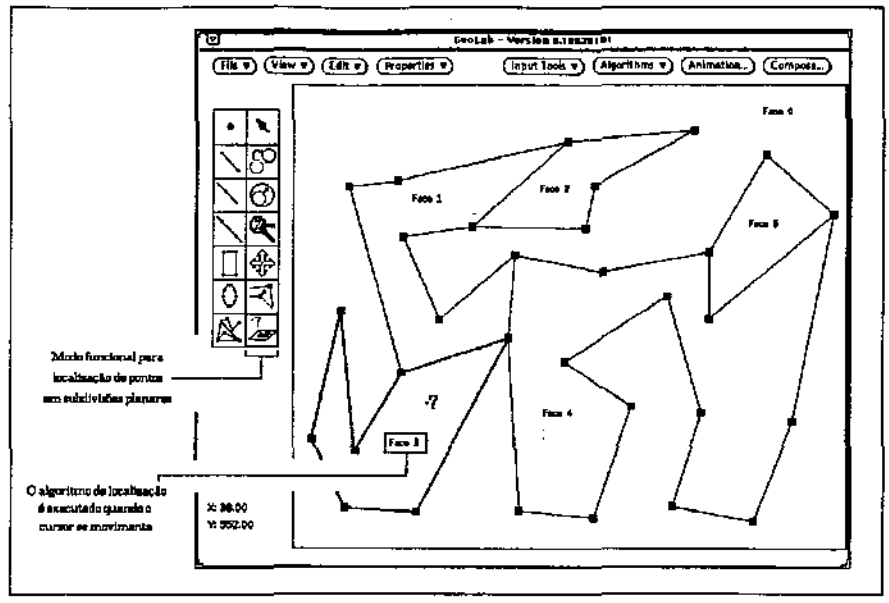

Figura 3.10: Implementação do localizador de pontos através de modos funcionais

GeoLab.

3.3.3 Modos Funcionais

A introdução de novos modos funcionais segue os mesmos passos da introdução de novos modos de criação interativa. A diferença, além do botão polivalente (na coluna da direita – veja figura 2.3) está na semântica das ações realizadas.

Através do sistema de visualização, é possível criar extensões aos modos de *Zoom* e *Scroll*, assim como a manipulação da lista de objetos do **GeoLab** permite a criação de extensões aos modos de seleção, translação, cópia e redimensionamento de objetos. É possível também criar novos modos para realizar rotação, escala, transformações geométricas sobre objetos, etc.

A despeito destas possibilidades, novos modos funcionais podem ser utilizados também para o mapeamento de algoritmos geométricos, notadamente aqueles que realizam tarefas que envolvem pre-processamento. Como exemplo, citamos o caso do localizador de pontos em subdivisões planares.

Ao acionar o modo funcional de localização de pontos, a subdivisão selecionada é preprocessada (de forma a proporcionar redução no tempo de busca), e aos movimentos do *mouse* são acionadas as rotinas de localização, demonstrando em "tempo real" o funcionamento do algoritmo. Desta forma, torna-se mais natural e realista avaliar o desempenho deste tipo de algoritmo (veja a figura 3.10).

3.3.4 Recomendações

Criar ferramentas flexíveis implica em relaxar algumas restrições, deixando a cargo do usuário-programador a tarefa de determinar algumas características no funcionamento da ferramenta.

Para que as ferramentas que venham a ser produzidas comportem-se de maneira homogênea com relação as demais, recomenda-se a aplicação dos filtros de eventos providos pelo ambiente, juntamente com a observância de algumas recomendações quanto ao estilo de funcionamento da interface. Tais recomendações são apresentadas nos apêndices desta tese.

Capítulo 4

Algoritmos Geométricos

A validade de nossa abordagem ao construir o **GeoLab** é atestada através da implementação e teste de algoritmos geométricos, conforme apresentado neste capítulo. Escolhemos implementar uma grande variedade de algoritmos fundamentais que servissem de base para outras implementações, utilizando-os por conseqüinte na implementação de alguns algoritmos de nível mais alto.

A implementação de algoritmos geométricos, apesar de ser tarefa extremamente interessante, não é de forma alguma uma tarefa trivial. Vários algoritmos descritos em poucas linhas de texto em "meta-linguagens" transformam-se em centenas de linhas de código, cujo teste e depuração tomam tempo considerável[1] [Dob88, Sed83, Sch91, For87a]. As dificuldades surgem em várias frentes, como a utilização de estruturas de dados complexas (e.g., *k-d trees*), a representação e o tratamento de objetos não realizáveis (e.g., pontos no infinito), representações e operações numéricas imprecisas (e.g., aritmética de ponto flutuante), etc. O **GeoLab** objetiva justamente amenizar estas dificuldades, proporcionando um suporte razoável para programação e manipulação de elementos gráficos, além de promover a reutilização do trabalho realizado através de bibliotecas compartilhadas. Acreditamos ser esta a abordagem que mais se aproxima da situação ideal, onde o trabalho de pesquisa em Geometria Computacional não apenas restringe-se à proposição e análise de complexidade de algoritmos em termos rigidamente teóricos, mas também à produção de resultados práticos que beneficiam a grande comunidade de aplicadores que necessita de ferramentas geométricas.

Abaixo apresentamos uma lista de todos os algoritmos presentemente implementados através do **GeoLab**, organizada em função do tipo de objeto manipulado. Nas seções que se seguem restringimos a apresentação aos mais interessantes, procurando informar suficientemente o leitor sobre aspectos de programação geométrica para facilitar futuramente incursões mais avançadas sobre o ambiente.

- Conjuntos de Pontos
  - Ponto de Menor Inclinação Relativa
  - Envoltória Convexa (Marcha de Jarvis, Varredura de Graham)
  - Cascas Convexas (Marcha de Jarvis modificada, Varredura de Graham modificada)
  - Diâmetro (força bruta, através de pares antipodais)
  - Par mais Próximo (força bruta, através de triangulação de Delaunay, divisão-e-conquista)

[1] Knight [Kni90] apresenta como conteúdo principal de sua dissertação de mestrado considerações sobre a implementação do algoritmo de Tarjan e van Wyk para a triangulação de polígonos simples em tempo $O(n \log\log n)$ [TW88].

  - Todos os Vizinhos mais Próximos (força bruta, através da triangulação de Delaunay)
  - Árvore Espalhada Euclidiana Mínima (através da triangulação de Delaunay)
  - Menor Círculo Envolvente (algoritmo de Shamos, algoritmo de Bhattacharya e Toussaint)
  - Maior Círculo Vazio (algoritmo de Preparata e Shamos)
  - Triangulação de Delaunay (algoritmo de Guibas e Stolfi)
  - Diagrama de Voronoi (algoritmo de Guibas e Stolfi)
  - Triangulação de Vizinho mais Distante (algoritmo de Guibas e Stolfi modificado pelo autor)
  - Diagrama de Voronoi de Vizinho mais Distante (algoritmo de Guibas e Stolfi modificado pelo autor)

- Segmentos
  - Teste de Pertinência de Pontos
  - Teste de Interseção de dois Segmentos
  - Cálculo da Interseção de dois Segmentos
  - Contagem de Interseções em um Conjunto de Segmentos (força bruta)
  - Cálculo de Interseções em um Conjunto de Segmentos (força bruta, varredura planar)

- Raios e Retas
  - Teste de Interseção entre dois Raios ou Retas
  - Cálculo da Interseção de dois Raios ou Retas
  - Contagem de Interseções em um Conjunto de Raios ou Retas (força bruta)
  - Cálculo de Interseções em um Conjunto de Raios ou Retas (força bruta)

- Círculos
  - Construção de um Círculo dados três Pontos

- Polígonos
  - *Winding Number*
  - Núcleo de um Polígono Simples (algoritmo de Lee e Preparata)
  - Pares Antipodais em Polígonos Convexos (algoritmo de Preparata e Shamos)
  - Envoltórias Convexas de:
    * Polígonos Genéricos (varredura de Graham)
    * Polígonos Simples (algoritmo de Lee)
    * Polígonos em Forma de Estrela (passo final da varredura de Graham)
  - Diâmetro (através da construção da envoltória convexa e cálculo de pares antipodais)

- Subdivisões Planares
  - Regularização de Subdivisões Planares
  - Triangulação de Subdivisões Planares
- Modos de Construção Interativa
  - Arestas Direcionadas
  - Subdivisões Planares (aplicação de operadores de Euler)
  - *Strings* para produção de ilustrações
  - Carregador de imagens para a produção de ilustrações
- Modos Funcionais
  - Localizador de Pontos em Subdivisões Planares (algoritmo de Kirkpatrick)
- Geradores de Entradas
  - Pontos Randômicos
    - Distribuídos na Área de Visualização
    - Internos a Objetos Geométricos
    - Coberturas sobre a Fronteira de Objetos Geométricos
  - Polígonos Randômicos
    - Polígonos Genéricos
    - Polígonos Simples
    - Polígonos em Forma de Estrela

A discussão sobre algoritmos está organizada da seguinte forma:

1. Definição do problema e sua complexidade.
2. Descrição a nível conceitual do(s) algoritmo(s).
3. Descrição da implementação (apenas partes mais interessantes).
4. Comparação de desempenho entre diferentes implementações.
5. Comparação quanto à complexidade de implementação.

As análises de complexidade limitam-se, na maioria dos casos, a mencionar cotas inferiores e superiores, reduções possíveis e algoritmos de alto nível. Provas para estes elementos são sugeridas superficialmente, indicando-se porém as fontes originais onde encontram-se detalhadas. Embora algumas das cotas inferiores apresentadas tenham provas para modelos computacionais mais poderosos, vamos assumir, por simplicidade, que tais cotas tenham validade para o modelo de Árvore Quadrática de Decisão de Yao [Yao81].

Incluímos inicialmente seções que tratam de dois problemas que surgem na implementação de algoritmos geométricos, a saber: a complexidade de implementação de algoritmos assintoticamente ótimos e o problema de robustez em implementações, sendo este último abordado apenas superficialmente neste trabalho.

4.1 Força Bruta *versus* Algoritmos Ótimos

A complexidade de implementação de algoritmos geométricos (aliada à falta de detalhamento dos mesmos na literatura) impõe a restrição mais sensível para o aproveitamento prático dos resultados emergentes em Geometria Computacional. Conforme Forrest aponta em [For87a], a principal razão para o distanciamento entre a teoria e a prática dos resultados é causado, por um lado, pela ênfase pouco realista dada a problemas geométricos por teóricos e, por outro lado, pela ausência de investigação (e publicação) por parte de aplicadores. A razão para a falta de divulgação de resultados inclui a proteção de interesses comerciais das aplicações. Sendo assim, seria preciso rever ambas as posições e, na medida do possível, aproximar estes dois ramos de produção de resultados.

Do ponto de vista acadêmico, onde torna-se mais fácil atuar devido ao limitado interesse comercial (e, em contrapartida, o objetivo de suprir a sociedade de resultados concretos), classificamos o **GeoLab** como uma das iniciativas de aproximação.

No tocante aos aspectos de implementação, tivemos a oportunidade de experimentar, em muitas situações, as dificuldades mencionadas, notadamente no início dos trabalhos, quando poucas ferramentas existiam. Temos a dizer, entretanto, que a partir do momento que tais ferramentas passaram a estar disponíveis, a dificuldade diminuiu sensivelmente, possibilitando a implementação de algoritmos reconhecidamente não triviais. É neste sentido que discordamos da seguinte afirmação, onde escreve-se a respeito de algoritmos para a construção da Árvore Espalhada Euclidiana Mínima de um conjunto de pontos[2] (pág. 467 [Sed83] – trecho traduzido):

> "... *Provou-se que é possível fazer melhor. O ponto é que a estrutura geométrica torna a maioria das arestas do grafo completo irrelevantes para o problema, e podemos eliminar a grande maioria delas antes de iniciarmos a construção da árvore espalhada Euclidiana mínima. De fato, foi provado que a árvore espalhada mínima é um subconjunto do grafo determinado pelas arestas do dual do diagrama de Voronoi. Sabemos que este grafo tem número de arestas proporcional a N, e tanto o algoritmo de Kruskal quanto o método de busca por prioridades funcionam bem em tais grafos esparsos. Então, em princípio, poderíamos computar o dual do diagrama de Voronoi (o que toma tempo proporcional a* $N \log N$*) e então aplicar qualquer um dos algoritmos citados para obter um algoritmo para o cálculo da árvore espalhada Euclidiana mínima que execute em tempo proporcional a* $N \log N$*. Entretanto, escrever um programa para computar o dual do diagrama de Voronoi é um desafio, mesmo para um programador experiente, e, portanto, tal abordagem tem grandes possibilidades de se mostrar impraticável.*"

Com a grande quantidade de ferramentas presentemente existentes no **GeoLab**, tal estratégia pôde ser aplicada com sucesso.

Outro ponto digno de nota é que, contando com as ferramentas corretas, a implementação de algoritmos ótimos pode revelar-se tão simples quanto a dos algoritmos ineficientes de força bruta (quase sempre preferidos devido à simplicidade de implementação). Os fragmentos de código apresentados nas figuras 4.1 e 4.2 ilustram esta afirmação para o algoritmo que determina o par de pontos mais próximos em um conjunto de n pontos no plano.

[2] *Euclidean Minimum Spanning Tree.*

```
Segment2D_ptr closest_pair(list(Point2D_ptr) &points) {
    Segment2D_ptr closest = 0;                                    // result
    if (points.size() > 1) {
      Point2D_ptr p1, p2;
      list_item item1, item2;
      double slaux, sl = MAXDOUBLE;                               // smallest length
      forall_list_items(item1, points)                            // O(n^2) algorithm
        forall_list_items(item2, points)
          if (item1 ≠ item2 &&
              (slaux = points[item1]→e_distance_sqr(*points[item2])) < sl) {
            p1 = points[item1];
            p2 = points[item2];
            sl = slaux;
          }
      closest = new Segment2D(*p1, *p2);
    }
    return closest;
}
```

Figura 4.1: Implementação de um algoritmo força bruta para o problema do par de pontos mais próximos

```
Segment2D_ptr closest_pair(list(Point2D_ptr) &points) {
    Segment2D_ptr it, closest = 0;                          // result
    if (points.size() > 1) {
      DelaunayTriangulation dt = delaunay_triangulation(points);
      list(Segment2D_ptr) ls = dt→edges();
      double slaux, sl = MAXDOUBLE;                         // extract the smallest
      forall(it, ls)                                        // edge among the edges
        if ((slaux = ls→length()) < sl) {                   // of the Delaunay Triangulation
          closest = it;                                     // of the given points in
          sl = slaux;                                       // O(n) time
        }
      }
    return res;
}
```

Figura 4.2: Implementação de um algoritmo ótimo para o problema do par mais próximo (através da triangulação de Delaunay)

4.2 Robustez em Implementações

O problema de robustez numérica em implementações foi tocado apenas superficialmente em nosso projeto. A razão para tal, além das restrições de tempo, devem-se principalmente à esparsa e inconclusiva literatura a respeito.

Notamos que pequenos cuidados de implementação, notadamente nas primitivas geométricas que compõem o ambiente, são suficientes para garantir um funcionamento satisfatório dos algoritmos implementados na grande maioria dos casos. Nestas primitivas, o ponto central considerado foi a introdução de tolerância nas comparações, ao estilo da proposta apresentada em [GSS89]. Outras técnicas não puderam ser incluídas nas implementações atuais ou por não serem gerais o suficiente, ou por demandarem uma reestruturação geral de todos os algoritmos implementados.

Ao leitor interessado, entretanto, indicamos os trabalhos de Hoffmann, Hopcroft e Karasick [HHK88], Edelsbrunner e Mücke [EM90], Dobkin e Silver [DS89], Hopcroft e Kahn [HK92], Schorn [Sch91], Hoffmann [Hof89], Knuth [Knu91] e Fang e Brüderlin [FB91] como referências para investigações futuras.

O trabalho de Peter Schorn [Sch91] merece especial atenção por aliar o desenvolvimento de um ambiente análogo ao **GeoLab** à questão de robustez em implementações de algoritmos geométricos. Como se conclui de seu trabalho, não existe um método único e definitivo a ser aplicado em todas as implementações e, em nosso caso, a investigação dos vários métodos possíveis foi preterida em prol de um ambiente mais versátil e completo. Vale lembrar que a possibilidade de introdução de novos tipos de objetos geométricos (e suas respectivas primitivas geométricas) de forma dinâmica (conforme apresentado no capítulo 3) facilita grandemente os estudos futuros nesta área.

4.3 Algoritmos para Conjuntos de Pontos

4.3.1 Envoltórias Convexas

Iniciamos a apresentação de algoritmos implementados com o problema da construção da envoltória convexa de um conjunto de n pontos no plano.

Definição 4.1 *A envoltória convexa de um conjunto S de n pontos no plano é o menor conjunto convexo que contém S, denotado $CH(S)$.*

Obviamente, este conjunto é um polígono convexo cujos vértices são pontos de S.

O fato dos vértices da envoltória convexa ocorrerem de forma ordenada (veja figura 4.3) aponta para uma conexão natural com o problema de ordenação. De fato, qualquer algoritmo que construa $CH(S)$ deve também ser capaz de ordenar (conseqüentemente uma cota inferior para a construção da envoltória convexa é $\Omega(n \log n)$ no modelo de Árvore Algébrica de Decisão). Uma prova informal segue:

Dados n números reais $x_1, x_2, \ldots, x_n$, todos positivos, obtém-se sua ordenação através da construção de uma envoltória convexa mapeando cada x_i ao ponto (x_i, x_i^2) e associando a este ponto o índice i. Todos os pontos obtidos através deste mapeamento estarão sobre a parábola $y = x^2$ (figura 4.4) e a envoltória convexa do conjunto será uma lista de pontos cuja ordenação pela coordenada x pode ser extraída em tempo linear, através de um passo sobre a lista [PS85].

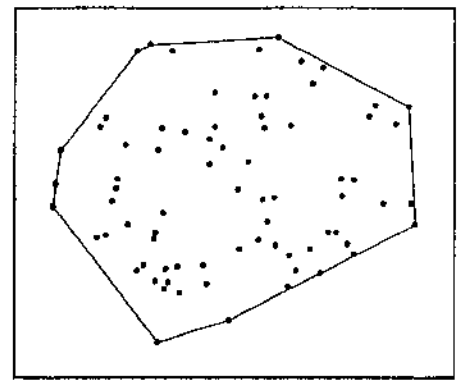

Figura 4.3: Envoltória convexa de um conjunto de pontos

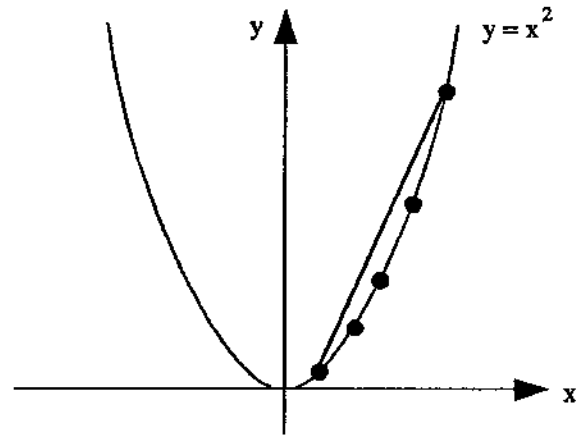

Figura 4.4: Mapeamento de valores sobre a parábola $y = x^2$ para realização da redução do problema da ordenação ao problema da construção da envoltória convexa

4.3.1.1 Varredura de Graham

O algoritmo proposto por Graham (Varredura de Graham) [Gra72] para este problema constrói a envoltória convexa de um conjunto de pontos em dois passos. No primeiro, os pontos são ordenados circularmente (e.g., sentido anti-horário) em torno de um ponto interno à envoltória. Este passo toma tempo $O(n \log n)$ no pior caso (utilizando-se um algoritmo ótimo para ordenação). Em seguida, os pontos ordenados são percorridos (varredura), executando-se os seguintes passos, baseando-se na posição relativa de três pontos consecutivos:

1. Sejam p_1, p_2, p_3 pontos consecutivos na lista ordenada, sendo p_1 o *ponto inicial* garantidamente pertencente à envoltória convexa (e.g., ponto de menor coordenada y).
2. Se p_2 está à esquerda de $\overrightarrow{p_1 p_3}$ então p_2 não pertence à envoltória e é eliminado; p_1 retrocede caso já não seja o próprio ponto inicial.
3. Se p_2 está à direita de $\overrightarrow{p_1 p_3}$ então p_2 é mantido e p_1 avança.

Neste segundo passo, um dado ponto p_i é considerado no máximo duas vezes, podendo ser eliminado (e não mais ser considerado) ou então mantido como ponto pertencente à envoltória. Neste último caso o algoritmo progride, até que o ponto inicial seja atingido (na implementação utilizamos uma lista duplamente encadeada). Sendo assim, a varredura toma tempo proporcional a n, e o o algoritmo como um todo executa em tempo $O(n \log n)$ (descrições formais e detalhadas desta afirmação podem ser encontradas em [Gra72] e [PS85]). A figura 4.5 ilustra alguns passos da varredura.

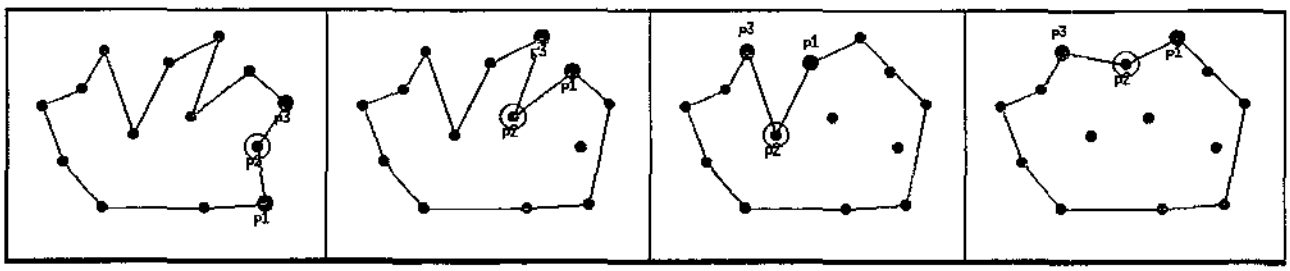

Figura 4.5: Alguns passos na construção da envoltória convexa através da Varredura de Graham

A implementação, neste caso, praticamente reflete a descrição do algoritmo (figura 4.6). Cabe apenas observar que o tratamento de pontos colineares é feito de forma simples, eliminando-se p_2 também quando este é colinear a p_1 e p_3.

4.3.1.2 Marcha de Jarvis

O algoritmo proposto por Jarvis (Marcha de Jarvis) [Jar73] baseia-se na idéia de "embrulho de presente" para determinar a envoltória convexa de um conjunto de pontos. Partindo-se de um ponto p reconhecidamente pertencente a $CH(S)$ (digamos, o ponto de menor coordenada y), encontra-se o próximo ponto pertencente à envoltória (em algum sentido, neste caso anti-horário) inspecionando-se todos os pontos. Este próximo ponto será aquele que juntamente com p determinar a reta de menor inclinação com relação ao eixo x (veja figura 4.7). Uma possível apresentação deste algoritmo é a seguinte:

```
list(Point2D_ptr) graham_scan(list(Point2D_ptr) &points) {
    if (points.size() > 2) {

      /* counterclockwise circular sorting of the given      */
      /* points around the point with smallest Y coordinate */

      Point2D *ref = points[it_get_min_Y(points)];
      set_circular_comp_reference(*ref);
      points.sort((int(*)(Point2D_ptr&, Point2D_ptr&))compare_function);

      list_item vi, vi1, vi2;
      SideType side;
      vi = points.first();
      do {
        vi1 = points.cyclic_succ(vi);
        vi2 = points.cyclic_succ(vi1);

        side = points[vi]→which_side(*points[vi1], *points[vi2]);
        if (side == Right || (side == Colinear && vi2 ≠ vi)) {
          points.del_item(vi1);
          /* backtrack if vi isn't at the starting point */
          if (vi ≠ points.first())
            vi = points.pred(vi);
        }
        else
          vi = points.succ(vi);
      } while (vi1 ≠ points.first());
    }
    return(points);
}
```

Figura 4.6: Implementação da Varredura de Graham

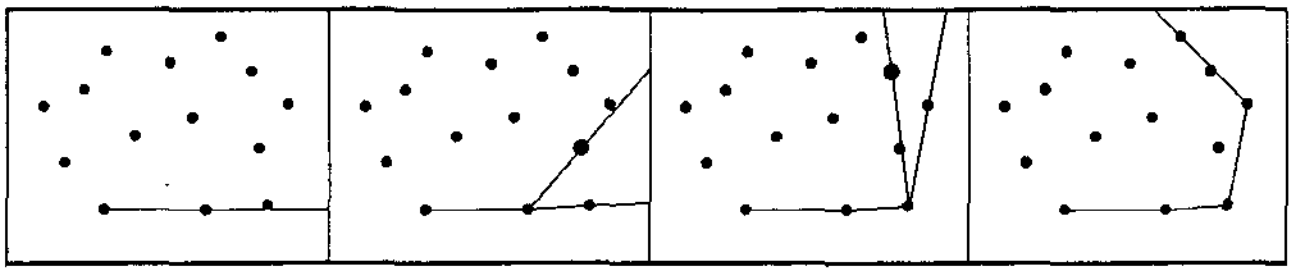

Figura 4.7: Passos da execução da Marcha de Jarvis

1. Encontre o ponto de menor coordenada y. Seja p este ponto. Adicione p à $CH(S)$ e torne-o o ponto corrente.
2. Seja l o eixo x.
3. Enquanto não tiver retornado a p:
   (a) Encontre q tal que $\overrightarrow{pq}$ seja a reta com menor inclinação com relação a l;
   (b) Adicione q a $CH(S)$ e faça q o ponto corrente. Seja $l = \overrightarrow{pq}$.

Embora a descrição mencione a utilização de cálculo de ângulos para a comparação de inclinações de retas, isto não é realmente necessário (figura 4.8). Dados três pontos p, q e r, a reta que passa por p e q tem menor inclinação que a reta que passa por p e r se r está a esquerda de $\overrightarrow{pq}$ (veja seção 3.1.1). Em caso de empates (i.e., p, q e r colineares), escolhe-se o ponto mais distante de p, de forma a eliminar pontos colineares intermediários.

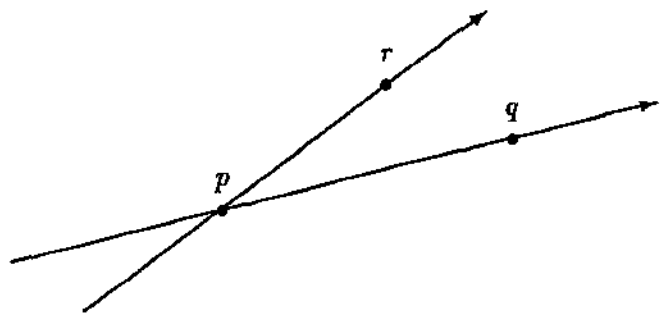

Figura 4.8: Determinação da inclinação relativa entre duas retas em função do posicionamento dos pontos que as determinam

A implementação da Marcha de Jarvis é apresentada na figura 4.9.

Para cada ponto sendo considerado (ponto corrente), todos os outros são inspecionados na busca do próximo, o que determina um tempo de execução de $O(n^2)$ no pior caso ($O(n)$ pontos de S em $CH(S)$). Para o caso em que k pontos do conjunto original pertençam a $CH(S)$, a complexidade pode ser melhor definida como $O(nk)$, o que torna o algoritmo de Jarvis assintoticamente mais eficiente que o algoritmo de Graham quando $k = o(\log n)$ [PS85].

Considerando distribuições uniformes de pontos e também o pior caso para o algoritmo de Jarvis (i.e., $O(n)$ pontos sobre a envoltória convexa), elaboramos uma comparação de performance (tempo de

```
list(Point2D_ptr) jarvis_march(list(Point2D_ptr) &points) {
    list(Point2D_ptr) hull;
    list_item low_ccw, low_y;

    if (!points.empty()) {
      low_y = it_get_min_Y(points);                    /* low_y is the point with */
                                                       /* smallest Y coordinate   */
      hull.append(points[low_y]);
      points.append(points[low_y]);
      points.del_item(low_y);

      while (!points.empty()) {
        low_ccw = it_lower_angle_ccw(points, *hull.tail());
        if (*points[low_ccw] == *hull.head())
          break;
        else
          hull.append(points[low_ccw]);
        points.del_item(low_ccw);
      }
      points.del_item(points.last());
    }

    remove_colinear_points(hull);
    return hull;
}
```

Figura 4.9: Implementação da Marcha de Jarvis

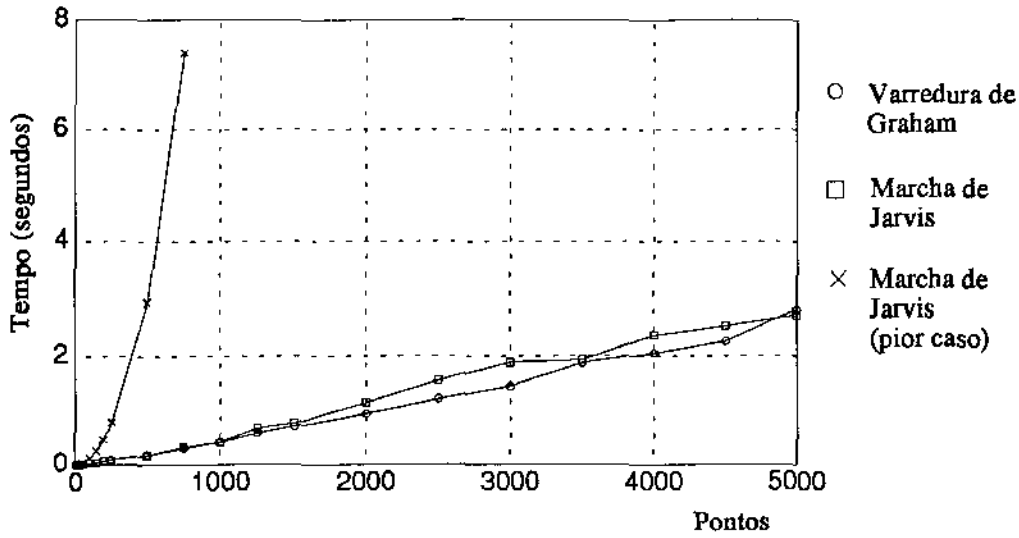

Figura 4.10: Comparação entre a Varredura de Graham e a Marcha de Jarvis

execução) para as duas implementações apresentadas. Note que n pontos uniformemente distribuídos em um quadrado ou retângulo tem $\theta(\log n)$ pontos na envoltória convexa final, o que resulta numa aproximação entre os tempos de execução da Varredura de Graham e da Marcha de Jarvis. O gráfico é apresentado na figura 4.10.

4.3.1.3 Cascas Convexas

Definição 4.2 *Dado um conjunto S de n pontos no plano, as cascas convexas deste conjunto correspondem ao conjunto $\{CH(S_i) \mid i = 0, 1, \ldots, d\}$ onde $S_0 = S$, $S_j = S_{j-1} - V(CH(S_{j-1}))$, $j = 1, 2, \ldots, d$, onde $V(CH(T))$ denota o conjunto de vértices de $CH(T)$, e $S_d = V(CH(S_d))$. A profundidade do conjunto é definida como sendo d, i.e., o número de cascas convexas resultantes deste processo [LP84].*

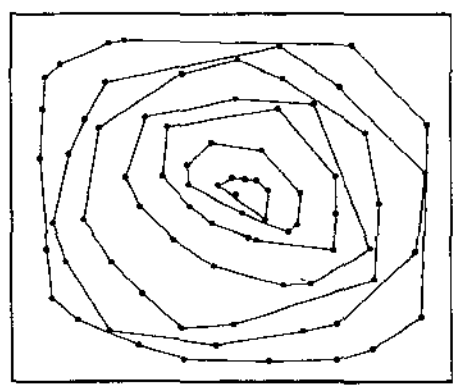

Figura 4.11: Cascas convexas de um conjunto de pontos

Algoritmos não ótimos podem ser conseguidos para este problema aplicando-se algoritmos para a construção de envoltórias convexas sucessivamente. Caso utilize-se a Marcha de Jarvis, o resultado é um algoritmo de complexidade $O(n^2)$ e, caso utilize-se a Varredura de Graham, o resultado é um

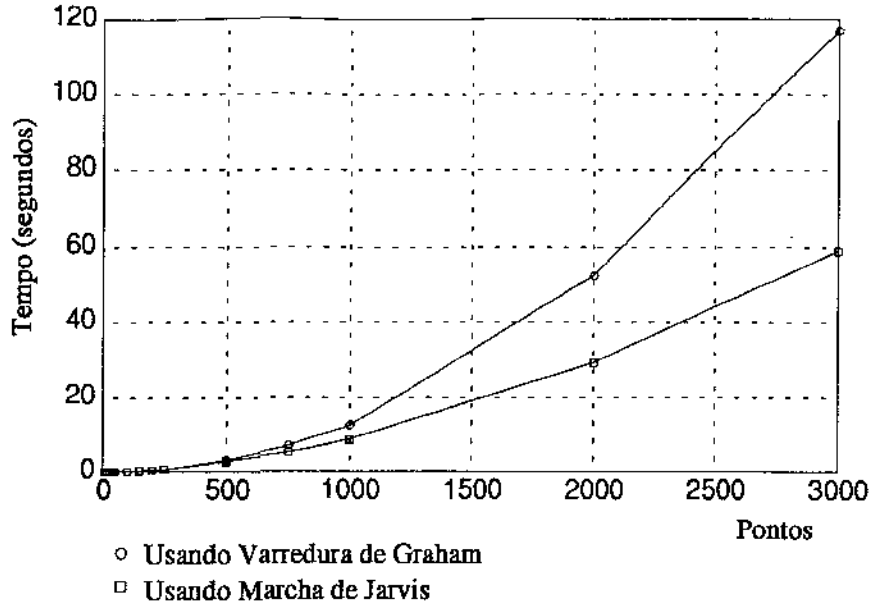

Figura 4.12: Comparação entre dois algoritmos não ótimos para construção de cascas convexas

algoritmo de complexidade $O(n^2 \log n)$. A comparação para estas duas alternativas, implementadas utilizando-se o **GeoLab**, é apresentada na figura 4.12.

Chazelle apresenta em [Cha83] um algoritmo para a construção de cascas convexas que executa em tempo $O(n \log n)$ em pior caso, e portanto ótimo (observe que uma cota inferior para o problema é $\Omega(n \log n)$, correspondente à construção da casca mais externa).

4.3.2 Problemas de Proximidade

4.3.2.1 Diâmetro de um Conjunto de Pontos

Definição 4.3 *Dado um conjunto S de n pontos no plano, define-se diâmetro de S (denotado por $diam(S)$) como sendo a distância máxima entre dois pontos de S.*

Uma cota inferior não trivial para este problema é obtida a partir da transformação em tempo linear do problema de disjunção de conjuntos para o problema de determinação do diâmetro [PS85]. Sejam $A = \{a_1, \ldots, a_n\}$ e $B = \{b_1, \ldots, b_n\}$ dois conjuntos de números reais não negativos. Testar se A e B são disjuntos requer $\Omega(n \log n)$ comparações [Rei72]. A transformação é obtida através do mapeamento dos conjuntos A e B para pontos no primeiro e terceiro quadrantes de uma circunferência de raio unitário C, da seguinte forma: a_j é mapeado na interseção do primeiro quadrante de C com a reta $y = a_j x$. De forma análoga, b_j é mapeado no terceiro quadrante de C (figura 4.13).

Seja T o conjunto destas $2n$ interseções. O conjunto T terá diâmetro igual a 2 se e somente se existirem dois pontos diametralmente opostos em C, o que significa, pelo mapeamento, que $A \cap B \neq \emptyset$.

Implementamos dois algoritmos para a solução deste problema. O primeiro resolve o problema através de força bruta (i.e., para cada ponto busca seu vizinho mais distante, escolhendo a maior distância assim determinada), com complexidade de tempo $O(n^2)$ em todos os casos.

O segundo utiliza-se do fato de que o diâmetro de um conjunto de pontos é igual ao diâmetro de sua envoltória convexa (veja [PS85], pág. 171). Desta forma, tendo-se implementado o algoritmo para cálculo do diâmetro de polígonos convexos, tudo que se faz necessário é calcular a envoltória convexa dos pontos (através da Varredura de Graham) em $O(n \log n)$. Na seção 4.4.3 descrevemos a

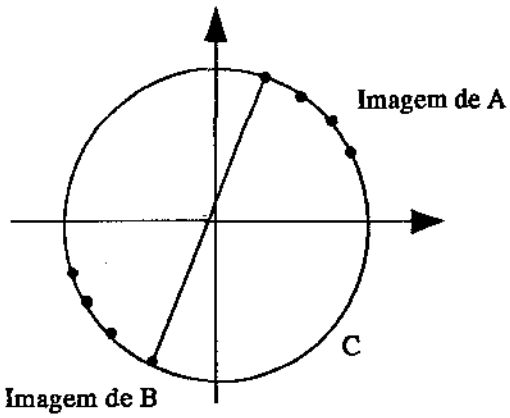

Figura 4.13: Mapeamento dos conjuntos A e B sobre um círculo unitário C

```
Segment2D_ptr diameter_set_of_points(list(Point2D_ptr) &points) {
    Polygon *ch = new Polygon(graham_scan(points));
    Segment2D_ptr result = diameter_convex_polygon(ch);
    delete ch;
    return result;
}
```

Figura 4.14: Implementação de um algoritmo ótimo para o cálculo do $diam(S)$

implementação do algoritmo para cálculo do diâmetro de polígonos convexos, que executa em tempo proporcional ao número de vértices do polígono, o que determina que o algoritmo para o cálculo do diâmetro de um conjunto de n pontos quaisquer executa em tempo $O(n \log n)$, sendo portanto ótimo (figura 4.14).

4.3.2.2 Cálculo do Par mais Próximo

Definição 4.4 *Dado um conjunto S de n pontos no plano, encontrar o par de pontos cuja distância seja mínima.*

Uma cota inferior para este problema é estabelecida através da transformação do problema de *unicidade de elementos* (mostrada ser $\Omega(n \log n)$ em [PS85]), e existem vários algoritmos que atingem este limite. Os três algoritmos implementados para este problema são apresentados a seguir:

- **Força Bruta**

  A primeira e mais simples implementação calcula as distâncias entre todos os pares de pontos. A complexidade de tempo é obviamente $O(n^2)$ em todos os casos (veja figura 4.1).

- **Através da Triangulação de Delaunay**

  Conforme apresentado adiante (seção 4.3.3.1), a triangulação de Delaunay de um conjunto de n pontos acumula informações de proximidade de todos estes pontos do conjunto. Todos os vizinhos mais próximos de um ponto p_i determinam arestas incidentes em p_i na triangulação. O número total de arestas é proporcional a n e, portanto, a extração da informação do par mais próximo desta triangulação pode ser realizada em tempo proporcional a n. A construção da triangulação

toma tempo $O(n \log n)$ e, conseqüentemente, o algoritmo como um todo é assintoticamente ótimo. O código correspondente à implementação deste algoritmo foi mostrado anteriormente na figura 4.2.

- **Divisão-e-Conquista**

O terceiro algoritmo implementado para o cálculo do par mais próximo de pontos em um conjunto de n pontos no plano foi o algoritmo de divisão-e-conquista apresentado em [PS85]. Neste algoritmo, o conjunto de entrada S é particionado em dois subconjuntos, S_1 e S_2, tal que os pontos de S_1 estejam à esquerda dos pontos de S_2, isto é, divide-se S por uma reta vertical l definida pela mediana das coordenadas x dos pontos de S. Resolve-se então recursivamente o problema para S_1 e S_2 obtando-se d_1 e d_2, distâncias mínimas entre elementos de S_1 e S_2, respectivamente.

Seja $d = min(d_1, d_2)$. Se o par mais próximo de S for determinado por algum $p \in S_1$ e algum $q \in S_2$, então certamente ambos encontram-se a uma distância máxima d com relação a l (figura 4.15).

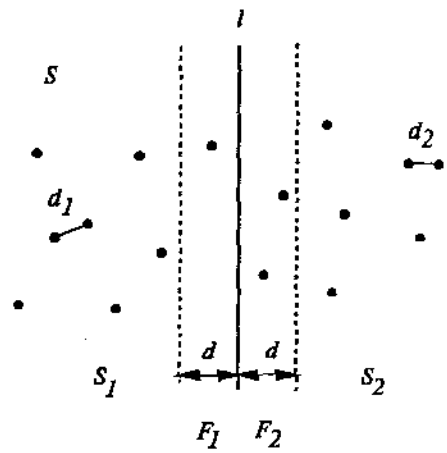

Figura 4.15: Busca do par mais próximo através de um algoritmo de divisão-e-conquista

Deve-se então verificar a presença do par mais próximo nas faixas horizontais F_1 e F_2. Para um determinado ponto p em F_1 existem no máximo 6 candidatos a serem considerados em F_2, como mostra a figura 4.16, e que podem ser determinados em tempo linear caso tenhamos os pontos ordenados pela coordenada y.

Uma descrição mais detalhada dos passos mencionados segue:

1. Particione S em dois subconjuntos S_1 e S_2 em função de uma reta vertical l determinada pela mediana das coordenadas x de S;
2. Encontre recursivamente os pares mais próximos em S_1 e S_2 e sejam d_1 e d_2 as respectivas distâncias entre estes pares;
3. Seja $d = min(d_1, d_2)$;
4. Seja F_1 o conjunto de pontos de S_1 que encontram-se a uma distância máxima d com relação a l. Seja F_2 o conjunto correspondente em S_2. Projete F_1 e F_2 sobre l e ordene-os pela coordenada y, produzindo as seqüências ordenadas F_1^* e F_2^*;

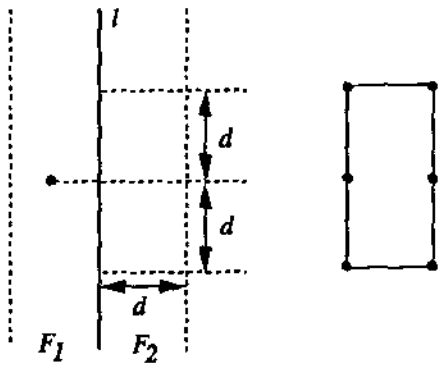

Figura 4.16: Número máximo de candidatos constante para cada ponto em F_1

5. Encontre o par mais próximo de pontos pq sendo $p \in F_1^*$ e $q \in F_2^*$ inspecionando-se, para cada ponto em F_1^* os pontos em F_2^* a uma distância máxima d de p. Enquanto um apontador avança em F_1^*, o apontador de F_2^* pode oscilar dentro de um intervalo de largura $2d$. Seja d_t a menor distância assim calculada caso existir ou, seja $d_t = +\infty$ caso contrário;
6. Seja $d_s = min(d, d_t)$ a solução final para o problema.

Os passos 1 e 5 tomam tempo proporcional a n. Os passos 3 e 6 tomam tempo constante. O passo 4 tomaria tempo $O(n \log n)$ caso a ordenação tivesse realmente que ser realizada a cada recursão do algoritmo, o que se pode evitar pré-ordenando-se o conjunto de entrada pela coordenada y, o que nos permite extrair a lista ordenada em tempo proporcional a n.

$T(n)$, o tempo total de execução do algoritmo, pode então ser definido como

$$T(n) = 2T(n/2) + O(n) = O(n \log n).$$

A apresentação da implementação correspondente ao algoritmo de divisão-e-conquista para o problema do par mais próximo foi aqui omitida por questões de espaço. Uma descrição completa e análise de complexidade deste algoritmo pode ser encontrada em [PS85].

O gráfico da figura 4.17 mostra uma comparação entre os três algoritmos implementados, considerando-se distribuições uniformes de pontos.

4.3.2.3 Todos os Vizinhos mais Próximos

O problema de determinar o par mais próximo em um conjunto de pontos pode ser generalizado:

Definição 4.5 *Dado um conjunto S de n pontos no plano, para cada um dos pontos em S determinar seu vizinho mais próximo.*

Uma cota inferior para este problema é obtida trivialmente através da redução do problema do par mais próximo, em tempo linear. Dado que tenhamos calculado todos os pares mais próximos, extrair o par mais próximo pode ser realizado em tempo proporcional a n e a cota inferior de $\Omega(n \log n)$ imediatamente se transfere.

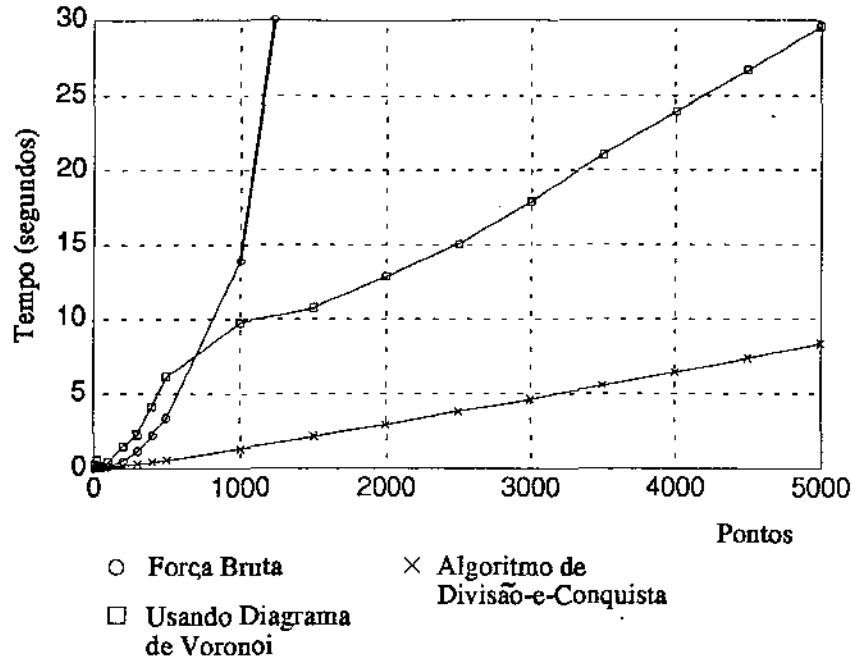

Figura 4.17: Comparação entre três implementações para o problema do par mais próximo

Implementamos dois algoritmos para a solução deste problema. O primeiro e mais simples executa em tempo $O(n^2)$ (força bruta) e o segundo, baseado na triangulação de Delaunay, toma tempo $O(n \log n)$, sendo portanto ótimo. Ilustramos apenas a implementação do algoritmo ótimo (figura 4.18).

A figura 4.19 ilustra o resultado do cálculo do par mais próximo e de todos os vizinhos mais próximos para um conjunto de pontos no plano.

4.3.2.4 Árvore Espalhada Euclidiana Mínima (AEEM)

Definição 4.6 *Dados n pontos no plano, construir uma árvore de comprimento total mínimo cujos vértices são os pontos de entrada.*

O problema da construção da Árvore Espalhada Euclidiana Mínima (figura 4.20) é um dos problemas clássicos de proximidade. Um algoritmo para sua solução baseia-se em uma estrutura fundamental em Geometria Computacional: a triangulação de Delaunay.

Uma cota inferior para este problema pode ser obtida tanto através da redução do problema do par mais próximo (i.e., a árvore contém a informação do par mais próximo) quanto da redução do problema de ordenação [PS85]. Considere um conjunto de n números reais $\{x_1, \ldots, x_n\}$. Cada x_i é mapeado em um ponto $(x_i, 0)$ e o conjunto de pontos assim formado possui uma única AEEM, tal que existe uma aresta entre os pontos $(x_i, 0)$ e $(x_j, 0)$ se e somente se x_i e x_j são consecutivos dentro do conjunto original ordenado. Estas reduções estabelecem uma cota inferior de $\Omega(n \log n)$ para o problema.

O estreito relacionamento entre o problema da construção da AEEM e a triangulação de Delaunay (dual do diagrama de Voronoi) é estabelecido pelos lemas que apresentamos a seguir, e dos quais decorre o algoritmo implementado.

```
list(Segment2D_ptr) all_nearest_neighbors(list(Point2D_ptr) &points) {
    list(Segment2D_ptr) result;

    if (points.size() > 1) {
      DelaunayTriangulation dt = delaunay_triangulation(points);
      GeoGraph g; GeoNode n; GeoEdge e;
      g→from_base(dt→to_base());                    // data structure conversion
      forall_nodes(n, g) {
        double slaux, sl = MAXDOUBLE;
        GeoEdge sle;
        forall_adj_edges(e, n)
          if ((slaux = e.length()) < sl) {
            sl = slaux;
            sle = e;
          }
        result.push(sle→segment());
      }
    }
    return result;
}
```

Figura 4.18: Implementação de um algoritmo ótimo para o problema de todos os vizinhos mais próximos que utiliza a triangulação de Delaunay

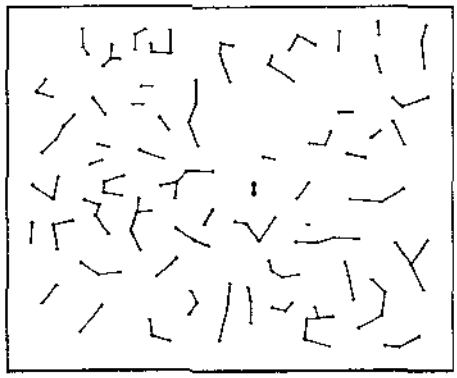

Figura 4.19: Par mais próximo e todos os vizinhos mais próximos para um conjunto de pontos

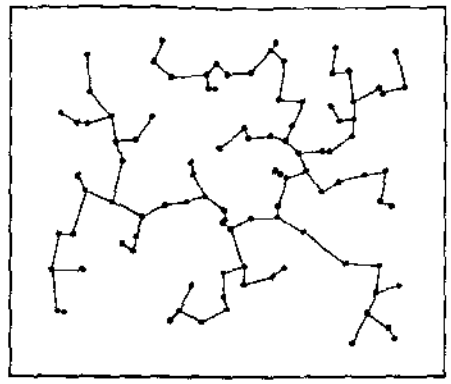

Figura 4.20: Árvore Espalhada Euclidiana Mínima de um conjunto de pontos no plano

Lema 4.1 *Seja $G = (V, E)$ um grafo com custos associados às arestas, e seja V_1, V_2 uma partição do conjunto V. Existe uma árvore espalhada mínima de G que contém a menor das arestas com um extremo em V_1 e outro em V_2 (veja [PS85] para uma prova).*

Dado um grafo G (digamos, o grafo completo sobre os n pontos de entrada) e o comprimento das arestas dado pela distância Euclidiana de seus vértices, o passo geral de um algoritmo para o cálculo da AEEM, baseando-se no lema anterior, seria:

1. Seja F uma floresta; inicialmente todos os vértices de G são árvores em F.

2. Selecione uma árvore T de F; encontre uma aresta (u', v') tal que $comprimento(u', v') = min\ \{comprimento(u, v) : u\ em\ T\ e\ v\ em\ F - T\}$.

3. Se T' é a árvore que contém v', retire T e T' de F substituindo-as pela árvore resultante de sua união através da aresta (u', v'). Quando apenas uma árvore restar em F o procedimento termina, enquanto houver mais de uma, repita passo 2.

Antes de analisarmos a complexidade deste algoritmo, é necessário observar que não se precisa do grafo completo (com $O(n^2)$ arestas) como entrada para o algoritmo. O seguinte lema nos ajuda a determinar que entrada seria necessária para o problema:

Lema 4.2 *Seja S um conjunto de pontos no plano, e seja $\Delta(p)$ o conjunto de pontos adjacentes a $p \in S$ na triangulação de Delaunay de S. Para qualquer partição S_1, S_2 de S, se $\overline{qp}$ for o menor segmento entre pontos de S_1 e pontos de S_2, então q pertence a $\Delta(p)$ (Uma prova para este lema pode ser encontrada em [PS85]).*

Em outras palavras, este lema nos diz que somente as arestas da triangulação de Delaunay necessitam ser consideradas. O grafo de entrada G para o algoritmo apresentado resume-se então a esta triangulação, cujo número de arestas é proporcional a n e toma tempo $O(n \log n)$ para ser construída (veja seção 4.3.3.1).

O algoritmo apresentado (de Cheriton e Tarjan [PS85]) pode então ser melhor descrito. Seja Q uma fila inicialmente contendo todas as árvores de *um* vértice:

1. Retire T' do início de Q;
2. Se T é a árvore resultante da união de T' com alguma T'' em Q através de uma aresta mínima, retire T'' de Q e insira T no final da fila Q.

Considere que para cada árvore T seja definido um valor inteiro denotando o estágio de T ($est(T)$), onde $est(T) = 0$ se $|T| = 1$ e $est(T) = min(est(T'), est(T'')) + 1$ se T resulta da união de duas árvores, T' e T''. Uma propriedade invariante interessante na fila de árvores é que os valores dos estágios de seus membros formam uma seqüência não-decrescente do começo ao final da lista. Diz-se que o estágio j foi completado quando T é retirada da fila e $est(T) = j$ e nenhuma outra árvore T' na fila tem $est(T') = j$. Note que ao se completar o estágio j, a fila contém no máximo $n/2^j$ membros, e portanto existem no máximo $\lceil \log_2 n \rceil$ estágios.

Desta forma, considerando o exame das n arestas da triangulação de Delaunay (ordenadas em função de seu comprimento para facilitar o trabalho de seleção e união de árvores em Q), o passo geral executa em tempo proporcional a n, sendo portanto a complexidade de tempo total do algoritmo $O(n \log n)$ (considerando-se também construção da triangulação).

4.3.3 A Construção de Diagramas de Voronoi

O diagrama de Voronoi (figura 4.22) constitui uma ferramenta em Geometria Computacional para representação de informações de proximidade e adjacência entre objetos geométricos. O diagrama e seu dual, a triangulação de Delaunay, são de enorme utilidade para a resolução eficiente de uma grande quantidade de problemas, notadamente problemas de proximidade, e isso nos incentivou a prover ao menos uma implementação para calculá-lo, que em seguida foi largamente utilizada em outros algoritmos.

Definição 4.7 *O diagrama de Voronoi de vizinho mais próximo de um conjunto S de n pontos no plano (denotado Vor(S)) corresponde a uma subdivisão planar de n regiões, cada uma correspondente a um ponto de S. A região que corresponde a um ponto p_i de S é um* polígono de Voronoi *$V(p_i)$ e é definida como:*

$$V(p_i) = \{r \mid r \in \Re^2 \wedge d(r, p_i) \leq d(r, p_j), \forall p_j \in S \wedge j \neq i\}.$$

Em outras palavras, $V(p_i)$ é o lugar geométrico dos pontos que estão mais próximos de p_i do que de qualquer outro ponto de S.

Algoritmos e estruturas de dados eficientes para a construção e a representação do diagrama de Voronoi tem sido o objeto de estudo de vários pesquisadores (veja por exemplo a extensa resenha de Aurenhammer [Aur91]). Dentre os resultados mais significativos encontram-se o algoritmo inicial proposto por Shamos [Sha78] (divisão e conquista), o algoritmo (divisão e conquista) e a estrutura de dados (*QuadEdge*) propostos por Guibas e Stolfi [GS85] e o algoritmo de Fortune (varredura planar) [For87b]. Estes algoritmos são todos assintoticamente ótimos e destaca-se a aplicação de mais de um paradigma em suas proposições.

```
SpanningTree *min_spanning_tree(UGRAPH(Point2D_ptr,bool) &G, edge_array(real) &cost) {
    node nv, nw;
    edge e;
    node_partition Q(G);

    list(edge) OEL = G.all_edges();
    C = &cost;
    OEL.sort(compare_function);

    /* marks all edges in the input graph that  also */
    /* belong to the Spanning Tree                   */

    forall (e, OEL) {
      nv = G.source(e);
      nw = G.target(e);
      if (!(Q.same_block(nv, nw))) {
        Q.union_blocks(nv, nw);
        G.assign(e, false);
      }
    }

    /* remove edges not in the final  Spanning Tree */

    forall (e, OEL) {
      if (G.inf(e))
        G.del_edge(e);
    }

    /* encapsulate the resulting graph in Geolab's Spanning  Tree structure */

    SpanningTree *res = new SpanningTree();
    res→set_graph(G);
    res→subtype(MinimumSpanningTree_t);

    return res;
}
```

Figura 4.21: Implementação do algoritmo para Árvore Espalhada Euclidiana Mínima

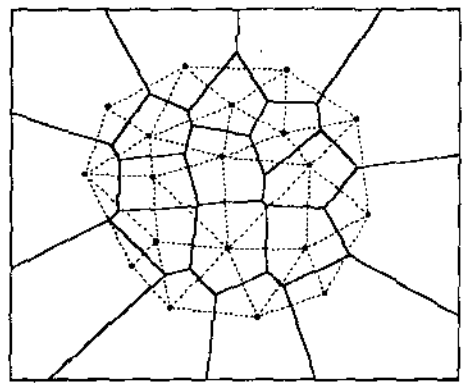

Figura 4.22: Diagrama de Voronoi (sólido) e a triangulação de Delaunay (tracejado) de um conjunto de pontos

Uma cota inferior para este problema pode ser estabelecida através de reduções do problema do par mais próximo ou mesmo através da redução do problema de ordenação. Dados n números reais $\{x_1, \ldots, x_n\}$ mapeados sobre uma mesma reta (digamos $(x_i, 0)$), o diagrama de Voronoi correspondente consiste de uma seqüência de $n-1$ bissetores paralelos separando os pontos adjacentes no conjunto ordenado, que pode então ser extraído em tempo linear. Assim sendo, transfere-se a cota inferior de $\Omega(n \log n)$ para o problema da construção do diagrama de Voronoi.

4.3.3.1 O algoritmo de Guibas e Stolfi

Do ponto de vista de implementação, escolhemos o algoritmo proposto por Guibas e Stolfi, juntamente com a estrutura de dados por eles introduzida. Esta estrutura, conhecida como *QuadEdge*, guarda as relações de adjacência do diagrama (sua topologia) além, é claro, de sua geometria, fornecendo mecanismos simples para sua construção e travessia.

O nó básico de uma *QuadEdge* consiste de uma "aresta quádrupla" que identifica uma aresta (e), a aresta simétrica ($eSym$), a aresta dual ($eRot$) e a aresta dual simétrica ($eRot^{-1}$ ou $eRotSym$), conforme ilustrado na figura 4.23.

A presença das arestas duais é uma das características vantajosas desta estrutura. O algoritmo de Guibas e Stolfi constrói a triangulação de Delaunay e, ao mesmo tempo, a topologia do grafo dual correspondente (i.e., $Vor(S)$) graças à presença das arestas duais.

Cada aresta guarda ainda informações sobre as arestas adjacentes. Da mesma forma que a estrutura *Winged-Edge* [MS82], as primeiras arestas no sentido anti-horário e no sentido horário (a partir do vértice de origem e também com relação ao vértice de destino) são atingíveis em tempo constante (figura 4.24), tanto para a aresta primal quanto para a dual.

A manipulação desta estrutura se dá através de apenas dois operadores, cujos nomes são *MakeEdge* e *Splice*. O primeiro é responsável pela criação de novas arestas e o segundo por sua união/separação (figura 4.25). Outras estruturas com possibilidades equivalentes valem-se de um conjunto de operadores significativamente maiores, como a *Winged-Edge* [MS82] ou a *Half-Edge* [Män88] que possuem pelo menos uma dúzia deles (conhecidos como operadores de Euler).

A implementação da estrutura *QuadEdge* é um interessante exercício de programação. Mostramos na figura 4.26 as classes em C++ correspondentes à nossa implementação. Uma vez implementada corretamente a estrutura, o algoritmo segue praticamente idêntico à proposição em [GS85] (e portanto

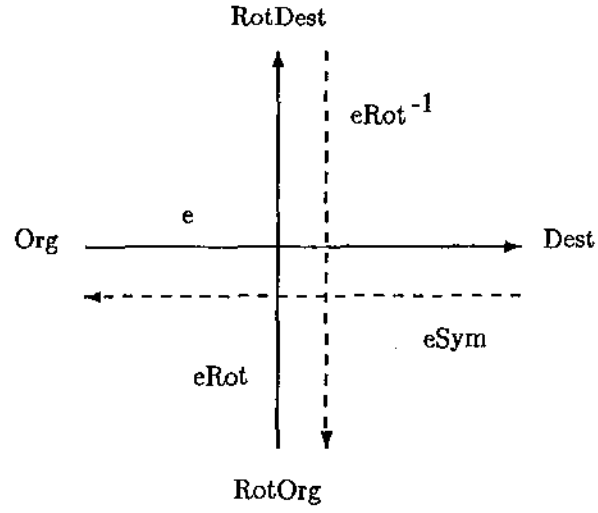

Figura 4.23: Unidade básica de informação de uma *QuadEdge*

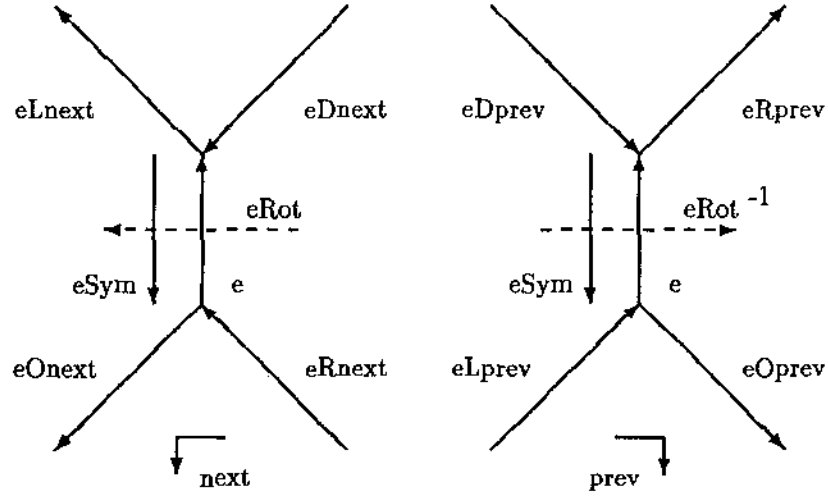

Figura 4.24: Relações de adjacência para arestas de uma *QuadEdge*

não será mostrado aqui).

4.3.3.2 Diagrama de Voronoi de Vizinho mais Distante

O conceito de diagrama de Voronoi de vizinho mais próximo pode ser generalizado de tal forma que um polígono de Voronoi torne-se o lugar geométrico dos pontos mais próximos a dois, três, ..., $n-1$ dos pontos do conjunto de entrada S do que aos outros pontos de S. Esta generalização torna possível a construção de estruturas de proximidade e adjacência que contenham informações como pontos mais distantes, k pontos mais próximos, etc. A formalização desta observação está a seguir.

Definição 4.8 *Um diagrama de Voronoi de ordem k é constituído por polígonos de Voronoi $V(T)$ definidos em função de subconjuntos T de S (onde $\mid T \mid = k$) satisfazendo:*

$$V(T) = \{p \in \Re^2 \mid \forall v \in T \wedge \forall w \in S - T, d(p,v) < d(p,w)\}.$$

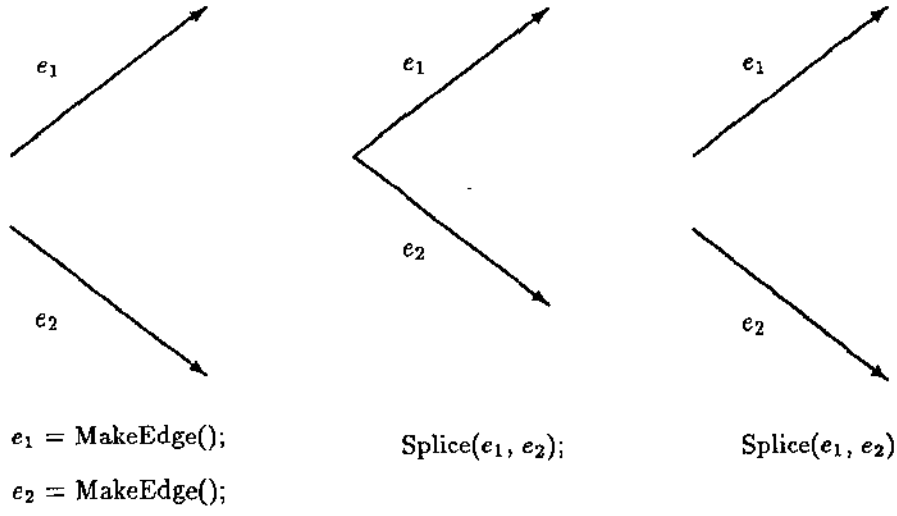

Figura 4.25: Funcionamento dos operadores *MakeEdge* e *Splice*

Em outras palavras, $V(T)$ é o lugar geométrico dos pontos p tal que todos os pontos em T estão mais próximos de p do que cada um dos pontos em $S - T$.

Em particular, o diagrama de Voronoi de ordem $n-1$ (n é o número de pontos da entrada) é chamado de diagrama de Voronoi de vizinho mais distante ($FVor(S)$), i.e., cada região neste diagrama corresponde ao lugar geométrico dos pontos mais próximos a $n-1$ pontos de S do que do outro ponto de S, digamos p_i, ou, equivalentemente, ao lugar geométrico dos pontos mais distantes de p_i do que de qualquer outro ponto de S (figura 4.27).

Da mesma forma com que vários problemas são resolvidos através do $Vor(S)$, um grande número de outros problemas são resolvidos através do $FVor(S)$ (e.g., menor círculo envolvente). Durante a implementação do algoritmo de Guibas e Stolfi para o cálculo de $Vor(S)$ notamos que, em função das propriedades duais entre os dois problemas, é possível modificar o algoritmo para que seja calculado em tempo ótimo o diagrama de Voronoi de vizinho mais distante. A cota inferior para a construção de $FVor(S)$ é também $\Omega(n \log n)$.

Basicamente, o algoritmo original foi modificado em função das seguintes propriedades:

Triangulação de Delaunay Um triângulo cujos vértices são pontos de S é dito triângulo de Delaunay se o círculo que o circunscreve é livre de pontos de S em seu interior.

Triangulação de vizinho mais distante Um triângulo cujos vértices são pontos de S é dito triângulo de vizinho mais distante se o círculo que o circunscreve contém todos os demais pontos de S em seu interior.

Observa-se que estas propriedades duais determinam uma diferença básica entre os problemas da construção de uma triangulação de Delaunay e da triangulação de vizinho mais distante (e conseqüentemente dos diagramas de Voronoi duais associados), que é a exclusão dos pontos internos à envoltória convexa do conjunto S da triangulação de vizinho mais distante [PS85, GS85]. Assim sendo, o algoritmo original foi reconstruído, incluindo-se um passo de *clean-up* na fase de *merge* de duas triangulações para remover os pontos internos à envoltória, além é claro da inversão no teste de que determina se um dado triângulo é um triângulo de Delaunay. No algoritmo apresentado a seguir,

```
⋮
typedef QEdge* QEdge_ptr;

class QEdge {
  public:
    // friendness
    friend class QuadEdge;
    friend void splice(QEdge_ptr, QEdge_ptr);
    friend QEdge_ptr connect_edges(QEdge_ptr, QEdge_ptr);
    friend void delete_incident_edges(QEdge_ptr);

    // constructors
    QEdge() { init(); }
    QEdge(const Point2D &porg, const Point2D &pdest) { make_edge(porg, pdest); }
    // destructor
    ~QEdge() { if (*_delete) delete_edge(); }

    // edge functions
    QEdge_ptr rot() { return Rot; }
    QEdge_ptr sym() { return Rot→Rot; }
    QEdge_ptr onext() { return Onext; }
    QEdge_ptr oprev() { return Rot→Onext→Rot; }
    QEdge_ptr lnext() { return Rot→Rot→Rot→Onext→Rot; }  // sym()->oprev()
    QEdge_ptr rprev() { return Rot→Rot→Onext; }          // sym()->onext()
    ⋮
    Point2D_ptr org() { return Org; }
    Point2D_ptr dest() { return Dest; }
    ⋮
  private:
    void make_edge(const Point2D &porg, const Point2D &pdest);
    void delete_edge();
    void init();

    bool *Flag;                  // traversal flag (edges)
    bool *OFlag;                 // traversal flag (vertices)
    Point2D *Org, *Dest;         // vertices
    QEdge_ptr Onext;             // next (counterclockwise) adjacent edge
    QEdge_ptr Rot;               // dual edge
    void *INFO;                  // user defined INFO field
    bool *_delete;               // private flag used by destructor
};
⋮
```

Figura 4.26: Implementação da estrutura *QuadEdge*

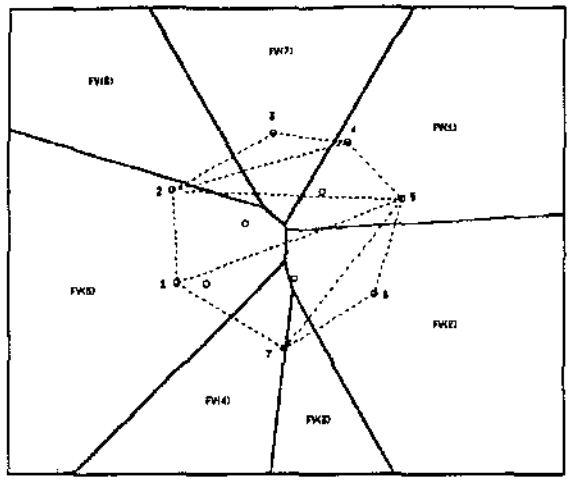

Figura 4.27: Diagrama de Voronoi de vizinho mais distante (e a triangulação dual) de um conjunto de pontos

as linhas 16–17, 28–38 executam a tarefa de remoção dos pontos internos à envoltória (e poderiam ser eliminadas caso a entrada fosse os vértices de um polígono convexo). As linhas 5–13, 18–27 foram modificadas para corretamente tratar pontos colineares. Maiores detalhes a respeito deste algoritmo são apresentados em [Jac92].

```
 1 PROCEDURE FVorTriangulation [S] RETURNS [le, re]:
 2   IF |S| = 2 THEN
 3     a <- MakeEdge[]; a.Org <- s1; a.Dest <- s2; RETURN [a, a.Sym]
 4   ELSEIF |S| = 3 THEN
 5     IF COLINEAR[s1, s2, s3] THEN
 6       a <- MakeEdge[]; a.Org <- s1; a.Dest <- s3; RETURN[a, a.Sym]
 7     ELSE
 8       a <- MakeEdge[]; b <- MakeEdge[]; Splice[a.Sym, b];
 9       a.Org <- s1; a.Dest <- b.Org <- s2; b.Dest <- s3;
10       IF CCW[s1, s2, s3] THEN c <- Connect[b, a]; RETURN [a, b.Sym]
11       ELSEIF CCW[s1, s3, s2] THEN c <- Connect[b, a]; RETURN [c.Sym, c]
12       FI
13     FI
14   ELSE
15     [ldo, ldi] <- FVorTriangulation[L];
       [rdi, rdo] <- FVorTriangulation[R];
16     {set uldi and urdi to be used to calculate the upper supporting line}
17     uldi <- ldi.Onext; urdi <- rdi.Oprev;
18     guard.ldi <- nil; guard.rdi <- nil;
19     DO
20       IF NOT RightOf[rdi.Org, ldi] AND guard.ldi <> ldi.Lnext THEN
21         IF guard.ldi = nil THEN guard.ldi = ldi FI;
22         ldi <- ldi.Lnext
23       ELSEIF NOT LeftOf[ldi.Org, rdi] AND guard.rdi <> rdi.Rprev THEN
24         IF guard.rdi = nil THEN guard.rdi = rdi FI;
```

```
25          rdi <- rdi.Rprev
26        ELSE EXIT FI
27      OD;
28      {calculate the upper supporting line}
29      guard.uldi <- nil; guard.urdi <- nil;
30      DO
31        IF NOT LeftOf[urdi.Org,uldi] AND guard.uldi <> uldi.Rprev THEN
32          IF guard.uldi = nil THEN guard.uldi = uldi FI;
33          uldi <- uldi.Rprev
34        ELSEIF NOT RightOf[uldi.Org,urdi] AND guard.urdi <> urdi.Lnext THEN
35          IF guard.urdi = nil THEN guard.urdi = urdi FI;
36          urdi <- urdi.Lnext
37        ELSE EXIT FI
38      OD;
39      basel <- Connect[rdi.Sym, ldi];
40      {remove the points that do not lie on the final convex hull}
41      ubaselOrg <- urdi.Sym.Dest; ubaselDest <- uldi.Org;
42      lcand = basel.Rprev.Sym.Onext;
        rcand = basel.Oprev.Sym.Oprev;
43      WHILE NOT lcand.Org = ubaselDest DO
44        t <- lcand.Rprev;
45        Remove_all_edges_out_of_lcand;
46        lcand <- t;
47      OD;
48      WHILE NOT rcand.Org = ubaselOrg DO
49        t <- rcand.Lnext;
50        Remove_all_edges_out_of_rcand;
51        rcand <- t;
52      OD;
53      IF ldi.Org = ldo.Org THEN ldo <- basel.Sym FI;
54      IF rdi.Org = rdo.Org THEN rdo <- basel FI;
55      DO
56        lcand <- basel.Sym.Onext;
57        IF Valid[lcand] THEN
58            WHILE OutCircle[basel.Dest, basel.Org,
                      lcand.Dest, lcand.Onext.Dest] DO
59            t <- lcand.Onext; DeleteEdge[lcand]; lcand <- t;
60          OD
61        FI;
62        rcand <- basel.Oprev;
63        IF Valid[rcand] THEN
64            WHILE OutCircle[basel.Dest, basel.Org,
                      rcand.Dest, rcand.Oprev.Dest] DO
65            t <- rcand.Oprev; DeleteEdge[rcand]; rcand <- t;
```

```
66          OD
67        FI;
68        IF NOT Valid[lcand] AND NOT Valid[rcand] THEN EXIT FI;
69        IF NOT Valid[lcand] OR
70           (Valid[rcand] AND
71              OutCircle[lcand.Dest, lcand.Org,
                 rcand.Org, rcand.Dest])
72          THEN
73            basel <- Connect[rcand, basel.Sym]
74          ELSE
75            basel <- Connect[basel.Sym, lcand.Sym]
76          FI
77        OD;
78        RETURN [ldo, rdo]
79      FI
80    END FVorTriangulation.
```

4.3.3.3 Aproximações para Objetos Genéricos

A construção de diagramas de Voronoi não se restringe a objetos do tipo ponto. Outros objetos para os quais definem-se relações de proximidade (e.g., segmentos, círculos, retângulos) também podem ter diagramas de Voronoi associados, e estes podem ser de grande utilidade prática. A implementação de algoritmos para a construção destes diagramas tende a ser muito mais complexa do que para pontos em função da introdução de bissetores não lineares (e.g., hipérboles).

Pode-se, por outro lado, simular tais diagramas[3] através do mecanismo de corbertura de pontos (vide seção 2.1.2.2) e da construção de diagramas para estas coberturas. Os diagramas resultantes, apesar de um grande número de bissetores "extras" introduzidos, sugere fortemente o resultado desejado, podendo inclusive ser aplicado na prática caso os erros decorrentes da aproximação não sejam relevantes (quanto maior a cobertura, melhor é a aproximação). Ilustramos esta interessante possibilidade através das figuras 4.28 e 4.29.

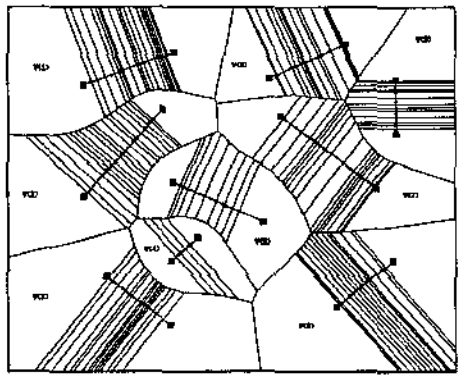

Figura 4.28: Simulação de um diagrama de Voronoi para um conjunto de segmentos

[3] Apenas para diagramas de Voronoi de vizinho mais próximo.

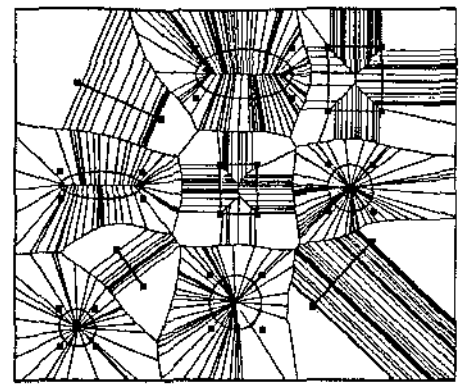

Figura 4.29: Simulação de um diagrama de Voronoi para um conjunto de segmentos, círculos, elipses e retângulos

4.3.4 Círculos Vazios e Envolventes

O uso de diagramas de Voronoi (seção 4.3.3.1) na resolução de problemas de proximidade foi aplicado mais uma vez com a implementação dos algoritmos para o cálculo do Menor Círculo Envolvente (MCE) e do Maior Círculo Vazio (MCV) de um conjunto de pontos no plano. Foi possível também analisar questões de eficiência (tempo de execução) para dois algoritmos para o cálculo do menor círculo envolvente (veja seção 2.1.2.3) através dos mecanismos de *benchmark* do ambiente e, em função disto, concluir que uma pequena variação na ordem dos componentes dos algoritmos pode resultar em constantes multiplicativas significantemente menores. De fato, um erro na proposição do algoritmo inicial de Shamos [Sha78] proporcionou o algoritmo de Bhattacharya e Toussaint [BT85], este último com constante multiplicativa menor em função da eliminação prévia dos pontos que não fazem parte da envoltória convexa do conjunto original, acarretando o cálculo do diagrama de Voronoi de vizinho mais distante com significativamente menos pontos (para distribuições normais ou uniformes), o que no final implica em menor tempo de processamento.

4.3.4.1 Menor Círculo Envolvente – MCE

Definição 4.9 *Dados n pontos no plano, encontrar o menor círculo que os envolve.*

Claramente, o menor círculo envolvente de um conjunto de pontos no plano é único (figura 4.30), e é dado ou pelo circuncentro de alguma tripla de pontos do conjunto ou por dois pontos (neste caso o diâmetro do conjunto). Um algoritmo de força bruta para o problema seria então examinar todas as triplas e pares de pontos, escolhendo o menor dos círculos assim determinados e que envolvesse todos os outros pontos. A implementação óbvia desta alternativa teria complexidade de tempo $O(n^4)$.

No campo de Pesquisa Operacional, este problema é conhecido como um problema minimax de localização de facilidades[4], no qual procuramos por um ponto $p_0 = (x_0, y_0)$ cuja maior distância a qualquer outro ponto do conjunto seja mínima, i.e.,

$$\min_{p_0} \max_{i} (x_i - x_0)^2 + (y_i - y_0)^2.$$

Os algoritmos que implementamos (descritos em linhas gerais na seção 2.1.2.3) executam em tempo $O(n \log n)$ e, em vista de um algoritmo de complexidade de tempo $\Theta(n)$ proposto por Megiddo, não

[4] *Minimax facilities location problem.*

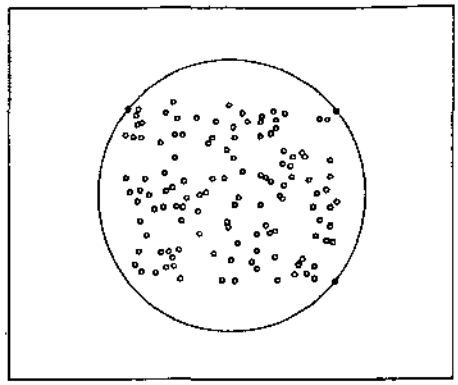

Figura 4.30: Menor Círculo Envolvente de um conjunto de pontos

são ótimos.

Ambos os algoritmos baseiam-se no diagrama de Voronoi de vizinho mais distante (seção 4.3.3.2). A grande diferença entre eles é que, no caso do algoritmo de Shamos, foi assumido erroneamente que o diâmetro do conjunto de pontos de entrada determina uma aresta no dual do diagrama de Voronoi de vizinho mais distante, o que não é verdadeiro, como mostra o contra-exemplo da figura 4.31. No caso, o diâmetro é dado por $\overline{BD}$.

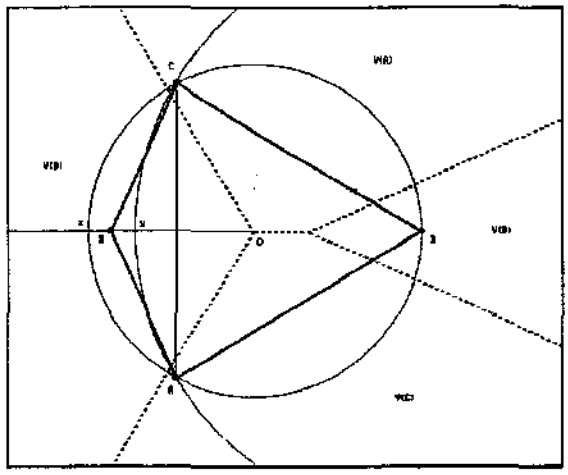

Figura 4.31: Contra exemplo para a afirmação de que o diâmetro de um conjunto de pontos determina uma aresta no dual do diagrama de Voronoi de vizinho mais distante

Apesar do erro na proposição de Shamos, surpreendentemente seu algoritmo ainda funciona de forma correta, uma vez que, se o diâmetro de um conjunto de pontos determinar o menor círculo envolvente, então certamente haverá uma aresta no dual do diagrama de Voronoi de vizinho mais distante unindo os dois vértices do diâmetro, e portanto este será corretamente encontrado. Caso contrário, então o diâmetro erroneamente calculado certamente será preterido, ou por não envolver todo o conjunto ou por existir um círculo envolvente ainda menor. Os teoremas que fundamentam este interessante resultado são apresentados em [BT85].

Apresentamos a seguir parte da implementação dos dois algoritmos mencionados. O gráfico comparativo do desempenho dos algoritmos não será colocado aqui por já ter sido apresentado na seção 2.1.2.3.

```
Circle_ptr smallest_enclosing_circle(list(Point2D_ptr) &points) {
    ⋮
    /* construct the Farthest-Neighbor Voronoi diagram of the given points */
    fvor = farthest_neighbor_voronoi_diagram(points);

    /* extract the edges of its dual (a triangulation) */
    dual_fvor_edges = fvor.dual().edges();

    /* get the longest edge */
    gs = longest_edge(dual_fvor_edges);

    /* test the given points for inclusion in */
    /* the circle having gs as diameter       */

    if (inclusion_test(points, Circle(gs)))
      MCE = Circle(gs);
    else {
      /* get the circles spawned by the faces of the */
      /* dual of the farthest-neighbor Voronoi diagram */
      fvor_circles = get_all_circles(fvor.dual());

      /* calculate the smallest circle so obtained */
      MCE = smallest_circle(fvor_circles);
    }
    ⋮
}
```

Figura 4.32: Implementação do algoritmo de Shamos para o problema MCE

```
Circle_ptr smallest_enclosing_circle(list(Point2D_ptr) &points) {
    ⋮
    /* construct the convex hull of the given points */
    /* and then calculate its diameter               */
    hull = new Polygon(graham_scan(points));
    diam = diameter_convex_polygon(hull);

    /* test the points in the hull for inclusion */
    /* in the circle having diam as diameter     */

    if (inclusion_test(hull→vertices(), Circle(diam)))
      MCE = Circle(diam)
    else {
      /* construct the Farthest-Neighbor Voronoi diagram of the hull vertices */
      fvor = farthest_neighbor_voronoi_diagram(hull→vertices());

      /* get the circles spawned by the faces of the */
      /* dual of the farthest-neighbor Voronoi diagram */
      fvor_circles = get_all_circles(fvor.dual());

      /* calculate the smallest circle so obtained */
      MCE = smallest_circle(fvor_circles);
    }
    ⋮
}
```

Figura 4.33: Implementação do algoritmo de Bhattacharya e Toussaint para o problema MCE

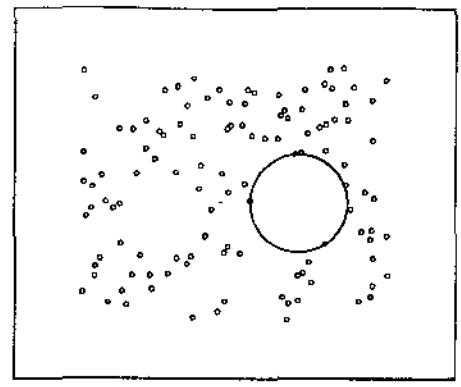

Figura 4.34: Maior Círculo Vazio de um conjunto de pontos

4.3.4.2 Maior Círculo Vazio - MCV

Definição 4.10 *Dado um conjunto S de n pontos no plano, encontrar o maior círculo livre de pontos em seu interior e cujo centro é restrito ao interior da envoltória convexa do conjunto.*

Este problema é o dual do problema anterior, sendo definido através de uma expressão *maxmin* da forma:

$$\max_{p_0 \in CH(S)} \min_i (x_i - x_0)^2 + (y_i - y_0)^2$$

e para o qual podem existir múltiplas soluções. A restrição quanto ao centro do menor círculo vazio ser interno à envoltória convexa faz-se necessária para que o problema tenha solução realizável (figura 4.34).

Uma cota inferior para este problema é $\Omega(n \log n)$ válida mesmo no caso unidimensional do problema. Neste caso, encontrar o maior círculo vazio reduz-se à busca do par de pontos consecutivos sobre uma dada linha cuja distância seja máxima (maior vão[5]).

O algoritmo implementado (figura 4.35) consiste em calcular o diagrama de Voronoi e a envoltória convexa do conjunto de entrada S (o primeiro cálculo feito em $O(n \log n)$ e o segundo em $O(n)$ através de um passo sobre o diagrama), em seguida calcular todas as interseções entre estas duas estruturas (feito em $O(n)$, que é número de interseções existentes) e em seguida buscar o maior círculo determinado ou pelas faces do dual do diagrama de Voronoi (triangulação de Delaunay) ou centrado nas interseções calculadas e limitados pelos vértices adjacentes da triangulação dual (passo também realizado em $O(n)$).

4.4 Polígonos

4.4.1 *Winding Number*

Definição 4.11 *Dado um polígono P, não necessariamente simples, e um ponto p, o winding number de P com relação a p é o número de revoluções completas em volta de p que um observador tem que realizar para percorrer completamente P em uma dada direção (e.g., no sentido anti-horário).*

[5] *Maximum gap.*

```
Circle_ptr largest_empty_circle(list(Point2D_ptr) &points) {
    ⋮
    /* first construct the Voronoi diagram of the given points */
    vor = voronoi_diagram(points);

    /* then extract the convex hull in O(n) time from the diagram */
    hull = extract_hull_from_voronoi_diagram(vor);

    /* construct a list with all circles centered in intersections */
    /* of the hull with the edges of the Voronoi diagram          */
    by_intersections = get_all_circles_by_intersections(hull, vor);

    /* get all circles spawned by the faces of the   */
    /* Voronoi diagram dual (Delaunay triangulation) */
    by_vertices = get_all_circles(vor.dual());

    /* select the largest empty circle, that is either one in the */
    /* by_intersections list or one in the by_vertices list       */

    MCV = largest_circle(by_intersections, by_vertices);
    ⋮
}
```

Figura 4.35: Implementação do algoritmo para o cálculo do MCV de um conjunto de pontos

O algoritmo para cálculo do *winding number*, cuja utilidade prática básica para os demais algoritmos é verificar a ordem com que são dados os vértices de um polígono, tem merecido significativa atenção por ser especialmente didático para mostrar problemas com precisão numérica e casos degenerados (e.g., p sobre o perímetro de P) que surgem mesmo em problemas simples em Geometria Computacional [Sch91].

4.4.2 Núcleo de um Polígono Simples

Definição 4.12 *O núcleo de um polígono simples P, $K(P)$, é o lugar geométrico dos pontos internos a P visíveis por todos os vértices de P, i.e., um segmento a partir de qualquer vértice de P com a outra extremidade em $K(P)$ está totalmente contido em P.*

O cálculo do núcleo de um polígono simples (figura 4.37) é um problema clássico para o qual existe uma solução ótima (tempo proporcional ao número de vértices) em função da ordem parcial estabelecida pelos vértices do polígono simples sendo processado.

O algoritmo que implementamos, proposto por Lee e Preparata [LP79], explora esta possibilidade através da construção incremental de $K(P)$ e, a cada iteração, o resultado parcial é transportado para a seguinte cuidadosamente mantendo-se informações que limitam em $O(n)$ o número de passos total do algoritmo, onde n é é o número de vértices de P.

A descrição que se segue, apesar de aparentemente simples, caracteriza-se por ter uma implementação extremamente extensa (da ordem de 1.000 linhas em código C++) e serve como padrão de

```
int winding_number(const list(Point2D_ptr) &Points, const Point2D &ref) {
    int wn = 0;
    Point2D_ptr u = Points.head();
    list_item it = Points.first();
    do {
      it = Points.cyclic_succ(it);
      Point2D_ptr v = Points[it];
      switch (ref.which_side(*u, *v)) {
        case Left:
          if (u→X() ≤ ref.X() && ref.X() < v→X())
            wn--;
          break;
        case Right:
          if (v→X() ≤ ref.X() && ref.X() < u→X())
            wn++;
          break;
      }
      u = v;
    } while(it ≠ Points.first());
    return wn;
}
```

Figura 4.36: Implementação do algoritmo para o cálculo do *winding number*

complexidade para uma grande quantidade de algoritmos em Geometria Computacional (a descrição, por outro lado, limita-se a uma ou duas páginas de texto).

Sua implementação, que consumiu apenas 4 dias desde a codificação até o final da depuração, mostra o quanto o **GeoLab** facilita a construção de algoritmos geométricos.

4.4.2.1 Conceitos e Nomeclatura

O algoritmo percorre ordenadamente os vértices de um polígono simples P (assume-se que os vértices são dados no sentido anti-horário) construindo uma seqüência de polígonos convexos $K_1, K_2, \ldots, K_{n-1}$. Estes polígonos podem ou não ser fechados. Um polígono K_i é formado pela interseção dos meios-

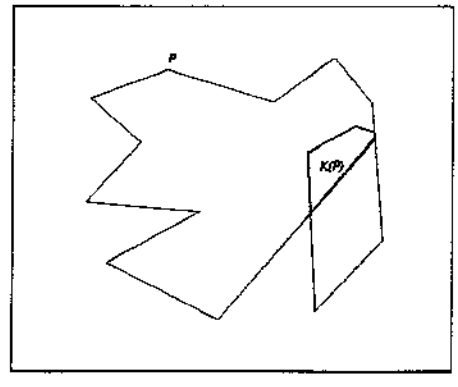

Figura 4.37: Núcleo de um polígono simples

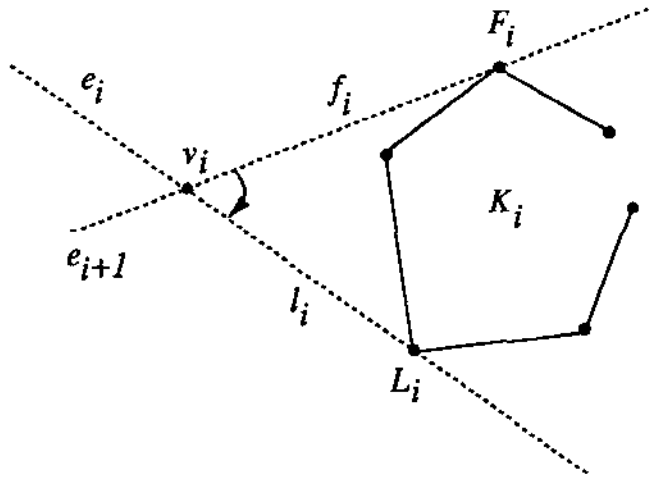

Figura 4.38: Ilustração da definição de F_i e L_i

planos à esqueda das arestas direcionadas $e_0, e_1, \ldots, e_n$. Obtém-se com esta construção que $K_{n-1} = K(P)$ e que $K_1 \supseteq K_2 \supseteq \ldots \supseteq K_i$, o que implica que existe um r tal que K_i é fechado se e somente se $i \geq r$.

Na notação utilizada a seguir, se os pontos w_i e w_{i+1} pertencem à reta que contém a aresta e_{s_i} de P, então $w_i e_{s_i} w_{i+1}$ denota o segmento entre w_i e w_{i+1} com a mesma direção de e_{s_i}. Quando um polígono K_i é aberto, suas duas extremidades são raios. $\Lambda e w$ denota um raio cujo ponto inicial é w e direcionado como a aresta e, enquanto $we\Lambda$ denota o raio complementar.

Durante o processamento, a fronteira de K_i é mantida como uma lista duplamente encadeada. Esta lista será linear ou circular, dependendo se K_i é aberto ou fechado, respectivamente. No primeiro caso, os elementos inicial e final da lista são chamados *cabeça* e *cauda* da lista, respectivamente.

Dentre os vértices de K_i distinguimos dois vértices, F_i e L_i definidos da seguinte forma: considere as duas linhas de suporte de K_i através de um vértice v_i de P. Sejam f_i e l_i dois raios sobre estas linhas que contenham os pontos de suporte, tais que o ângulo no sentido horário de f_i para l_i no setor do plano que contém K_i não é maior que π (figura 4.38). O vértice F_i é o ponto comum a f_i e K_i cuja distância a v_i é máxima. Define-se L_i de forma análoga. Estes dois vértices tem importância fundamental na construção de K_{i+1} a partir de K_i.

Um vértice v_i de P é dito *côncavo* se v_{i+1} está à direita da reta contendo e_i e direcionada no mesmo sentido, i.e., o ângulo interno do vértice v_i é maior que π. v_i é dito *convexo* caso contrário.

4.4.2.2 O Algoritmo

- **Inicialização**

  Trivialmente, se P não tiver um vértice côncavo então $K(P) = P$ e o algoritmo termina.

  Caso contrário, seja v_0 (vértice inicial) um vértice côncavo de P.

  Seja K_1 a interseção dos meios-planos à esquerda das retas contendo e_0 e e_1 e direcionadas da mesma forma, i.e., $K_1 = \Lambda e_1 v_0 e_0 \Lambda$. Seja F_1 um ponto no infinito de $\Lambda e_1 v_0$ e L_1 um ponto no infinito de $v_0 e_0 \Lambda$ (figura 4.39).

- **Passo Geral**

  1. v_i é côncavo (figuras 4.40 (a, b))

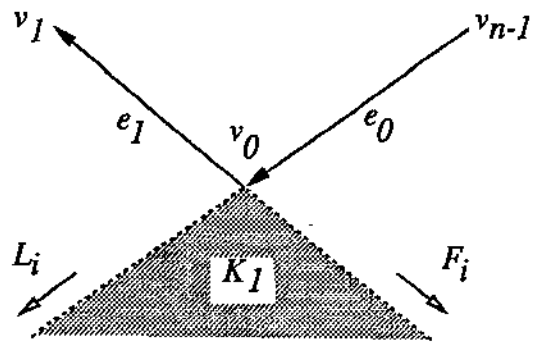

Figura 4.39: Inicialização de K_1

(a) F_i está à direita ou sobre $\Lambda e_{i+1}v_{i+1}$ (figura 4.40 (a))
Percorre-se K_i no sentido anti-horário a partir de F_i até que se encontre uma aresta $(w_{t-1}w_t)$ interceptando $\Lambda e_{i+1}v_{i+1}$ ou até que se atinja L_i sem encontrar tal aresta. Neste último caso, o algoritmo termina ($K(P) = \emptyset$). No primeiro caso, realiza-se as seguintes ações:

i. Encontra-se a interseção w' de $(w_{t-1}w_t)$ com $\Lambda e_{i+1}v_{i+1}$.

ii. Percorre-se K_i no sentido horário a partir de F_i até que se encontre uma aresta (w_{s-1}, w_s) interceptando $\Lambda e_{i+1}v_{i+1}$ em um ponto w'' (garantido se K_i é fechado) ou (somente quando K_i é aberto) se chegue ao final da lista sem encontrar tal aresta. No primeiro caso, sendo $K_i = \alpha w_s \ldots w_{t-1}\beta$ (onde α e β são seqüências alternantes de arestas e vértices), faz-se $K_{i+1} \leftarrow \alpha w'' e_{i+1} w'\beta$; no segundo caso ($K_i$ é aberto), testa-se se K_{i+1} será aberto ou fechado. Se a inclinação de $\Lambda e_{i+1}v_{i+1}$ estiver compreendida entre as inclinações dos raios (cabeça e cauda) de K_i, então $K_{i+1} \leftarrow \Lambda e_{i+1}w'\beta$ é também aberto. Caso contrário, inicia-se uma busca no sentido horário sobre K_i a partir da cauda da lista até que se encontre uma aresta $(w_{r-1}w_r)$ que intercepta $\Lambda e_{i+1}v_{i+1}$ em um ponto w''; sendo $K_i = \gamma w_{i-1}\delta w_r \eta$, faz-se $K_{i+1} \leftarrow \delta w'' e_{i+1} w'$ e K_{i+1} passa a ser fechado (a lista torna-se circular).

A seleção de F_{i+1} é feita da seguinte forma: se $\Lambda e_{i+1}v_{i+1}$ tem apenas uma interseção com K_i, então $F_{i+1} \leftarrow$ (*ponto no infinito de* $\Lambda e_{i+1}v_{i+1}$); caso contrário, $F_{i+1} \leftarrow w''$. Para determinar L_{i+1}, percorre-se K_i no sentido anti-horário a partir de L_i até que se encontre um vértice w_u tal que w_{u+1} está à esquerda de $v_{i+1}(v_{i+1}w_u)\Lambda$, ou a lista que representa K_i é esgotada. No primeiro caso, faz-se $L_{i+1} \leftarrow w_u$; no outro caso, que pode acontecer apenas se K_i for aberto, faz-se $L_{i+1} \leftarrow L_i$.

(b) F_i está à esquerda de $\Lambda e_{i+1}v_{i+1}$ (figura 4.40 (b))
Neste caso, $K_{i+1} \leftarrow K_i$ mas F_i e L_i precisam ser atualizados. Para determinar F_{i+1} percorre-se K_i no sentido anti-horário a partir de F_i até que se encontre um vértice w_t tal que w_{t+1} está à direita de $v_{i+1}(v_{i+1}w_t)\Lambda$; faz-se então $F_{i+1} \leftarrow w_t$. A determinação de L_{i+1} se faz como no caso 1a.

2. v_i é convexo (figuras 4.41 (a, b))

(a) L_i está à direita ou sobre $v_i e_{i+1}\Lambda$ (figura 4.41 (a))
Percorre-se K_i no sentido horário a partir de L_i até que se encontre uma aresta (w_{t-1}, w_t) interceptando $v_i e_{i+1}\Lambda$ ou até que se atinja F_i sem encontrar tal aresta.

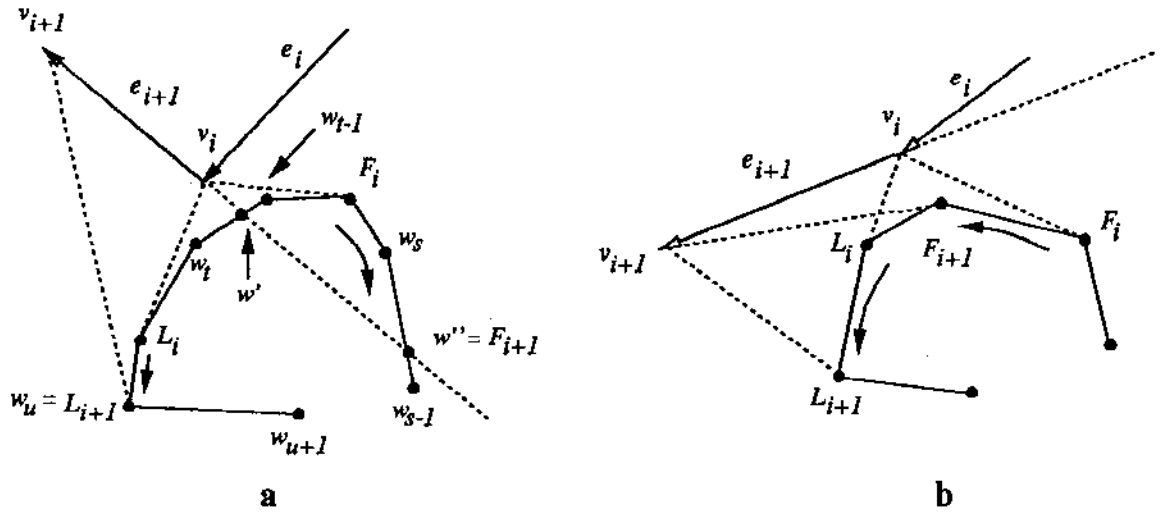

Figura 4.40: Passo geral quando v_i é côncavo

Neste último caso, o algoritmo termina ($K(P) = \emptyset$). No primeiro caso, realiza-se as seguintes ações:

i. Encontra-se a interseção w' de $(w_{t-1}w_t)$ com $v_ie_{i+1}\Lambda$.

ii. Percorre-se K_i no sentido anti-horário a partir de L_i até que se encontre uma aresta $(w_{s-1}w_s)$ interceptando $v_ie_{i+1}\Lambda$ em um ponto w'' (garantido se K_i for fechado) ou (somente quando K_i é aberto) se chegue ao final da lista sem encontrar tal aresta. No primeiro caso, sendo $K_i = \alpha w_t \ldots w_{s-1}\beta$, faz-se $K_{i+1} \leftarrow \alpha w' e_{i+1} w''\beta$; no segundo caso ($K_i$ é aberto), testa-se se K_{i+1} será aberto ou fechado. Se a inclinação de $v_ie_{i+1}\Lambda$ estiver compreendida entre as inclinações dos raios (cabeça e cauda) de K_i, então $K_{i+1} \leftarrow \alpha w' e_{i+1}\Lambda$ é também aberto. Caso contrário, inicia-se uma busca no sentido anti-horário a partir da cabeça da lista até que se encontre uma aresta $(w_{r-1}w_r)$ que intercepta $v_ie_{i+1}\Lambda$ em um ponto w''; sendo $K_i = \gamma w_{r-1}\delta w_i\eta$, faz-se $K_{i+1} \leftarrow \delta w' e_{i+1} w''$ e K_{i+1} passa a ser fechado (a lista torna-se circular).

A seleção de F_{i+1} e L_{i+1} depende da posição de v_i com relação ao raio $v_ie_{i+1}\Lambda$ e também do número de interseções de $v_ie_{i+1}\Lambda$ com K_i. Distinguem-se dois casos:

i. $v_ie_{i+1}\Lambda$ intercepta K_i em w' e w''. Se $v_{i+1} \in [v_ie_{i+1}w']$, então F_{i+1} é selecionado como em 1b; senão, $F_{i+1} \leftarrow w'$. Se $v_{i+1} \in [v_ie_{i+1}w'']$, então $L_{i+1} \leftarrow w''$; senão, L_{i+1} é selecionado como em 1a, exceto que percorre-se K_i no sentido anti-horário a partir de w''.

ii. $v_ie_{i+1}\Lambda$ intercepta K_i apenas em w'. Se $v_{i+1} \in [v_ie_{i+1}w']$, então F_{i+1} é selecionado como em 1b; senão, $F_{i+1} \leftarrow w'$. $L_{i+1} \leftarrow$ (*ponto no infinito de* $v_ie_{i+1}\Lambda$).

(b) L_i está à esquerda de $v_ie_{i+1}\Lambda$ (figura 4.41 (b))
Neste caso, $K_{i+1} \leftarrow K_i$. F_{i+1} é determinado como em 1b. Se K_i for fechado, então L_{i+1} é determinado como em 1a; senão, $L_{i+1} \leftarrow L_i$.

Uma completa prova de corretude e análise de complexidade deste algoritmo pode ser encontrada em [LP79].

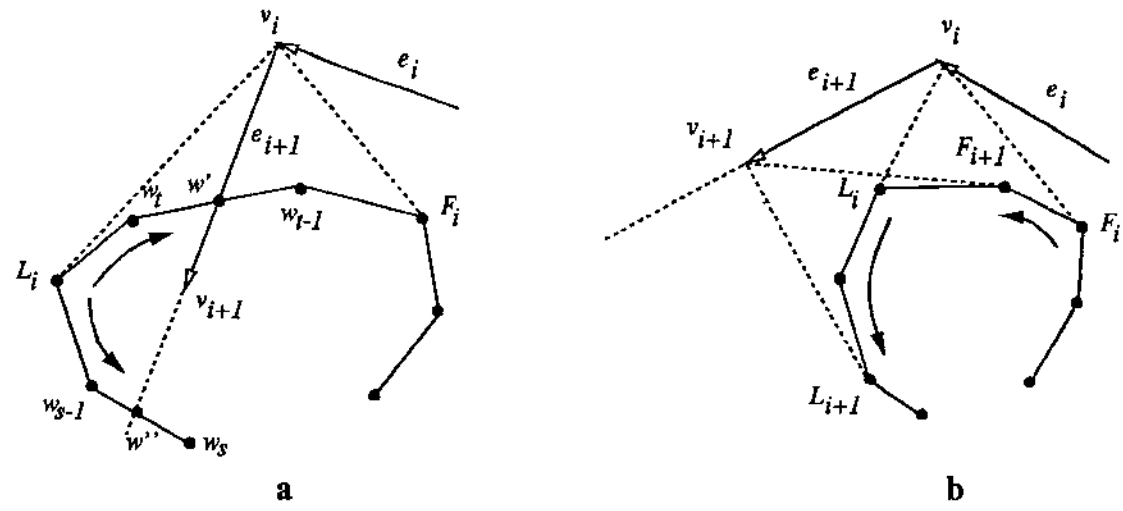

Figura 4.41: Passo geral quando v_i é convexo

4.4.2.3 Implementação

Apesar dos passos gerais terem sido introduzidos na descrição do algoritmo, as principais dificuldades ficam "escondidas" ou na falta de detalhes ou nas primitivas mencionadas na descrição. Algumas destas dificuldades com relação a este particular algoritmo são:

1. Como representar pontos no infinito (as extremidades de K_i aberto)?

2. O teste da posição relativa de F_i e L_i com relação a $\Lambda e_{i+1} v_{i+1}$ (e outros testes similares) é descrito de forma geral, e aplicado tanto para F_i ou L_i atingíveis quanto no infinito em alguma direção. Como resolver coerentemente os dois tipos de testes (que de fato são implementados de formas diferentes) e como identificar que tipo de teste aplicar?

3. O que se deve manter na lista que representa K_i? Vértices, arestas ou raios? Quanto aos K_i abertos, como representar consistentemente os raios (cabeça e cauda) da lista utilizando-se uma estrutura de dados geral?

4. Menciona-se no algoritmo que algumas buscas são realizadas em K_i, mas por vezes atualiza K_{i+1} antes de se realizar tais buscas. Uma observação mais cuidadosa desta atitude revela que, se quisermos manter a otimalidade do algoritmo, então a ordem com que são feitas as atualizações e as buscas terá que ser redefinida, já que não pode realmente haver vários K_i (o que demandaria cópia de estruturas entre cada iteração e implicaria na deterioração da eficiência do algoritmo).

As respostas para estes problemas encontram-se na vasta quantidade de facilidades para programação existentes no **GeoLab**. Pontos são representados internamente através de coordenadas homogêneas, facilitando a representação de pontos no infinito. Esta mesma representação interna cria uma coerente abstração nas primitivas (funções) envolvidas, como no teste da posição relativa entre pontos. Na implementação em questão, optou-se por armazenar vértices na lista que representa K_i. Esta lista pode ser tratada tanto como aberta ou fechada. Quando aberta, vértices nas extremidades indicam a presença (implícita) de raios, tratados também homogeneamente dada a relação hierárquica e polimórfica entre retas, raios e segmentos em nossa implementação.

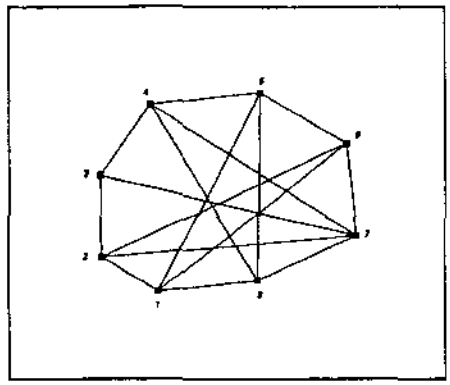

Figura 4.42: Pares antipodais de um polígono convexo

A apresentação da implementação correspondente ao algoritmo de Lee e Preparata foi aqui omitida por questões de espaço.

4.4.3 Pares Antipodais

Definição 4.13 *Dado um polígono convexo P, calcular todos os seus pares antipodais, que são pares de vértices de P que admitem linhas de suporte paralelas distintas.*

O cálculo dos pares antipodais (figura 4.42) de um polígono convexo encontra aplicação em problemas relacionados com o diâmetro (par de vértices mais distantes) de conjuntos de pontos e de polígonos. Como exemplo, dado um conjunto S de n pontos no plano, o diâmetro de S pode ser calculado em tempo $O(n \log n)$ da forma:

1. $P \leftarrow CH(S)$;
2. $diam(S) = diam(P) \leftarrow \min(AntiPairs(P))$.

O algoritmo implementado foi proposto por Shamos [Sha78], e calcula todos os pares antipodais de um polígono convexo em tempo linear (o número de pares antipodais é proporcional ao número de vértices do polígono), e portanto ótimo. A idéia é bastante simples: A ordem cíclica (horária neste caso) assumida para os vértices de P induzem um direcionamento das arestas, o que permite interpretá-los como vetores. Transladando estes vetores para a origem (figura 4.43) temos que os vértices de P são mapeados em setores.

Na discussão que se segue, recorreremos à figura 4.43. Para encontrar o par de vértices antipodais correspondentes a uma dada região, primeiramente cria-se uma reta D que passe pela origem dos vetores e atravesse a região em questão. Considera-se então uma rotação de D no sentido horário. No início, D define o par antipodal $4-7$. Este par continuará sendo o par corrente até que D passe por algum outro vetor no diagrama. Quando isto acontece no exemplo, o primeiro vetor ultrapassado por D é o vetor 78, onde o par antipodal muda de $4-7$ para $4-8$. Similarmente, quando D passa pelo vetor 81 o par passa a ser $4-1$. Quando D atinge 45 o par antipodal corrente torna-se $5-1$. Como D sempre atravessa dois setores, podemos manter um apontador para cada um destes setores (inicializados em tempo linear) na busca do primeiro par antipodal) de tal forma que para encontrar o próximo par antipodal seleciona-se a fronteira que estiver mais próxima e atualiza-se o apontador de setor correspondente para o próximo setor, reportando-se um novo par antipodal, e então repete-se

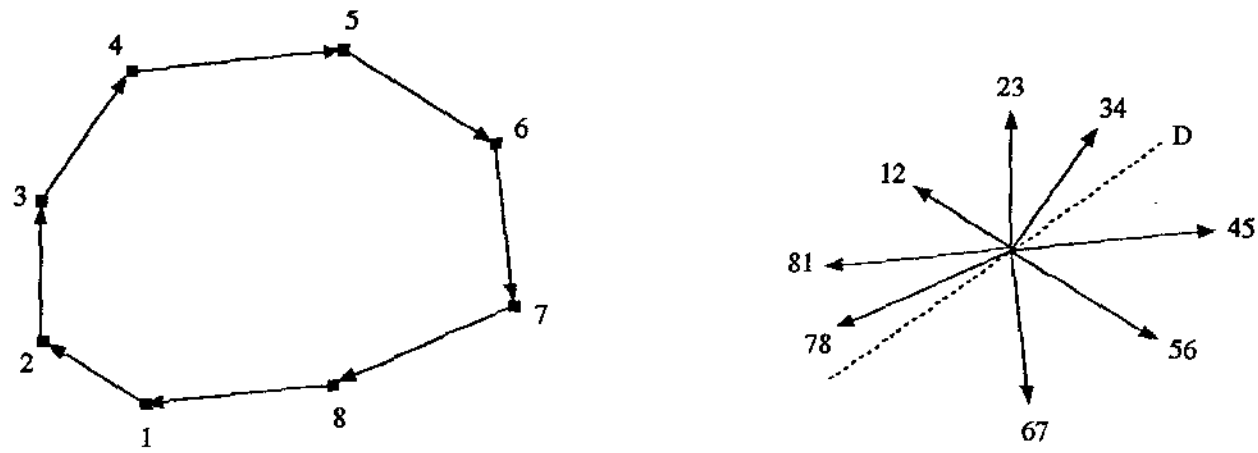

Figura 4.43: Mapeamento de arestas em vetores e vértices em setores para a proposição do algoritmo de Shamos

a seleção. Os apontadores percorrerão cada um dos vértices (sendo que sempre estarão em setores opostos) e portanto em tempo proporcional a n todos os pares antipodais serão reportados.

A implementação apresentada na figura 4.44 (detalhada em [PS85]) utiliza-se do cálculo da área de triângulos para determinar o avanço dos apontadores.

4.4.4 Envoltória Convexa de Polígonos Simples

Outro conjunto interessante de problemas relacionados com polígonos é a construção das envoltórias convexas dos vértices de polígonos não convexos. A definição para o problema foi apresentada na seção 4.3.1, com a diferença que o conjunto S de pontos é formado pelos vértices do polígono P em questão, *na mesma ordem que ocorrem em P*. Apesar de termos implementado algoritmos para vários tipos de polígonos (genéricos, simples e em forma de estrela), restringimo-nos aqui a comentar apenas a construção da envoltória convexa de polígonos simples.

O algoritmo proposto por Lee [Lee83] explora a existência de propriedades especiais no conjunto de entrada (i.e., simplicidade) para a produção de um algoritmo de tempo linear para o cálculo da envoltória convexa dos vértices de um polígono simples[6]. Nós apresentamos o algoritmo e sua implementação a seguir.

4.4.4.1 Notações e Conceitos

Denota-se o polígono simples sendo processado por uma lista de vértices, i.e., $P = v_0, v_1, \ldots, v_n$ tal que (v_i, v_{i+1}) é uma aresta de P[7]. Assume-se que o interior de P está à esquerda das arestas direcionadas, o que assume por sua vez que os vértices são dados no sentido anti-horário. A lista de vértices deve ser duplamente encadeada, tal que $CCW(v_i)$ denote o sucessor de v_i no sentido anti-horário (i.e., v_{i+1}) e $CW(v_i)$ o sucessor de v_i no sentido horário (i.e., v_{i-1}). Assume-se que o vértice inicial v_0 é o vértice de P de menor coordenada y.

[6] Observa-se que a cota inferior de tempo $\Omega(n \log n)$ não mais se aplica, já que os pontos não estão em posição geral, o que sugere a existência de outros problemas (muitos ainda a serem descobertos) onde a presença de propriedades especiais (e.g., simplicidade, monotonicidade) podem proporcionar algoritmos mais eficientes que aqueles para casos gerais.

[7] Índices são tomados módulo n.

```
⋮
// assume that the polygon vertices (points) are given
// in counterclockwise order and that there are more than
// three vertices

p = points.last();
q = points.cyclic_succ(p);

// find the first antipodal pair and set p and q as opposite pointers
// that will move around the polygon (rotation of the D line in the algorithm)
while (points[points.cyclic_succ(q)]→area(*points[p], *points[points.cyclic_succ(p)]) >
        points[q]→area(*points[p], *points[points.cyclic_succ(p)]))
  q = points.cyclic_succ(q);

q0 = q;
while(q ≠ points.first()) {
  p = points.cyclic_succ(p);
  pairs.push(new Segment2D(*points[p], *points[q]));
  while (points[points.cyclic_succ(q)]→area(*points[p], *points[points.cyclic_succ(p)]) >
          points[q]→area(*points[p], *points[points.cyclic_succ(p)])) {
    q = points.cyclic_succ(q);
    if (p ≠ q0 || q ≠ points.first())
      pairs.push(new Segment2D(*points[p], *points[q]));
    else {
      stop = true;
      break;
    }
  }
  if (!stop && points[points.cyclic_succ(q)]→area(*points[p], *points[points.cyclic_succ(p)]) ==
                 points[q]→area(*points[p], *points[points.cyclic_succ(p)]))
    if (p ≠ q0 || q ≠ points.last())
      pairs.push(new Segment2D(*points[p], *points[points.cyclic_succ(q)]));
}
⋮
```

Figura 4.44: Implementação do algoritmo de Shamos para o cálculo dos pares antipodais de um polígono convexo

A envoltória convexa de P é representada por uma lista circular tal que $PRED(v_i)$ denota o predecessor de v_i. Se os vértices v_{i_j} e $v_{i_{j+1}}$ pertencentes a $CH(P)$ não são vértices extremos de uma mesma aresta, então $v_{i_j}, CCW(v_{i_j}), CCW(CCW(v_{i_j})), \ldots, v_{i_{j+1}}$ definem uma região fechada denominada *lóbulo*[8], sendo a aresta $(v_{i_j}, v_{i_{j+1}})$ denominada *alça*[9]. A porção de um lóbulo sem sua alça é referenciada como *corpo*[10] do lóbulo (figura 4.45). Se v_{i_j} e $v_{i_{j+1}}$ são vértices extremos de uma mesma aresta de P, então $(v_{i_j}, v_{i_{j+1}})$ definem um lóbulo degenerado com corpo vazio.

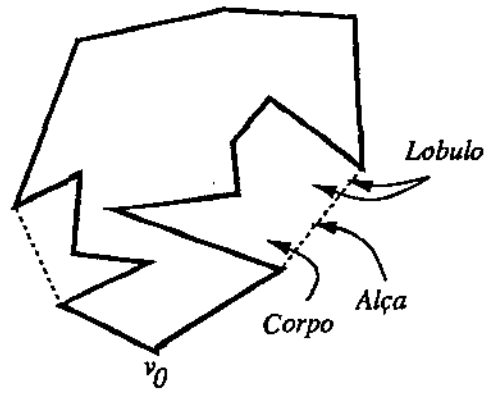

Figura 4.45: Ilustração das definições de *lóbulo*, *alça* e *corpo*.

Durante a execução do algoritmo, a lista circular que mantém a envoltória convexa sendo construída é também tratada como pilha, contendo em um dado passo os vértices sabidamente na envoltória juntamente com os vértices sendo examinados, *menos* aqueles que pertencem ao corpo de lóbulos já processados (que obviamente não pertencem à envoltória final). Mais precisamente, se o conteúdo da pilha (chamada configuração C da pilha) consiste dos vértices $v_{i_0}, v_{i_1}, \ldots, v_{i_k}$, $0 = i_0 < i_1 < \ldots < i_k$, então $C = v_{i_0}, v_{i_1}, \ldots, v_{i_k}$ é por si só um polígono convexo. Quando o último vértice v_{n-1} for examinado e pertencer ou à configuração $C = v_{i_0}, \ldots, v_{i_k}$, $i_k = n-1$, ou ao lóbulo $L(v_{i_k}, v_{i_0})$, então $C = v_{i_0}, v_{i_1}, \ldots, v_{i_k}$ será a envoltória convexa de P.

4.4.4.2 O Algoritmo

- Inicialização

  Sendo v_0 o vértice de menor coordenada y, e portanto um vértice sabidamente na envoltória final, v_0 é empilhado inicialmente. O próximo vértice $v_1 = CCW(v_0)$, é também empilhado (será verificado). Associada a essa configuração inicial, determina-se uma reta direcionada $l(C)$ passando pelos dois vértices do topo da pilha, chamada *linha corrente*. A aresta determinada por estes dois vértices é chamada *aresta corrente*.

- Passo Geral

  Percorrendo-se os vértices de P a partir de $CCW(CCW(v_0))$ (chama-se o vértice v_j sendo considerado de *vértice corrente* – figura 4.46), distinguem-se dois tipos de configurações:

  1. v_j não está estritamente à esquerda de $l(C)$ (figura 4.47)

[8] *Lobe*.
[9] *Handle*.
[10] *Body*.

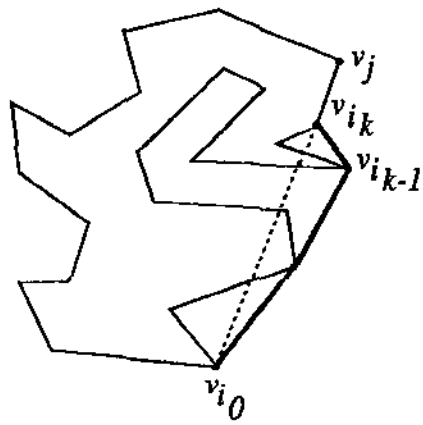

Figura 4.46: Exemplo de configuração

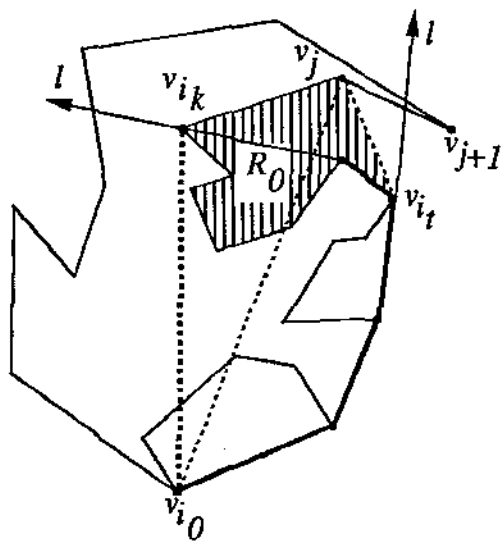

Figura 4.47: Tratamento do primeiro caso – v_{i_j} não está estritamente à esquerda de $l(C)$

Neste caso, o vértice v_{i_k} do topo da pilha e possivelmente alguns vértices abaixo na pilha devem ser descartados por não mais pertencerem a uma configuração válida. Atualiza-se a pilha descartando-se tais vértices até que se encontre v_{i_t} tal que v_j está estritamente à esquerda de $\overrightarrow{v_{i_{t-1}} v_{i_t}}$ ou até que se chegue a v_0. Depois da atualização obtém-se uma nova configuração $C' = v_{i_0}, v_{i_1}, \ldots, v_{i_t}, v_j$ repetindo-se então o passo geral para v_{j+1}.

2. v_j está à esquerda de $l(C)$ (figuras 4.48)

   Neste caso, distinguem-se ainda duas situações:

   (a) v_j está na região R_0 definida pelo lóbulo $L(v_{i_{k-1}}, v_{i_k})$ (figura 4.48 (a))
   Neste caso, v_{i_k} e v_j não mais pertencem à envoltória convexa. De fato, todos os vértices que sucedem v_j e estão em R_0 não pertencem à envoltória e podem ser ignorados. Busca-se então, nos vértices seguintes, por um vértice v_s de P tal que v_s esteja fora de R_0. Constrói-se então uma nova configuração $< C', v_s >$ com v_s estritamente à direita de $l(C)$ e o passo geral novamente se repete.

   (b) v_j não está na região R_0

      i. v_j é interno ao polígono convexo correspondente à configuração corrente (v_j não

está estritamente à direita de $\overrightarrow{v_{i_k} v_{i_0}}$ - figura 4.48 (b))
Neste caso, decide-se que v_j não pertence à envoltória convexa final. De fato, todos os vértices que sucedem v_j que são internos à configuração corrente (região R_1) podem ser ignorados. Busca-se então o próximo vértice v_s tal que v_s esteja fora da região R_1, obtendo-se uma nova configuração $< C', v_s >$ que voltará a ser processada pelo passo geral.

ii. v_j está estritamente à direita de $\overrightarrow{v_{i_k}, v_{i_0}}$ (região R_2 - figura 4.48 (c))
Neste caso, cria-se uma nova configuração $C' =< C, v_{j+1} >$ (pois v_j é empilhado por pertencer à envoltória convexa contruída até aqui) e o passo geral se repete.

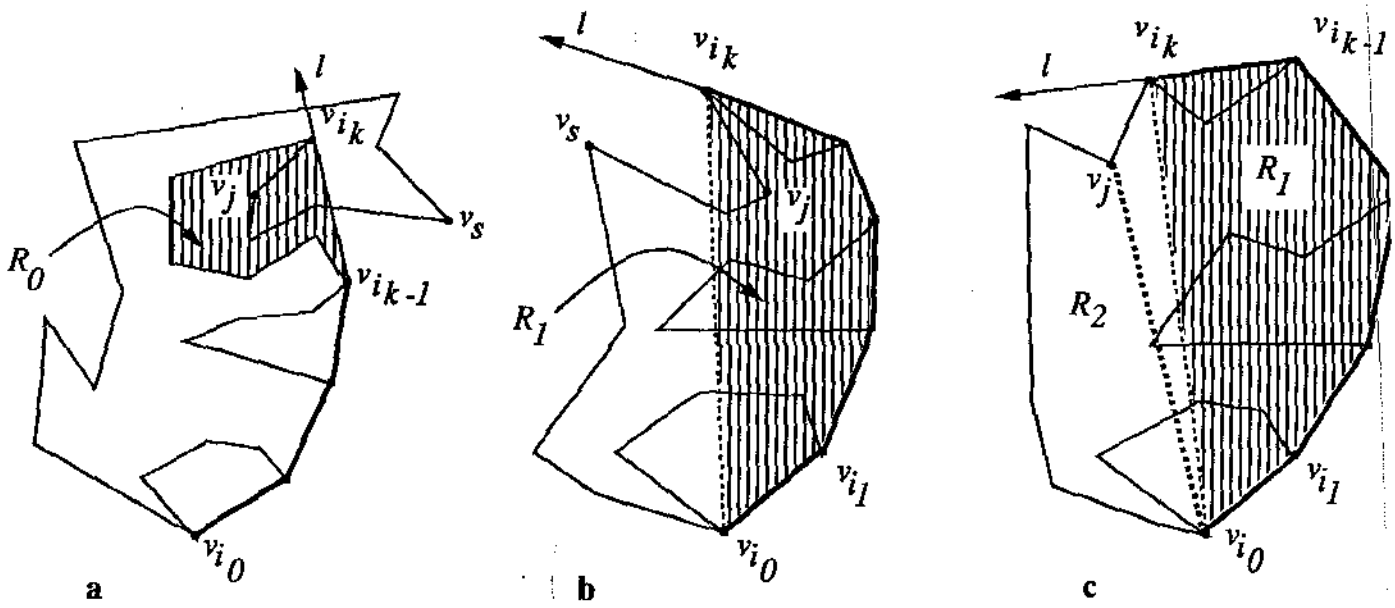

Figura 4.48: v_j à esquerda de $l(C)$. (a) v_j na região R_0; (b) v_j na região R_1; (c) v_j na região R_2.

Uma prova de corretude juntamente com uma análise de complexidade deste algoritmo pode ser encontrada em [Lee83].

4.4.4.3 Implementação

Trechos da implementação do algoritmo de Lee são apresentados na figura 4.49.

```
list(Point2D_ptr) ch_simple_polygon(list(Point2D_ptr) &points) {
    list(Point2D_ptr) result;

    // omited code: remove colinear points, assert that points are given in
    //              counterclockwise order, set v0 as the point with smallest
    //              y coordinate and push v0 and v1 into the stack

    next = points.cyclic_succ(points.succ(points.first()));

    l1 = *points.head();
    l2 = *points[points.succ(points.first())];

    while (next ≠ points.first()) {
      if (!points[next]→left_turn(l1, l2)) {
        update(stack, points, next, l1, l2);
        close_lobe(stack, points, next, l1, l2, w);
      }
      else {
        if (!points[next]→right_turn(*stack.head(), *points.head()))
          // next in R1
          do {
            next = points.cyclic_succ(next);
          } while (next ≠ points.first() &&
                   !points[next]→right_turn(*stack.head(), *points.head()));
        else {
          // next in R2
          Point2D ll1 = *stack.head(), ll2 = *points[next];
          close_lobe(stack, points, next, ll1, ll2);
          l1 = ll1; l2 = ll2;
        }
      }
    }

    ⋮
    return points;
}
```

Figura 4.49: Implementação do algoritmo de Lee para o cálculo da envoltória convexa de polígonos simples

4.5 Suporte para Criação de Ilustrações

Outra aplicação derivada da possibilidade de ligação dinâmica de novos modos de criação interativa e tipos de objetos geométricos é a introdução de suporte gráfico e textual para a criação de ilustrações em trabalhos científicos.

As figuras apresentadas até aqui foram geradas graças a esse suporte, que vem sendo ampliado e aprimorado no **GeoLab**. A colocação de rótulos em objetos geométricos é a única facilidade para ilustração proporcionada internamente pelo **GeoLab**, e todas as outras ferramentas que complementam esta facilidade são ligadas dinâmicamente.

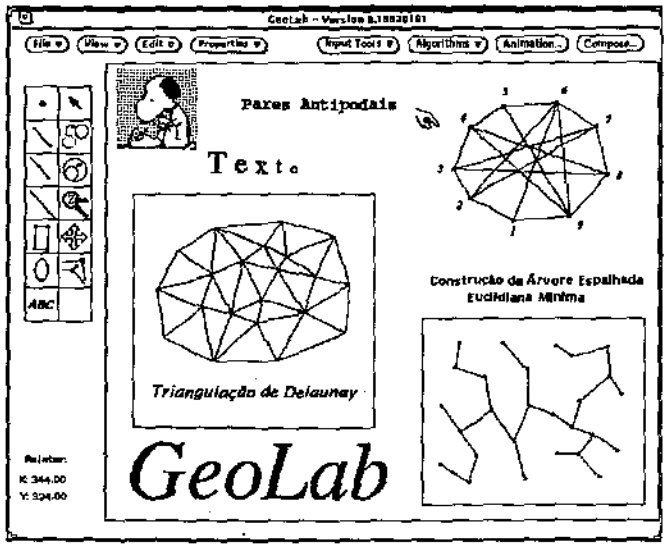

Figura 4.50: Exemplo de ilustração produzida pelo **GeoLab**

Apesar de contarmos ainda com um número muito reduzido destas ferramentas, estamos certos que, com a abordagem utilizada, este conjunto crescerá à medida das necessidades, possibilitando formas mais simples de se produzir ilustrações para trabalhos em Geometria Computacional.

Capítulo 5

Conclusão

O trabalho investigativo com uma conotação essencialmente prática realizado durante um período ininterrupto de 18 meses resultou em um novo ambiente de desenvolvimento de algoritmos geométricos bastante elaborado e completo. O sucesso deste empreendimento pode ser medido parcialmente pela diversidade de trabalhos futuros agora factíveis e de algoritmos a serem implementados para estudos práticos, já que nosso objetivo era precisamente o de desenvolver um ambiente que proporcionasse a oportunidade de se abordar tais problemas de forma sistemática e coerente.

Antes de mencionar quais são os trabalhos futuros, faz-se necessário ressaltar a importância de todas as ferramentas utilizadas na confecção do **GeoLab**.

5.1 Ferramentas Utilizadas na Construção do GeoLab

5.1.1 Ambiente Hospedeiro

Todo trabalho de implementação para a construção do **GeoLab** foi realizado utilizando estações de trabalho do tipo Sun Sparcstation sob o controle do sistema operacional UNIX. Esta plataforma revelou-se extremamente eficiente tanto em termos de performance (velocidade de processamento) quanto da existência de elaboradas ferramentas de programação, muito embora a robustez destas ferramentas deixasse a desejar em alguns casos.

Em particular, os recursos de programação como os utilitários Make (para compilação dos módulos do ambiente), Guide (geração automática de interfaces para o sistema XView) e o SCCS (*Source Code Control System*) foram de extrema utilidade, ajudando na organização e acelerando o trabalho das pessoas envolvidas com o **GeoLab**. Outros utilitários desenvolvidos aqui mesmo no Departamento de Ciência da Computação da UNICAMP auxiliaram em muito nosso trabalho, como o programa desenvolvido por Ana Cláudia Biazetti para geração automática de *makefiles* ("Make-Make").

Recursos disponíveis no sistema operacional, como o suporte para bibliotecas compartilhadas, também determinaram alguns aspectos do ambiente de forma decisiva, muito embora acreditemos que nelas faltem várias extensões.

O ambiente de janelas por sua vez (XView) ainda está longe de ser considerado um produto acabado, falando no caso da versão 2.0 que temos em mãos. Existem alguns *bugs* que ainda deverão ser reparados, e seriam desejáveis extensões para suportar linguagens orientadas a objetos como C++, sem que para isso tivéssemos que recorrer a artifícios indiretos. Especificamente, seria desejável ter suporte direto para a associação de *callbacks* a métodos de classes em C++.

5.1.2 Orientação a Objetos

O paradigma de orientação a objetos nos proporcionou os recursos necessários para uma organização consistente e flexível, permitindo a inclusão no **GeoLab** de recursos inexistentes nos projetos afins de que se tem conhecimento. A introdução de algoritmos e objetos geométricos de forma dinâmica é um destes recursos, sendo que isto só foi possível graças ao estabelecimento de protocolos para estas entidades juntamente com a utilização das bibliotecas compartilhadas, suportadas pelo sistema operacional.

5.2 Trabalhos Futuros

Várias são as frentes ainda a serem exploradas no **GeoLab**. A principal, acreditamos, é a ampliação do conjunto de bibliotecas existentes, o que proporcinará reais contribuições à comunidade geradora de algoritmos para os problemas tratados em Geometria Computacional.

Do ponto de vista de expansão do próprio ambiente, os seguintes aspectos deverão merecer maior atenção no futuro:

- Composição de Algoritmos
- Bibliotecas de Estruturas de Dados e Algoritmos de Uso Geral
- Geradores de Entrada
- Tolerância a Falhas e Coleta de Lixo
- Geometria Espacial, Projetiva e Suporte para Diferentes Métricas

5.2.1 Composição de Algoritmos

O mecanismo para construção de macros presente atualmente no **GeoLab** provou ser promissor, no sentido de permitir o agrupamento de algoritmos básicos para a construção de algoritmos cada vez mais complexos.

O próximo passo a ser dado diz respeito à geração automática de código para "algoritmos compostos", propiciando uma ferramenta ímpar na programação de algoritmos geométricos. O desafio para a realização de tal tarefa é a especificação de como serão introduzidas estruturas condicionais e repetitivas em composições.

5.2.2 Bibliotecas de Estruturas de Dados e Algoritmos de Uso Geral

A versão 2.0 de C++ distribuída pela Sun Microsystems ainda não implementa o recurso de *templates*, que permite a construção de tipos polimórficos diretamente em C++, sem ter que utilizar truques através de preprocessadores como cpp. A nova versão já anunciada pelo grupo de Bjarne Stroustrup (e que já existe no mercado) terá o recurso de *templates* e será então uma boa investida iniciar a construção de uma biblioteca de algoritmos e estruturas de dados de uso geral que então poderá passar a ser utilizada no **GeoLab**, eventualmente substituindo as bibliotecas de domínio público atualmente utilizadas (e.g., LEDA).

5.2.3 Geradores de Entrada

Os geradores de entrada presentes atualmente no **GeoLab** facilitam o teste de uma grande quantidade de algoritmos geométricos. Outros geradores, adicionados como algoritmos externos, para geração de objetos como segmentos, círculos, virão a facilitar ainda mais o trabalho de teste de algoritmos geométricos.

5.2.4 Tolerância a Falhas e Coleta de Lixo

Suporte para tornar o ambiente imune a erros fatais ocorridos a nível de algoritmos externos deverá também ser abordado no futuro, à medida que novos algoritmos forem introduzidos e se ampliar as ferramentas para depuração existentes. Um maior controle da memória feito pelo **GeoLab** o tornaria mais robusto, especialmente no caso de algoritmos externos mal construídos.

Técnicas que facilitem a produção de algoritmos geométricos numéricamente consistentes e robustos como o SoS deverão também merecer maior atenção [Sch91, EM90].

5.2.5 Geometria Espacial, Projetiva e Suporte para Diferentes Métricas

O suporte geométrico do ambiente, assim como os algoritmos atualmente implementados referem-se em sua maioria a problemas planares, onde concentram-se a maioria dos problemas fundamentais em Geometria Computacional.

Estendendo-se o sistema de visualização utilizado pelo **GeoLab** (*World*), pode-se criar suporte para a manipulação de objetos não planares, o que possibilitará que novos problemas sejam estudados através do ambiente.

Geometria projetiva e suporte para a utilização de diferentes métricas deverão ser considerados em futuras extensões do **GeoLab**.

GeoLab – Guia de Referência

Os apêndices que se seguem introduzem, em termos práticos, os modos de utilização do ambiente e de suas ferramentas. A apêndice A discute os diversos componentes da interface gráfica do ambiente, como ferramentas e modos de operação. Neste apêndice são descritos também os mecanismos para animação e composição de algoritmos (este último em uma versão experimental).

O apêndice B introduz os conceitos e técnicas de programação associados ao **GeoLab**. São apresentadas as ferramentas geométricas (e.g., estruturas geométricas, primitivas e protocolos de comunicação com entidades geométricas) e os recursos de integração e produção de bibliotecas de algoritmos geométricos, juntamente com exemplos relacionados ao atual conjunto de algoritmos geométricos implementados.

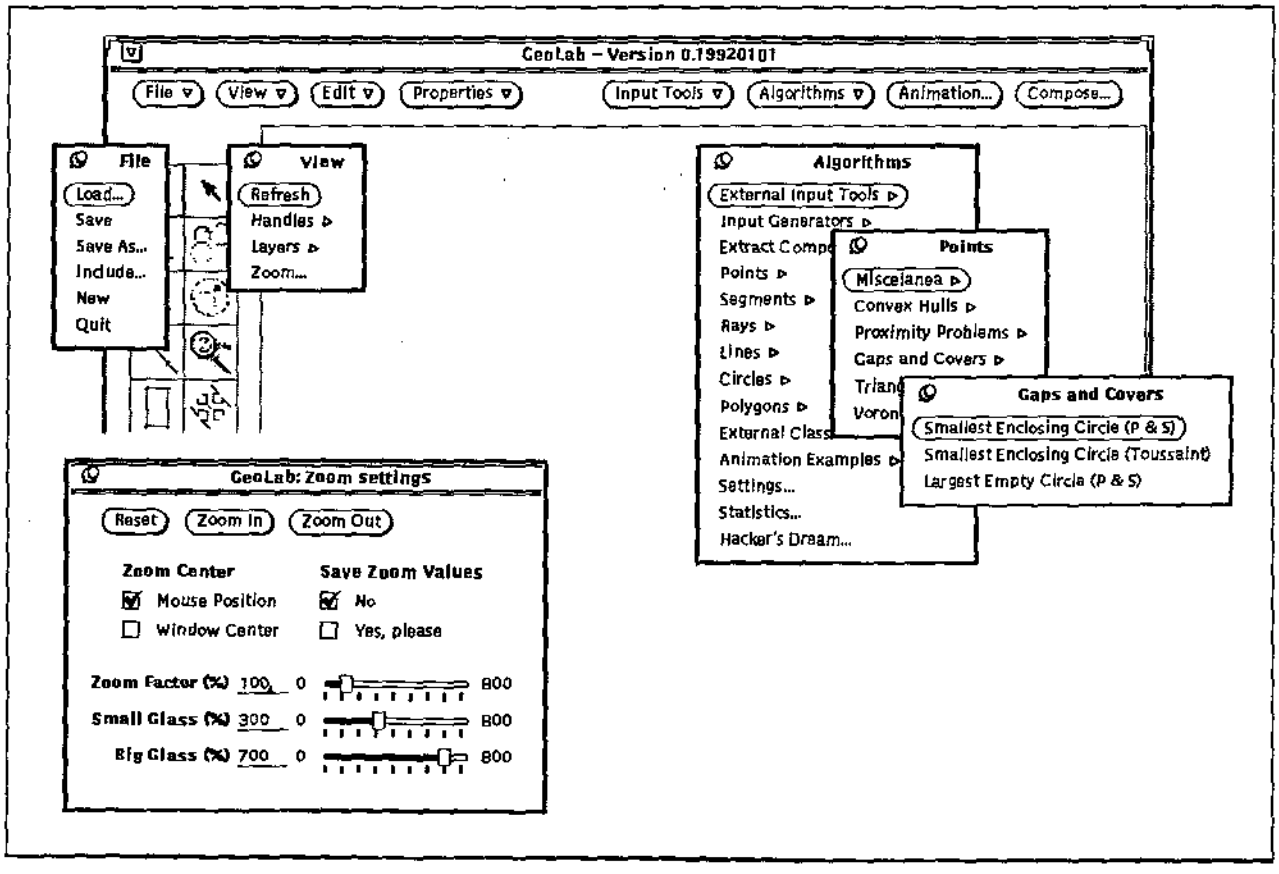

Figura 5.1: **GeoLab** – Um ambiente de desenvolvimento de algoritmos em Geometria Computacional

Apêndice A

Interface Gráfica

A interação com o **GeoLab** dá-se em dois níveis: manipulação gráfica e programação. A nível de manipulação gráfica, dispõe-se de um conjunto relativamente grande de ferramentas pré-definidas (i.e., disponíveis independentemente de algoritmos e objetos geométricos) para facilitar aspectos importantes como a construção de modelos geométricos, geração automática de entradas, animação e composição de algoritmos. Complementam estas ferramentas vários outros recursos, comuns à maioria das aplicações, tais como armazenamento de objetos em memória secundária, manipulação de cores e atributos de desenho, etc.

A.1 Editor e Modos de Operação

Modelos geométricos são construídos através de um editor gráfico, através do qual interage-se com uma base de dados para a representação de objetos geométricos. A interação com o editor (veja figura A.1) se dá através dos dispositivos de entrada do computador (*mouse* e teclado) e este responde visualmente a todas as ações sobre estes dispositivos.

Seguindo-se as convenções do sistema de janelas utilizado (i.e., OpenWindows), os botões do *mouse* (três ao todo) tem funções bem definidas:

Botão da Esquerda – Atua em todos os modos de operação, sendo o principal na maioria deles.

Botão do Meio – Estende alguns modos de operação.

Botão da Direita – Sempre utilizado para menus de opções, junto tanto ao editor quanto aos botões de acesso às ferramentas.

Além dos três botões, algumas teclas possibilitam variações em comandos que possuem mais de duas alternativas. Ainda seguindo o modelo de interface do ambiente de janelas, as teclas *Ctrl* e *Shift* são usadas como qualificadores. Seqüências de teclas (chamadas *hot-keys*) possibilitam acesso simplificado a algumas ferramentas, permitindo uma maior rapidez no trabalho à medida que aprende-se a utilizar o **GeoLab**.

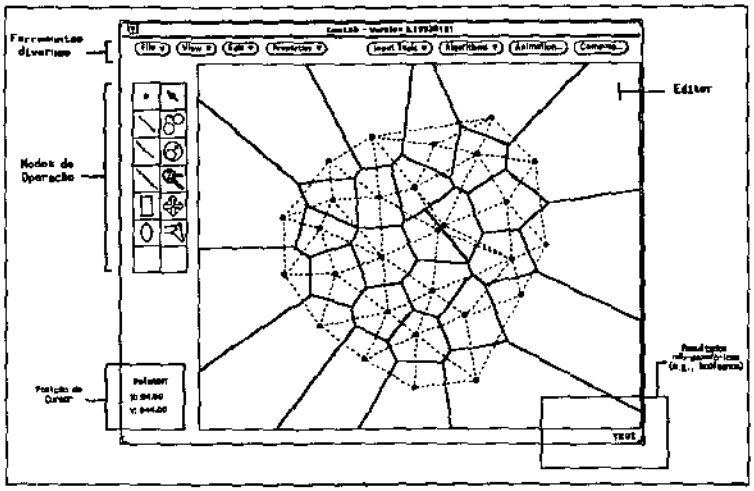

Figura A.1: Visão geral da interface gráfica do **GeoLab** destacando-se o editor gráfico e os modos de operação

Relacionado com o editor gráfico existem diferentes modos de operação que determinam as atitudes a serem executadas pelo editor a dadas ações do usuário. Os modos de operação estão separados em dois tipos: **modos de construção interativa** e **modos funcionais**. Modos de construção interativa compreendem os modos de operação destinados à construção de objetos geométricos, de forma interativa. No painel de controle dos modos de operação (figura A.1), os modos de construção interativa correspondem à coluna da esquerda.

Modos funcionais definem os modos de operação que atuam sobre o editor para a realização de tarefas não construtivas como a seleção de objetos, *Zoom*, e até animação de algoritmos (um dos dois tipos disponíveis). A coluna da direita do painel de controle dos modos de operação corresponde aos modos funcionais.

Conforme observa-se na figura A.1, para os dois modos de operação existem botões "em branco" situados na base das colunas. Estes botões são controlados por mecanismos (descritos no apêndice B) que possibilitam a criação e a introdução dinâmica de novos modos de criação interativa e de novos modos funcionais.

A.1.1 Desenho de Objetos Geométricos

Selecionando-se um dos modos de criação interativa, toda ação sobre o editor gráfico através do *mouse* resulta no início do processo de criação de algum objeto geométrico. Para objetos simples (i.e., pontos, segmentos, raios, círculos, etc.) a ação de construção restringe-se a pressionar um botão construtivo (da esquerda ou do meio), arrastar o dispositivo e então liberar o botão pressionado. Outros objetos geométricos adicionados através de novos modos de criação interativa podem ou não seguir este padrão, de acordo com a complexidade do objeto em questão.

As teclas de qualificação (e.g., *Shift*) podem ser utilizadas durante a tarefa para restringir os movimentos do cursor a uma dada direção (i.e., horizontal ou vertical), permitindo a fácil criação de objetos com propriedades especiais tais como pontos colineares, segmentos verticais ou horizontais, quadrados (restrição sobre retângulos) ou círculos (restrição sobre elipses).

Como mencionamos, a coluna da esquerda do painel de controle dos modos de operação contém os modos de criação interativa, sendo que existe a possibilidade de introduzir-se novos modos de forma dinâmica. Dentre os modos já existentes encontram-se aqueles sobre os quais concentram-se a grande maioria dos algoritmos fundamentais desenvolvidos em Geometria Computacional. A tabela A.1 descreve quais são estes objetos e quais são as variações possíveis para sua construção em função de diferentes combinações de botões do *mouse* e teclas pressionadas.

	Botão do *mouse*		Teclado
	Esquerda	Meio	Tecla *Shift* pressionada
⊙	Pontos	Idem	Colineares
	Segmentos	Idem	Paralelos aos eixos
	Raios	Idem	Paralelos aos eixos
	Retas	Idem	Paralelos aos eixos
	Retângulos	Retângulos centralizados no ponto inicial	Lados iguais (Quadrados)
	Elipses	Elipses centralizadas no ponto inicial	Focos coincidentes (Círculos)

Tabela A.1: Variações possíveis para os modos de criação interativa de objetos básicos

A.1.2 Seleção de Objetos

A aplicação de ferramentas sobre os objetos geométricos existentes no editor gráfico baseia-se no conceito de **objeto selecionado**. A seleção de objetos realiza-se através de um dos modos funcionais primitivos e possui variações de acordo com os botões e teclas pressionados, analogamente aos modos de criação interativa.

Seleciona-se um particular objeto pressionando-se o botão da esquerda do *mouse* sobre o objeto (em qualquer parte dele). Vários objetos podem ser selecionados ao mesmo tempo através de um retângulo de seleção (criado da mesma forma que objetos do tipo retângulo) que envolva completamente os objetos em questão.

Identifica-se quais são os objetos que estão selecionados em um determinado momento através das alças[1] dos objetos, que tornam-se preenchidas quando o objeto está selecionado (figura A.2). Se a apresentação das alças estiver desabilitada (veja seção A.4) não há como distinguir visualmente quais são os objetos selecionados.

A tabela A.2 ilustra as diversas variações possíveis no modo de seleção.

[1] *Handles.*

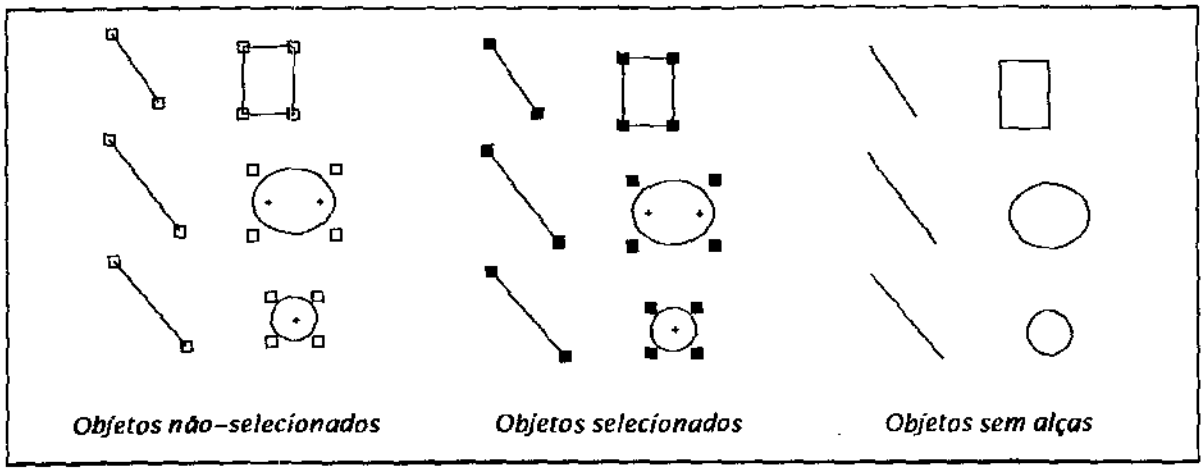

Figura A.2: Objetos selecionados diferenciam-se dos outros através das alças

Botão do *mouse*		Teclado
Esquerda	Meio	Botão do Meio + tecla *Shift*
Seleciona um objeto ou um grupo	Estende seleção	Inverte status de seleção dos objetos

Tabela A.2: Variações possíveis no modo de seleção de objetos

A.1.3 Translação e Cópia de Objetos

Em virtude do estabelecimento de pré-requisitos que devem ser atendidos por todos os objetos geométricos (seção B.2), tornou-se possível incluir entre os modos funcionais básicos opções variadas como translação, cópia e redimensionamento de objetos. Translação ou cópia de objetos geométricos constituem um dos modos funcionais disponíveis (veja tabela A.3).

A cópia ou a translação de objetos pode ser feita de forma individual (i.e., um objeto por vez) ou em grupo. Caso aplique-se sobre um objeto não selecionado, então a translação (ou cópia) atua unicamente neste objeto, ao passo que se aplicado sobre um objeto selecionado, a translação (ou cópia) é realizada sobre **todos** os objetos selecionados no momento. Ao contrário do modo de redimensionamento de objetos (a seguir), qualquer parte do objeto (i.e., não só as alças) pode ser utilizada no processo.

A.1.4 Redimensionamento de Objetos

Uma vez finalizada a construção de um objeto geométrico, a única forma de alterar sua geometria (além, é claro, de destruir e criar novamente o objeto) é alterar seus parâmetros através do modo de

Botão do *mouse*		Teclado
Esquerda	Meio	Tecla *Shift* pressionada
Translada	Copia	Restringe movimentos (horizontal ou vertical)

Tabela A.3: Variações possíveis no modo de translação e cópia de objetos

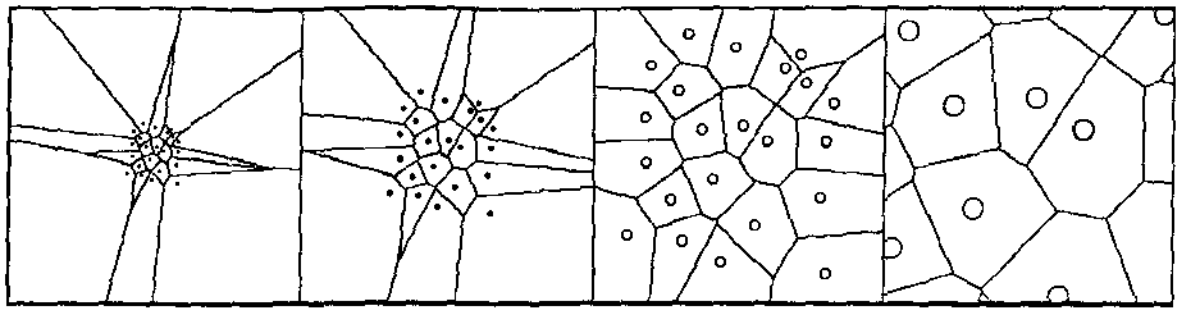

Figura A.3: Diferentes níveis de aproximação de objetos obtidos através do modo de *Zoom*

Combinações de botões de *mouse* e teclado			
Estações Coloridas		Estações Monocromáticas	
Esquerda/(+*Shift*)	Meio/(+*Shift*)	Esquerda/(+*Shift*)	Meio/(+*Shift*)
Amplia/Sem efeito	Reduz/Sem efeito	Amplia/Lente Pequena	Reduz/Lente Grande

Tabela A.4: Variações possíveis no modo de *Zoom*

redimensionamento. Quando acionado, o modo de redimensionamento permite que seja selecionada uma alça de um objeto que, ao ser movimentada, provoca alterações na geometria do objeto.

O processo de redimensionamento de objetos é bastante simplificado, não sendo diferenciadas as operações sobre objetos selecionados e objetos não-selecionados. Em ambos os casos a ação afeta apenas o objeto alvo.

Da mesma forma que a translação e a cópia de objetos, as teclas de restrição de movimento podem ser utilizadas para auxiliar na manutenção de propriedades especiais tais como colinearidade. Os botões do *mouse* (esquerda e meio) assumem as mesmas características dos modos de criação interativa para o objeto sendo redimensionado.

A.1.5 *Zoom*

Permite-se que o usuário determine a "proximidade" dos objetos existentes no editor ao plano virtual de visualização (tela). O efeito de alterar este tipo de propriedade é equivalente ao modo de *Zoom* encontrado em vários aplicativos gráficos e torna mais simples e agradável inspecionar os modelos geométricos sendo trabalhados em vários níveis de detalhes (figura A.3).

O modo funcional de *Zoom* diferencia-se dos demais modos funcionais pois não se aplica diretamente aos objetos existentes no editor. Ao contrário, o modo de *Zoom* atua sobre o sistema de coordenadas, alterando seus parâmetros, e provocando o efeito de ampliação ou redução dos objetos. A geometria dos objetos não é alterada pelo modo de *Zoom*.

A tabela A.4 descreve as diferentes formas de acionamento do modo de *Zoom*.

Além destes modos, o **GeoLab** dispõe de um painel de controle para o acionamento indireto e a configuração do modo de *Zoom*, que é descrito na seção A.3.3.

Botão do *mouse*		Teclado
Esquerda	Meio	Tecla *Shift* pressionada
Movimenta o plano	—	Restringe movimentos (horizontal ou vertical)

Tabela A.5: Variações possíveis no modo de *Scroll*

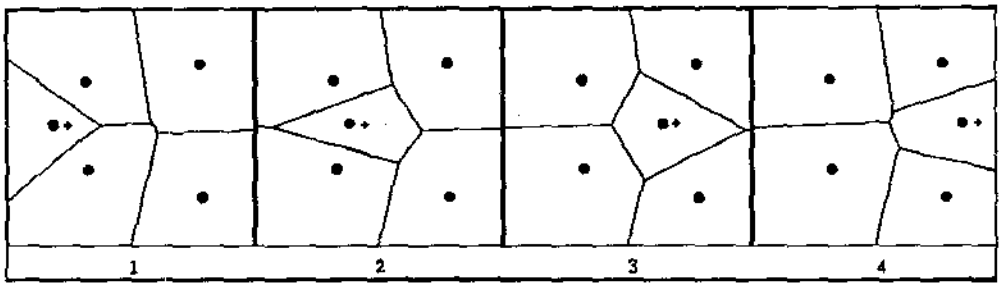

Figura A.4: Quadros de animação via movimentação dinâmica do algoritmo que constrói o diagrama de Voronoi de um conjunto de pontos

A.1.6 *Scroll*

Pode-se alterar a posição dos objetos com relação à janela do editor através do modo funcional de *Scroll*. Este modo permite que se considere o plano de desenho como sendo ilimitado, já que não há restrições (além, é claro, do poder de representação de valores da máquina).

Para deslocar o plano de visualização seleciona-se o modo de *Scroll* e arrasta-se o cursor sobre o plano. A tabela A.5 descreve as possíveis formas de efetuar-se a movimentação do plano de visualização.

A.1.7 Animação por Movimentação Dinâmica

Animação de algoritmos é um dos recursos mais importantes existentes no (e inerentes ao) **GeoLab**. O fato de criarmos e manipularmos objetos geométricos (com natural representação pictórica) permite a construção de processos de visualização do funcionamento de algoritmos auxiliando, entre outras coisas, o entendimento dos mesmos.

Existem duas formas de animação de algoritmos disponíveis. A primeira delas, aplicável a qualquer algoritmo, é obtida através de um dos modos funcionais primitivos do **GeoLab** (a segunda é abordada mais adiante nas seções A.3.4 e B.1.2).

A animação através do modo funcional denominado "dynamic-move" consiste da execução repetitiva do último algoritmo solicitado (veja seção A.6.1 para informações de como solicitar um algoritmo) sobre um conjunto de entrada sofrendo constantes modificações. Trata-se de um método através do qual permite-se visualizar resultados intermediários de um processamento que, quando aplicado em pequenos conjuntos de entrada, proporciona uma visão animada de seu resultado (figura A.4).

O funcionamento desta ferramenta é bastante simples: seleciona-se o conjunto de entrada e ativa-se o modo funcional de movimentação dinâmica. Movimenta-se (ou redimensiona-se) então um dos objetos que constituem a entrada através dos botões do *mouse* e, durante este processo, o algoritmo

Combinações de botões de *mouse* e teclado		
Esquerda/(+*Ctrl*)	Meio/(+*Ctrl*)	Tecla *Shift*) pressionada
 Translada/Redimensiona	Copia/Redimensiona[2]	Restringe movimentos

Tabela A.6: Variações possíveis na manipulação de conjuntos de entrada para o modo de movimentação dinâmica

será executado seguidamente, e o ambiente se encarregará de mostrar os resultados que forem obtidos. Para possibilitar uma grande quantidade de variações na forma de alterar o conjunto de entrada utiliza-se os botões do *mouse* (esquerda e meio) juntamente com as teclas auxiliares (*Ctrl* e *Shift*). A tabela A.6 descreve as variações possíveis.

O modo funcional de animação por dinamização dinâmica pode ainda tirar grande vantagem de algoritmos dinâmicos que suportem inserção e deleção de elementos eficientemente.

A.1.8 Modos de Construção Interativa e Modos Funcionais Externos

Como mencionado anteriormente, o **GeoLab** permite a inclusão de novos modos funcionais através de botões polivalentes existentes nas colunas que definem modos de operação. Conforme descrito na seção B.3, que trata dos aspectos de programação destas ferramentas, tais extensões comportam-se, em linhas gerais, da mesma forma que os modos de operação primitivos não exigindo, portanto, maiores cuidados com sua utilização.

A introdução destas extensões, do ponto de vista do usuário final, é conseguida através do acionamento de algoritmos especiais (seção A.6.1) que, por sua vez, instalam os novos modos no ambiente. Estes, quando acionados, passam a receber o fluxo de entrada resultante das ações do usuário e então realizam as tarefas que estiverem sendo solicitadas.

Até o presente momento, foram criados alguns modos de criação interativa para possibilitar a construção de objetos geométricos que não constam das opções primitivas (e.g., subdivisões planares) e também modos para a introdução de texto e figuras no editor, proporcionando interessantes recursos para a produção de ilustrações (figura A.5).

Modos funcionais externos tem sido usados para a implementação e teste de algoritmos que realizam pré-processamento, e para os quais deseja-se testar o comportamento em situações próximas ao de uso real (e.g., um localizador de pontos em subdivisões planares).

Estes modos encontram-se disponíveis aos usuários que, por sua vez, podem facilmente criar outros que satisfaçam suas próprias necessidades sem, no entanto, ter que recompilar o ambiente para tal (veja seções B.3 e B.3). Isto só se faz possível graças à possibilidade de ligação dinâmica de novos algoritmos e objetos geométricos ao ambiente.

A.1.9 Acesso a Ferramentas via Seqüências de Teclas

Algumas ferramentas disponíveis na interface gráfica, especialmente as mais utilizadas, podem ser acessadas também através de seqüências de teclas (usualmente a tecla *CTRL* seguida de uma letra).

[2]Diferentes objetos podem ter mais de um modo de redimensionamento, tal qual ocorre nos modos de criação interativa.

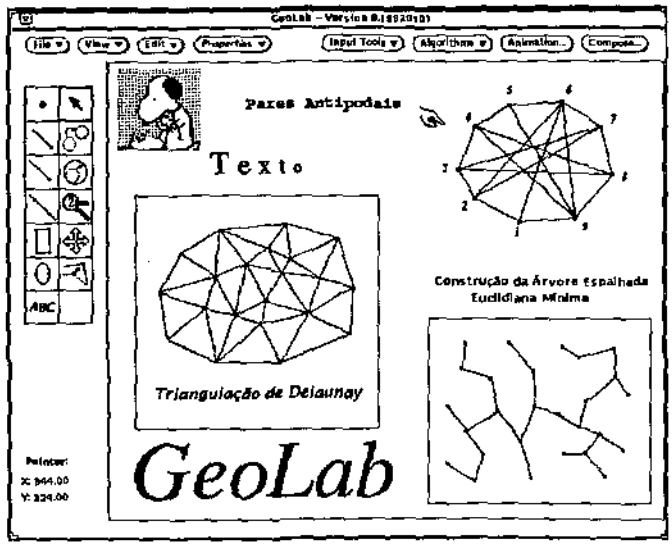

Figura A.5: Exemplo de ilustração produzida no **GeoLab** através de modos de construção interativa incorporados dinamicamente

Este recurso acelera e torna mais agradável o uso de vários recursos do ambiente, principalmente para usuários experientes. As *hot-keys* atuais são mostradas na tabela A.7.

Seqüência de Teclas	Ação associada
`Delete`	Remove objetos selecionados do editor (*Delete*)
`CTRL-U`	Desfaz última operação de remoção (*Undelete*)
`CTRL-A`	Seleciona todos os objetos (*Select All*)
`CTRL-R`	Redesenha o conteúdo do editor (*Refresh*)
`CTRL-H`	Alterna o status de apresentação das alças dos objetos (*Toggle Handles*)
`TAB`	Rotaciona objetos (*Layers - Rotate All*)

Tabela A.7: Seqüências de teclas para acesso direto a ferramentas da interface

A.2 Manipulação de Arquivos

Como toda aplicação que manipula dados armazenáveis, o **GeoLab** dispõe de recursos para gravar e ler o conteúdo do editor gráfico em dispositivos de memória secundária.

Além dos recursos comuns, o **GeoLab** conta ainda com um recurso muito útil durante a fase de depuração de algoritmos e mesmo na manutenção do próprio ambiente. Trata-se de uma rotina de interceptação de sinais que filtra a ocorrência de erros fatais de execução (e.g., *segmentation fault*, *bus error*, *illegal instruction*, etc) e, antes de finalizar (de forma um pouco mais graciosa) a execução do ambiente, grava o conteúdo do editor gráfico para que o eventual erro de programação possa ser reproduzido e detectado em execuções futuras, além de garantir que o processo de criação do modelo

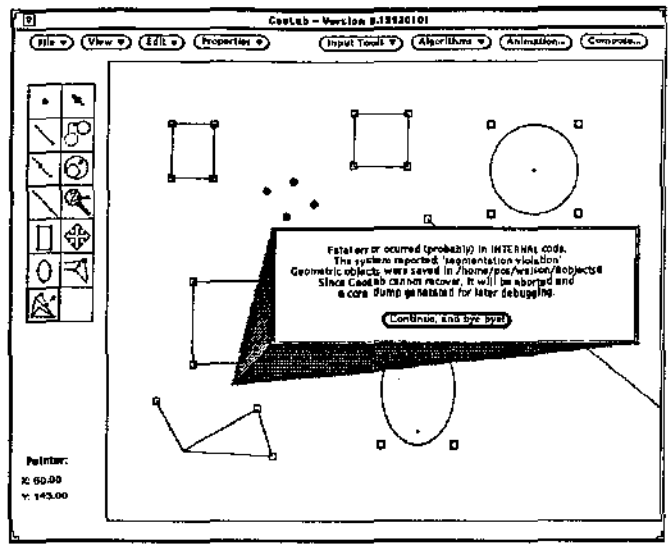

Figura A.6: Janela de aviso registrando a ocorrência de um erro fatal no **GeoLab**

geométrico não se perca acidentalmente.

Os objetos são colocados em um arquivo no diretório raiz do usuário, e podem ser carregados por vias normais. No momento do salvamento é ainda comunicado ao usuário se o erro ocorreu interna ou externamente ao ambiente (algoritmos externos) além, é claro, do tipo de erro ocorrido (figura A.6).

Usualmente, todos os objetos do editor são salvos seqüencialmente em um único arquivo, segundo o formato abaixo:

`ObjID`	*Identificação do Objeto*
`[AlgorithmName ModuleName]`	*Origem do Objeto (objetos não primitivos)*
`ObjName ObjSubtypeID`	*Nome e Subtipo do objeto*
`ObjData`	*Dados do Objeto*
`ObjA1 ObjA2 ObjA3 ObjA4 ObjA5 ObjA6 ObjA7 ObjA8`	*Atributos*

onde:

`ObjA1 = handles(on/off)`	`ObjA2 = selected(on/off)`	`ObjA3 = glued(on/off)`
`ObjA4 = not deletable(on/off)`	`ObjA5 = foreground`	`ObjA6 = background`
`ObjA7 = line width`	`ObjA8 = line style`	

Em alguns casos, como no caso de figuras (i.e., *pixmaps* utilizados em ilustrações) que geralmente ocupam razoável espaço em disco, utiliza-se o nome do arquivo que contém a figura ao invés dos dados da figura propriamente ditos. Isto permite que vários arquivos de objetos do **GeoLab** contenham referências a uma mesma base de dados (no caso, uma figura) sem acarretar excessivo gasto de memória secundária. Sempre que o caso não demandar a duplicação de informações sugere-se proceder desta forma (fala-se aqui da implementação de novos objetos geométricos a serem incorporados futuramente no **GeoLab**).

O formato textual adotado pelo **GeoLab** para o salvamento de seus objetos permite ainda que os arquivos de dados sejam utilizados para verificações quanto a características dos objetos e mesmo editados manualmente, facilitando o criação de ferramentas textuais diversas.

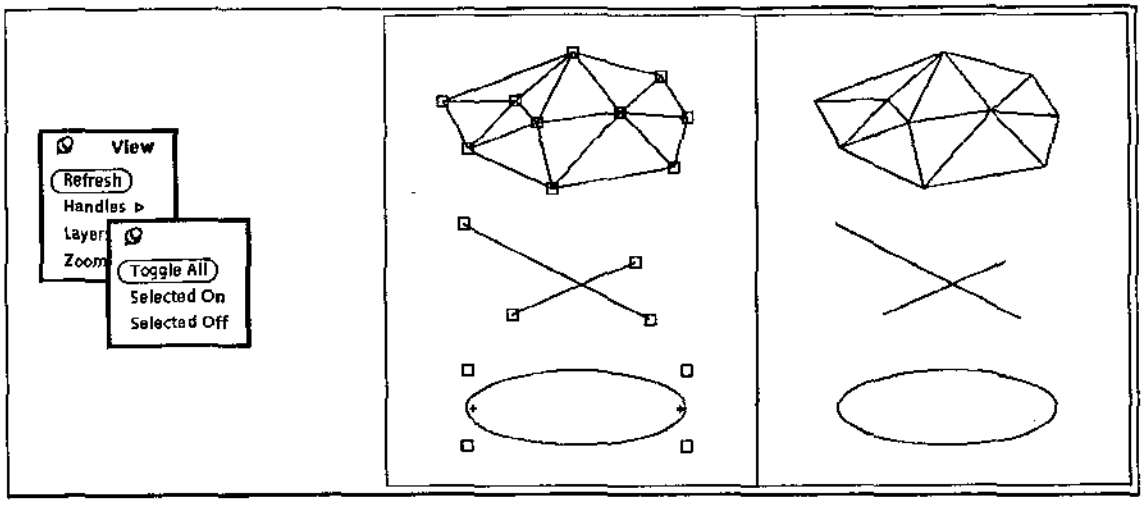

Figura A.7: (a) Menu de opções para a manipulação das alças dos objetos; (b) Exemplo de visões com e sem alças

A.3 Manipulação da Área de Visualização

A forma e a ordem com que são apresentados os objetos geométricos no editor gráfico pode ser configurada de acordo com a necessidade do usuário durante uma seção com o **GeoLab**. Algumas possibilidades atuam sobre o sistema de coordenadas (e.g., *Zoom*), sobre a base de dados de objetos geométricos (e.g., posição relativa entre objetos) ou ainda sobre atributos de controle de desenho de objetos geométricos (controle de apresentação das alças de objetos).

A.3.1 Alças

Conforme mencionado na seção A.1, as alças de um objeto geométrico constituem uma forma de determinar o fato de um objeto estar selecionado, ou então o local a ser "tocado" no processo de redimensionamento do objeto.

No menu de opções disponível (figura A.7 (a)) permite-se habilitar/desabilitar a apresentação das alças de todos os objetos, ou então apenas dos objetos selecionados. Esta é uma facilidade útil quando se deseja visualizar de forma mais "limpa" o conteúdo do editor gráfico (figura A.7 (b)).

A.3.2 Posição Relativa de Objetos

Outra possibilidade de manipulação do conteúdo do editor gráfico diz respeito ao posicionamento relativo entre os objetos geométricos que contém. No caso de objetos sobrepostos, pode-se alterar a ordem de busca determinando que objetos sejam sejam trazidos para a camada superior ou então colocados em camadas sob a dos demais objetos. O menu de opções (*Layers*) dispõe de ferramentas que manipulam todos (rotação geral) ou apenas os objetos selecionados (figura A.8).

A.3.3 Parametrização de *Zoom*

O modo funcional de *Zoom* pode ser configurado através de um painel de controle que permite alterações em seus parâmetros (como taxa de ampliação e redução). Pode-se ainda restaurar o sistema

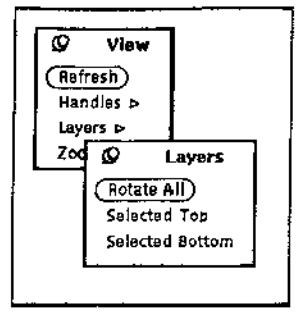

Figura A.8: Menu de opções para alteração da posição relativa entre objetos

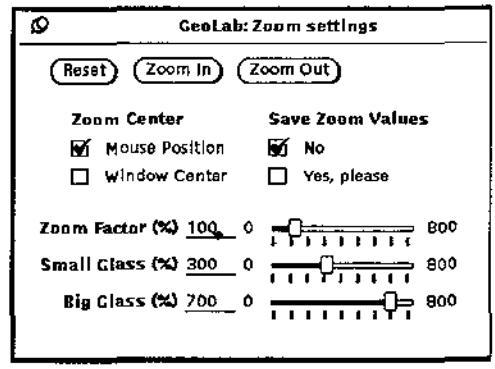

Figura A.9: Painel de controle para as opções do modo de *Zoom*

de coordenadas ao seu estado inicial (*Reset*), armazenar planos de visualização ou então determinar que a ampliação/redução seja relativa à posição do cursor ou centralizada com relação à janela do editor (figura A.9).

A.3.4 Painel de Controle para Animação de Algoritmos

A animação de algoritmos propriamente dita (i.e., animação do funcionamento do algoritmo) é controlada através de um painel que permite o acesso aos parâmetros de uma animação (figura A.10). Dentre estes parâmetros destacam-se:

- **Profundidade de Animação** – Nível de detalhes que deseja-se visualizar durante a animação. Algoritmos geométricos na maioria das vezes são utilizados como blocos de construção de outros algoritmos e uma correta organização de uma animação (feita pelo usuário-programador do **GeoLab**) permite a visualização da animação em diferentes níveis de abstração. A profundidade de uma animação deve ser determinada antes dela ser disparada.

- **Tempo de exposição** – Algoritmos animados possuem marcas de parada (*break points*) que permitem ao ambiente acelerar ou reduzir a velocidade de apresentação de uma animação. O tempo de exposição dos quadros pode ser alterado durante a execução de uma animação. Permite-se ainda executar um algoritmo animado passo a passo.

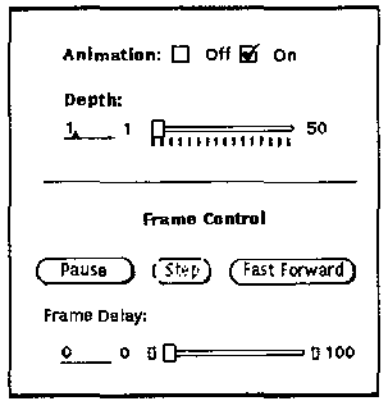

Figura A.10: Painel de controle para animação de algoritmos

A.4 Atributos de Objetos Geométricos

Objetos geométricos podem ser caracterizados, individualmente ou em grupo, da forma que melhor entender o usuário através de um conjunto de atributos editáveis através da interface. Os atributos editáveis de um objeto são:

- Alças – O usuário pode determinar se quer ou não que sejam mostradas (esta possibilidade foi abordada na seção A.3.1).
- Cor e Tipo de Traço.
- Rótulos – O **GeoLab** possui uma capacidade limitada para a introdução de texto associado a objetos geométricos através da utilização de rótulos.
- Objetos Removíveis e Não-Removíveis – Pode-se determinar que um objeto ou um grupo de objetos não sejam removíveis do editor com os comandos de destruição de objetos. Esta capacidade é útil quando se fazem experiências repetidas sobre um mesmo conjunto.
- Objetos Fixos – Objetos podem ser "colados" ao editor, evitando que possam ser movidos ou redimensionados.

A interface gráfica proporciona vários caminhos para o acesso a estes atributos. Pode-se, por exemplo, alterá-los através de um menu de opções ou através de uma janela de configuração (que se abre em resposta a um duplo-clique do botão da esquerda do *mouse* sobre o objeto em questão), mostrados na figura A.11.

Objetos geométricos, segundo seu protocolo (seção B.2.2.1), são capazes de produzir descrições textuais de si mesmos de forma legível para permitir avaliação de valores e eventual depuração de conteúdo (diferentemente do formato de gravação descrito na seção A.2 onde o resultado pode não ser facilmente legível). Para obter tal descrição deve-se selecionar os objetos para os quais se quer uma descrição textual e então pressionar o botão *Dump* na janela de configuração.

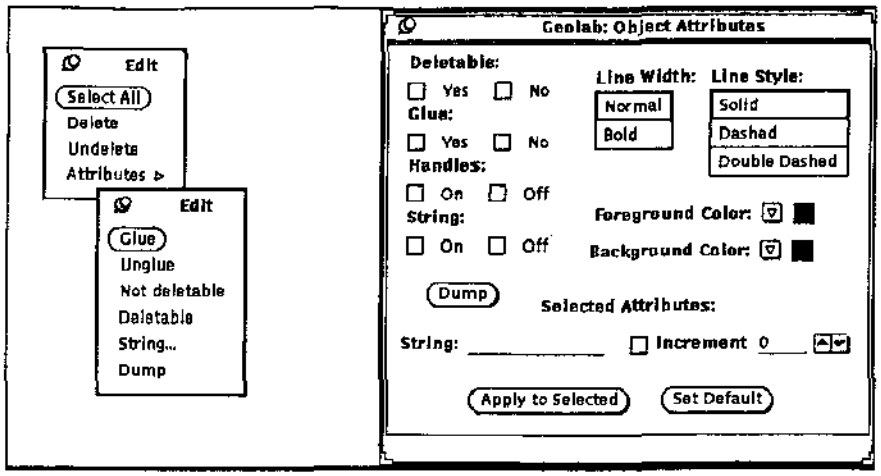

Figura A.11: Menu de opções e janela de configuração para a edição dos atributos de objetos geométricos

A.5 Geradores de Entrada

Entradas automaticamente geradas para algoritmos geométricos podem ser conseguidas através de geradores de entrada primitivos disponíveis no **GeoLab**. Estes geradores fazem parte do conjunto de ferramentas do **GeoLab** e podem ser complementados através da introdução de algoritmos geométricos especiais, cuja função é produzir entradas, nos mesmos moldes de algoritmos geométricos externos apresentados na seção B.1 (veja também seção A.5.4).

Os geradores primitivos, basicamente para pontos e polígonos, são acessíveis através da opção *Input Tools* presente na interface. Todos os geradores são configuráveis em vários aspectos, de forma equivalente à apresentada na seção A.6.2, com a diferença de que nos geradores de entrada é possível determinar o tempo máximo de execução de um gerador (*timeout*), que pode ser útil quando se deseja gerar grande quantidade de objetos dentro de uma limitação de tempo.

A.5.1 Geradores de Pontos Randômicos

A capacidade básica do gerador de pontos randômicos é a geração de pontos distribuídos uniformemente sobre uma determinada região do plano (figura A.12). A ferramenta possibilita a geração de pontos sobre o retângulo definido pela área visível do editor gráfico ou então no interior de polígonos. Neste último caso, deve ser provido ao gerador uma entrada constituída de polígonos selecionados no editor.

A quantidade de pontos e o tempo de execução da ferramenta podem ser controlados através do painel de configuração do gerador de pontos randômicos apresentado na figura A.13.

A.5.2 Cobertura para Objetos Geométricos

Outra possibilidade com relação a pontos é a geração de coberturas randômicas para objetos em geral. Uma cobertura é um conjunto de pontos distribuídos sobre a fronteira de um objeto e, neste caso, esta distribuição é aleatória. A entrada para o gerador de coberturas deve consistir de objetos (qualquer

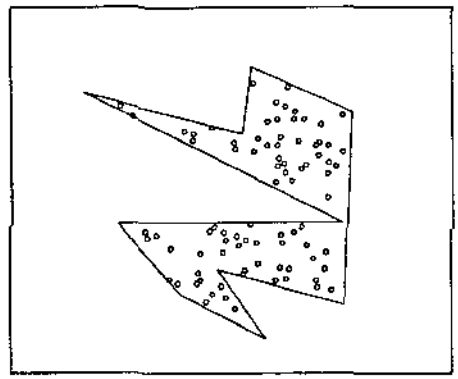

Figura A.12: Pontos aleatoriamente gerados no interior de polígonos através do gerador de pontos randômicos

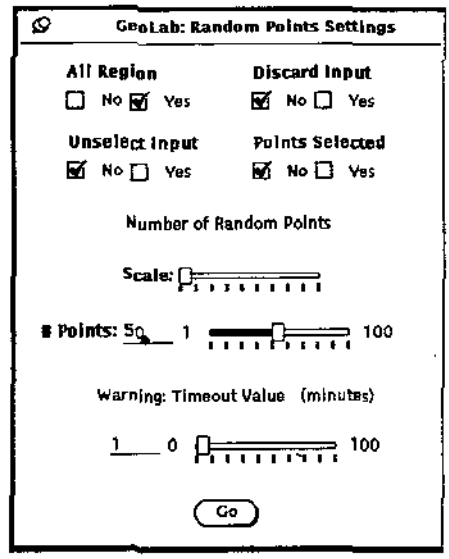

Figura A.13: Painel de configuração dos parâmetros do gerador de pontos randômicos

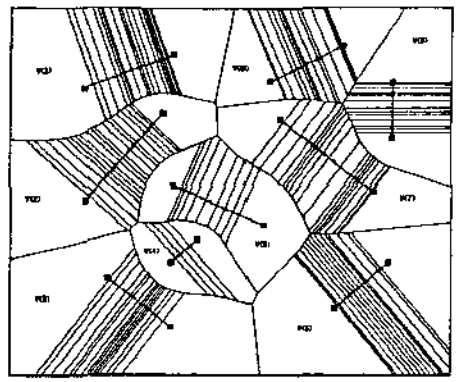

Figura A.14: Diagrama de Voronoi aproximado para um conjunto de segmentos obtido através de coberturas randômicas sobre os segmentos

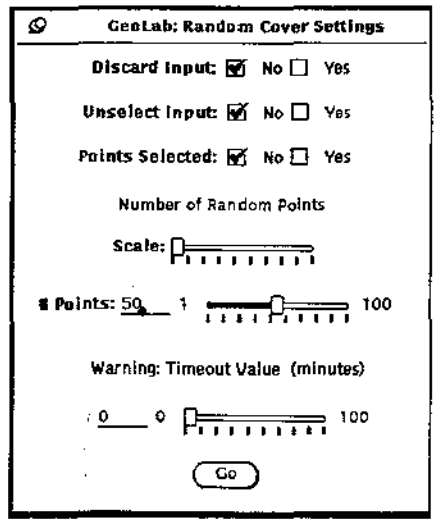

Figura A.15: Painel de configuração dos parâmetros do gerador de coberturas randômicas

objeto pode ser coberto) selecionados no editor. Extensões ao gerador de coberturas randômicas incluirão num futuro próximo a geração de pontos distribuídos equidistantemente.

Esta ferramenta pode ser muito útil para a avaliação de algoritmos com casos extremamente degenerados de entrada (e.g., todos os pontos colineares ou co-circulares). Outra possibilidade é utilizar as coberturas randômicas para a criação de estruturas aproximadas, como é o caso do diagrama de Voronoi de segmentos apresentado na figura A.14.

A quantidade de pontos e o tempo de execução da ferramenta podem ser controlados através do painel de configuração apresentado na figura A.15.

A.5.3 Geradores de Polígonos Randômicos

Além de pontos, os geradores internos compreendem também a geração automática de alguns tipos de polígonos. Tendo como entrada um conjunto de pontos selecionados (vértices), o gerador de polígonos é capaz de construir polígonos genéricos (i.e., a ordem dos vértices é uma permutação simples da entrada), polígonos simples, em forma de estrela (a ordem dos vértices é circular com relação a um ponto interno escolhido aleatoriamente) ou polígonos convexos.

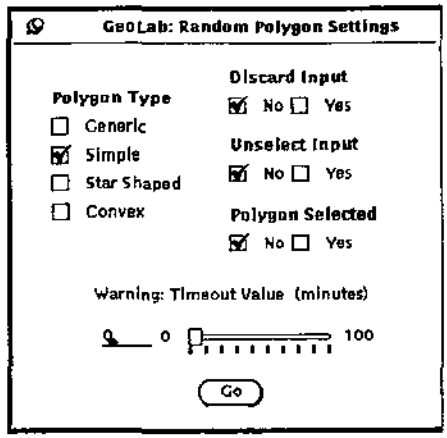

Figura A.16: Painel de configuração dos parâmetros do gerador de polígonos randômicos

A configuração dos parâmetros é feita através do painel apresentado na figura A.16.

A.5.4 Geradores Internos *versus* Geradores Externos

A escolha dos geradores apresentados para fazer parte das ferramentas primitivas do ambiente teve como motivo principal a necessidade destes para testes de uma grande quantidade de algoritmos fundamentais. Mesmo assim, muitos outros algoritmos dependem de outros tipos de entradas e portanto o ambiente deveria de prever a criação de novos geradores.

Para tanto, o ambiente permite que novos geradores de entrada sejam adicionados dinamicamente da mesma forma com que são adicionados algoritmos geométricos genéricos (seção B.1) e portanto garante que as necessidades dos usuários poderão ser supridas à medida que forem aparecendo, e de forma simples. Com os mecanismos de programação criados para a incorporação de algoritmos externos, mesmo a criação de janelas de configuração torna-se simples de ser realizada, garantindo a funcionalidade do gerador e ilustrando que o **GeoLab** é de fato um ambiente desenvolvido com atenção a flexibilidade.

A.6 Algoritmos Geométricos

Algoritmos geométricos são entidades externas ao **GeoLab**, construídos utilizando suas facilidades e convenções e que são incorporados a ele *de forma dinâmica*.

Algoritmos geométricos são organizados em **bibliotecas compartilhadas**[3] construídas utilizando-se facilidades proporcionadas pelo SunOS e que podem ser ligadas a qualquer outra aplicação (i.e., não só o **GeoLab**).

A forma de incorporação destas bibliotecas compartilhadas ao **GeoLab** se dá através de um mecanismo chamado **ligação dinâmica**[4] utilizado pelo **GeoLab** e executado imediatamente após a carga do ambiente.

O usuário determina quais (e em que ordem) são os algoritmos que devem ser considerados e quais são as bibliotecas compartilhadas que devem ser ligadas através de um arquivo de configuração

[3] *Shared Libraries.*
[4] *Dynamic Link.*

```
/* Isto é um comentário */

TITLE "Algorithms"                          /* Título do menu principal */

"Option #1" "link_function_name1" "module_name1"
"Option #2" "link_function_name2" "module_name2"

  MENU "Submenu"                            /* Título de um submenu */
    "Option #3" "link_function_name3" "module_name3"
    "Option #4" "link_function_name4" "module_name4"
  END
```

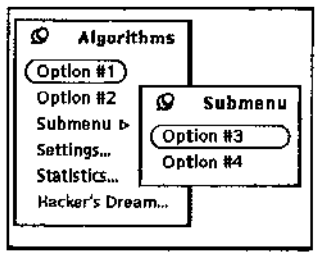

Figura A.17: Arquivo de configuração e o menu de algoritmos correspondente, construído pelo **GeoLab**

chamado `.geolab-menu`. Este arquivo é estruturado em termos de uma pequena linguagem que, entre outras coisas, define também qual será o formato do menu de opções de algoritmos presente no **GeoLab**.

A figura A.17 mostra um arquivo de configuração e o menu de algoritmos associado criado pelo **GeoLab**.

As linhas que definem entradas selecionáveis no menu de algoritmos tem três campos que indicam:

1. O nome da opção no menu de algoritmos;
2. O nome da função de ligação correspondente ao algoritmo desejado; e
3. O nome da biblioteca compartilhada onde se encontra tal função.

A função de ligação (veja seção B.1) define qual o algoritmo a ser considerado dentro da biblioteca compartilhada sendo acessada, e normalmente é descrita nos *header files* das bibliotecas.

O nome da biblioteca (com opcional localização dentro do sistema de arquivos) é utilizado para a carga e ligação dinâmica da biblioteca pelo **GeoLab**.

Usuários podem ter vários arquivos de configuração. Isto permite que se trabalhe em diferentes conjuntos de problema com o **GeoLab** e torna-se um fator de organização no trabalho que é desenvolvido. A localização do arquivo `.geolab-menu` é importante para sua leitura pelo **GeoLab**. O arquivo é inicialmente procurado no diretório corrente (i.e., onde foi invocado o **GeoLab**). Caso não seja encontrado, é então buscado no diretório raiz do usuário. Não encontrado, o **GeoLab** busca no diretório definido na variável de ambiente `GEOLAB_MENU`. Se a busca falhar em todos os casos, o **GeoLab** entra em funcionamento sem o menu de algoritmos.

Outra variável de ambiente que pode ser utilizada para ajudar na construção do arquivo de configuração é a variável `GEOLAB_ALGORITHMS`, que indica a localização das bibliotecas compartilhadas, de tal forma que o nome completo (*full path name*) não tenha que ser inserido no arquivo `.geolab-menu`. O nome completo pode, porém, ser inserido no arquivo de configuração quando se deseja *overwrite* a localização de uma biblioteca padrão.

A.6.1 Seleção de Objetos e Opções Disponíveis

Logo após sua carga, o **GeoLab** lê o arquivo de configuração e monta seu menu de algoritmos. Este menu é acessado pelo botão da direita do *mouse* pressionado sobre o editor gráfico ou então através do botão *Algorithms* disponível na interface. A seleção de algoritmos deste menu segue o padrão do ambiente de janelas utilizado (*OpenLook*).

As opções do menu de algoritmos podem ou não estar disponíveis em um determinado momento (disponibilidade sensível a contexto). O fator principal que determina se uma dada opção está disponível ou não são os objetos selecionados no momento. O ambiente controla, em função da seleção de objetos, quais os algoritmos aplicáveis ao conjunto selecionado, auxiliando o usuário no processo de escolha de algoritmos.

Outro fator que pode tornar uma opção indisponível em um dado momento é o status da animação de algoritmos (seção A.3.4). Caso a opção de animação esteja acionada, então aqueles algoritmos que não possuem código específico para animação tornam-se indisponíveis, independentemente do conjunto de objetos selecionados.

O último algoritmo aplicado sobre um conjunto selecionado torna-se o **algoritmo corrente**, sobre o qual o modo de animação por movimentação dinâmica atuará (seção A.1.7). É importante notar apenas que, se a entrada selecionada para o modo de animação por movimentação dinâmica for incompatível com o algoritmo corrente, então a animação simplesmente não irá acontecer.

A.6.2 Configuração para Algoritmos Externos

O comportamento do editor pode ser configurado no que diz respeito ao tratamento da entrada consumida e da saída produzida por algoritmos geométricos[5] através de um painel de controle invocado através da opção denominada *Settings* situada na base do menu de algoritmos (esta é uma opção fixa, independente do conteúdo do menu que é definido através do arquivo de configuração). É possível, dentre outras coisas, configurar (figura A.18):

- **Descarte da entrada** – A entrada a ser passada para o algoritmo geométrico pode ou não ser descartada através da configuração desta opção.
- **Cancelamento da seleção da entrada** – Outra forma de fazer com que a entrada seja desconsiderada nas próximas chamadas a algoritmos geométricos (sem no entanto ser destruída) é conseguida simplesmente configurando esta opção para fazê-la não selecionada.
- **Seleção da saída** – No caso de vários algoritmos serem aplicados em seqüência (especialmente no caso de composição de algoritmos – vide seção A.7) pode ser interessante que o resultado de um algoritmo seja introduzido no editor já selecionado.

[5] Os geradores de entrada primitivos também são configuráveis da mesma forma, possuindo entretanto painéis de controle próprios e independentes.

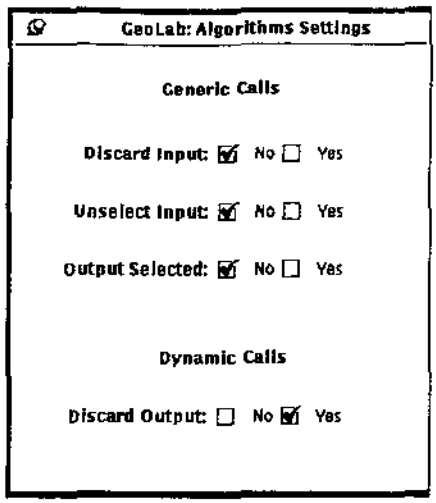

Figura A.18: Janela de configuração para algoritmos geométricos externos

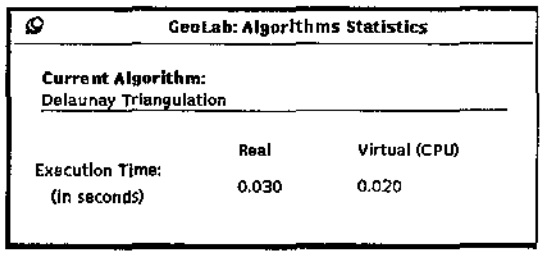

Figura A.19: Janela de monitoramento de tempo de execução de algoritmos geométricos externos

- **Descarte da saída** – No caso do resultado final de um algoritmo dentro do processo de movimentação dinâmica, é possível solicitar ao ambiente que descarte tal resultado. Normalmente, quando o modo de movimentação dinâmica é usado com a exclusiva finalidade de animação este é o comportamento desejado.

A.6.3 Controle do Tempo de Execução de um Algoritmo

A opção *Statistics* presente permanentemente no menu de algoritmos aciona um painel de controle cuja função é ajudar no monitoramento do tempo de execução de algoritmos geométricos externos (figura A.19).

Isto é particularmente interessante na comparação de diferentes algoritmos para um mesmo problema, o que permite uma análise prática das constantes multiplicativas associadas a tais algoritmos e também a análises de caso médio, sempre mais difíceis de considerar teoricamente.

Os tempos avaliados são:

- **Real** – Tempo medido em um relógio contínuo global. Depende fortemente da carga (número de processos concorrendo pela CPU, espaço em disco, etc) da máquina no momento.
- **Virtual** – Tempo ideal de execução, supondo que o processo sendo avaliado fosse o único a executar, sem ser escalonado ou paginado (praticamente independente da carga da máquina).

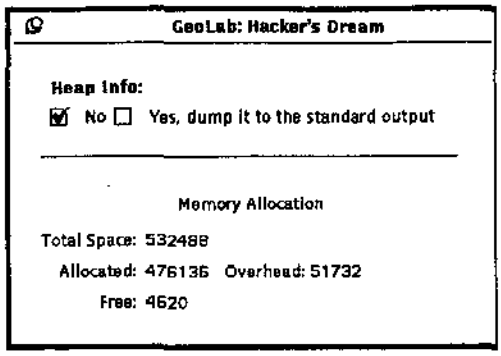

Figura A.20: Janela de monitoração de memória alocada por algoritmos externos

A.6.4 Controle de Memória Alocada por um Algoritmo

Enquanto o monitoramento do tempo de execução pode ser útil na análise de desempenho dos algoritmos, a análise de memória consumida por um algoritmo serve, principalmente, para a depuração de algoritmos.

Normalmente, deixar de desalocar memória em máquinas multiprocessamento (onde geralmente a memória disponível é grande) é um erro de programação que passa desapercebido. Eventualmente ocorrem quedas da aplicação por este motivo, mas dificilmente chega-se rápido à solução do problema. Para ajudar neste sentido, a memória e o estado do *heap* podem ser monitorados através da opção *Hacker's Dream*[6] disponível também no menu de opções de algoritmos (figura A.20).

As informações a respeito da memória alocada dinamicamente por processos é obtida através de recursos dependentes do SunOS descritas na seção 3V das páginas de manual (`malloc`).

A.7 Composição de Algoritmos

A possibilidade de criar "macros" com algoritmos geométricos constitui mais uma facilidade provida pelo ambiente.

Esta ferramenta, ainda em fase experimental, permite que sejam "gravadas" seqüências de chamadas a algoritmos externos (inclusive com os parâmetros relacionados ao tratamento das entradas e saídas – seção A.6.2) para posteriormente serem aplicadas sobre novos conjuntos de objetos geométricos.

O painel de controle da ferramenta, com apenas três botões de controle, permite a realização do processo de composição que consiste em iniciar uma gravação, executar uma seqüência de algoritmos geométricos externos configurando o comportamento das entradas e saídas intermediárias e finalmente finalizar uma gravação. A qualquer momento então, dado um outro conjunto de objetos selecionados, a tecla *Play* dispara a execução da composição (figura A.21).

Por ser uma ferramenta experimental, da qual se espera um grande desenvolvimento ainda (particularmente no que diz respeito à geração automática de código), algumas restrições são feitas:

1. Não se pode animar via movimentação dinâmica uma composição. Embora a animação comum funcione perfeitamente, não há suporte para mudança de cores nas várias fases da animação, o que facilitaria sua visualização.

[6] O nome é uma alusão aos problemas que surgem durante a depuração de um algoritmo geométrico.

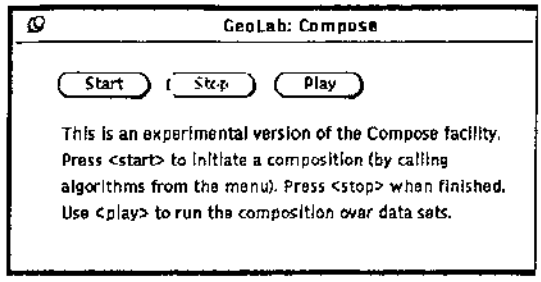

Figura A.21: Painel de controle para composição de algoritmos

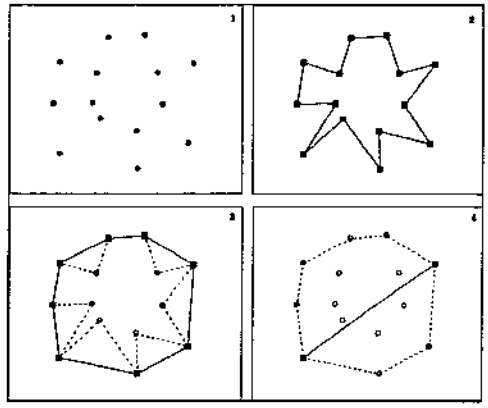

Figura A.22: Passos da execução de uma composição para o problema do cálculo do diâmetro de um conjunto de pontos no plano

2. Só é possível uma composição por vez, e composições não são armazenadas em disco. Uma composição só sobrevive até a criação de outra.

3. Modos de construção interativa, modos funcionais e geradores de entrada externos, por serem todos "tipos especiais" de algoritmos geométricos externos, podem, sem restrições, fazer parte de uma composição. O resultado de uma composição que inclua modos externos não é desastroso, mas também não traz qualquer benefício.

Para ilustrar, considere o exemplo do algoritmo para o cálculo do diâmetro de um conjunto de pontos em tempo $O(n \log n)$. Neste algoritmo, dado um conjunto de n pontos no plano, constrói-se um polígono em forma de estrela cujos vértices são os pontos da entrada em tempo $O(n \log n)$, obtém-se a envoltória convexa dos pontos em tempo linear a partir do polígono em forma de estrela e então, também em tempo linear, calcula-se o diâmetro do polígono convexo (figura A.22).

Apêndice B

Interface de Programação

A principal característica da programação de algoritmos geométricos através do **GeoLab** diz respeito à incorporação destes algoritmos ao ambiente de forma dinâmica. Para tanto foram estabelecidos protocolos a serem utilizados na comunicação entre as entidades dinamicamente inseridas (algoritmos e objetos geométricos) e o ambiente de desenvolvimento.

Do ponto de vista prático, tais protocolos são estabelecidos por um conjunto de métodos definido nas super-classes que dão origem às hierarquias de abstração tanto para objetos quanto para algoritmos geométricos. Para a criação destas hierarquias (e também de todo o ambiente) utiliza-se a linguagem de programação C++ [Str87].

Parte da funcionalidade e da flexibilidade do **GeoLab** deve-se às facilidades específicas da máquina utilizada (estações de trabalho do tipo Sun Sparcstation) assim como de seu sistema operacional (SunOS). Facilidades como *dynamic link* e *shared libraries*, embora também existam em outras plataformas, são de boa qualidade e de fácil acesso no equipamento citado, o que nos possibilitou elaborar consistentemente os mecanismos que caracterizam o **GeoLab**.

Este apêndice aborda de forma prática os aspectos de programação relacionados com o **GeoLab**. Nas seções que se seguem, com exceção da última (B.6), são tratados os aspectos relacionados com algoritmos geométricos, objetos geométricos, modos de construção interativa e modos funcionais. Em todas estas seções são mencionadas as características e os requisitos de programação associados (i.e., programação **utilizando** o **GeoLab**). Na última seção, entretanto, fala-se do próprio **GeoLab** e de sua organização, descrevendo os diversos módulos (em torno de 50) que o compõem e as ferramentas utilizadas, constituindo-se em uma espécie de guia para as eventuais (e inevitáveis) manutenções futuras.

B.1 Algoritmos Geométricos

B.1.1 Hierarquia de Algoritmos e Protocolo de Comunicação

Algoritmos geométricos são objetos que derivam de uma classe chamada `Algorithm` e que encapsulam e abstraem o acesso às funções que implementam algoritmos que consomem e/ou produzem objetos geométricos.

O mapeamento das implementações de algoritmos geométricos em classes possibilita ao ambiente uma comunicação com o algoritmo através de um protocolo e também permite ao programador estruturar seus algoritmos de forma a organizar e economizar código (através dos mecanismos de herança

da programação orientada a objetos). A super-classe `Algorithm` define o protocolo de mensagens a ser provido por subclasses derivadas (tabela B.1).

Método	*Tarefa*
is_able_to_handle	verificar se o algoritmo é aplicável a um dado conjunto de entrada
animation	verificar se o algoritmo possui código para animação
binder	extrair os dados de entrada passados pelo **GeoLab** e organizá-los da forma que for adequada para o algoritmo
unbinder	desalocar memória e realizar tarefas de *clean-up* em modos de construção interativa e modos funcionais externos
instantiation	produzir instâncias de objetos geométricos externos ao **GeoLab** e que são criados pelo algoritmo

Tabela B.1: Protocolo de mensagens para Algoritmos Geométricos

As mensagens deste protocolo são utilizadas tanto a nível de controle de acesso a algoritmos (mensagens `is_able_to_handle` e `animation`) quanto de transmissão e coleta de dados processados (`binder`). Além destas, existe ainda a mensagem `instantiation` cuja finalidade é dar suporte à criação de novos tipos de objetos geométricos (veja seção B.2).

Em termos gerais, implementar um algoritmo através do **GeoLab** segue os seguintes passos:

1. Implementar o algoritmo propriamente dito, utilizando-se para isso dos objetos geométricos (e respectivas primitivas) e estruturas providas pelo **GeoLab** (descritos nas seções B.1.4 e B.2), assim como bibliotecas de algoritmos e estruturas de dados provenientes de outras fontes (e.g., LEDA [MN89]).

2. Mapear o algoritmo implementado em função da classe `Algorithm`, provendo implementação para o protocolo de comunicação através do qual o **GeoLab** irá se comunicar com o algoritmo.

3. Produzir uma função de ligação cuja função é retornar uma instância do algoritmo mapeado, que servirá como *handle* para o **GeoLab** referenciar o algoritmo.

4. Gerar um arquivo de dependências para então construir um *Makefile* que se encarregará de executar os passos necessários para a geração de bibliotecas compartilhadas.

5. Introduzir uma linha no arquivo de configuração (`.geolab-menu`) para que o **GeoLab** faça a ligação do novo algoritmo, e torne-o disponível em seu menu de opções.

6. Incorporar código extra para animação do algoritmo utilizando-se as facilidades do **GeoLab**[1].

B.1.2 Aspectos de Programação de Animações de Algoritmos

Conforme mencionado na seção anterior, a animação *per se* de um algoritmo geométrico é obtida através do acréscimo de código ao algoritmo. O ambiente dispõe de uma série de ferramentas, assim como de uma biblioteca para facilitar o processo de animação (em desenvolvimento), que simplificam

[1]Este passo na verdade pode, e muitas vezes deve, ser realizado concorrentemente com o primeiro, tendo sido aqui discriminado para uma maior simplicidade na explanação.

a produção deste código adicional, assim como induzem (caso as recomendações sejam seguidas) à produção de bibliotecas passíveis de serem utilizadas em outros projetos.

Basicamente, o ambiente dispõe de duas macros que realizam o controle do código inserido entre elas e determinam se tal código deve ou não ser executado em função da profundidade de animação corrente (`animation_depth`) e da profundidade de referência configurada pelo usuário do algoritmo através da interface gráfica do ambiente (`reference_depth`). Estas macros são: `ANIMATION_BEGIN` e `ANIMATION_END`.

Além destas, existem duas instruções que podem ser colocadas nas "ilhas" de animação para que o ambiente controle pontos de parada (utilizado nas execuções passo-a-passo) e também no controle da velocidade da apresentação da animação, que são `frame_stop_mark` e `frame_delay_mark`.

Os parâmetros requeridos para a utilização das facilidades mencionadas fazem parte do protocolo de mensagens seguido pelo ambiente. Estes parâmetros (*handle* do sistema de coordenadas, profundidade de animação e profundidade de referência) podem ser livremente manipulados pelo programador (sem efeito externo, já que são passados por valor) e tal manipulação proporciona a organização de uma animação em níveis (i.e., o programador incrementa a profundidade de animação quando utiliza um algoritmo de "nível mais baixo" para alguma tarefa). Para que as bibliotecas possam ser utilizadas em outras aplicações (onde naturalmente o código para animação não tem utilidade nem suporte), o primeiro cuidado a ser tomado pelo usuário-programador é inicializar automaticamente estes parâmetros (através de recursos de C++) para que nestes casos tais parâmetros possam ser desconsiderados.

Para o caso de um algoritmo vir a ser utilizado em outras aplicações através de seus fontes (i.e., escolhe-se compilar o algoritmo e incorporá-lo estaticamente a outras aplicações) recomenda-se também a utilização de compilação condicional cercando-se código específico para animação, tal que este seja automaticamente expurgado.

A seção seguinte encarrega-se de exemplificar os casos aqui mencionados.

B.1.3 Exemplo Prático

O exemplo apresentado a seguir demonstra o processo de produção de um algoritmo para o cálculo do par mais próximo de pontos em um conjunto de pontos no plano[2].

1. Implementação do algoritmo propriamente dito. Note que os tipos `Point2D` e `Segment2D` são objetos geométricos definidos pelo **GeoLab**, enquanto o tipo polimórfico `list` é definido por uma biblioteca chamada LEDA, assim como a macro `forall`.

```
Segment2D_ptr closest_pair_bf(list(Point2D_ptr) &points){

    // Entrada: uma lista de apontadores para pontos (Point2D)
    // Saida  : um apontador para um segmento criado que une os dois
    //          pontos mais próximos

    list_item it_p, it_cl;
    Segment2D_ptr res = 0;

    if (points.size() > 1) {
      Point2D_ptr p1 = 0, p2 = 0;
      double aux, menor = MAXDOUBLE;
      forall_list_items(it_p, points) {
        forall_list_items(it_cl, points) {
          if (it_p ≠ it_cl &&
              (aux = points[it_p]→e_distance_sqr(*points[it_cl])) < menor) {
            p1 = points[it_p]; p2 = points[it_cl];
            menor = aux;
          }
        }
      }
      res = new Segment2D(*p1, *p2);
    }

    return res;
}
```

[2]Escolheu-se este problema simples para ressaltar o processo de produção de algoritmos para o **GeoLab**, e não a complexidade de implementação de algoritmos geométricos.

2. Mapeamento do algoritmo implementado na classe `Algorithm` definida pelo **GeoLab**. O protocolo de comunicação entre ambiente e algoritmo é aqui implementado. Os parâmetros `World`, `animation_depth` e `reference_depth` são utilizados na produção de animações. Sua inicialização automática possibilita que sejam ignorados.

```
class ClosestPairBF : public Algorithm {
    bool animation() { return true; }

    bool is_able_to_handle(const SelectedObjects &SO,
                           const GraphicObjects &GO) {
      list_item it;

      forall(it, SO) {
        if (GO[it]→type() ≠ Point2D_t) {
          SO.init_iterator();
          return false;
        }
      }

      return true;
    }

    ResultObjects binder(const SelectedObjects &SO, const GraphicObjects &GO,
                         World *w = 0,
                         int animation_depth = 0, int reference_depth = 0) {
      ResultObjects result;
      Segment2D_ptr cpair;

      if (!SO.empty()) {
        if ((cpair = closest_pair_bf(extract_points(SO, GO))) ≠ 0)
          result.push(cpair);
      }

      return result;
    }
};
```

3. Incorporação de *include files* e produção da função de ligação, cuja tarefa é devolver instâncias do algoritmo mapeado. O código mostrado a seguir pressupõe que os exemplos mostrados nos passos anteriores coexistem em um mesmo arquivo.

```
#include "External.h"                    // obrigatório a todo módulo externo
#include "Algorithm.h"

#include "basic.h"                       //
#include "List.h"                        // headers do LEDA

#include "Point2D.h"                     //
#include "Segment2D.h"                   // objetos manipulados


// Abaixo é produzido um tipo "lista de pontos" segundo a sintaxe apresentada
// no manual do LEDA, acrescido de código de controle quanto a múltiplas
// definições.

// A linha "declare(list,Point2D)" produz o tipo "lista de pontos".
// Os "ifdef's" que circundam o "declare" tem por função evitar que
// seja feito mais de uma vez, o que causaria erro de compilação.

// A convenção e' a seguinte:
//
// declare(tad,tipo) -> _tipo_tad (nos ifdef's)

#ifndef _Point2D_ptr_list
#define _Point2D_ptr_list
declare(list,Point2D_ptr)
#endif ~_Point2D_ptr_list

...

Algorithm* closest_pair_bf_link() {
    return new ClosestPairBF();
}
```

4. Produção do *Makefile* para a geração de uma *shared library* a ser ligada dinamicamente ao **GeoLab**. Para o exemplo, um possível *Makefile* seria o apresentado a seguir (veja também seção B.5).

```
# Header para Makefile de módulos ligados dinamicamente ao GeoLab
#
# Por:
#     Welson R. Jacometti
#

DOBJ =
CFLAGS = -I$(GUIDEHOME)/include -I$(OPENWINHOME)/include -pic
LDFLAGS = -L$(GUIDEHOME)/lib -L$(OPENWINHOME)/lib \
          -lguide -lguidexv -lxview -lolgx -lX -lm -ll -ldl
CC = CC

OBJS= \
                $(DOBJ)proximity.o

$(DOBJ)proximity.so: $(OBJS)
        ld $(LDFLAGS) -assert pure-text -o $(DOBJ)proximity.so $(OBJS)

$(DOBJ)proximity.o: proximity.C \
                            Algorithm.h \
                            Coords2D.h \
                            External.h \
                            ExternalObj_id.h \
                            List.h \
                            Obj.h \
                            Point2D.h \
                            Segment2D.h \
                            String.h \
                            World.h \
                            basic.h \
                            global.h \
                            handle.h \
                            proximity.h
        $(CC) $(CFLAGS) -c proximity.C -o $(DOBJ)proximity.o
```

5. Introdução da linha descritiva do novo algoritmo no arquivo de configuração `.geolab-menu` (veja também seção B.4).

```
TITLE "Algorithms"

MENU "Points"
  MENU "Convex Hulls"
       "Jarvis' March" "jarvis_march" "ConvexHull.so"
       "Graham's Scan" "graham_scan" "ConvexHull.so"
  END
  MENU "Proximity Problems"
       "Closest Pair" "closest_pair_bf" "proximity.so"          <=
  END
END
```

6. Incorporação de código específico para animação. A varia'vel `_GEOLAB_ANIMATION` é utilizada para compilação condicional, devendo ser definida com `-D_GEOLAB_ANIMATION` quando o algoritmo destinar-se ao **GeoLab**. Notem os parâmetros inicializados no algoritmo.

```
Segment2D_ptr closest_pair_bf(list(Point2D_ptr) &points,
                              World *w = 0, int animation_depth = 0,
                              int reference_depth) {

    /* NOTE A COLOCAÇÃO DE PARÂMETROS PARA ANIMAÇÃO */

    // Entrada: uma lista de apontadores para pontos (Point2D)
    // Saida  : um apontador para um segmento criado que une os dois
    //          pontos mais próximos

    list_item it_p, it_cl;
    Segment2D_ptr res = 0;
#ifdef _GEOLAB_ANIMATION
    ANIMATION_BEGIN(w, animation_depth, reference_depth)
      proximity_animation_init(w);
    ANIMATION_END(w, animation_depth, reference_depth)
#endif
    if (points.size() > 1) {
      Point2D_ptr p1 = 0, p2 = 0;
      double aux, menor = MAXDOUBLE;
      forall_list_items(it_p, points) {
        forall_list_items(it_cl, points) {
#ifdef _GEOLAB_ANIMATION
          ANIMATION_BEGIN(w, animation_depth, reference_depth)
            anim_segment→set(*points[it_p], *points[it_cl]);
            anim_s→draw(w, XorMode);
            frame_stop_mark(w);
            anim_s→draw(w, XorMode);
          ANIMATION_END(w, animation_depth, reference_depth)
#endif
          if (it_p ≠ it_cl &&
              (aux = points[it_p]→e_distance_sqr(*points[it_cl])) < menor) {
#ifdef _GEOLAB_ANIMATION
            ANIMATION_BEGIN(w, animation_depth, reference_depth)
              if (p1) {
                anim_segment→set(*p1, *p2);
                anim_s→draw(w, XorMode);
              }
            ANIMATION_END(w, animation_depth, reference_depth)
#endif
```

```
            p1 = points[it_p]; p2 = points[it_cl];
#ifdef _GEOLAB_ANIMATION
            ANIMATION_BEGIN(w, animation_depth, reference_depth)
              anim_segment→set(*p1, *p2);
              anim_s→draw(w, XorMode);
              frame_stop_mark(w);
            ANIMATION_END(w, animation_depth, reference_depth)
#endif
            menor = aux;
          }
        }
      }
      res = new Segment2D(*p1, *p2);
    }

    return res;
}
```

B.1.4 Biblioteca de Algoritmos Geométricos

Os algoritmos implementados até o momento (outubro/92) complementam o conjunto de ferramentas geométricas que podem ser utilizadas para a produção de mais implementações. Dentre os mais relevantes destacamos:

- **Convexidade** – Envoltórias convexas (algoritmos de Jarvis e de Graham) e cascas convexas para conjuntos de pontos. Envoltórias convexas de polígonos simples (Lee) e em forma de estrela. Núcleo de polígonos simples (Lee e Preparata).

- **Proximidade** – Diâmetro de conjuntos de pontos e de polígonos, par mais próximo (diversos algoritmos), todos os vizinhos mais próximos, árvore espalhada euclidiana mínima, diagrama de Voronoi (de vizinho mais próximo e de vizinho mais distante), triangulações (Delaunay e dual do diagrama de Voronoi de vizinho mais distante).

- **Interseção** – Teste de interseção de segmentos, raios e retas. Cálculo de interseções entre segmentos, raios e retas.

- **Diversos** – *Winding number*, pares antipodais em polígonos convexos.

- **Modos de Construção Interativa** – Desenhador de subdivisões planares, desenhador de segmentos direcionados, desenhador de *Strings* e desenhador de imagens.

- **Modos Funcionais** – Localizador de pontos em subdivisões planares.

B.2 Objetos Geométricos

B.2.1 Hierarquia de Objetos

Objetos geométricos são organizados através de hierarquias de classes em C++. A razão para tal, além de proporcionar uma organização para tais objetos, é o estabelecimento de um protocolo de mensagens com o qual o ambiente comunica-se com os objetos permitindo, entre outras coisas, que novos objetos sejam inseridos facilmente e consistentemente com relação às ferramentas primitivas disponíveis na interface do **GeoLab**.

Da forma com que foram projetados e implementados, os objetos geométricos são descritos não através de uma, mas duas hierarquias de classes. Em uma hierarquia, cuja super-classe chama-se `Obj` (definida em `Obj.h`), estão organizados os objetos geométricos na forma "pura", i.e., contendo somente os dados e funções essenciais para sua representação geométrica. Na outra hierarquia, cuja super-classe é chamada `GraphicObj` (também em `Obj.h`) estão os dados e funções que complementam as informações geométricas de tal forma que o **GeoLab** possa manipulá-los graficamente (i.e., atributos como cor, tipo de traço, etc, além das funções que os manipulam).

A razão para o estabelecimento destas duas hierarquias é a separação entre um tipo abstrato de dado (i.e., um objeto "puro") de um tipo utilizado apenas dentro do contexto do **GeoLab** (i.e., um objeto "gráfico"). De fato, com esta separação consegue-se que os algoritmos geométricos manipulem apenas os objetos puros, tornando-se um pouco mais independentes do ambiente (figura B.1).

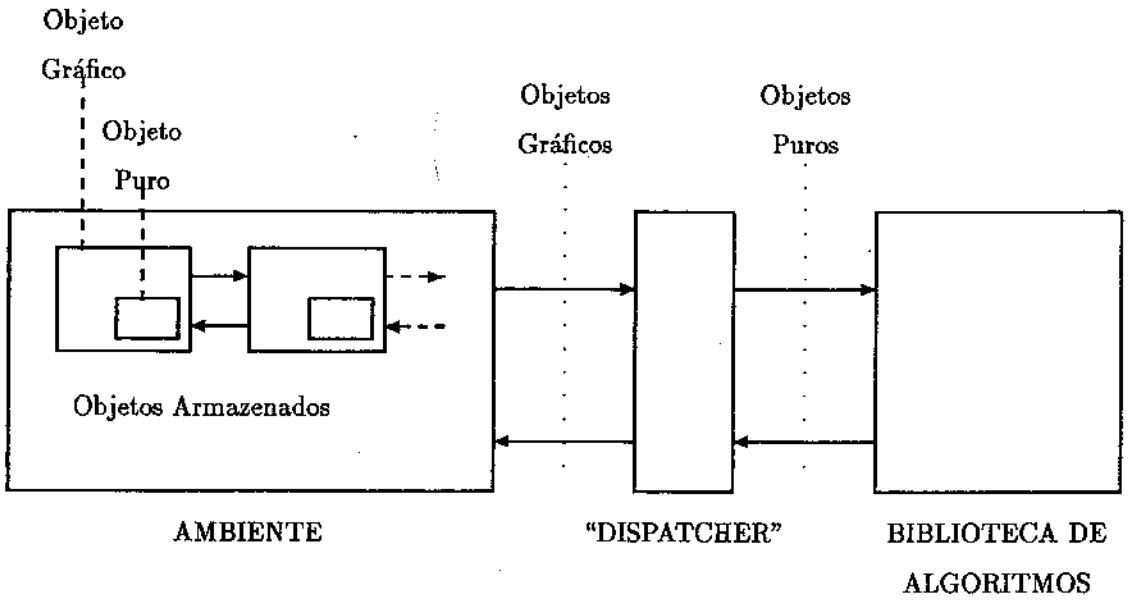

Figura B.1: Transmissão de dados entre ambiente e algoritmos geométricos externos – bibliotecas "independentes" em função da existência de uma dupla hierarquia de objetos

Do ponto de vista funcional, estas hierarquias (que poderíamos chamar daqui por diante de dupla hierarquia, já que existe uma relação de 1 para 1 entre seus componentes) estão relacionadas em dois sentidos:

1. Objetos gráficos possuem internamente um campo que aponta para o objeto puro (i.e, objetos gráficos contém objetos puros, não se tratando de derivação de classes).

2. Objetos puros "sabem" quem são seus objetos gráficos através de um método especial que simula um construtor virtual (i.e., pode-se pedir para um objeto puro produzir uma instância do objeto gráfico relacionado).

Assim sendo, o protocolo ao qual nos referimos desmembra-se em dois: um protocolo puro e um protocolo gráfico. O atendimento destes protocolos para cada novo objeto geométrico produzido é suficiente para que o ambiente possa manipulá-los.

B.2.2 Suporte à Programação

Os objetos atualmente existentes são aqueles mais utilizados por uma grande quantidade de algoritmos fundamentais em geometria computacional: Pontos, Segmentos, Raios, Retas, Retângulos, Quadrados, Elipses, Círculos, Polígonos, Diagramas de Voronoi, Triangulações de Delaunay e Subdivisões Planares.

Para cada um destes objetos existe, além dos protocolos definidos em função do próprio ambiente, métodos e funções que constituem a base para a programação geométrica que os utiliza. Dizemos serem estas as primitivas geométricas do ambiente.

Não existe um padrão para a organização destas primitivas. Isto, muito embora fosse desejável, não é possível pois cada objeto possui um conjunto de operações relacionadas muito particulares. Assim sendo, cada objeto define, do ponto de vista geométrico, suas primitivas. A tabela B.2 resume as primitivas geométricas para objetos do tipo ponto. Sugere-se que os usuários-programadores em potencial inspecionem as demais bibliotecas de objetos geométricos para a obtenção de mais detalhes sobre primitivas.

B.2.2.1 Protocolos para Objetos Geométricos

Esta seção apresenta os protocolos mencionados nas seções anteriores, descrevendo com mais detalhes alguns componentes.

O protocolo para objetos puros é apresentado na tabela B.3. A discussão sobre as mensagens deste protocolo (i.e., métodos da classe `Obj`) pode ser dividida de acordo com as utilidades de cada conjunto:

1. **Qualificação e identificação de objetos puros**

   Cada objeto é capaz de informar seu tipo (e.g., `Segment2D_t`) e subtipo (e.g., `Vertical_t`). O tipo de um objeto é uma qualidade estática de um objeto, definida na codificação da classe que o implementa. Duas implementações diferentes para um mesmo tipo podem retornar o mesmo valor a uma chamada ao método `type` desde que tais implementações concordem não só nos protocolos padrão como também no conjunto de primitivas geométricas. Isto permite que virtualmente qualquer tipo de experimentação com algoritmos externos no que diz respeito à análise de eficiência, robustez e consistência com diferentes escolhas de representação interna para objetos geométricos. Derivações de classes também encaixam-se nesta possibilidade.

   Além de um tipo estático, todo objeto pode ser qualificado através de um subtipo, alterável durante o tempo de vida do objeto, o que permite a simulação da migração de tipos evitando, em alguns casos, esforço de programação na definição de novas especializações de um mesmo objeto. Como exemplo, considere os objetos da classe `Polygon`, que podem ser qualificados como genéricos (`Generic_t`), simples (`Simple_t`), em forma de estrela (`Starshaped_t`), convexos (`Convex_t`), entre outros, sem que existam realmente classes para todas estas especializações de

Método	*Tarefa*
lessX	verificar se a coordenada X é menor que a de um dado ponto
lessY	verificar se a coordenada Y é menor que a de um dado ponto
equalX	verificar se dois pontos tem a mesma coordenada X
equalY	verificar se dois pontos tem a mesma coordenada Y
equal	verificar se dois pontos são iguais
lexicoXless	comparar lexicograficamente as coordenadas X Y de dois pontos
lexicoYless	comparar lexicograficamente as coordenadas Y X de dois pontos
circular_less	comparar circularmente (sentido anti-horário) dois pontos
left_turn	verificar se o ponto está a esquerda de uma reta direcionada dada por dois pontos
right_turn	verificar se o ponto está a direita de uma reta direcionada dada por dois pontos
colinear	verificar se três pontos são colineares
which_side	calcular a posição de um ponto com relação a uma reta direcionada
area	calcular a área de um triângulo (dados três pontos)
inCircle	calcular a posição de um ponto com relação a um círculo dado por três pontos
e_distance	calcular a distância euclidiana entre dois pontos
e_distance_sqr	calcular o quadrado da distância euclidiana entre dois pontos
angle	calcular o ângulo formado pela reta dada por dois pontos e o eixo X
gravitycentre	calcular o centro de gravidade de um triângulo (dados três pontos)
incentre	calcular o incentro de um triângulo
baricentre	calcular o baricentro de um triângulo
middle	calcular o ponto médio de um segmento dado por dois pontos

Tabela B.2: Resumo de primitivas geométricas para objetos do tipo Ponto em duas dimensões

polígono. As primitivas geométricas neste caso são gerais a todos os subtipos, mas nada impede que sejam construídos novos métodos para um particular subtipo, desde que, para efeito de consistência, estes métodos verifiquem internamente sua própria aplicabilidade.

2. **Auto-reprodução e associação à hierarquia gráfica**

   Todo objeto puro deve ser capaz de se auto-reproduzir, i.e., gerar uma nova instância de seu tipo cujo conteúdo seja idêntico ao seu. Isto é utilizado pelo ambiente para realizar a cópia de objetos através do modo funcional de cópia de objetos (seção A.1.3). A presença do método `reproduct` ocorre tanto nos objetos puros quanto nos objetos gráficos, para que o encapsulamento de cada um seja mantido.

   Para efeito de relacionamento entre a dupla hierarquia, todo objeto puro deve conhecer o objeto gráfico relacionado, e isto se faz suprindo um método que simula um construtor virtual (não suportado naturalmente em C++) nos objetos puros. Isto é utilizado basicamente no processo de comunicação com os algoritmos externos que, idealmente, devem manipular apenas objetos puros, mantendo o máximo de independência possível do ambiente.

3. **Conversão de e para uma representação base**

   Devido ao fato de que um mesmo objeto pode ser implementado de inúmeras maneiras (e.g., uma subdivisão planar pode ser representada através de uma QuadEdge, de uma HalfEdge,

Método	*Tarefa*
type	retornar o tipo do objeto (e.g., `Segment2D_t`)
subtype	retornar (ou ajustar) o subtipo do objeto (e.g., `Vertical_t`)
new_GraphicObj	construtor virtual simulado
reproduct	auto-duplicador
to_base	converter a estrutura em uma representação base (HalfEdge)
from_base	extrair informações a partir de uma representação base
load	recuperar as informações sobre o objeto
save	escrever (de forma recuperável) as informações do objeto
print	produzir uma descrição textual do objeto
translate	transladar o objeto para uma dada posição
rotate	rotacionar o objeto
scale	ampliar ou reduzir o objeto
draw	desenhar o objeto em um dispositivo gráfico
draw_handles	desenhar as alças do objeto
select_test	verificar se o objeto está sendo apontado pelo *mouse* ou se está totalmente contido dentro de um retângulo de seleção
distance_to_point	calcular a menor distância de um ponto ao objeto
obj_volatile	retornar (ou ajustar) a condição de objeto *volátil* (veja seção 3.1.4.3)

Tabela B.3: Protocolo de mensagens para objetos geométricos "puros"

WingedEdge, RadialEdge, Doubly Connected Edge List, etc) e, para garantir que qualquer necessidade de um particular algoritmo geométrico quanto à representações específicas possa ser facilmente atendida e ainda, procurando minimizar o potencial grande número de conversores entre estruturas, todo objeto deve ser capaz de converter-se em uma representação base, com potencial para representar uma grande quantidade, senão a totalidade, dos objetos de uso prático. Além disso, todo objeto deve ser capaz de reconstruir seu conteúdo a partir de uma representação base. Até o presente momento, a representação escolhida foi a HalfEdge proposta por Mäntylä [Män88].

A utilização destes conversores (com os devidos cuidados dada à flexibilidade e liberdade proporcionadas), assegura um alto grau de comunicação entre componentes a priori incompatíveis, através do **GeoLab**.

Outra possibilidade, advinda deste mecanismo, é a manipulação de memória secundária como mostrado a seguir.

4. **Armazenamento/recuperação em memória secundária e descrição textual de objetos**

Dado um descritor de arquivo, um objeto deve ser capaz de armazenar seu conteúdo de forma que se possa recuperá-lo futuramente. Para a maioria dos objetos esta é uma tarefa simples que não demanda significativo esforço de programação. Para alguns, entretanto, esta pode ser uma tarefa bastante complexa. Nestes casos, dada a existência dos conversores mencionados anteriormente, torna-se extremamente fácil armazenar e recuperar um objeto: converte-se o objeto para a representação base e armazena-se o resultado (através das operações disponíveis na representação base). A recuperação se faz executando-se os passos na ordem inversa.

Do ponto de vista de esforço de programação, observa-se que a introdução de conversores não reduz o trabalho, mas apenas transfere-o para um lugar onde é mais útil.

Todo objeto deve também ser capaz de apresentar uma descrição textual legível (por um usuário) de seu conteúdo. Tal recurso é extremamente útil no processo de depuração tanto de algoritmos geométricos quanto dos próprios objetos geométricos.

5. **Translação, rotação e escala**

   Respondendo à necessidades básicas para sua manipulação gráfica, todo objeto puro deve responder às mensagens de translação (`translate`), rotação (`rotate`) e escala (`scale`). Tais mensagens servem de suporte àquelas existentes nos objetos gráficos, mantendo o encapsulamento de representações internas utilizadas nos objetos puros.

6. **Desenho de um objeto e de suas alças**

   Dados os parâmetros gráficos (*handle* para o sistema de coordenadas e contexto gráfico), objetos puros devem responder a requisições para desenhar seu conteúdo e/ou suas alças. Da mesma forma que as mensagens para translação, rotação e escala, estas mensagens servem de suporte para aquelas existentes nos objetos gráficos.

7. **Teste de seleção e distância a um dado ponto**

   Também relacionado com a interface (embora de forma genérica o bastante para evitar dependências), existem métodos nos objetos puros que verificam a condição de seleção (`select_test`) e também calculam sua distância mínima a um dado ponto.

   No caso do teste de seleção, dado um ponto (usualmente uma posição na tela), o objeto verifica se alguma de suas alças coincide com o ponto. Dado um retângulo, é verificado se todas as alças do objeto são internas ao retângulo.

   O cálculo de distância complementa (em termos de suporte para interface) o teste de seleção. Objetos podem ser selecionados através de cliques sobre suas arestas caso tenham implementado a função que calcula sua distância mínima a um dado ponto.

8. **Tempo de vida de objetos puros**

   A comunicação entre os algoritmos externos e o ambiente realiza-se através de objetos geométricos puros. Para permitir uma maior flexibilidade deste modelo, foram implementados "objetos geométricos" para a representação de dados não realmente geométricos como inteiros, reais, booleanos e cadeias de caracteres (*strings*).

   Tais "objetos geométricos" comunicam valores que usualmente são apresentados e então descartados. Para limitar o tempo de vida de um objeto desta forma, utiliza-se o método `obj_volatile` para caracterizá-lo como volátil. O processo de composição de algoritmos pode exigir uma mudança nesta abordagem, não tendo sido isto ainda investigado.

Já o protocolo para objetos gráficos, além de uma conexão com todas as mensagens do protocolo "puro", implementa o conjunto de mensagens apresentada na tabela B.4. Existem mensagens analogamente agrupadas em função de suas utilidades:

Método	*Tarefa*
int_start	iniciar a criação interativa do objeto
int_change	alterar o último valor registrado na criação interativa
int_end	finalizar a criação interativa do objeto
resize_start	iniciar o redimensionamento do objeto
resize_change	alterar o último valor registrado no redimensionamento
resize_end	finalizar o redimensionamento do objeto
select	selecionar o objeto
unselect	desfazer uma seleção
selected	verificar se um objeto está selecionado
glue	fixar as coordenadas do objeto
unglue	desfazer uma operação de glue
glued	verificar se um objeto está fixado
anchor	tornar um objeto não-removível[3]
unanchor	desfazer uma operação de anchor
anchored	verificar se um objeto é não-removível
set_handle	tornar as alças visíveis/invisíveis
set_color	alterar os atributos de cor do objeto gráfico
set_line_attr	alterar o tipo de traço usado para desenhar o objeto gráfico
set_draw_mode	ajustar modo de desenho para o objeto (e.g., cópia de bits, ou-exclusivo)
reproduct	auto-duplicador
load	recuperar as informações sobre o objeto
save	escrever (de forma recuperável) as informações do objeto
type	retornar o tipo do objeto
subtype	retornar (ou ajustar) o subtipo do objeto
get_external	identificar o algoritmo e o módulo onde tal objeto foi produzido para o caso de objetos externos
outros	todos os métodos existentes para objetos puros são também acessíveis pelos objetos gráficos através da interceptação de chamadas (note que não se trata de herança, e sim de uma dupla hierarquia)

Tabela B.4: Protocolo de mensagens para objetos geométricos "gráficos"

1. **Construção interativa**

   Os objetos gráficos implementam os recursos básicos para sua construção interativa através dos métodos `int_start`, `int_change` e `int_end`. Para objetos simples (e.g., os objetos primitivos providos pelo **GeoLab**) tais métodos correspondem às ações de pressionar, arrastar e soltar o botão do *mouse*. Para objetos complexos (e.g., subdivisões planares), tal associação pode ser diferente, mas ainda assim pode-se organizar consistentemente a semântica associada a cada método.

2. **Redimensionamento de objetos**

   Dado que um objeto necessite ser redimensionado, o ambiente utiliza os métodos `resize_start`, `resize_change` e `resize_end` que devem ser supridos pelo objeto.

   Tanto na construção interativa quanto no redimensionamento, escolheu-se mapear as ações do

usuário em um conjunto de mensagens de objetos gráficos em função da organização proporcionada e também do relacionamento lógico entre estes e os objetos puros. Observa-se que, apesar de não ter acesso direto ao objeto puro (i.e., o encapsulamento de um objeto puro não deve ser quebrado pelo objeto gráfico associado), um objeto gráfico é o ente mais próximo para realizar tais tarefas.

3. **Atributos de seleção**

   Complementando os testes de seleção dos objetos puros, existem os métodos para manipular atributos próprios dos objetos gráficos. No caso de seleção, objetos gráficos mantêm internamente um *flag* que indica se o objeto está ou não selecionado, manipulado através de `select` e `unselect`. O método `selected` é usado para inspecionar o valor do *flag*.

4. **Objetos não-removíveis e objetos fixos**

   Objetos não-removíveis são objetos que não são afetados pelas operações de remoção disponíveis na interface do ambiente. Objetos fixos são objetos que não podem sofrer translação ou redimensionamento. Objetos fixos são qualificados, desqualificados e inspecionados por `glue`, `unglue` e `glued`, respectivamente. Objetos não-removíveis (ancorados) são analogamente manipulados por `anchor`, `unanchor` e `anchored`.

5. **Atributos genéricos**

   As alças de um objeto gráfico podem ser habilitadas ou desabilitadas através do método `set_handle`. Cor e tipo de traço são manipulados através de `set_color` e `set_line_attr`, respectivamente. Cores e tipos de traço são definidos nos módulos `color.h` e `line_settings.h`.

   O modo de desenho de um objeto gráfico é manipulado através de `set_draw_mode`. Modos possíveis são cópia de bits (`CopyMode`), ou-exclusivo (`XorMode`), etc.

6. **Auto-reprodução**

   Objetos gráficos possuem atributos que são reproduzidos internamente através do método `reproduct`. Para duplicar o objeto puro associado, é utilizada uma chamada ao seu método `reproduct`.

7. **Armazenamento/recuperação em memória secundária**

   Objetos gráficos armazenam seus dados através de métodos próprios, e solicitam ao objeto puro associado a armazenagem dos dados geométricos, analogamente ao processo de auto-reprodução.

8. **Tipos, subtipos e procedência de objetos**

   Da mesma forma que os objetos puros, os objetos gráficos possuem tipo e subtipo. Os valores retornados por tais métodos são os mesmos que o objeto puro associado retorna (e obtidos do mesmo). Isto permite que diferentes objetos puros sejam utilizados com um mesmo objeto gráfico (restrições quanto aos modos de criação possivelmente restringem a classe de objetos puros intercambiáveis aos objetos de uma mesma sub-árvore de derivação).

   Além disso, objetos geométricos externos (veja seção B.2.2.2) armazenam internamente sua procedência (i.e., o algoritmo ou o modo de criação interativa que o produziu), de forma a permitir que o método `instantiation` do protocolo da fonte produtora possa ser chamado

quando novas instâncias destes objetos necessitem serem criadas (e.g., no processo de recuperação de objetos externas de memória secundária).

É muito importante notar que, embora o protocolo para objetos gráficos seja extenso, poucos métodos deste protocolo necessitam ser realmente redefinidos e reimplementados em classes derivadas pois existe uma implementação *default* na super-classe que cobre uma grande maioria deles. O mecanismo de herança de C++ proporciona este benéfico reaproveitamento de código e assegura que somente os métodos dependentes de representação (e.g., métodos para construção interativa) necessitem ser reimplementados na maioria dos casos.

B.2.2.2 Introdução de Novos Objetos Geométricos

A criação de novos tipos de objetos geométricos, em termos de programação, se dá através da introdução de novas classes na dupla hierarquia.

Do ponto de vista de utilização, novas instâncias de objetos definidos externamente ao **GeoLab** (i.e., não compilados juntamente com o ambiente) são introduzidas a partir de algoritmos geométricos externos (note que os modos de construção interativa externos são tipos especiais de algoritmos geométricos).

Tais objetos são corretamente manipulados pelo ambiente em função de sua descendência das super-classes que o **GeoLab** conhece, e em função disso há o suporte de resolução de métodos virtuais (o protocolo) na linguagem utilizada.

Para cada novo objeto programado, deve-se proporcionar identificadores únicos para seu tipo e eventuais subtipos particulares. Tal tarefa deve sempre ser realizada incluindo dados em uma base comum a todos os demais objetos e armazenada em um arquivo chamado `ExternalObj_id.h`.

B.3 Modos de Criação Interativa e Modos Funcionais Externos

B.3.1 Tratamento de Eventos

A introdução de novos modos no **GeoLab** fundamenta-se no modelo de tratamento de eventos desenvolvido juntamente com o ambiente. A idéia é permitir que algoritmos externos instalem seus próprios tratadores de eventos, os quais são acionados (por redirecionamento do ambiente) quando o respectivo modo é selecionado junto à interface.

O método `binder` do algoritmo externo é o responsável pelo cadastramento de uma função que deve ser acionada quando o modo for selecionado e esta função deve então realizar a introdução dos tratadores de eventos (filtros) responsáveis pelo modo. O desenho (*icon*) a ser mostrado no botão polivalente do modo em questão também deve ser suprido através do método `binder`, como no exemplo abaixo:

```
void install_draw_arrow_events(World *w) {
    w→add_event_proc(draw_arrow_events);
    w→add_event_proc(fixed_movement_events);
}

static unsigned short draw_arrow_icon[] = {
```

```
#include "ICONS/draw_arrow.icon"
};

ResultObjects draw_arrow::binder(const SelectedObjects&, const GraphicObjects&, World*, int,
int) {
    ResultObjects dummy;
    install_external_object_mode(draw_arrow_icon, install_draw_arrow_events);
    return dummy;
}
```

Para os modos de criação interativa utiliza-se a ferramenta `install_external_object_mode` e para os modos funcionais `install_external_funcion_mode`. Ambos os modos encaixam-se na descrição a seguir.

O método `animation`, que informa ao ambiente da existência de código para animação em algoritmos externos, deve ser ajustado, excepcionalmente neste caso, para retornar *true*. Isto é devido ao fato de que modos externos não se encaixam no modelo de animação adotado, e devem estar sempre disponíveis, qualquer que seja o status de animação (i.e., ligada ou desligada).

O método `is_able_to_handle`, que informa ao ambiente da aplicabilidade do algoritmo em função da seleção corrente, tem a mesma função que em algoritmos externos genéricos, observando-se apenas que seu acionamento ocorre quando o modo for acionado (i.e., no painel de modos) e não quando o modo for instalado.

O método `unbinder` pode ser utilizado quando o modo externo necessita ser informado que foi desativado. Isto é particularmente útil no caso de algoritmos que realizam preprocessamento (e.g., para desalocar memória e atualizar estruturas internas). `unbinder` é acionado automaticamente pelo ambiente apenas para modos externos (i.e., modos de construção interativa e modos funcionais), podendo assumir outras funções no caso de algoritmos genéricos, ao gosto do programador.

Já no método `instantiation`, deve-se considerar a produção de objetos geométricos externos ao ambiente. Se for o caso, então comandos para a produção de novas instâncias devem ser incorporados, como no exemplo abaixo:

```
Obj* draw_arrow::instantiation(ObjType t) {
    // um switch pode ser mais adequado
    // caso vários objetos externos diferentes
    // sejam produzidos pelo novo modo
    if (t == Arrow2D_t)
      return new Arrow2D();
    else
      return 0;
}
```

B.3.2 Interação com o Sistema de Visualização

As facilidades para manipulação de eventos e também do sistema de coordenadas utilizado para apresentação gráfica de informações geométricas são proporcionadas pelo sistema de visualização chamado

World. Este sistema provê uma abstração mais efetiva do ambiente gráfico utilizado além de incluir diversos recursos adicionais, como *Zoom*, *Scroll*, etc.

Basicamente, um *World* é associado à janela de desenho do **GeoLab** (*Canvas*) e todo acesso a esta janela passa a ser feito através deste *World*.

Dentre as principais facilidades existentes, destacamos os seguintes grupos (detalhados em `World.doc` – veja seção B.6):

1. **Manipulação do sistema de coordenadas**

   Um método chamado `define(x1, y1, x2, y2)` permite que faixas de valores reais sejam associados aos eixos coordenados, inclusive com a possibilidade de definição da orientação destes. Outros métodos permitem busca e atualização destas faixas.

2. **Abstração das primitivas de desenho do ambiente X_Windows**

   Praticamente todas as funções do XLib [Nye89] são acessíveis através de *World*, com simplificações nos parâmetros e algumas extensões.

3. **Tratamento de eventos**

   World pode ser designado como roteador do fluxo de eventos de uma aplicação através dos métodos `set_repaint_proc` e `set_event_proc`. Funções tratadoras de eventos podem ser "cadastradas" junto ao *World* para que façam parte de uma lista de filtros, chamados em seqüência na ocorrência de eventos, com a possibilidade de consumir eventos em qualquer nível. Isto é particularmente interessante em grandes aplicações, pois pode-se modularizar os tratadores de eventos, torná-los intercambiáveis e mesmo assegurar que os tratadores são independentes uns dos outros.

   Tratadores podem ser adicionados com `add_event_proc`, retirados com `del_event_proc`, acionados manualmente com `call_event_proc`, etc.

4. **Facilidades para *Zoom* e *Scroll***

   World provê facilidades para *Zoom* como os métodos `zoom_in` e `zoom_out` que atuam sobre o sistema de coordenadas. Vários planos de *Zoom* podem ser mantidos através dos métodos `zoom_in_stack` e `zoom_out_stack`. Facilidades análogas existem para *Scroll*.

B.3.3 Modos de Construção Interativa

B.3.3.1 Manipulação da Lista de Objetos Geométricos

A principal especificidade dos modos de criação interativa externos diz respeito à manipulação da base de dados de objetos geométricos do **GeoLab**. Para simplificar a tarefa, uma vez terminada a criação de um objeto deve-se introduzi-lo no ambiente através da função `install_externally_created_object` (definida em `External.h`).

Continuando o exemplo iniciado na seção B.3.1, um tratador de eventos com a inclusão de um objeto no seu término seria:

```
bool draw_arrow_events(World *world, Xv_Window, Event *event) {
    static GraphicArrow2D *current_arrow = 0;
```

```
    static Int_Mode mode = LeftButton;

    double X = world→virtual_xcoord(event_x(event));
    double Y = world→virtual_ycoord(event_y(event));

    switch (event_action(event)) {
      case ACTION_SELECT:
      case ACTION_ADJUST:
        if (event_is_down(event) && !current_arrow) {
          if (event_action(event) == ACTION_SELECT)
            mode = LeftButton;
          else
            mode = MiddleButton;
          current_arrow = new GraphicArrow2D();
          current_arrow→set_all(world, FOREGROUND, BACKGROUND, 0, 0, HandlesOn);
          current_arrow→int_start(world, X, Y, mode);
          current_arrow→draw(world, XorMode);
        }
        else
          if (current_arrow) {
            current_arrow→draw(world, XorMode);
            current_arrow→int_end(world, X, Y, mode);
            current_arrow→set_line_attr(world, LINE_WIDTH, LINE_STYLE);
            current_arrow→draw(world, CopyMode);
            install_externally_created_object(current_arrow);
            current_arrow = 0;
          }
        break;

      case LOC_DRAG:
        if (current_arrow) {
          current_arrow→draw(world, XorMode);
          current_arrow→int_change(world, X, Y, mode);
          current_arrow→draw(world, XorMode);
        }
        break;
    }

    // retorna true APENAS se for desejado CONSUMIR o evento
    return false;
}
```

B.3.3.2 Recomendações

A facilidade de introduzir novos modos de criação interativa tem o indesejável efeito de possibilitar que o funcionamento das ferramentas seja determinado não mais pelo ambiente, mas pelo usuário-programador. Recomenda-se então que se evitem as torres de Babel seguindo-se o padrão de interface descrito no apêndice A, além dos filtros padronizados disponíveis no **GeoLab** (para restrição de movimentos, compressão de eventos, etc).

Alguns dos filtros existentes são:

1. Compressor de eventos – Tratador responsável por comprimir eventos de movimentação do *mouse*.

2. Restritor de Movimentos – Responsável por auxiliar o desenho de objetos sobre eixos verticais ou horizontais fixos.

3. Desenhador de objetos simples – Objetos como pontos, segmentos, círculos são manipulados por um mesmo tratador de eventos.

4. Controlador de área de visualização – Operações como *Zoom* e *Scroll* possuem tratadores próprios, mas que podem ser reutilizados caso deseje-se acrescentar opções a estas funções.

B.3.4 Modos Funcionais

B.3.4.1 Sugestões para Algoritmos Geométricos com Preprocessamento

Algoritmos que realizam preprocessamento podem ser mapeados de forma bastante interessante em modos funcionais externos. Para tanto, basta implementar tais algoritmos normalmente, e então suprir pequenos tratadores de eventos que realizam as chamadas às funções de cálculo, atualização, localização, etc, que utilizam-se do preprocessamento.

Localização de pontos, *Range-Search*, manutenção de envoltórias convexas e vários outros tipos de algoritmos de tempo-real podem ser mapeados com sucesso neste modelo.

B.3.4.2 Recomendações

Modos funcionais merecem os mesmos cuidados mencionados para os modos de construção interativa. Deve-se evitar também, nos dois casos, a produção de efeitos colaterais derivados da manipulação indevida de dados pertencentes ao ambiente (como as listas de objetos geométricos), já que as demais ferramentas não esperam este tipo de comportamento.

B.4 O Arquivo *.geolab-menu*

A ligação de novas bibliotecas com o ambiente é controlada pelo usuário através de um arquivo de configuração chamado `.geolab-menu`. Neste arquivo estão identificadas as bibliotecas juntamente com os algoritmos destas bibliotecas que devem ser considerados (i.e., se existir mais de um algoritmo em uma biblioteca, não é necessário que todos sejam considerados). Além disso, elaborou-se uma pequena linguagem de descrição destes componentes de forma a possibilitar ao ambiente a construção de seu menu de opções para algoritmos através de menus e submenus.

A sintaxe desta linguagem é bastante simples, como pôde ser verificado na seção B.1.3. Existem três tipos de comandos de estruturação das opções controlados por três palavras reservadas:

`TITLE` A palavra reservada `TITLE` seguida de uma cadeia de caracteres delimitada por aspas duplas introduz o título que será afixado no cabeçalho do menu de opções. Não há restrições quanto ao número de vezes que um título é definido, muito embora apenas a última ocorrência surtirá efeito.

`MENU` A palavra `MENU` seguida de uma cadeia de caracteres delimitada por aspas duplas indica que as opções subseqüentes devem ser organizadas em um submenu, identificado pela cadeia.

`END` Um submenu é terminado pela palavra reservada `END`. Pode-se suceder `END` com uma cadeia de caracteres opcional para identificar o `MENU` associado.

As opções são introduzidas em triplas que contém as seguintes informações:

"Opção no Menu" "Função de Ligação" "Nome da Biblioteca Compartilhada"

Comentários de uma linha são permitidos e delimitados por `/*` e `*/`. Abaixo mais um exemplo de um possível arquivo de configuração:

```
/* Isto é um comentário */

TITLE "Algorithms"

MENU "Points"
  MENU "Convex Hulls"
    "Jarvis' March" "jarvis_march" "ConvexHull.so"
    "Graham's Scan" "graham_scan" "ConvexHull.so"
  END
  MENU "Proximity Problems"
    "Diameter" "diameter_set_of_points" "Diameter.so"
    "Closest Pair" "closest_pair_dc" "Proximity.so"
    "All Nearest Neighbors" "all_nn_vd" "Proximity.so"
    "Euclidean Minimum Spanning Tree" "emst" "MinSpanningTree.so"
  END

MENU "Polygons"
  "Winding Number" "winding_number" "MiscClasses.so"
  "Kernel" "kernel_simple_polygon" "Kernel.so"
  "Antipodal Pairs" "antipodal_pairs" "Diameter.so"
  MENU "Diameter"
    "Generic Polygon" "diameter_generic_polygon" "Diameter.so"
    "Simple Polygon" "diameter_simple_polygon" "Diameter.so"
    "StarShaped Polygon" "diameter_star_shaped_polygon" "Diameter.so"
    "Convex Polygon" "diameter_convex_polygon" "Diameter.so"
```

```
  END
END
```

A interpretação desta pequena linguagem foi implementada utilizando-se o gerador de analisadores léxicos chamado `lex`.

B.4.1 Variáveis de Ambiente

A construção e a localização do arquivo de configuração podem ser auxiliadas através da definição de duas variáveis de ambiente utilizadas pelo **GeoLab**: `GEOLAB_MENU` e `GEOLAB_ALGORITHMS`.

B.4.1.1 Diferentes Ambientes de Trabalho

Em `GEOLAB_MENU` pode ser colocada uma localização *default* para o arquivo `.geolab-menu`.

Além desta facilidade, um mesmo usuário pode organizar seu trabalho sobre diferentes conjuntos de problemas mantendo vários arquivos `.geolab-menu`. O acesso a estes vários arquivos pode ser manipulado mesmo sem o auxílio de `GEOLAB_MENU`, já que o ambiente procura primeiro no diretório corrente, em seguida no diretório raiz do usuário e só então no local indicado pela variável.

B.4.1.2 Localização de Algoritmos Geométricos no Sistema de Arquivos

Em `GEOLAB_ALGORITHMS` pode ser colocada a localização no sistema de arquivos onde encontram-se as bibliotecas compartilhadas citadas no arquivo `.geolab-menu`. Desta forma evita-se colocar o nome completo (*full path name*) das bibliotecas possibilitando economia de espaço e flexibilidade na manipulação do sistema de arquivos.

B.5 Criação de *Shared Libraries*

No ambiente onde foi desenvolvido o **GeoLab**, a criação de bibliotecas compartilhadas é conseguida através da utilização de parâmetros de compilação e ligação (*link*) disponíveis no compilador e no ligador.

A compilação de um módulo que irá se tornar uma biblioteca compartilhada se faz em dois passos: o primeiro, compilando o módulo para que o resultado seja uma biblioteca cujo código é independente de posição. Isto é conseguido com os *flags* `-pic` (*position independent code*) e `-c`.

Em seguida, deve-se usar o ligador sobre o arquivo objeto da forma:

```
ld -assert pure-text <arq>.o -o <arq>.so
```

O parâmetro `-assert pure-text` serve como instrução para o ligador verificar se os arquivos objeto sendo ligados são independentes de posição (i.e., foram compilados com `-pic`).

Para a produção de bibliotecas compartilhadas pelo **GeoLab** estes dois passos são suficientes. As chamadas de sistema (*system calls*) *dlopen*, *dlclose*, *dlsym* e *dlerror* são utilizadas pelo **GeoLab** para ligar dinamicamente a biblioteca ao seu espaço de endereçamento e a partir daí, tendo acionado a função de ligação do algoritmo (que retorna uma instância de um objeto do tipo `Algorithm`), todo acesso às bibliotecas é feito através do protocolo de algoritmos.

B.5.1 Shared Libraries de Uso Geral

Embora apenas a produção de um *shared object* (`.so`) seja suficiente para o **GeoLab** (pois as *system-calls* utilizadas não manipulam as demais estruturas descritas a seguir), quando se quer que as bibliotecas sejam utilizáveis por outros projetos fazem-se necessários alguns outros cuidados.

Para complementar a geração dos `.so`, resta ainda produzir um arquivo (aqui referenciado como `.sa`) responsável por informar ao sistema a respeito dos dados inicializados exportáveis pelo *shared object* associado.

Dados inicializados exportados são variáveis e constantes globais aos módulos que compõem o *shared object* e que **não** são estáticos ao módulo (i.e., não possuem a cláusula *static* precedendo-os).

A produção do `.sa` é simples. Basta isolar o código correspondente aos dados inicializados exportados e então compilar tal código normalmente (de forma a produzir um arquivo no formato executável).

B.5.2 Limitações Impostas pelo Sistema

O sistema peca ainda por não permitir que bibliotecas compartilhadas ligadas manualmente através das chamadas de sistema *dlopen*, *dlclose*, *dlsym* e *dlerror* sejam também compostas por outras bibliotecas compartilhadas, i.e., que uma biblioteca compartilhada que use outra biblioteca compartilhada possa ser ligada de forma simples com sucesso.

Até o presente momento, este problema tem sido contornado simplesmente adicionando ao **GeoLab** os módulos e estruturas mais utilizados por algoritmos externos (fazendo com que estes não precisem ser ligados também aos algoritmos externos), mas isto tem o indesejável efeito de tornar o ambiente cada vez maior (em termos de memória ocupada) tirando-lhe um pouco de sua flexibilidade inerente.

A solução para o problema provavelmente virá juntamente com as próximas versões do SunOS.

B.6 Manutenção do Ambiente

Abaixo são enumerados os módulos que compõem o **GeoLab** juntamente com um pequeno resumo de seu conteúdo e utilidade. Atualmente, os módulos que compõem o **GeoLab** somam em torno de 25.000 linhas de código C++ enquanto cerca de outras 25.000 linhas correspondem aos algoritmos correntemente implementados.

Mensagens de erro que podem ocorrer durante a execução do **GeoLab** sempre identificam o módulo e a localização (função ou método) onde o problema foi detectado, auxiliando no processo de manutenção. Nomes de módulos iniciados por letras maiúsculas definem novas classes, enquanto módulos iniciados por minúsculas contém (usualmente) somente funções.

Algorithm.h 130 Definição da classe `Algorithm`, utilizada no mapeamento de algoritmos geométricos.

B_stack.h 90 Definição da classe `b_stack` (*bounded stacks*) incorporada da biblioteca LEDA e utilizada por `Bb_tree`.

Bb_tree.h 270 Definição da classe `bb_tree` (*BB-[alpha]Trees*) utilizada por `Range_tree.h`.

Bb_tree.C 710 Implementação dos métodos da classe `bb_tree`.

Circle.h 190 Definição da classe `Circle`, um dos objetos geométricos utilizados pelo **GeoLab**.

Circle.C 160 Implementação dos métodos da classe `Circle`.

Coords2D.h 100 Definição da classe `Coords2D`, que implementa a abstração de coordenadas homogêneas para objetos de duas dimensões,

utilizada em todas as definições de objetos geométricos do **GeoLab**.

Coords2D.C 120 Implementação dos métodos da classe `Coords2D`.

D2_dictionary.h 150 Definição da classe `d2_dictionary` utilizada em `Interval_set.h`.

Ellipse.h 170 Definição da classe `Ellipse`, um dos objetos geométricos utilizados pelo **GeoLab**.

Ellipse.C 310 Implementação dos métodos da classe `Ellipse`.

Interval_set.h 90 Definição da classe `interval_set`, utilizada pelos geradores de entrada do **GeoLab**.

Interval_set.C 40 Implementação dos métodos da classe `interval_set`.

Line2D.h 170 Definição da classe `Line2D` (reta em duas dimensões).

Line2D.C 290 Implementação dos métodos da classe `Line2D`.

List.h 330 Definição da classe `list` (lista duplamente encadeada).

List.C 540 Implementação dos métodos da classe `List`.

Obj.h 560 Definição da super-classe `Obj`, raiz da hierarquia de objetos geométricos puros do **GeoLab**.

Obj.C 80 Implementação dos métodos da classe `Obj`.

Point2D.h 330 Definição da classe `Point2D` (ponto em duas dimensões).

Point2D.C 530 Implementação dos métodos da classe `Point2D`.

Polygon.h 180 Definição da classe `Polygon` (polígono).

Polygon.C 410 Implementação dos métodos da classe `Polygon`.

Range_tree.h 210 Definição da classe `range_tree` (*d-dimensional Range Trees*), utilizada por `D2_dictionary.h`.

Range_tree.C 840 Implementação dos métodos da classe `range_tree`.

Ray2D.h 100 Definição da classe `Ray2D` (raio em duas dimensões).

Ray2D.C 190 Implementação dos métodos da classe `Ray2D`.

Rectangle.h 190 Definição da classe `Rectangle` (retângulo).

Rectangle.C 180 Implementação dos métodos da classe `Rectangle`.

SList.h 200 Definição da classe `slist` (listas encadeadas), utilizada em `Stack.h`.

SList.C 120 Implementação dos métodos da classe `slist`.

Segment2D.h 170 Definição da classe `Segment2D` (segmento em duas dimensões).

Segment2D.C 280 Implementação dos métodos da classe `Segment2D`.

Square.h 120 Definição da classe `Square` (quadrado), derivada de `Rectangle`.

Square.C 70 Implementação dos métodos da classe `Square`.

Stack.h 70 Definição da classe `Stack` (pilha), utilizado no analisador léxico do menu de configuração, em `algorithms.C`.

String.h 50 Definição da classe `String` (cadeia de caracteres).

String.C 90 Implementação dos métodos da classe `String`.

World.h 400 Definição da classe `World`, utilizado pelo **GeoLab** para manter, controlar e prover acesso às facilidades do dispositivo gráfico.

World.C 450 Implementação dos métodos da classe `World`.

algorithms.h 110 Definições para o módulo de controle de algoritmos externos e de construção do menu de opções.

algorithms.C 570 Módulo de controle de algoritmos externos e da construção do menu de opções.

animation.h 40 Definições para o módulo de controle de animação de algoritmos.

animation.C 270 Módulo de controle de eventos durante a animação de algoritmos.

basic.h 640 Definições para os módulos importados do LEDA.

basic.C 640 Inicializações e funções de apoio aos módulos do LEDA.

benchmark.h 30 Definições para o módulo de controle de tempo de execução de algoritmos externos.

benchmark.C 50 Módulo de controle dos *timers* da máquina para efeito de *benchmark* de algoritmos.

color.h 40 Definições para o módulo de controle e alocação de cores do **GeoLab**.

color.C 100 Funções e variáveis globais de apoio para a manipulação de cores.

compose.h 30 Definições para o módulo de composição de algoritmos.

compose.C 90 Manutenção da lista de algoritmos que compõe uma composição.

dynamic.h 30 Definições para o módulo de animação por movimentação dinâmica.

dynamic.C 270 Controlador de eventos para animação por movimentação dinâmica.

edit_mode.h 90 Definições para o módulo de controle dos modos de operação do **GeoLab**.

edit_mode.C 550 Módulo de controle dos modos de operação. Controla acesso aos botões polivalentes a ajusta os diversos tipos de cursores utilizados pelo **GeoLab**.

geolab_stubs.C 1920 Implementação de todas as funções que respondem às ações do usuário sobre a interface gráfica do ambiente (*callbacks*).

geolab_ui.h 500 Definição da interface gráfica do **GeoLab**.

geolab_ui.C 4610 Implementação dos componentes da interface gráfica do **GeoLab**.

global.h 40 Definições globais utilizadas no **GeoLab**.

global.C 20 Inicializações das variáveis globais do **GeoLab**.

handle.h 60 Definições para o módulo de desenho de alças de objetos gráficos.

handle.C 40 Módulo de desenho de alças de objetos gráficos.

input_tools.h 60 Definições para os geradores de entrada.

input_tools.C 400 Implementação dos geradores de entrada do **GeoLab**.

insert_simple.h 30 Definições para o módulo de controle dos modos de construção interativa primitivos.

insert_simple.C 70 Módulo de controle de eventos nos modos de construção interativa primitivos.

instantiation.h 50 Definições para o módulo de instanciação de objetos geométricos internos e externos.

instantiation.C 90 Funções para instanciação de objetos geométricos internos e externos.

line_settings.h 40 Definições para o módulo de controle de estilo e tipo de traço.

line_settings.C 20 Funções para manipulação de estilo e tipo de traço no desenho de objetos geométricos.

load_save.h 40 Definições para o módulo de armazenamento e recuperação de objetos gráficos em memória secundária.

load_save.C 120 Funções para armazenamento e recuperação de objetos gráficos do editor.

move_copy.h 30 Definições para o módulo de controle do modo de cópia e translação de objetos.

move_copy.C 90 Tratador de eventos no modo de translação e cópia de objetos.

objects.h 100 Definição das funções de controle da base de dados de objetos geométricos do **GeoLab**.

objects.C 720 Módulo de controle da base de dados de objetos gráficos do **GeoLab**.

quit.h 30 Definições para o módulo de término de execução.

quit.C 40 Funções de controle de término de execução do ambiente.

resize.h 30 Definições para o módulo de controle do modo de redimensionamento de objetos.

resize.C 80 Tratador de eventos no modo de redimensionamento de objetos.

scroll.h 30 Definições para o módulo de *Scroll*.

scroll.C 60 Tratador de eventos no modo de *Scroll*.

select.h 30 Definições para o módulo de seleção de objetos.

select.C 100 Tratador de eventos no modo de seleção de objetos.

tools.h 30 Ferramentas genéricas para cálculos numéricos com tolerância.

tools.C 40 Implementação das ferramentas genéricas para cálculos com tolerância.

trapsignal.h 40 Definições para o módulo de interceptação de sinais do **GeoLab**.

trapsignal.C 90 Tratador de exceções do **GeoLab**.

utilities.h 30 Ferramentas genéricas para manipulação de entrada e saída de dados.

utilities.C 30 Implementação das ferramentas genéricas para manipulação de entrada e saída de dados.

zoom.h 40 Definições para o módulo de *Zoom*.

zoom.C 330 Tratador de eventos no modo de *Zoom*.

Para algoritmos e modos de operação externos, assim como novos objetos geométricos incorporados dinamicamente, existem ainda dois módulos:

External.h 140 Agrupamento das definições que devem ser importadas por todos os algoritmos geométricos externos.

ExternalObj_id.h 100 Arquivo de definição de identificadores únicos para objetos geométricos externos. Deve sempre ser atualizado quando novos objetos são criados.

B.6.1 Documentação Básica e *Header Files*

Abaixo encontram-se alguns módulos que devem ser consultados por usuários-programadores do **GeoLab**. `World.doc` descreve os recursos do sistema de visualização utilizado. `Obj.h` especifica as super-classes das hierarquias de objetos geométricos e `Algorithm.h` especifica a super-classe da hierarquia de algoritmos geométricos externos.

B.6.1.1 `World.doc`

```
-------------------------------------------------------------------------

        Modulo: World   -   11.10.1991   -   Welson R. Jacometti

-------------------------------------------------------------------------

Arquivos: World.h
          World.C

Funcao: possibilitar o uso de sistemas de coordenadas reais
        junto ao ambiente grafico (Canvas).

Flags: o flag -lm deve ser usado.

-------------------------------------------------------------------------

                                DESCRICAO

-------------------------------------------------------------------------


Construtores:
=============

  1. World(Canvas);

    Cria uma instancia de World associado a Canvas. Este tipo de
  instanciacao faz com que as coordenadas minimas e  maximas nos
  eixos X e Y sejam as mesmas do Canvas (suas dimensoes), exceto
  que o eixo Y e' invertido, de forma que a origem do sistema de
  coordenadas seja a canto inferior esquerdo da tela.


  2. World(Canvas, double x1, double y1, double x2, double y2);

    Cria uma instancia de World associado a Canvas e tambem define
  os valores minimos e maximos das coordenadas em X e Y.


  3. World(const World&);
  4. World& operator = (const World&);

                          ---------------

                              ATENCAO

                          ---------------

    Os metodos 3 e 4 NAO sao permitidos para objetos do tipo World pois
  nao existe meio seguro e eficiente de se manter a semantica da associacao
  entre World e Canvas (incluido repaint_proc e event_proc) consistente.
```

```
Assim sendo, qualquer tentativa como as ilustradas abaixo (ou variacoes
possiveis) provocam uma mensagem de erro e fim de programa:

World w1 = w2;

ou

World w1(canvas);
World w2(outro_canvas);

...

w2 = w1;


Funcoes de atualizacao e busca de valores internos:
======= == =========== = ===== == ======= ========

1. void restart();

   (Re)inicializa valores internos, para prever problemas como
redimensionamento de janela, etc.

2. bool define();

   (Re)define os valores de World para aqueles que seriam gerados
pela instanciacao de World(Canvas), i.e., coordenadas equivalentes
'as da View Window, porem com o eixo Y invertido.

3. bool define(double x1, double y1, double x2, double y2);

   Permite (re)definir os valores correntes de World. Note que
apenas os valores eventualmente 'empilhados' devido a zoom_in_stack
nao sao afetados. Para que a mudanca afete tambem 'initial_values',
a funcao reset() deve ser chamada antes para liberar as outras
views.

4. void get_values(double &x1, double &y1, double &x2, double &y2);

   Retorna os valores correspondentes ao world corrente. Para
inspecionar 'initial_values' pode ser necessario usar reset().

5. void get_view(double &width, double &height);

   Retorna a largura e altura da porcao visivel do canvas associado. Os
os valores sao dados em pixels.

6. void get_paint(double &width, double &height);
```

```
   Retorna a largura e altura da janela de desenho (canvas_paint_window).
  Os valores sao dados em pixels.

  7. Canvas get_canvas();

   Retorna o canvas associado a World.

  8. Xv_Window get_paint();

   Retorna canvas_paint_window do canvas associado. Note que este metodo
  economiza um xv_get pois o valor e' previamente calculado (instanciacao).
  Note tambem que esta funcao e' "overloaded" com aquela que retorna as
  dimensoes do paint_window.

  9. Display *get_display();

   Retorna display associado a canvas, por sua vez associado a
  World.

 10. Window get_window();

   Retorna window associado a canvas, por sua vez associado a
  World. (Equivalente a xv_get(canvas_paint_window(canvas), XV_XID)).


Funcoes de conversao entre sistemas de coordenadas:
======= == ========== ===== ======== == ===========

  1. int xcoord(double virtual_x); e
  2. int ycoord(double virtual_y);

   Convertem coordenadas virtuais para coordenadas inteiras (reais).
   A conversao se da' obviamente segundo o sistema de coordenadas
  especificado na instanciacao (ou redefinicao) dos valores minimos
  e maximos para cada coordenada e sua relacao com as dimensoes
  do canvas associado.


  3. double virtual_xcoord(int real_x); e
     double virtual_ycoord(int real_y);

   Funcao inversa, ou seja, obtem os valores virtuais de coordenadas
  reais.


Funcoes de desenho:
======= == =======

  As seguintes funcoes de desenho tem por objetivo substituir
```

```
aquelas que XLib oferece, de forma a tornar transparente a
transformacao de coordenadas, e tambem aproveitar-se do
relacionamento entre Canvas, Window e Display para criar
uma melhor abstracao de Canvas.

  1. void Flush();

  2. void Sync();

  3. Pixmap CreatePixmap(Drawable drawable, int width, int height, int depth);

  4. void FreePixmap(Pixmap pixmap);

  5. void CopyArea(Drawable src, Drawable dest, GC gc, int src_x, int src_y,
                  int width, int height, int dest_x, int dest_y);

  6. void ClearWindow();

  7. void DrawPoint(GC, double x, double y);

  8. void DrawLine(GC, double x1, double y1, double x2, double y2);

  9. void DrawRectangle(GC, double x1, double y1, double x2, double y2);

 10. void FillRectangle(GC, double x1, double y1, double x2, double y2);

 11. void DrawArc(GC, double x1, double y1, double x2, double y2,
                 int arc_i = 0, int arc_f = 23040 /* 360 x 64 */);

 12. void DrawArc(GC, double x, double y, double radius, int arc_i = 0, arc_f = 23040);

    Versao overloaded de DrawArc convencional. Desenha circulos a partir
  de centro e raio.

 13. void FillArc(GC, double x1, double y1, double x2, double y2,
                 int arc_i = 0, int arc_f = 23040);

 14. void FillArc(GC, double x, double y, double radius, int arc_i = 0, arc_f = 23040);

 15. void DrawString(GC, double x, double y, char *str, int strlen);

 16. void DrawCenterString(GC, double x, double y, char *str, int strlen);

 17. XFontStruct* LoadQueryFont(char *name);

 18. SetFont(GC gc, Font f);

Funcoes relacionadas com contexto grafico:
======= ============ === ======== =======
```

Contextos graficos estao relacionados com o Display em questao. Aproveitando-se entao do encapsulamento do valor de Display, foram tambem incorporadas funcoes de manipulacao de contextos graficos, de forma a simplificar seu uso. O uso destas funcoes porem nao e' obrigatorio. Nada impede que as funcoes correlatas possam ser chamadas, mesmo porque nao sao todas elas que foram trazidas para a interface de world.

A esta altura, alguem pode se perguntar: "E quanto 'a eficiencia?"

Pois bem, notem que (segundo meu sentimento) as rotinas mais utilizadas e tambem aquelas que sao apenas "shortcuts" para outras funcoes ou:

a. Sao declaradas inline (o que causa sua expansao textual dentro do programa)

b. Sao definidas e declaradas no header file, o que equivale a uma declaracao inline.

Assim sendo, apesar de criar uma abstracao relativamente grande, o modulo nao deixa a desejar nas questoes de eficiencia.

1. `GC defaultGC();`

   Retorna o GC default associado com o canvas.

2. `GC CreateGC(unsigned long mask, XGCValues &values);`

3. `GC CreateGC(unsigned long mask, XGCValues *values);`

4. `void ChangeGC(GC gc, unsigned long mask, XGCValues &values);`

5. `void CopyGC(GC gcsource, unsigned long mask, GC gcdest);`

6. `void FreeGC(GC gc);`

7. `void SetForeground(GC gc, unsigned long color);`

8. `void SetBackground(GC gc, unsigned long color);`

9. `void SetFunction(GC gc, int function);`

10. `void SetPlaneMask(GC gc, int mask);`

11. `void SetLineAttributes(GC gc, int width, int style, int cap, int join);`

```
Funcoes de ZOOM e de SCROLL:
======= == ==== = == ======
```

As funcoes de zoom e scroll realizam mudancas internas no sistema de coordenadas de forma a proporcionarem ZOOM e SCROLL nos objetos ali desenhados. Por exemplo, para fazer ZOOM IN, basta diminuir proporcionalmente a faixa de valores validos nos eixos X e Y de tal forma que os objetos sejam desenhados "esticados". As outras rotinas funcionam de forma similar. Obs: Notem que "repaint" se faz necessario para visualizar o efeito destas funcoes.

1. void reset();

Retorna o sistema de coordenadas aos valores com que foi istanciado. Todos os valores de zoom_in_stack ou zoom_out_stack sao destruidos.

2. bool zoom_in(double x, double y, double factor);

Realiza zoom_in de toda a janela centrada em x,y proporcionalmente a factor. O resultado da funcao e' TRUE se o zoom foi completado sem problemas, e FALSE caso contrario.

3. bool zoom_out(double x, double y, double factor);

Funcao reversa analoga a zoom_in().

4. bool zoom_in_stack(double x, double y, double factor);

Funcao identica a zoom in, exceto que a view atual e' salva para que seja possivel retornar 'aquele estado atraves de zoom_previous() ou zoom_out_stack(). Se o zoom anterior tiver sido um zoom_out_stack, entao a funcao simplesmente faz um zoom_previous.

5. bool zoom_out_stack(double x, double y, double factor);

6. bool zoom_previous();

Restaura o sistema de coordenadas vigente antes de um zoom_in_stack ou zoom_out_stack.

7. double ratio_x();

Da' o valor da razao entre as diferencas dos valores maximo e minimo atuais e dos valores maximo e minimo iniciais com relacao a coordenada X. Valores negativos indicam uma inversao do sentido do uso do sistema de coordenadas.

8. double ratio_y();

```
   Idem a ratio_x(), porem para a coordenada Y.

 9. void pscroll_begin(GC);

   Esta funcao deve ser utilizada se for desejavel o uso de pixmap no
  processo de scroll. Basicamente, no momento em que pscroll_begin() e'
  chamado, e' feita uma "foto" da area de desenho (canvas ou pixmap) que
  entao passa a ser mostrada atraves da funcao pscroll().

10. void pscroll_end(int final_x, int final_y);

   Termina o processo de scroll e ajusta o sistema de coordenadas internamente.

11. void pscroll_same_size(int to_x, int to_y);

   Funcao para scroll iterativo onde view e' igual ao paint_window.

12. void pscroll_paint_bigger(int to_x, int to_y);

   Funcao para scroll iterativo onde view e' menor que paint_window.

13. void scroll(int horz_amount, int vert_amount);

   Faz um deslocamento horizontal/vertical de uma certa quantidade de
  pixels, especificado por amount. Se amount for positivo, entao
  o deslocamento se da' para a esquerda. Se negativo, o deslocamento
  e' para direita.


  Obs1: Nas funcoes de zoom, fatores de zoom validos estao na faixa de
        1 a n, sendo que um fator de zoom n faz com que os objetos sejam
        ampliados ou reduzidos de n vezes.

  Obs2: Nas funcoes de scroll, a quantidade de pixels a ser movida e'
        ajustada coerentemente de acordo com o fator de zoom corrente.


  Obs3: Os parametros to_x, to_y, final_x, final_y, horz_amount, vert_amount
        devem ser dados sempre em funcao do NUMERO DE PIXELS envolvido no
        scrolling, e nao em valores correspondentes a coordenadas virtuais.

Funcoes relacionadas a eventos:
======= ============ = =======

  1. void compress_drag(Event *event, int to_retain);

   Comprime eventos de LOC_DRAG associados a paint_window. A funcao descarta
```

os eventos repetidos, deixando apenas a quantidade especificada em to_retain. No final, o parametro contem o primeiro evento da fila comprimida.

2. void set_repaint_proc(f_repaint);

```
typedef void (*f_repaint)(World *);
```

Associa uma funcao externa aos eventos de repaint do canvas. A partir de uma chamada a set_repaint_proc(), eventos de repaint sao interceptados por World.

3. f_repaint get_repaint_proc();

Retorna o valor da variavel interna repaint_proc. Caso nao tenha sido previamente setada com o endereco de alguma funcao externa, entao seu valor e' NULL.

4. void call_repaint_proc();

Faz com que uma chamada explicita 'a funcao de repaint do usuario ocorra.

5. void set_event_proc(f_events);

```
typedef bool (*f_events)(World *, Xv_Window, Event *);
```

Associa uma funcao tratadora de eventos externa a World. Caso ja' existisse uma ou mais funcoes associadas, estas sao descartadas.

6. void reset_event_proc();

Descarta todas menos a primeira funcoes de evento associadas.

7. void add_event_proc(f_events);

Adiciona uma funcao de evento na lista das funcoes tratadoras de eventos de World. A funcao recem adicionada passa a ser a primeira a ser chamada (comportamento de pilha).

8. void del_event_proc();

Retira a ultima funcao de evento da lista, exceto quando so' existe uma, quando nada e' feito.

9. void call_event_proc(Event *);

Dispara o mecanismo de chamadas sucessivas 'as funcoes tratadoras de eventos associadas a World.

```
Obs1: Em funcao da estrategia escolhida para realizar a interceptacao de
      callbacks do xview ter considerado, primariamente, a questao de
      eficiencia, restringe-se o uso das facilidades acima quando temos
      multiplas instancias de World associadas a um mesmo canvas por considerar
      somente as funcoes (repaint, events) da ultima instancia associada.

Obs2: Nao existe encadeamento de funcoes de repaint.

Obs3: Nao e' feita verificacao de consistencia nos valores passados a
      set_repaint_proc e set_event_proc (eficiencia!). Assim sendo, valores
      absurdos podem provocar resultados absurdos (NULL por exemplo).

Obs4: Uma vez chamados set_repaint_proc e/ou set_event_proc, chamadas do
      XView so' deixarao de ser interceptadas se os valores CANVAS_REPAINT_PROC
      e WIN_EVENT_PROC forem alterados diretamente em Canvas e
      canvas_paint_window(Canvas).


At last, but not least:

  A convencao adotada para as funcoes tratadoras de eventos e' a seguinte:

  bool (*f_events)(World *, Xv_Window, Event *);

    . retorna TRUE se NAO deseja que as funcoes subsequentes sejam
      notificadas sobre o evento (ESTE NORMALMENTE E' UM CASO ESPECIAL!).

    . retorna FALSE caso contrario, mesmo que tenha tratado o evento.
      (NO CASO DO GEOLAB, ESTE E' O COMPORTAMENTO DEFAULT).

  A sequencia de chamadas na lista de funcoes associadas termina quando
  uma delas retorna TRUE, ou quando a lista se esgota.

-------------------------------------------------------------------------

                                BUGS

Descricao - Observacao          Data          Usuario     Resolvido? (S/N)
-------------------------------------------------------------------------

0. Em fase de testes.           11.10.91      Welson      Sim
1. Scroll                       25.10.91      Welson      Nao

   Em estacoes que possuem apenas 8M de memoria, e video colorido,
   pode ser extremamente lento o scrol com pixmap (no caso do paint
   window ser de proporcoes consideraveis). Isto se da' pela enorme
```

```
   quantidade de 'swaps' que tem de ser realizada pela maquina.

2. Scroll                          18.11.91        Welson          Nao

   Pode ser que ocorram pequenas discrepancias no formato do objeto
   apos um scroll quando a proporcao de zoom nao for 1:1 devido a
   imprecisao do modelo aritmetico utilizado. Um paleativo simples
   e' colocar a disposicao do usuario um botao de "refresh" que limpa
   a tela e redesenha os objetos.

-------------------------------------------------------------------------

                                EXTENSOES

Descricao - Sugestao               Data            Usuario     Implementado? (S/N)
-------------------------------------------------------------------------

1. CLIP                            11.10.91        Welson      Nao

   Fazer "clipping" de objetos, ou descobrir uma forma segura de fazer com
   que XView o faca. Nao necessario se usa estacao comum. Estacoes com ace-
   lerador grafico (GX) 'capotam' feio se nao for feito.

-------------------------------------------------------------------------
```

B.6.1.2 Obj.h

```
/*
 *
 * Obj.h - definicao da superclasse Obj, da qual descendem todos os
 *         objetos geometricos 'puros' (que nao contem dados e metodos
 *         extras, como os de interface), e da superclasse GraphicObj
 *         que incorpora elementos do tipo Obj e prove recursos de
 *         interface.
 *
 *
 *      Data    Hora    Nome                    Modificacao/Motivo
 *
 *    19.11.91  13:40   Welson R. Jacometti     Arquivo criado
 *
 */

#ifndef _OBJ_DEFINED
#define _OBJ_DEFINED

#include <iostream.h>
#include <stdio.h>
```

```
#include <sys/param.h>
#include "tools.h"
#include "World.h"
#include "String.h"
#include "utilities.h"
#include "basic.h"
#include "List.h"
#include "attributes.h"

#define TOO_FAR 10e+10

// forward delcarations
class Obj;
class GraphicObj;

typedef Obj* Obj_ptr;
typedef GraphicObj* GraphicObj_ptr;

#ifndef _Obj_ptr_list
#define _Obj_ptr_list
declare(list,Obj_ptr)
#endif ~_Obj_ptr_list;

#define GCALL 0xFFFFFFFF

/*****************************************************************************

  Para tentar diminuir o problema de nao termos a versao 3.0 do compilador
  C++, todos os metodos virtuais indefinidos em qualquer classe base sera'
  implementado de forma a produzir uma mensagem de erro do tipo:

  Warning: 'object type': 'method' not implemented

  Isso livra-me de ter que ficar contornando o problema da versao 2.0 nao
  permitir heranca sem redefinicao de metodos virtuais por mais de um ni-
  vel (somente uma classe base por hierarquia). Fique claro apenas que
  nestes casos, os metodos nao implementados realmente NAO tem que ser
  implementados, por tratar-se de uma classe abstrata.

*****************************************************************************/

// As constantes abaixo referem-se ao conjunto de objetos geometricos que
// sao conhecidos (em tempo de compilacao) pelo GeoLab. Infelizmente nao
// tive tempo para pensar em um metodo mais elaborado (automatico) e seguro
// para fazer esta identificacao, tao necessaria nas funcoes que conectam
// a interface do GeoLab aos algoritmos, ligados dinamicamente.
//
// IMPORTANTE:
//
```

```
// Objetos externos ao GeoLab (aqueles que sao introduzidos pelos algoritmos
// e que nao sao necessariamente conhecidos pelo GeoLab) devem ser unicamente
// identificados para que nao ocorram conflitos entre diferentes tipos. Minha
// sugestao e' que todo novo objeto introduzido externamente seja cadastrado
// em um arquivo 'ExternalObj_id.h' de forma que fique razoavelmente facil
// dar um novo "numero" aos novos objetos.

typedef int ObjType;

const ObjType MIN_INTERNAL_OBJ_ID = 0;
const ObjType MAX_INTERNAL_OBJ_ID = 1000;

const ObjType Obj_t             = 0;
const ObjType Point2D_t         = 1;
const ObjType Segment2D_t       = 2;
const ObjType Ray2D_t           = 3;
const ObjType Line2D_t          = 4;
const ObjType Rectangle_t       = 5;
const ObjType Square_t          = 6;
const ObjType Circle_t          = 7;
const ObjType Ellipse_t         = 8;
const ObjType Polygon_t         = 9;

// subtypes
const ObjType Vertical_t        = 10;
const ObjType Horizontal_t      = 11;
const ObjType Generic_t         = 12;
const ObjType Simple_t          = 13;
const ObjType StarShaped_t      = 14;
const ObjType Convex_t          = 15;
const ObjType Monotone_t        = 16;

// Funcao auxiliar, que diz se um objeto e' ou nao interno ao GeoLab

inline bool geolab_knows_object(ObjType t) {
    return range(t, MIN_INTERNAL_OBJ_ID, MAX_INTERNAL_OBJ_ID);
}

// Classe "Objeto Geometrico", raiz da hierarquia e herdado por todos seus
// componentes.

class Obj {
  public:
    // construtores
    Obj(char *s = "Obj",  ObjType t = Generic_t, bool v = false)
                                : is_a(s), stype(t), volati(v) {}

    // destrutor
    ~Obj() {}
```

```
// construtor virtual (simulado) que estabelece o relacionamento
//
//                    Obj --> GraphicObj

virtual GraphicObj* new_GraphicObj();

// funcao para auto-reproducao de um objeto -> necessario quando
// queremos, por exemplo, copiar um objeto na interface e este
// objeto e' externo ao GeoLab.

virtual Obj* reproduct() const {
    warning(what_is(), "reproduct");
    return 0;
}

// metodos para armazenamento secundario
virtual void load(FILE *);
virtual void save(FILE *);

virtual ObjType type() const {
    warning(what_is(), "type");
    return Obj_t;
}

virtual ObjType subtype() const { return stype; }

virtual ObjType subtype(ObjType v) { return stype = v; }

// transformacoes geometricas aplicadas a um objeto particular
virtual void translate(double to_x, double to_y) {
    warning(what_is(), "translate");
    cout << "to X: " << to_x << "   to Y: " << to_y << endl;
}

virtual void rotate(double orig_x, double orig_y, double angle) {
    warning(what_is(), "rotate");
    cout << "Origin at: " << orig_x << "," << orig_y << endl;
    cout << "Rotation: " << angle << " radians" << endl;
}

virtual void scale(double factor) {
    warning(what_is(), "scale");
    cout << "Scale factor: " << factor << endl;
}

// metodos para descricao textual do objeto
virtual void print() const {
    warning(what_is(), "print");
```

```
}

char* what_is() const { return is_a.value(); }

// metodos para desenho do objeto
virtual void unfill_handles(World *, bool /*show_handle*/) const {
    warning(what_is(), "unfill_handles");
}

virtual void draw_handles(World *, GC, bool /*highlight*/, bool /*show_handle*/) const {
    warning(what_is(), "draw_handles");
}

virtual void draw(World *, GC) const {
    warning(what_is(), "draw");
}

// metodos para teste de selecao (auxilio 'a interface, mantem abstracao)
virtual bool select_test(double x, double y) const {
    warning(what_is(), "select_test");
    cout << "Reference point: (" << x << "," << y << ")" << endl;
    return false;
}

virtual bool select_test(double x1, double y1, double x2, double y2) const {
    warning(what_is(), "select_test");
    cout << "Bounding box: (" << x1 << "," << y1 << ") ("
                              << x2 << "," << y2 << ")" << endl;
    return false;
}

// funcao para calculo de distancia de um ponto ao objeto, utilizado pela
// interface para selecionar objetos, mas tambem util em varios outros casos
virtual double distance_to_point(double x, double y) const {
    warning(what_is(), "distance_to_point");
    cout << "Point is at: (" << x << "," << y << ")" << endl;
    return TOO_FAR;
}

// metodos para query & set de objetos volateis
bool obj_volatile() { return volati; }
bool obj_volatile(bool value) { return volati = value; }

// metodos para InputTools

virtual bool point_inside_implemented () { return false; }
virtual bool point_inside(double x, double y) const {
    warning(what_is(), "point_inside");
    cout << "Point is at:(" << x << "," << y << ")" << endl;
```

```
        return false;
    }

    virtual bool bounding_box_implemented () { return false; }
    virtual void bounding_box (double& x1, double& y1,
                               double& x2, double& y2) const {
        warning(what_is(), "bounding_box");
        cout << "Bounding box: (" << x1 << "," << y1 << ") ("
             << x2 << "," << y2 << ")" << endl;
    }

    virtual bool get_segments_implemented () { return false; }
    virtual list(Obj_ptr) get_segments () const {
        warning(what_is(), "get_segments");
        return 0;
    }

    // Methods for information retrieval.

     virtual bool get_points_implemented() { return false; }
     virtual list(Obj_ptr) get_points() const {
         warning(what_is(), "get_points");
         return 0;
     }

  protected:
    String is_a;
    ObjType stype;
    bool volati;
};

/*
 *
 * GraphicObj
 *
 *
 */

class GraphicObj {
  public:
    // construtores
    GraphicObj() { obj = new Obj(); init(); }
    GraphicObj(Obj *o) { obj = o; init(); }

    // destrutor
    ~GraphicObj() {
        if (gco) {                      // Isto e' um tanto perigoso, mas
          gc_world->FreeGC(gco);        // nao da' problema enquanto nao
          gc_world->FreeGC(gch);        // se tem World's temporarios.
```

```
        }
        delete obj;
    }

    // Metodos para construcao interativa do objeto

    // Como a construcao se da' atraves de um modelo bem restrito de entrada
    // de dados, que sao as acoes do mouse, a criacao interativa de um objeto
    // geometrico deve seguir as fases: "int_start" para iniciar o processo de
    // criacao, onde sao passadas as coordenadas (virtuais) do primeiro click,
    // "int_change" quando o mouse e' arrastado para se acertar a posicao do
    // objeto e "int_end" quando o processo for encerrado. Para aumentar o
    // conjunto possivel de possibilidades destes metodos, existe um
    // parametro opcional (Mode) que pode vir a ser utilizado pela
    // interface para especificar diferentes maneiras de construcao
    // interativa de um mesmo objeto (e.g., circulo dado por centro e
    // raio e circulo inscrito em um quadrado).
    // A semantica do uso destas operacoes nao e' a mesma para todos os
    // objetos devido diferentes necessidades, nao sendo estabelecida pelo
    // ambiente nenhuma norma neste sentido.

    virtual void int_start(World *, double X, double Y, Int_Mode) {
        warning(what_is(), "int_start");
        cout << "X: " << X << "   Y: " << Y << endl;
    }

    virtual void int_change(World *,double X, double Y, Int_Mode) {
        warning(what_is(), "int_change");
        cout << "X: " << X << "   Y: " << Y << endl;
    }

    virtual void int_end(World *,double X, double Y, Int_Mode) {
        warning(what_is(), "int_end");
        cout << "X: " << X << "   Y: " << Y << endl;
    }

    // Metodos para redimensionamento de objetos

    // Na maioria dos objetos (principalmente os objetos simples) o
    // redimensionamento se faz disparando a construcao interativa a
    // partir das dimensoes atuais do objeto (usa-se neste caso
    // dos metodos ja' construidos int_start, int_change e int_end).
    // Em outros casos porem, o redimensionamento nao repete os
    // passos da construcao interativa (por exemplo, quando nao
    // e' disponivel tal construcao) e nestes casos o redimensionamento
    // pode simplesmente afetar partes do objeto - ao gosto do implementador.

    virtual void resize_start(World *, double X, double Y, Int_Mode) {
        warning(what_is(), "resize_start");
```

```
    cout << "X: " << X << "   Y: " << Y << endl;
}

virtual void resize_change(World *, double X, double Y, Int_Mode) {
    warning(what_is(), "resize_change");
    cout << "X: " << X << "   Y: " << Y << endl;
}

virtual void resize_end(World *, double X, double Y, Int_Mode) {
    warning(what_is(), "resize_end");
    cout << "X: " << X << "   Y: " << Y << endl;
}

// metodos auxiliares
virtual void translate(double to_x, double to_y) {
    obj->translate(to_x, to_y);
}

virtual void rotate(double orig_x, double orig_y, double angle) {
    obj->rotate(orig_x, orig_y, angle);
}

virtual void scale(double factor) {
    obj->scale(factor);
}

virtual void unfill_handles(World *w) {
    obj->unfill_handles(w, show_handle);
}

virtual void draw_handles(World *w) {
    obj->draw_handles(w, gch, is_selected, show_handle);
}

virtual void draw(World *w, DrawMode dm);

virtual void get_string(String &s) {s = label;}

virtual void set_string(const String &s) { label = s; }

virtual bool where_draw_string (double &x, double &y) {
    warning(what_is(), "where_draw_string");
    cout << "where: (" << x << "," << y << ")" << endl;
    return false;
}

// metodos para armazenamento secundario
virtual void load(FILE *, World *);
virtual void save(FILE *);
```

```
// metodos para descricao textual do objeto
void print() const { obj->print(); }
char* what_is() const { return is_a.value(); }

// metodos de selecao
virtual void select() {
    is_selected = true;
}

virtual void unselect() {
    is_selected = false;
}

virtual bool selected() {
    return is_selected;
}

virtual bool select_test(double x, double y) {
    return obj->select_test(x, y);
}

virtual bool select_test(double x1, double y1, double x2, double y2) {
    return obj->select_test(x1, y1, x2, y2);
}

virtual void glue() {
    is_glued = true;
}

virtual void unglue() {
    is_glued = false;
}

virtual bool glued() {
    return is_glued;
}

virtual void anchor() {
    is_anchored = true;
}

virtual void unanchor() {
    is_anchored = false;
}

virtual bool anchored() {
    return is_anchored;
}
```

```
virtual double distance_to_point(double x, double y) {
    return obj->distance_to_point(x, y);
}

// metodos para manipula,c~ao de atributos
virtual void set_attributes(World *w, Attributes& attr);

virtual Attributes get_attributes();

// metodos para manipulacao do GC do objeto (auxiliares a varias outras)
virtual void set_handle(bool sh) {
    show_handle = sh;
}

bool handle_is_on() {
    return show_handle;
}

virtual void set_color(World *w, unsigned long fg, unsigned long bg) {
    gc_init(w);
    w->SetForeground(gco, fg); w->SetBackground(gco, bg);
    w->SetForeground(gch, fg); w->SetBackground(gch, bg);
    foreground = fg; background = bg;
}

virtual void get_color(unsigned long &fg, unsigned long &bg) {
    fg = foreground; bg = background;
}

virtual void set_line_attr(World *w, int lw, int ls) {
    gc_init(w);
    w->SetLineAttributes(gco, lw, ls, 0, 0);    // not gch
    line_width = lw; line_style = ls;
}

virtual void get_line_attr(int &lw, int &ls) {
    lw = line_width; ls = line_style;
}

virtual void set_all(World *w, unsigned long fg, unsigned long bg,
                     int lw, int ls, bool sh) {
    set_color(w, fg, bg);
    set_line_attr(w, lw, ls);
    set_handle(sh);
}

virtual GC get_gco() { return gco; }
virtual GC get_gch() { return gch; }
```

```
virtual void set_draw_mode(World *w, DrawMode dm) {
    gc_init(w);

    if (dm == CopyMode) {
      w->SetFunction(gco, GXcopy);
      w->SetFunction(gch, GXcopy);
    }
    else {
      w->SetFunction(gco, GXxor);
      w->SetFunction(gch, GXxor);
    }
}

// funcao para auto-reproducao de um objeto -> necessario quando
// queremos, por exemplo, copiar um objeto na interface e este
// objeto e' externo ao GeoLab.

virtual GraphicObj* reproduct() {
    Obj *reproducted_obj = obj->reproduct();
    if (reproducted_obj) {
      GraphicObj *go = reproducted_obj->new_GraphicObj();
      go->ext_function = ext_function;
      go->ext_module = ext_module;
      go->label = label;
      go->show_handle = show_handle;
      go->foreground = foreground;
      go->background = background;
      go->line_width = line_width;
      go->line_style = line_style;
      if (gco) {
        go->gc_world = gc_world;
        go->gco = gc_world->CreateGC(0, NULL);
        gc_world->CopyGC(gco, GCALL, go->gco);
        go->gch = gc_world->CreateGC(0, NULL);
        gc_world->CopyGC(gch, GCALL, go->gch);
      }
      return go;
    }
    else
      return 0;
}

// suporte para identificacao da origem (onde foi criado) de objetos
// externos, util para recuperacao de objetos de memoria secundaria

ObjType type() { return obj->type(); }

ObjType subtype() { return obj->subtype(); }
```

```
  ObjType subtype(ObjType v) { return obj->subtype(v); }

  void set_external(char *function, char *module) {
      ext_function = function;
      ext_module = module;
  }

  char* get_external_function() const { return ext_function.value(); }

  char* get_external_module() const { return ext_module.value(); }

  // suporte para objetos volateis

  bool obj_volatile() { return obj->obj_volatile(); }
  bool obj_volatile(bool value) { return obj->obj_volatile(value); }

  Obj *obj;

protected:
  void init() {
      is_a = String("Graphic") + String(obj->what_is());
      show_handle = !Default_Obj_Attributes.HANDLES;
      is_glued = !Default_Obj_Attributes.GLUE;
      is_anchored = Default_Obj_Attributes.DELETABLE;
      foreground = Default_Obj_Attributes.FOREGROUND;
      background = Default_Obj_Attributes.BACKGROUND;
      line_width = Default_Obj_Attributes.LINE_WIDTH;
      line_style = Default_Obj_Attributes.LINE_STYLE;
      is_selected = false;
      saved_dm = NoMode;
      gch = gco = 0;
      gc_world = 0;
  }

  void gc_init(World *w) {
      if (!gco) {
        gc_world = w;
        gco = w->CreateGC(0, NULL);
        gch = w->CreateGC(0, NULL);
      }
  }

  String is_a;
  String ext_function, ext_module;
  String label;
  bool show_handle;
  bool is_selected;
  bool is_glued, is_anchored;
  DrawMode saved_dm;
```

```
    unsigned long foreground, background;
    int line_width, line_style;
    GC gco, gch;
    World *gc_world;        // destrutor precisa para poder desalocar GC's
};

#endif "_OBJ_DEFINED;
```

B.6.1.3 Algorithm.h

```
/*
 *
 * Algorithm.h - definicao da superclasse Algorithm, da qual devem descender
 *               todos os algoritmos implementados e introduzidos no GeoLab.
 *               Embora pareca o contrario, o objetivo da hierarquia de algo-
 *               ritmos NAO e' restringir, e sim simplificar o trabalho com
 *               o ambiente tornando mais facil a reutilizacao de codigo,
 *               permitindo uma organizacao logica mais eficiente quanto as
 *               ferramentas de trabalho disponiveis e tambem ajudando na
 *               tarefa de se adequar o comportamento das ferramentas aos
 *               'guidelines' do GeoLab.
 *
 *               Do ponto de vista de reutilizacao externa, a introducao do
 *               conceito de hierarquia de algoritmos nao necessariamente
 *               desvirtua nosso objetivo inicial de proporcionar ferramentas
 *               independentes do ambiente. Na verdade, no ponto de vista do
 *               aplicador (i.e., aquele que usa o ambiente como bancada de
 *               desenvolvimento de ferramentas), tal conceito trata-se apenas
 *               de um conjunto minimo de regras a serem atendidas para a
 *               utilizacao do ambiente, e nao de "exportacao" de ferramentas,
 *               o que depende fortemente das necessidades especificas do
 *               aplicador.
 *
 *
 *      Data    Hora    Nome                    Motivo
 *
 *    29.03.92  13:40   Welson R. Jacometti     Arquivo criado
 *
 */

#ifndef _ALGORITHM_DEFINED
#define _ALGORITHM_DEFINED

#include "global.h"
#include "utilities.h"
#include "basic.h"
#include "List.h"
#include "Obj.h"
```

```
#include "World.h"

/*****************************************************************************

  Para tentar diminuir o problema de nao termos a versao 3.0 do compilador
  C++, todos os metodos virtuais indefinidos em qualquer classe base sera'
  implementado de forma a produzir uma mensagem de erro do tipo:

  Warning: 'object type': 'method' not implemented

  Isso livra-me de ter que ficar contornando o problema da versao 2.0 nao
  permitir heranca sem redefinicao de metodos virtuais por mais de um ni-
  vel (somente uma classe base por hierarquia). Fique  claro  apenas que
  nestes casos, os metodos nao implementados realmente NAO tem  que  ser
  implementados, por tratar-se de uma classe abstrata.

*****************************************************************************/

#ifndef _Obj_ptr_list
#define _Obj_ptr_list
declare(list,Obj_ptr)
#endif ~_Obj_ptr_list;

#ifndef _GraphicObj_ptr_list
#define _GraphicObj_ptr_list
declare(list,GraphicObj_ptr)
#endif ~_GraphicObj_ptr_list;

#ifndef _list_item_list
#define _list_item_list
declare(list,list_item)
#endif ~_list_item_list;

typedef list(Obj_ptr) ResultObjects;
typedef list(GraphicObj_ptr) GraphicObjects;
typedef list(list_item) SelectedObjects;

class Algorithm {
  public:
    // is_able_to_handle() e' usada para determinar se o algoritmo pode ser
    // chamado para o conjunto de objetos selecionados no GeoLab. Isto e' util
    // para manter consistencia em relacao ao funcionamento dos algoritmos e
    // tambem para o GeoLab poder atualizar quais sao as opcoes disponiveis no
    // menu de algoritmos.

    virtual bool is_able_to_handle(const SelectedObjects &, const GraphicObjects &) {
        warning("Algorithm", "is_able_to_handle");
        return false;
    }
```

```
    // animation() indica ao ambiente se o algoritmo possui ou nao qualquer
    // tipo de animacao (mesmo que em niveis inferiores). E' usada para tornar
    // mais facil a selecao de algoritmos quando a opcao de animacao esta'
    // selecionada.

    virtual bool animation() {
        return false;                // no warning, false is default
    }

    // funcao de ligacao entre a implementacao do algoritmo e o ambiente.
    // A tarefa de binder() resume-se a checar (opcionalmente) se o algoritmo
    // esta' apto a manipular a entrada que esta' lhe sendo fornecida, extrair
    // esta entrada em estruturas especificas e tambem realizar conversoes
    // entre diferentes representacoes caso seja necessario.

    virtual ResultObjects binder(const SelectedObjects &, const GraphicObjects &, World *,
                                 int animation_depth = 0, int reference_depth = 0) {
        ResultObjects res;
        warning("Algorithm", "binder");
        cout << "Animation Depth: " << animation_depth << "    "
             << "Reference Depth: " << reference_depth << endl;
        return res;
    }

    // funcao atraves da qual algoritmos externos, notadamente modos de
    // desenho e modos funcionais, sao notificados pelo ambiente que devem
    // encerrar suas atividades em virtude de mudanca de modo de operacao
    // por parte do usuario. Muito util no caso de algoritmos que necessitam
    // operacao ininterrupta (consistencia) ou no caso de algoritmos
    // que realizam preprocessamento (ex: modos funcionais mapeando
    // algoritmos geometricos) e que, ao mudar-se de modo de operacao,
    // devem desalocar memoria, necessitando para isso serem notificados.

    virtual void unbinder(World *) {}

    // funcao de re-instanciacao de tipos criados externamente ao GeoLab
    // e introduzidos por meio de algoritmos geometricos ou de modos de
    // desenho externos. Deve retornar uma instancia do tipo t para que
    // o GeoLab seja capaz de realizar, por exemplo, carga do objeto
    // a partir de memoria secundaria.

    virtual Obj* instantiation(ObjType t) {
        warning("Algorithm", "instantiation");
        cout << "Object Type: " << t << endl;
        return 0;
    }
};
```

```
// Uma vez que sao externos, os objetos componentes da hierarquia de algoritmos
// nao estao instanciados imediatamente apos a sua ligacao com o ambiente.
// Assim sendo, faz-se necessaria uma funcao de ligacao (de fato, somente
// utilizada no momento do link externo) que informe ao GeoLab qual e' a
// instancia a ter seus metodos invocados nas tarefas subsequentes.
// Portanto, a cada algoritmo - digamos "alg" - deve haver uma funcao
// chamada "alg_link()" que retorna um pointer para uma instancia do
// algoritmo. Veja em External.h e nos arquivos de documentacao
// mais detalhes sobre o esquema de ligacao externa.

#endif "_ALGORITHM_DEFINED;
```

Bibliografia

[AC68] D. V. Ahuja e S. A. Coons. Geometry for construction and display. *IBM Syst. J.*, 7:188–205, 1968.

[AHU74] A. V. Aho, J. E. Hopcroft, e J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[Amo92] R. V. Amorim. AnimA – Um ambiente para animação de algoritmos. Tese de Mestrado, DCC - IMECC - UNICAMP, 1992. a ser defendida.

[And78] K. R. Anderson. A reevaluation of an efficient algorithm for determining the convex hull of a finite planar set. *Inform. Process. Lett.*, 7:53–55, 1978.

[And79] A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Inform. Process. Lett.*, 9(5):216–219, 1979.

[Aur91] F. Aurenhammer. Voronoi diagrams: a survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23:345–405, 1991.

[Bac86] M. J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall, Englewood Clifs, NJ, 1986.

[Bau75] B. G. Baumgart. A polyhedron representation for computer vision. Em *Proc. AFIPS Natl. Comput. Conf.*, volume 44, pp. 589–596, 1975.

[Bro79] K. Q. Brown. Voronoi diagrams from convex hulls. *Inform. Process. Lett.*, 9:223–228, 1979.

[Bro87] M. H. Brown. *Algorithm Animation.* ACM Distinguished Dissertations Series. MIT Press, Cambridge, MA, 1987.

[Bro92] M. H. Brown. Zeus: A system for algorithm animation and multi-view editing. Relatório Técnico 75, Digital Systems Research Center, Palo Alto, CA, Fevereiro de 1992.

[BT85] B. K. Bhattacharya e G. T. Toussaint. On geometric algorithms that use the furthest-point Voronoi diagram. Em G. T. Toussaint, editor, *Computational Geometry*, pp. 43–61. North-Holland, Amsterdam, Netherlands, 1985.

[Cav88] P. R. Cavalcanti. Operadores de Euler e sistemas de modelagem geométrica para CAD. Relatório não publicado, Outubro de 1988.

[Cav90] P. R. Cavalcanti. Subdivisões planares. Relatório não publicado, Dezembro de 1990.

[CGL83] B. Chazelle, L. J. Guibas, e D. T. Lee. The power of geometric duality. Em *Proc. 24th Annu. IEEE Sympos. Found. Comput. Sci.*, pp. 217–225, 1983.

[Cha83] B. M. Chazelle. Optimal algorithms for computing depths and layers. Em *Proc. 21st Allerton Conf. Commun. Control Conput.*, pp. 427–436, October de 1983.

[Che90] S. Chern. What is geometry? *The American Mathematical Monthly*, 8(97):679–686, Outubro de 1990.

[Cou85] B. Courington. The unix system: A sun technical report. Relatório técnico, Sun Microsystems, 1985.

[DL87] D. P. Dobkin e M. J. Laszlo. Primitives for the manipulation of three-dimensional subdivisions. Em *Proc. 3rd Annu. ACM Sympos. Comput. Geom.*, pp. 86–99, 1987.

[Dob88] D. P. Dobkin. Computational geometry: then and now. Em R. A. Earnshaw, editor, *Theoretical Foundations of Computer Graphics and CAD*, volume F40 de *NATO ASI*, pp. 71–109. Springer-Verlag, 1988.

[DS89] D. P. Dobkin e D. Silver. Applied computational geometry: Towards robust solutions of basic problems. *J. Comput. Syst. Sci.*, 40:70–87, 1989.

[Ede87] H. Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer-Verlag, Heidelberg, West Germany, 1987.

[EM90] H. Edelsbrunner e E. P. Mücke. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graph.*, 9:66–104, 1990.

[ES85] H. Edelsbrunner e R. Seidel. Voronoi diagrams and arrangements. Em *Proc. 1st Annu. ACM Sympos. Comput. Geom.*, pp. 251–263, 1985.

[EY88] L. W. Ericson e C. K. Yap. The design of LINETOOL, a geometric editor. Em *Proc. 4th Annu. ACM Sympos. Comput. Geom.*, pp. 83–92, 1988.

[FB91] S. Fang e B. Brüderlin. Robustness in geometric modeling tolerance-based methods. *Lecture Notes in Computer Science*, 553:85–102, 1991.

[FD82] J. D. Foley e A. Van Dam. *Fundamentals of Interactive Computer Graphics.* Addison-Wesley, Reading, MA, 1982.

[For87a] A. R. Forrest. Computational Geometry and Software Engineering: Towards a Geometric Computing Environment. Em D. F. Rogers e R. A. Earnshaw, editores, *Techniques for Computer Graphics*, pp. 23–37. Springer-Verlag, 1987.

[For87b] S. J. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[Fur91] C. A. Furuti. Um compilador para uma linguagem de programação orientada a objetos. Tese de Mestrado, DCC - IMECC - UNICAMP, 1991.

[GNU92] GNU. GNU C++ Class Library - GNU Free Software Foundation. Available under communication to gnu@prep.ai.mit.edu, 1992.

[GOP90] K. Gorlen, S. Orlow, e P. Plexico. *Data Abstraction and Object-Oriented Programming in C++*. John Wiley and Sons, 1990. NIH Class Library Rev. 3.0 definition.

[Gra72] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Inform. Process. Lett.*, 1:132–133, 1972.

[GS85] L. J. Guibas e J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Trans. Graph.*, 4:74–123, 1985.

[GSS89] L. J. Guibas, D. Salesin, e J. Stolfi. Epsilon geometry: building robust algorithms from imprecise computations. Em *Proc. 5th Annu. ACM Sympos. Comput. Geom.*, pp. 208–217, 1989.

[GY90] R. Graham e F. Yao. A whirlwind tour of computational geometry. *Amer. Math. Monthly*, 97(8):687–701, 1990.

[Har87] S. Harrington. *Computer Graphics - A Programming Approach*. McGraw-Hill, New York, NY, 1987.

[Hel90] D. Heller. *XView Programming Manual*, volume 7. O'Reilly & Associates, Inc., Sebastapol, CA, 1990.

[HHK88] C. M. Hoffmann, J. E. Hopcroft, e M. S. Karasick. Towards implementing robust geometric computations. Em *Proc. 4th Annu. ACM Sympos. Comput. Geom.*, pp. 106–117, 1988.

[HK92] J. E. Hopcroft e P. J. Kahn. A paradigm for robust geometric algorithms. *Algorithmica*, 7:339–380, 1992.

[Hof89] C. M. Hoffmann. The problem of accuracy and robustness in geometric computation. *Computer*, 22(3):31–42, March de 1989.

[Jac91] W. R. Jacometti. Um ambiente para desenvolvimento de algoritmos em geometria computacional. Proposta de Dissertação de Mestrado – DCC – IMECC – UNICAMP, 1991.

[Jac92] W. R. Jacometti. A note on primitives for the manipulation of general subdivisions and the computation of voronoi diagrams. Relatório Técnico 4, DCC – IMECC – UNICAMP, Campinas, SP, Junho de 1992.

[Jar73] R. A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Inform. Process. Lett.*, 2:18–21, 1973.

[Jon89] J. Jones. *Introduction to the X Window System*. Prentice Hall, Englewood Cliffs, NJ, 1989.

[KMM+90] A. Knight, J. May, M. McAffer, T. Nguyen, e J.-R. Sack. A computational geometry workbench. Em *Proc. 6th Annu. ACM Sympos. Comput. Geom.*, p. 370, 1990.

[Kni90] A. Knight. Implementation of algorithms in a computational geometry workbench. Tese de Mestrado, School of Computer Science, Carleton University, Abril de 1990.

[Knu68] D. E. Knuth. *Fundamental Algorithms*, volume 1 de *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 1968.

[Knu73] D. E. Knuth. *Sorting and Searching*, volume 3 de *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 1973.

[Knu91] D. E. Knuth. Axioms and Hulls. *Lecture Notes in Computer Science*, 606, 1991.

[KP84] B. W. Kernigham e R. Pike. *The Unix Programming Environment*. Prentice-Hall, Englewood Clifs, NJ, 1984.

[KR78] B. W. Kernigham e D. M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, 1978.

[KS86] D. G. Kirkpatrick e R. Seidel. The ultimate planar convex hull algorithm? *SIAM J. Comput.*, 15:287–299, 1986.

[Lee83] D. T. Lee. On finding the convex hull of a simple polygon. *Internat. J. Comput. Inform. Sci.*, 12:87–98, 1983.

[Lip91] S. Lippman. *C++ Primmer*. Addison-Wesley, Reading, MA, 2 edição, 1991.

[LP78] D. T. Lee e F. P. Preparata. The all nearest-neighbor problem for convex polygons. *Inform. Process. Lett.*, 7:189–192, 1978.

[LP79] D. T. Lee e F. P. Preparata. An optimal algorithm for finding the kernel of a polygon. *J. ACM*, 26:415–421, 1979.

[LP84] D. T. Lee e F. P. Preparata. Computational geometry: a survey. *IEEE Trans. Comput.*, C-33:1072–1101, 1984.

[Män88] M. Mäntylä. *An Introduction to Solid Modeling*. Computer Science Press, 1988.

[Man89] U. Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley, Reading, MA, 1989.

[Mic90] Sun Microsystems. *OpenLook – Graphical Users Interface - Aplication Style Guidelines*. Addison-Wesley, Reading, MA, 1990.

[MN89] K. Mehlhorn e S. Näher. LEDA - A Library of Efficient Data Types and Algorithms. *Lecture Notes in Computer Science*, 379:88–106, 1989.

[MS82] M. J. Mäntylä e R. Sulonen. GWB: A solid modeler with Euler operators. *IEEE Comput. Graph. Appl.*, 2(5):17–31, 1982.

[NS79] W. M. Newman e R. F. Sproull. *Principles of Interactive Computer Graphics*. McGraw-Hill, New York, NY, 1979.

[Nye89] A. Nye. *XLib Programming Manual*, volume 1. O'Reilly & Associates, Inc., Sebastapol, CA, 1 edição, 1989.

[Pal87] R. Palkovic. Sunpro: The sun programming environment - a sun technical report. Relatório técnico, Sun Microsystems, 1987.

[PS85] F. P. Preparata e M. I. Shamos. *Computational Geometry: an Introduction.* Springer-Verlag, New York, NY, 1985.

[RA76] D. F. Rogers e J. A. Adams. *Mathematical Elements for Computer Graphics.* McGraw-Hill, New York, NY, 1976.

[Rei72] E. M. Reingold. On the optimality of some set algorithms. *J. ACM*, 19:649–659, 1972.

[Sch90] P. Schorn. An object-oriented workbench for experimental geometric computation. Em *Proc. 2nd Canad. Conf. Comput. Geom.*, pp. 172–175, 1990.

[Sch91] P. Schorn. *Robust Algorithms in a Program Library for Geometric Computation.* Tese de Doutorado, Swiss Federal Institute of Technology (ETH) Zürich, 1991.

[Sed83] R. Sedgewick. *Algorithms.* Addison-Wesley, Reading, MA, 1983.

[Sha75] M. I. Shamos. Computational geometry (notebook). Collection of Problems, Maio de 1975.

[Sha78] M. I. Shamos. *Computational Geometry.* Tese de Doutorado, Dept. Comput. Sci., Yale Univ., 1978.

[Str87] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, MA, 1987.

[Tou85] G. T. Toussaint, editor. *Computational Geometry.* North-Holland, Amsterdam, Netherlands, 1985.

[Tou86] G. T. Toussaint. Computing largest empty circle with location constraints. *Internat. J. Comput. Inform. Sci.*, 12:347–358, 1986.

[TW88] R. E. Tarjan e C. J. Van Wyk. An $O(n \log\log n)$-time algorithm for triangulating a simple polygon. *SIAM J. Comput.*, 17:143–178, 1988. Erratum in 17(1988), 106.

[vLW80] J. van Leeuwen e D. Wood. Dynamization of decomposable searching problems. *Inform. Process. Lett.*, 10:51–56, 1980.

[Wei86] K. J. Weiler. *Topological Structures for Geometric Modeling.* Tese de Doutorado, Rensselaer Polytechnic Institute, Agosto de 1986.

[Xav92] D. T. C. Xavier. Uma modelagem e implementação do princípio de múltiplas visões com reatividade. Tese de Mestrado, DCC - IMECC - UNICAMP, 1992.

[Yao81] A. C. Yao. A lower bound to finding convex hulls. *J. ACM*, 28:780–787, 1981.

[Zlo75] M. M. Zloof. Query by example. Em *Proc. National Comput. Conf.*, Anahein, CA, June de 1975.