Instituto de Computação
Universidade Estadual de Campinas

Computação Exata em Geometria Projetiva Orientada e Tratamento de Degenerações

César Nivaldo Gon
Julho 1996

Instituto de Computação
Universidade Estadual de Campinas

Computação Exata em Geometria Projetiva Orientada e Tratamento de Degenerações

César Nivaldo Gon
Julho 1996

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

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Gon, César Nivaldo
G586c Computação exata em geometria projetiva orientada e tratamento de degenerações / César Nivaldo Gon. -- Campinas, [SP : s.n.], 1996.

Orientador : Pedro Jussieu de Rezende
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Computação gráfica. 2. Visualização. 3. Algoritmos. 4. Perturbação (Matemática). 5. Geometria projetiva. I. Rezende, Pedro Jussieu. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Computação Exata em Geometria Projetiva Orientada e Tratamento de Degenerações

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por César Nivaldo Gon e aprovada pela Banca Examinadora.

Campinas, 30 de julho de 1997.

Prof. Dr. Pedro J. de Rezende
(Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Tese de Mestrado defendida e aprovada em 30 de julho de 1996

pela Banca Examinadora composta pelos Profs. Drs.

Prof (a). Dr (a). GILBERTO CÂMARA NETO

Prof (a). Dr (a). JORGE STOLFI

Prof (a). Dr (a). LUIZ EDUARDO BUZATO

Prof(a). Dr(a). PEDRO JUSSIEU DE REZENDE

A Leone e Sílvia

Abstract

One of the greatest challenges in computational geometry today is to build the bridge between theory and practice which requires tools for the robust implementation of the algorithms that populate the literature. We attempt here to contribute a step in this direction.

In the first part of this thesis, we present an extension to the technique of symbolic perturbation for oriented projective geometry. We describe the implementation of a library based on this technique which consists of geometric primitives sufficient for programming a large class of robust geometric algorithms, using exact arithmetic.

In the second part, we describe the design of GeoPrO: a distributed programming environment for geometric visualization. We present an overview of its classes that allow for easy extensibility and portability. Due to a client-server architecture, comprised of a kernel, with multiple contexts, applications and visualizers, GeoPrO supports distributed execution over a heterogeneous network. Visualizers are currently available for the planar and the spheric models of the oriented projective geometry, running on Silicon Graphics workstations, while another is being implemented in Java for multi-platform support.

Agradecimentos

Este trabalho de pesquisa foi desenvolvido com suporte financeiro parcial de Capes, CNPq, ProTeM-CC — projeto GEOTEC, e Fapesp.

Eu gostaria de expressar a minha gratidão, em primeiro lugar, ao meu orientador Pedro Jussieu de Rezende pela inestimável ajuda na realização deste trabalho.

Eu também agradeço a Ilídio J. Antunes pela valorosa contribuição prestada, o atento revisor Guilherme Albuquerque Pinto e a todos que, de alguma forma, contribuiram para a conclusão desta tese. Acrescento aqui, o meu agradecimento aos membros da banca pelos comentários e sujestões visando o aperfeiçoamento do presente texto.

Ao saudoso amigo Luiz Claudio Rosa, que nos deixou abreviadamente.

Finalmente, eu gostaria de agradecer à minha mãe Leone e à minha futura esposa Sílvia pelo incansável e entusiástico apoio, sem o qual este trabalho não teria se concretizado.

César N. Gon
Julho de 1996

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

Um grupo de pesquisadores ativos (*CG Impact Task Force*) [7] iniciou recentemente um debate sobre o futuro da geometria computacional (GC), focado principalmente na necessidade de que GC se torne uma ferramenta prática para aplicações que requerem computação geométrica.

Existe um longo caminho a percorrer para que os sólidos resultados teóricos e a maturidade dos fundamentos algorítmicos alcançados na área se traduzam em implementações de soluções robustas e eficientes para problemas reais. O desafio é fazer a conexão entre teoria e prática, priorizando o desenvolvimento de aplicações que resolvem problemas através de computação geométrica e não apenas algoritmos para problemas teóricos com ênfase em eficiência assintótica.

Dentro desta ótica, parte do esforço deve ser direcionado à produção de ferramentas voltadas para o desenvolvimento de aplicações práticas que envolvem computação geométrica. É neste contexto que se insere a presente tese.

A primeira parte do presente trabalho (capítulos 3 e 4) apresenta uma extensão da técnica de perturbação simbólica para geometria projetiva orientada. Como forma de validação, descrevemos a implementação de uma biblioteca baseada nesta técnica que consiste de primitivas geométricas suficientes para programação de algoritmos geométricos robustos, usando aritmética exata.

A segunda parte deste trabalho (capítulos 5 e 6) descreve um ambiente de *software* para implementação de algoritmos geométricos. O ambiente GeoPrO foi projetado como uma ferramenta de auxílio ao estudo, desenvolvimento e depuração de aplicações na área de geometria computacional. Além disso, o ambiente foi estruturado de modo a permitir a sua utilização em projetos de outras áreas que realizam algum tipo de computação geométrica e onde podem ser de grande valia recursos de visualização geométrica (p.ex., programação linear, robótica, GIS, etc.).

O capítulo 2 apresenta uma série de programas/bibliotecas destinados à implementação e visualização de algoritmos geométricos. No capítulo 3, apresentamos conceitos preliminares sobre robustez, incluindo: (1) perturbação simbólica para tratamento geral de degenerações, (2) geometria projetiva orientada e (3) computação exata. No capítulo 4, descrevemos a biblioteca GeoPrOLib para implementação de algoritmos geométricos robustos. O capítulo 5 descreve o projeto do ambiente GeoPrO, incluindo os seus objetivos, suas características computacionais e a metodologia de utilização dos recursos providos pelo ambiente. O capítulo 6 apresenta uma breve descrição de possíveis extensões do presente trabalho. Finalmente, as conclusões são apresentadas no capítulo 7.

Capítulo 2

Trabalhos relacionados

GeoLab [11, 1]. Foi desenvolvido como um ambiente de programação para implementação, teste e animação de algoritmos geométricos. GeoLab utiliza bibliotecas compartilhadas de algoritmos e uma abordagem incremental para a composição de novos tipos de objetos geométricos e módulos funcionais externos acessíveis via ligação dinâmica. Foi escrito em C++ e roda sobre a plataforma SunOS/XView.

LEDA [16]. É uma biblioteca de tipos de dados e algoritmos implementada segundo uma abordagem orientada a objetos através de um conjunto de classes C++. Uma pequena parte da biblioteca é dedicada à geometria computacional, com algoritmos para resolução de problemas geométricos no plano euclidiano. LEDA está disponível para as mais diversas plataformas sendo considerada um modelo para o desenvolvimento de bibliotecas de *software*.

GeoSheet [13]. É uma ferramenta para visualização de algoritmos geométricos em ambientes distribuídos. GeoSheet provê visualização interativa de programas para depuração através de uma interface para entrada e saída de objetos geométricos baseada no programa Xfig. Uma aplicação pode utilizar diversas "planilhas" para realizar a visualização remota de objetos geométricos. A implementação foi realizada em C++.

XYZ GeoBench [19, 20]. Foi projetado como um ambiente de programação geométrica, provendo ferramentas para criação, edição, e manipulação de objetos geométricos. A ênfase é na robustez de implementação de algoritmos fundamentais. Ele é composto de uma interface gráfica e uma biblioteca de algoritmos, sendo implementado em *Object Pascal* para Macintosh.

GeomView [15]. É uma interface para visualização e manipulação de objetos geométricos, podendo ser utilizada como um visualizador para objetos estáticos ou para objetos dinamicamente produzidos por outros programas. GeomView roda sobre estações IRIS Silicon Graphics e estações NeXT.

Capítulo 3

Preliminares

A implementação de algoritmos geométricos sobre o espaço euclidiano feita da maneira clássica mostra-se, em geral, bem mais complexa do que o projeto teórico destes algoritmos deixa transparecer.

Para garantir a corretude de um algoritmo, o implementador precisa tratar de maneira consistente os casos degenerados intrínsecos ao problema geométrico e ao algoritmo em questão. Além disso, o programador deve contornar uma série de singularidades causadas pelas limitações do modelo geométrico utilizado. Finalmente, a robustez do algoritmo estará ameaçada por problemas de precisão numérica, cuja análise é bastante complexa e particular para cada algoritmo.

O projeto teórico de algoritmos em geometria computacional tipicamente assume certas hipóteses sobre a entrada, como, por exemplo, que ela está em posição geral. O tratamento dos casos que violam estas hipóteses é tedioso e intrincado. Mesmo quando estes casos são abordados durante a discussão teórica do algoritmo, resta ainda uma tarefa não trivial ao implementador.

Métodos baseados na simulação de uma perturbação conceitual dos dados de entrada têm sido tema de trabalhos recentes na área de geometria computacional [4, 6]. Essa perturbação infinitesimal tem por função eliminar os casos degenerados, fazendo com que qualquer algoritmo projetado sob a hipótese de uma entrada não degenerada funcione para instâncias arbitrárias. Desse modo, essas técnicas eliminam a necessidade de lidar com casos degenerados de maneira particular.

Efetuado o tratamento consistente das degenerações, o projeto de um algoritmo que resolve um determinado problema ainda deverá superar eventuais restrições impostas pelo modelo geométrico utilizado.

A utilização de *coordenadas homogêneas*, que implicitamente define a *geometria projetiva clássica*, é uma tentativa de extensão do espaço euclidiano para tratamento geral das

limitações implícitas a este modelo geométrico, permitindo a simplificação de fórmulas, redução do número de casos particulares, unificação e extensão de conceitos e a utilização do conceito de dualização como ferramenta poderosa para teorização e projeto de algoritmos na área de geometria computacional.

A *geometria projetiva orientada* é um modelo geométrico alternativo proposto por Stolfi [22] com tratamento consistente de linhas e planos orientados, ângulos com sinal, segmentos, direções, conjuntos convexos e muitos outros conceitos que a geometria projetiva clássica não suporta em toda generalidade.

A utilização da geometria projetiva orientada e de técnicas de perturbação simbólica para eliminação de casos degenerados asseguram uma considerável simplificação na implementação dos algoritmos geométricos, mas não garantem a robustez necessária para aplicações que realmente urgem de respostas exatas.

De maneira geral, os modelos teóricos utilizados para formulação de algoritmos geométricos são o *real RAM* e os modelos de árvores de decisão [18], que pressupõem aritmética real exata. Na prática, a implementação destes algoritmos é realizada em algum modelo de precisão fixa, como as aritméticas de inteiros e ponto flutuante, padrões nos sistemas computacionais disponíveis atualmente.

Para conciliar teoria e prática, alguns autores propõem a utilização de *computação exata* [23, 24, 8] para a implementação das primitivas necessárias aos algoritmos geométricos, em detrimento das técnicas *ad hoc* de contorno de problemas de precisão numérica, como a utilização de limiares "epsilon" (geralmente uma constante utilizada nas comparações) que, na melhor da hipóteses, diminuem a probabilidade de falha.

O objetivo desta parte introdutória do trabalho é mostrar a viabilidade da integração consistente e harmoniosa entre geometria projetiva orientada, perturbação simbólica para tratamento de casos degenerados e computação exata, dentro de uma perspectiva voltada para o projeto e implementação de algoritmos em geometria computacional.

3.1 Perturbação simbólica

A implementação de algoritmos em geometria computacional normalmente reserva ao programador a árdua tarefa de prover um tratamento consistente aos vários casos particulares que podem ocorrer.

Como ilustração, tomemos o algoritmo que determina se um ponto p é interior a um polígono simples $\mathcal{P}$ através do cálculo do número de interseções entre uma linha horizontal e as arestas do polígono. Este algoritmo pode ser descrito da seguinte forma: seja r a semi-reta horizontal com limite esquerdo p. Calcule n como sendo o número de interseções entre r e as arestas do polígono $\mathcal{P}$. Se n é impar, p está no interior de $\mathcal{P}$. Caso contrário, p está no exterior de $\mathcal{P}$.

O lado esquerdo da figura 3.1 mostra a ocorrência dos casos que precisam ser tratados consistentemente pelo algoritmo.

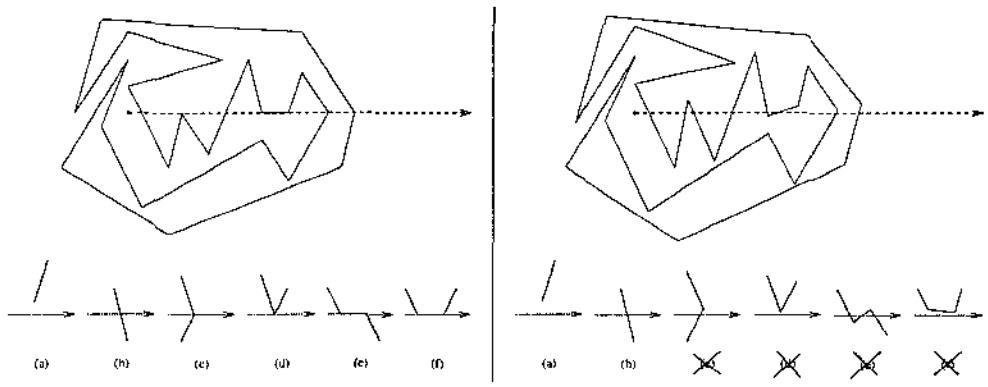

Figura 3.1: Os diferentes casos num algoritmo de localização de pontos (lado esquerdo) e a remoção dos casos degenerados via perturbação simbólica (lado direito).

Edelsbrunner e Mücke [4] propuseram um técnica geral para tratamento de dados de entrada degenerados para algoritmos geométricos, denominada *Simulation of Simplicity* (SoS).

Intuitivamente, SoS simula uma perturbação dos dados de entrada que elimina todos os casos degenerados. É importante frisar que a perturbação nunca é realmente calculada - ela é assumida ser arbitrariamente pequena, mas suficiente para simular a topologia não degenerada. Uma outra interpretação para a técnica é uma maneira geral de efetuar o tratamento dos casos degenerados no nível mais baixo de um algoritmo, ou seja, nos procedimentos que implementam as operações primitivas necessárias.

SoS considera somente primitivas topológicas, ou seja, operações que testam uma dada entrada e classificam-na dentro de um número constante de possíveis casos. Sem perda

de generalidade, a discussão pode ser restrita a primitivas com três possíveis saídas: $+1$, 0, -1, onde 0 indica um caso degenerado.

Se pensarmos em uma operação primitiva como uma função f que mapeia um ponto hiper-dimensional (cujas coordenadas descrevem os objetos de entrada) em $+1$, 0 ou -1, então $f^{-1}(0)$ representa o conjunto das entradas degeneradas. Um requerimento para este conjunto é que a sua medida neste espaço hiper-dimensional seja zero – caso contrário, não seria razoável chamar seus pontos de degenerados. Partindo desta idéia, Edelsbrunner e Mücke fazem uma análise topológica do mapeamento de $f^{-1}(0)$ no espaço nd-dimensional, procurando compreender melhor a natureza dos casos degenerados e, ao mesmo tempo, mostrando porque sempre existe uma perturbação suficientemente pequena que remove todos os casos degenerados.

Uma entrada é *simples* se ela não possui degenerações. A *simplicidade* é simulada pela aplicação de uma particular perturbação a um conjunto $P = \{p_0, p_1, \ldots, p_{n-1}\}$ de n objetos geométricos

$$p_i = (\pi_{i,1}, \pi_{i,2}, \ldots, \pi_{i,d}), \qquad 0 \leq i \leq n-1,$$

cada qual especificado por d parâmetros. É importante que cada objeto tenha um índice único entre 0 e $n-1$. Os objetos estão em posição arbitrária, ou seja, não necessariamente em posição geral. A perturbação de P é realizada substituindo cada parâmetro por um polinômio em ε. Assim, define-se:

$$P(\varepsilon) = \{p_i(\varepsilon) = (\pi_{i,1}(\varepsilon), \pi_{i,2}(\varepsilon), \ldots, \pi_{i,d}(\varepsilon)) \mid 0 \leq i \leq n-1\},$$

onde

$$\pi_{i,j}(\varepsilon) = \pi_{i,j} + \varepsilon(i,j), \qquad 0 \leq i \leq n-1,\ 1 \leq j \leq d,$$

e $\varepsilon(i,j)$ é um polinômio em ε que tende a zero quando ε tende a zero. A escolha do polinômio $\varepsilon(i,j)$ deve ser baseada em três requisitos:

1. A entrada perturbada $P(\varepsilon)$ deve ser simples se $\varepsilon > 0$ é suficientemente pequeno.
2. $P(\varepsilon)$ deve manter todas as propriedades não-degeneradas do conjunto original P.
3. O *overhead* de computação causado pela simulação $P(\varepsilon)$ deve ser desprezível.

O lado direito da figura 3.1 esboça o resultado hipotético da utilização de uma perturbação conceitual dos dados para o algoritmo de localização de pontos descrito anteriormente.

O trabalho de Edelsbrunner e Mücke concentra-se principalmente na primitiva para determinação de qual dos semi-espaços definidos por um hiperplano em $\mathbb{R}^d$ (determinado

pelos pontos $x_{i_1}, x_{i_2}, \ldots, x_{i_d}$) contém um ponto $x_{i_{d+1}}$. O caso degenerado ocorre exatamente quando $x_{i_{d+1}}$ pertence ao hiperplano.

A existência de um apropriado conjunto perturbado de pontos é assegurada através da utilização do polinômio em ε:

$$\varepsilon(i,j) = \varepsilon^{2^{i \cdot d - j}}$$

que, pode-se verificar em [4], satisfaz às condições 1., 2. e 3. acima.

Emiris e Canny [5, 6] estenderam estas idéias e apresentaram um método simples e eficiente para a implementação de quatro primitivas básicas e fundamentais na construção de algoritmos geométricos: (1) ordem de coordenadas, (2) orientação de pontos com respeito a hiperplanos, (3) transversalidade e (4) teste de ponto em esfera. O método é baseado na utilização de uma perturbação linear dada por:

$$\pi_{i,j}(\varepsilon) = \pi_{i,j} + \varepsilon.i^j$$

A ênfase deste trabalho está na eficiência da computação destas primitivas através da avaliação dos sinais de determinantes perturbados da forma $\det(A + \varepsilon.B)$. Esse polinômio é essencialmente o polinômio característico de $A.B^{-1}$, que pode ser computado de maneira bastante eficiente.

Por fim, vale notar que o método de perturbação SoS baseado em curvas polinomiais de grau elevado pode, potencialmente, ser aplicado a um maior número de primitivas e, sob este aspecto, pode ser considerado mais geral que o método de Emiris e Canny [14].

As pesquisas nesta área estão direcionadas para o desenvolvimento de técnicas aplicáveis a um maior número de primitivas geométricas, à redução da complexidade computacional envolvida e ao estudo dos limites de aplicabilidade dessa abordagem.

3.2 Modelo geométrico

3.2.1 Geometria projetiva orientada

O uso de *coordenadas homogêneas com sinal* implicitamente define uma geometria, denominada *geometria projetiva orientada*, para a qual há (pelo menos) dois modelos geométricos convenientes: o modelo esférico e o modelo do *espaço projetivo orientado* $\mathbf{T}^d$ para dimensão d [22, 2].

O modelo esférico

O *modelo esférico* de $\mathbb{T}^2$ consiste de uma esfera unitária $\mathbb{S}^2$ de $\mathbb{R}^3$, que é o conjunto de pontos (x, y, w) de $\mathbb{R}^3$ tais que $x^2 + y^2 + w^2 = 1$.

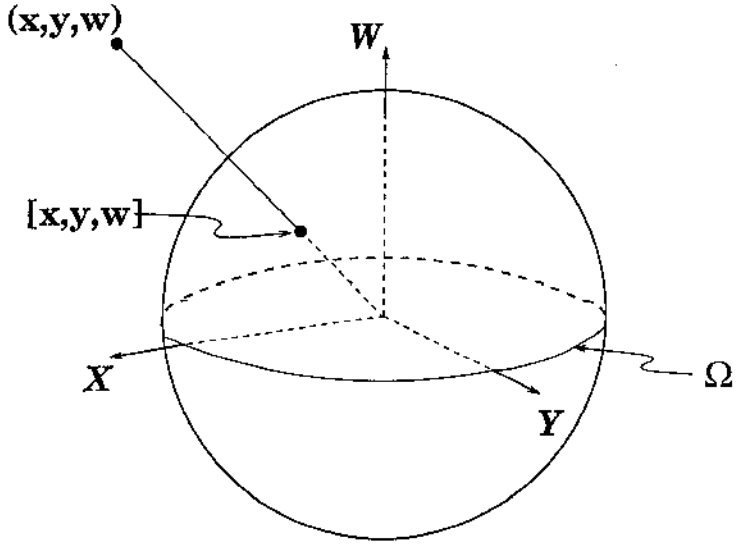

Figura 3.2: Modelo esférico para $\mathbf{T}^2$

Como mostrado na figura 3.2, no modelo esférico, o ponto de $\mathbf{T}^2$ com coordenadas homogêneas com sinal $[x, y, w]$ é representado pela projeção do ponto (x, y, w) de $\mathbb{R}^3$ sobre a esfera $\mathbb{S}^2$, na direção do centro da mesma, ou seja,

$$\frac{(x, y, w)}{\sqrt{x^2 + y^2 + w^2}}$$

Por convenção, os pontos com $w > 0$ estão no *aquém*, ou seja, no hemisfério norte da esfera (assumindo-se norte na direção $w > 0$). Simetricamente, os pontos com $w < 0$ estão no *além*, ou seja, no hemisfério sul da esfera.

Os pontos no infinito de $\mathbb{T}^2$ estão no círculo onde a esfera é cortada pelo plano $w = 0$, ou seja, sobre a linha no infinito Ω, tornando o aquém e o além duas regiões disjuntas separadas por Ω.

Dois pontos p e $\neg p$ são denominados *antípodas* se estão diametralmente opostos na esfera.

Uma linha reta no plano $\mathbb{T}^2$ é definida por três *coeficientes homogêneos* $\langle \mathcal{X}, \mathcal{Y}, \mathcal{W} \rangle$, sendo que um ponto genérico $[x, y, w]$ está nesta linha se, e somente se, $\mathcal{X}x + \mathcal{Y}y + \mathcal{W}w = 0$. Esta equação define um plano de $\mathbb{R}^3$ que passa pela origem e, portanto, corta a esfera num círculo de raio máximo. Desse modo, uma reta de $\mathbb{T}^2$ corresponde a um círculo máximo de $\mathbb{S}^2$.

O modelo plano

O modelo do plano projetivo orientado $\mathbf{T}^2$ consiste de duas cópias de $\mathbb{R}^2$ (o aquém e o além) e de um ponto no infinito $d\infty$ para toda direção d em $\mathbb{R}^2$.

A figura 3.3 apresenta uma reta na direção d no modelo $\mathbb{T}^2$, ou seja, duas cópias de uma reta em $\mathbb{R}^2$ e dois pontos no infinito $+d\infty$ e $-d\infty$.

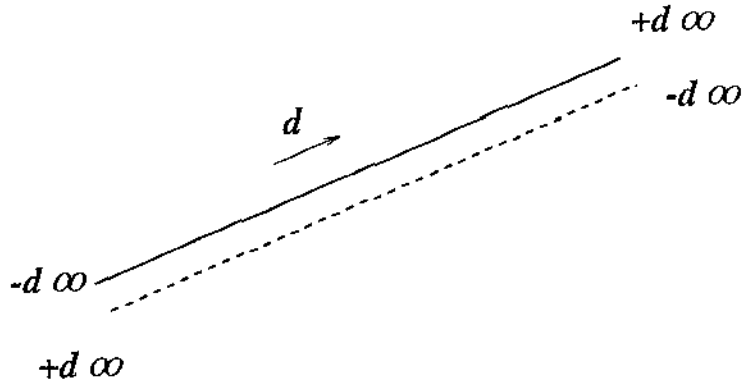

Figura 3.3: Uma linha reta em $\mathbf{T}^2$

A geometria projetiva clássica, como extensão da geometria euclidiana, introduz uma série de benefícios, como a simplificação de fórmulas, a redução do número de casos particulares, a unificação e extensão de conceitos e a utilização do conceito de dualização como ferramenta poderosa para teorização e projeto de algoritmos na área de geometria computacional.

A geometria projetiva orientada ratifica e amplia estas vantagens através do tratamento consistente de linhas e planos orientados, ângulos com sinal, segmentos, direções, conjuntos convexos e muitos outros conceitos que a geometria projetiva clássica não suporta em toda generalidade (veja [22]).

3.3 Geometria projetiva orientada com perturbação simbólica

3.3.1 Orientação de pontos em coordenadas homogêneas

No artigo de introdução da técnica *Simulation of Simplicity* (SoS) [4], H. Edelsbrunner e E. P. Mücke escolheram pontos no espaço euclidiano d-dimensional para um estudo mais detalhado da aplicação de SoS. Além disso, a discussão sobre a utilização de perturbação simbólica com coordenadas homogêneas é restrita às primitivas baseadas no conceito de orientação de pontos *finitos*.

Conforme apresentado em [4], suponha que desejamos determinar a orientação de uma seqüência de d pontos finitos e um ponto p no infinito. Um ponto genérico d-dimensional em coordenadas homogêneas pode ser representado por $d+1$ coordenadas, ou seja,

$$p = (\pi_1, \pi_2, \ldots, \pi_d, \pi_{d+1})$$

p é considerado um ponto *no infinito* se, e somente se, $\pi_{d+1} = 0$ e $\pi_i \neq 0$ para algum $i \leq d$. Desse modo, podemos ver p como o limite de

$$p(\delta) = (\pi_1, \pi_2, \ldots, \pi_d, \delta)$$

quando δ tente a zero.

Se usarmos $p(\delta)$ no lugar de p, teremos orientações distintas para $\delta > 0$ ou $\delta < 0$, mostrando a impossibilidade de generalizar-se a noção de orientação a pontos no infinito.

Em termos mais gerais, a mudança de coordenadas cartesianas para coordenadas homogêneas, significa abandonar o espaço euclidiano e utilizar um espaço geométrico estritamente maior, representado pela geometria projetiva clássica.

Apesar de todos os benefícios introduzidos pela geometria projetiva clássica, uma desvantagem é a inexistência de uma definição consistente para o conceito de orientação, largamente utilizado em geometria computacional. Isto decorre do fato do plano projetivo clássico não ser orientável, ou seja, não há nenhuma maneira de se definir sentido "horário" e sentido "anti-horário" consistentemente para todo o plano $\mathbb{P}^2$. Dada uma seqüência de três pontos quaisquer em posição geral sobre o plano $\mathbb{P}^2$, a "quina" determinada por esta seqüência pode ser continuamente transportada sobre o plano projetivo de tal modo que ela retorne à sua posição original, porém com o seu sentido invertido.

Finalmente, utilizar a geometria projetiva clássica (por meio de coordenadas homogêneas inteiras) sem um tratamento geral para pontos no infinito implica na mera utilização da geometria euclidiana com coordenadas racionais, onde todo o tratamento de pontos no infinito tem que ser feito de maneira particular, ou seja, caso a caso. Veremos a seguir que em $\mathbb{T}^2$ essa limitação não ocorre.

3.3.2 Orientação de pontos em geometria projetiva orientada

A geometria projetiva orientada permite a definição do conceito de orientação em toda a sua generalidade, sem a necessidade de qualquer tratamento especial para pontos no infinito e, conseqüentemente, permite a utilização de perturbação simbólica para tratamento de degeneração para todos os pontos do espaço orientado $\mathbb{T}^d$.

Para decidir a orientação de uma seqüência de $d+1$ pontos em coordenadas homogêneas, define-se:

$$\Delta = \begin{pmatrix} \pi_{i_0,1} & \pi_{i_0,2} & \cdots & \pi_{i_0,d+1} \\ \pi_{i_1,1} & \pi_{i_1,2} & \cdots & \pi_{i_1,d+1} \\ \vdots & \vdots & \ddots & \vdots \\ \pi_{i_d,1} & \pi_{i_d,2} & \cdots & \pi_{i_d,d+1} \end{pmatrix}$$

e a matriz perturbada:

$$\Delta(\varepsilon) = \begin{pmatrix} \pi_{i_0,1} + \varepsilon(i_0,1) & \pi_{i_0,2} + \varepsilon(i_0,2) & \cdots & \pi_{i_0,d+1} + \varepsilon(i_0,d+1) \\ \pi_{i_1,1} + \varepsilon(i_1,1) & \pi_{i_1,2} + \varepsilon(i_1,2) & \cdots & \pi_{i_1,d+1} + \varepsilon(i_1,d+1) \\ \vdots & \vdots & \ddots & \vdots \\ \pi_{i_d,1} + \varepsilon(i_d,1) & \pi_{i_d,2} + \varepsilon(i_d,2) & \cdots & \pi_{i_d,d+1} + \varepsilon(i_d,d+1) \end{pmatrix}$$

Espaço projetivo clássico

Para o espaço projetivo clássico, a orientação de uma seqüência $(p_{i_0}, p_{i_1}, \ldots, p_{i_d})$ de pontos com $p_{i_v} = (\pi_{i_v,1}, \pi_{i_v,2}, \ldots, \pi_{i_v,d+1})$ é *positiva* se:

$$sign(\det(\Delta(\varepsilon))) = \prod_{v=0}^{d} sign(\pi_{i_v,d+1}(\varepsilon))$$

O lado direito da fórmula acima, é o ajuste para que a noção de orientação "euclidiana" seja estendida para os pontos finitos do espaço projetivo clássico. Porém, como já foi dito, esta definição não pode ser utilizada para pontos no infinito.

Espaço projetivo orientado

No espaço projetivo orientado $\mathbb{T}^d$, a orientação de uma seqüência $(p_{i_0}, p_{i_1}, \ldots, p_{i_d})$ de pontos com $p_{i_v} = (\pi_{i_v,1}, \pi_{i_v,2}, \ldots, \pi_{i_v,d+1})$ é *positiva* se:

$$\det \Delta(\varepsilon) > 0$$

e *negativa* caso contrário (já que $\det \Delta(\varepsilon)$ nunca é zero). Esta definição não é restrita a pontos finitos, podendo também ser utilizada com pontos no infinito.

Portanto, a utilização de geometria projetiva orientada com perturbação simbólica resolve o problema da generalização da noção de orientação para pontos no infinito, apresentado em [4].

3.4 Computação exata

Algoritmos geométricos são normalmente descritos sob aritmética exata com números reais (modelo *real RAM*). Como os sistemas computacionais não provêem aritmética exata para reais, a implementação desses algoritmos é feita utilizando-se algum método alternativo. A aritmética de ponto flutuante é uma substituição bastante comum e conveniente, não havendo, porém, nenhuma técnica simples que garanta a confiabilidade dos programas resultantes.

L. Guibas, D. Salesin e J. Stolfi propuseram uma técnica, denominada *Epsilon Geometry*, para tratamento de erros computacionais em algoritmos geométricos oriundos do uso de aritmética de precisão finita [10]. O método proposto combina técnicas de aritmética intervalar e análise de erro regressiva para calcular a solução exata de uma versão da entrada perturbada de maneira não infinitesimal.

A utilização de *computação exata* [23] é uma alternativa para a implementação de algoritmos geométricos *provadamente* robustos. A aritmética de inteiros é suficiente para muitos algoritmos geométricos. Além disso, a utilização de coordenadas homogêneas inteiras implicitamente estende o domínio dos dados para os números racionais, e raramente números fora deste domínio são de fato necessários.

O paradigma da computação exata garante que:

1. todos os objetos (matematicamente descritos) são representados de maneira exata;

2. todas as decisões condicionais que definem o fluxo de execução do programa são livres de erro.

Assim, temos que *aritmética de precisão múltipla*[1] claramente satisfaz 1., mas não garante 2., ou seja, ela é uma condição necessária mas não suficiente para garantir que um método computacional é exato.

O preço a ser pago pela utilização de computação exata é perda de desempenho, mas isto é inevitável para aplicações onde somente respostas (realmente) exatas fazem sentido. Além disso, este também é um dos motivos pelos quais a utilização de computação exata sempre foi descartada como alternativa para a solução de problemas de robustez em aplicações onde o tempo de execução é considerado crítico. Diversos trabalhos têm sido apresentados objetivando melhorar a eficiência dos métodos de computação exata, alguns com resultados certamente expressivos [8, 24].

[1] Do Inglês: *multiprecision arithmetic.*

Capítulo 4

A biblioteca GeoPrOLib

GeoPrOLib é uma biblioteca de primitivas geométricas para produção de aplicações sobre o plano projetivo orientado, utilizando perturbação simbólica para tratamento geral dos casos degenerados e aritmética exata para garantir robutez às operações efetuadas.

As principais características da biblioteca são:

- GeoPrOLib é implementada por uma família de classes C++, podendo ser utilizada praticamente com qualquer compilador C++ (p. ex., Sun, Gnu, AIX e HPUX).
- Os objetos básicos possuem uma especificação consistente e suficientemente abstrata para permitir que a utilização funcional de um objeto independa dos detalhes internos de sua implementação.
- Os objetos são hierarquizados de forma a permitir que derivações e *templates* estendam e especializem a biblioteca de acordo com as necessidades do usuário.
- O conjunto de primitivas implementadas é o descrito em [4].
- O escopo é restrito aos números inteiros e racionais, através da utilização de coordenadas homogêneas inteiras com sinal.

4.1 Primitivas

Para detalhes sobre estas primitivas, o leitor pode consultar [4].

Ordenação. Decide a ordem entre duas quantidades expressas por coordenadas dos pontos de entrada. O caso degenerado ocorre quando estas quantidades coincidem. Para o plano projetivo orientado temos:

$PredX(p_i; p_j) = verdadeiro$ se, e somente se,

$$\begin{vmatrix} \pi_{i,1} + \varepsilon(i,1) & \pi_{i,3} + \varepsilon(i,3) \\ \pi_{j,1} + \varepsilon(j,1) & \pi_{j,3} + \varepsilon(j,3) \end{vmatrix} < 0$$

$PredY(p_i; p_j) = verdadeiro$ se, e somente se,

$$\begin{vmatrix} \pi_{i,2} + \varepsilon(i,2) & \pi_{i,3} + \varepsilon(i,3) \\ \pi_{j,2} + \varepsilon(j,2) & \pi_{j,3} + \varepsilon(j,3) \end{vmatrix} < 0$$

Orientação e Transversalidade. Determina a orientação de 3 pontos em $\mathbb{T}^2$, ou seja, decide qual dos semi-planos definidos por uma reta em $\mathbf{T}^2$ (dada pelos pontos p_i, p_j) contém um ponto p_k. O caso degenerado ocorre exatamente quando p_k pertence à reta. No contexto dual, a mesma primitiva decide em qual dos semi-planos definidos pela reta $p_i^\perp$ está a interseção entre as retas $p_j^\perp, p_k^\perp$. O caso degenerado ocorre exatamente quando esta interseção pertence à reta $p_i^\perp$. Para o plano projetivo orientado, temos:

$Positive(p_i; p_j; p_k) = verdadeiro$ se, e somente se,

$$\begin{vmatrix} \pi_{i,1} + \varepsilon(i,1) & \pi_{i,2} + \varepsilon(i,2) & \pi_{i,3} + \varepsilon(i,3) \\ \pi_{j,1} + \varepsilon(j,1) & \pi_{j,2} + \varepsilon(j,2) & \pi_{j,3} + \varepsilon(j,3) \\ \pi_{k,1} + \varepsilon(k,1) & \pi_{k,2} + \varepsilon(k,2) & \pi_{k,3} + \varepsilon(k,3) \end{vmatrix} > 0$$

Teste de interior de esfera. Decide, dados 4 pontos (p_i, p_j, p_k, p_l), se o ponto p_l é interior à esfera definida por p_i, p_j, p_k em $\mathbb{T}^2$.

$InSphere(p_i; p_j; p_k; p_l) = verdadeiro$ se, e somente se,

$$\det \Delta_2(\varepsilon) \cdot \det \Delta_3(\varepsilon) > 0$$

onde:

$$\det \Delta_2(\varepsilon) = \begin{vmatrix} \pi_{i,1} + \varepsilon(i,1) & \pi_{i,2} + \varepsilon(i,2) & \pi_{i,3} + \varepsilon(i,3) \\ \pi_{j,1} + \varepsilon(j,1) & \pi_{j,2} + \varepsilon(j,2) & \pi_{j,3} + \varepsilon(j,3) \\ \pi_{k,1} + \varepsilon(k,1) & \pi_{k,2} + \varepsilon(k,2) & \pi_{k,3} + \varepsilon(k,3) \end{vmatrix}$$

$$\det \Delta_3(\varepsilon) = \begin{vmatrix} \pi_{i,1} + \varepsilon(i,1) & \pi_{i,2} + \varepsilon(i,2) & \pi_{i,3} + \varepsilon(i,3) & \frac{\pi_{i,1}^2 + \pi_{i,2}^2}{\pi_{i,3}} + \varepsilon(i,4) \\ \pi_{j,1} + \varepsilon(j,1) & \pi_{j,2} + \varepsilon(j,2) & \pi_{j,3} + \varepsilon(j,3) & \frac{\pi_{j,1}^2 + \pi_{j,2}^2}{\pi_{j,3}} + \varepsilon(j,4) \\ \pi_{k,1} + \varepsilon(k,1) & \pi_{k,2} + \varepsilon(k,2) & \pi_{k,3} + \varepsilon(k,3) & \frac{\pi_{k,1}^2 + \pi_{k,2}^2}{\pi_{k,3}} + \varepsilon(k,4) \\ \pi_{l,1} + \varepsilon(l,1) & \pi_{l,2} + \varepsilon(l,2) & \pi_{l,3} + \varepsilon(l,3) & \frac{\pi_{l,1}^2 + \pi_{l,2}^2}{\pi_{l,3}} + \varepsilon(l,4) \end{vmatrix}$$

Capítulo 5

O ambiente GeoPrO

Este capítulo descreve o projeto do ambiente GeoPrO, incluindo os objetivos do projeto, suas características computacionais e a metodologia de utilização dos recursos providos pelo ambiente.

GeoPrO é um ambiente de *software* para implementação de algoritmos geométricos. O ambiente GeoPrO foi projetado como uma ferramenta de auxílio ao estudo, desenvolvimento e depuração de aplicações na área de geometria computacional. Além disso, o ambiente foi estruturado de modo a permitir a sua utilização em projetos de outras áreas que realizam algum tipo de computação geométrica e onde podem ser de grande valia recursos de visualização geométrica (p.ex., programação linear, robótica, GIS, etc.).

A arquitetura aberta[1] do ambiente é o primeiro passo no sentido de permitir a sua evolução através de contribuições dos próprios usuários do sistema. Além disso, serão criados canais de comunicação via Internet de modo a permitir uma maior interação entre os usuários, incluindo um repositório (acessível via *WWW*) para contribuições dos usuários e distribuição de novas versões do GeoPrO.

[1] No sentido de arquitetura concebida para ser extensível

5.1 Por que um ambiente de visualização geométrica?

Tradicionalmente, algoritmos geométricos manipulam objetos dotados de descrições e funcionalidades geométricas. Por exemplo, um objeto que representa uma subdivisão planar pode ter associado a ele, além de sua descrição geométrica (vértices, faces e arestas), métodos que implementam funcionalidades como a localização de pontos, percursos, etc.

Por outro lado, é raro encontrarmos métodos para visualização ou entrada destes objetos através de uma interface gráfica. A implementação *ad-hoc* dessas funcionalidades é normalmente tediosa e específica para cada plataforma, o que obriga o implementador a desviar sua atenção para detalhes relacionados ao dispositivo gráfico, à definição de uma interface com o usuário, etc.

A utilização de programas dedicados à visualização geométrica [1, 20, 15] também apresenta uma série de limitações, como a dificuldade de integração entre a aplicação do usuário e o programa de visualização. Além disso, estes programas normalmente estão disponíveis para apenas uma plataforma e/ou linguagem de programação e os recursos de visualização e entrada de objetos não são facilmente estendíveis para incorporar características que atendam às necessidades específicas das aplicações.

O ambiente GeoPrO foi desenvolvido como um ambiente aberto tendo como meta os seguintes requisitos:

- *Facilidade de integração entre as aplicações e o ambiente.* A utilização de uma abordagem cliente-servidor, possibilita que, em um ambiente de rede[2], *qualquer aplicação* possa facilmente utilizar os recursos de visualização do GeoPrO.

- *Reaproveitamento de código.* Isto é conseguido através de uma metodologia, baseada no paradigma da programação orientada a objetos, de integração entre classes geométricas existentes e o ambiente GeoPrO, descrita na seção 5.4.1.

- *Abstração de dados.* A interação entre os objetos da aplicação e o ambiente é feita através da utilização de um protocolo de descrição geométrica (descrito na seção 5.2.1) que deixa transparente ao programador detalhes não relacionados ao algoritmo em questão.

- *Extensibilidade.* O ambiente foi concebido para ser estendido de acordo com as necessidades do usuário. A idéia é prover um conjunto mínimo de ferramentas para que o usuário-programador derive soluções de visualização e entrada de dados cada vez mais sofisticadas.

[2]A versão atual do ambiente é baseada no protocolo TCP/IP.

5.2 A estrutura básica do ambiente GeoPrO

Em um alto nível de abstração, o ambiente consiste de três partes básicas: o *núcleo* do GeoPrO, os *visualizadores geométricos* e as *aplicações* do usuário. Uma aplicação pode ser a implementação de um algoritmo geométrico ou um módulo de um sistema maior para o qual se deseje algum tipo de visualização geométrica (veja figura 5.1).

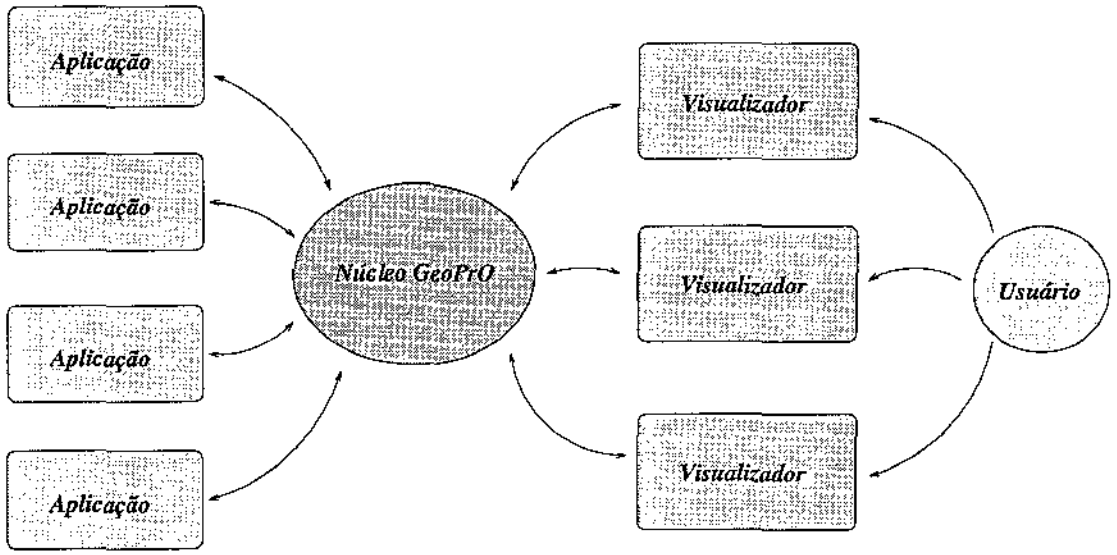

Figura 5.1: Componentes do ambiente GeoPrO

Quando uma aplicação deseja usar as facilidades do ambiente, ela conecta-se ao núcleo do GeoPrO, tornando-se um novo *cliente*. Então, esta aplicação passa a ter à disposição uma série de recursos para realizar a visualização de objetos geométricos ou para obter entradas de dados específicas do usuário. Através desses recursos, o algoritmo pode enviar requisições para um conjunto de visualizadores que são responsáveis pelo desenho dos objetos e por oferecer uma interface gráfica amigável para que o usuário possa manipulá-los ou atender a um pedido de entrada de dados. O usuário pode optar por usar um visualizador já incluído no ambiente GeoPrO ou por implementar um novo visualizador especialmente projetado para a sua aplicação. Detalhes sobre o funcionamento das diversas partes do ambiente e a integração entre elas serão vistos em seções adiante.

Operação em rede GeoPrO foi projetado para trabalhar eficientemente em um ambiente de rede. O formato de transmissão dos comandos das aplicações para o núcleo e deste para os visualizadores é definido de modo a ser independente da plataforma de *hardware*. Assim, o núcleo, as aplicações e os visualizadores podem estar rodando distribuídos

em máquinas de uma rede heterogênea. Por exemplo, o usuário pode rodar os seus algoritmos em uma máquina paralela e ver o resultado geométrico ser mostrado em um visualizador rodando em uma estação Silicon Graphics, com o núcleo sendo executado em uma terceira plataforma.

As seções seguintes detalham a estrutura básica do ambiente GeoPrO.

5.2.1 Objetos geométricos primitivos

Existe uma grande gama de algoritmos geométricos para o plano que lidam com objetos razoavelmente complexos como triangularizações, diagramas de Voronoi, árvores espalhadas, etc. Contudo, em geral estes objetos podem ser construídos a partir de um pequeno número de objetos geométricos primitivos.

GeoPrO provê um conjunto básico de objetos geométricos, e todas as operações de desenho e de entrada de dados são feitas através desse conjunto de objetos. Desse modo, qualquer objeto descrito em termos desses objetos primitivos pode interagir com o ambiente. Assim, o usuário é encorajado a escrever as suas próprias bibliotecas de objetos mais complexos, descrevendo-os a partir dos objetos básicos do GeoPrO.

Os objetos primitivos são:

- *pontos* - uma lista de um ou mais pontos,
- *linhas* - uma lista de uma ou mais retas,
- *segmentos* - uma lista de um ou mais pares de vértices interpretados como segmentos de retas,
- *círculos* - uma lista de um ou mais pares de vértices interpretados como centro e ponto na borda do círculo,
- *linha poligonal* - uma seqüência de vértices descrevendo uma linha poligonal (possivelmente aberta),
- *polígono* - uma seqüência de vértices descrevendo um polígono.

Esses objetos básicos são hierarquizados da forma descrita na figura 5.2.
A partir da superclasse *NetObj*, são derivadas três subclasses:

- *NetSingle:* classe abstrata que representa os objetos descritos por uma seqüência de vértices, ou seja, *NetPoints*, *NetLines*, *NetPolygonalLine* e *NetPolygon*. Cada vértice é representado por uma instância da classe *Coord*, ou seja, uma coordenada homogênea com sinal.

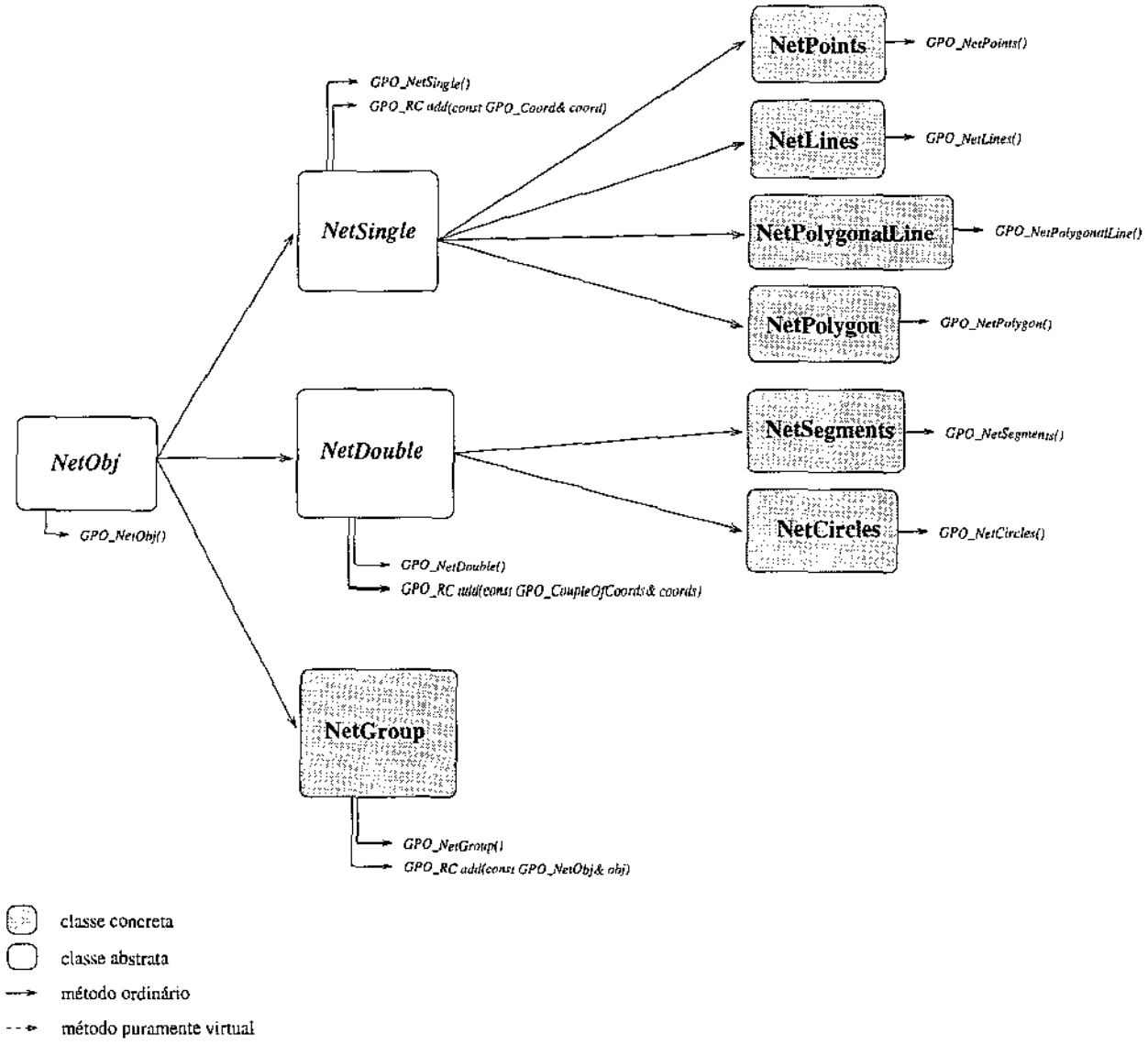

Figura 5.2: Hierarquia dos objetos geométricos primitivos

- *NetDouble:* classe abstrata que representa os objetos descritos por uma seqüência de pares de vértices, ou seja, *NetSegments* e *NetCircles*. Cada par de vértices é representado por uma instância da classe *CoupleOfCoords*, ou seja, um par de coordenadas homogêneas com sinal.

- *NetGroup:* representa a entidade de agrupamento de objetos. Uma instância dessa classe descreve uma seqüência de instâncias da superclasse *NetObj* ou de suas classes derivadas.

Esse conjunto de objetos básicos define o que chamaremos de *protocolo de descrição geométrica* (PDG) e toda descrição geométrica dentro do ambiente GeoPrO é feita através deste protocolo.

5.3 O Núcleo do GeoPrO

Uma aplicação do usuário pode enviar *comandos* para os visualizadores, que podem ser ações ou requisições. As primeiras determinam mudanças nos objetos mostrados e as últimas representam pedidos de novos objetos.

O *núcleo* do GeoPrO pode ser visto como uma interface entre a aplicação do usuário e os visualizadores. O principal objetivo do núcleo é rotear os comandos da aplicação para os visualizadores. Além disso, ele também é responsável por manter toda informação sobre o conjunto de objetos geométricos já criados (e não removidos) pelas aplicações. Esses objetos são agrupados de maneira disjunta em *contextos*. Cada aplicação está associada a um único contexto do núcleo, que compreende o escopo dos objetos "visíveis" para aquela aplicação. Um visualizador também tem o seu escopo restrito a um único contexto.

A idéia de múltiplos contextos é permitir que diversos grupos de aplicações e visualizadores compartilhem os recursos de um único núcleo sem que haja nenhum tipo de interferência entre eles. Para isso, basta que estes grupos se associem a diferentes contextos. O suporte a múltiplos contextos é discutido na seção 5.6.1.

5.3.1 O funcionamento do núcleo

Esta seção apresenta uma visão geral do modo de funcionamento do núcleo. Detalhes sobre a implementação das classes que compõem o núcleo podem ser obtidos no apêndice B.

Internamente, o núcleo do GeoPrO pode ser visto como uma máquina de estados que responde a eventos de oito tipos distintos.

Os eventos gerados pelas aplicações são:

- *Cadastramento de aplicações*
- *Inserção de objetos geométricos*
- *Remoção de objetos geométricos*
- *Requisição de objetos geométricos*
- *Descadastramento de aplicações*

Os eventos gerados pelos visualizadores são:

- *Cadastramento de visualizadores*
- *Resposta à requisição de objetos geométricos*
- *Descadastramento de visualizadores*

Cadastramento de aplicações

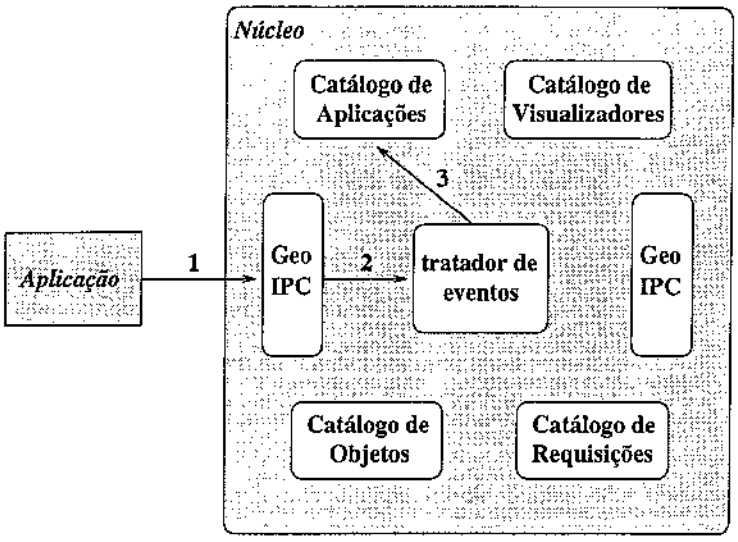

Figura 5.3: A operação de cadastramento de uma aplicação

Este evento representa a operação de cadastramento de uma aplicação a um determinado contexto do núcleo.

Os índices da figura 5.3 representam as seguintes ações:

1. A aplicação envia uma mensagem de cadastramento para o núcleo, especificando o seu *nome* e o *contexto* a que deseja conectar-se.

2. O tratador de eventos do núcleo reconhece esta mensagem como um evento válido.

3. O núcleo cadastra esta aplicação, que passa a dispor dos recursos do ambiente GeoPrO. Se o contexto ao qual esta aplicação está associada ainda não existe, ele é criado.

Inserção de objetos geométricos

Este evento permite a inserção de objetos em um contexto do núcleo. Quando um objeto é inserido no núcleo ele se torna *persistente*[3], passando a ter um identificador único. A partir de então ele será mostrado por todos os visualizadores cadastrados àquele contexto. A persistência dos objetos de visualização é discutida na seção 5.6.1.

Os índices da figura 5.4 representam as seguintes ações:

[3]O termo *persistente* não está sendo usado com a conotação normalmente assumida em tópicos relacionados à área de banco de dados, mas sim com o significado que o texto sugere.

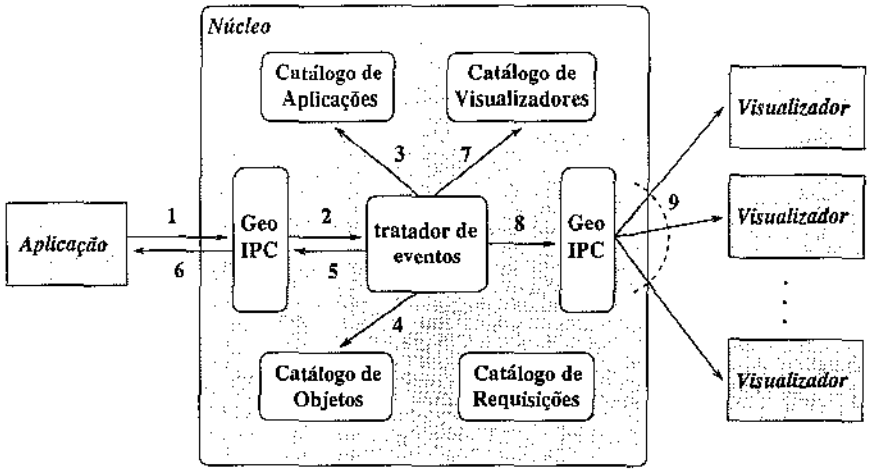

Figura 5.4: A operação de inserção de objetos geométricos

1. A aplicação envia uma mensagem ao núcleo contendo a descrição do objeto a ser inserido no contexto associado a ela.

2. O tratador de eventos do núcleo reconhece esta mensagem como um evento válido.

3. Através de uma consulta ao catálogo de aplicações, é identificado o contexto ao qual a aplicação está associada, que passa a ser o *contexto corrente.*

4. O objeto recebe um identificador único e é inserido no catálogo de objetos geométricos do núcleo, sendo associado ao contexto corrente.

5. O núcleo prepara uma mensagem de resposta à aplicação contendo o identificador único atribuído ao objeto.

6. A aplicação recebe esse identificador e associa-o ao objeto em questão. Maiores detalhes sobre o funcionamento das aplicações serão vistos na seção 5.4.

7. O catálogo de visualizadores é consultado para obtenção do conjunto de visualizadores associados ao contexto corrente.

8. O núcleo prepara uma mensagem para divulgar o novo objeto a todos os visualizadores associados ao contexto corrente.

9. Essa mensagem é enviada aos visualizadores, que passam a mostrar o novo objeto.

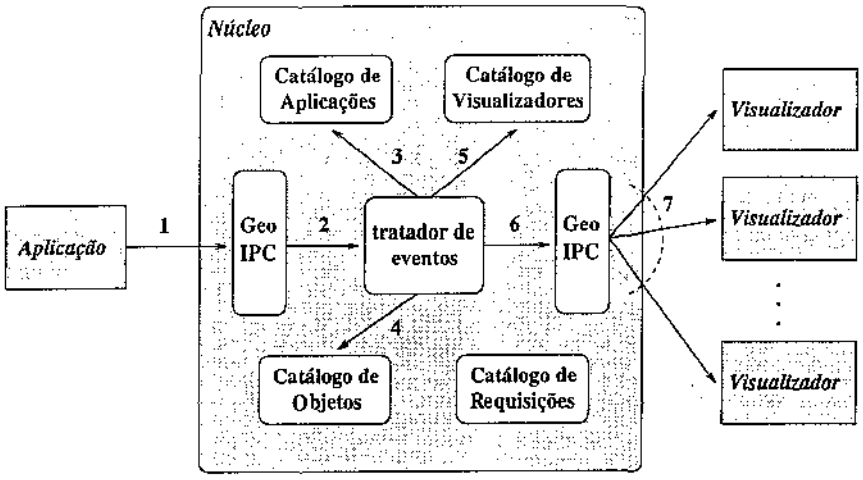

Figura 5.5: A operação de remoção de objetos geométricos

Remoção de objetos geométricos

O evento de remoção de objetos permite que uma aplicação remova um objeto do contexto associado a ela no núcleo. Quando um objeto é removido do núcleo, ele deixa de ser mostrado nos visualizadores cadastrados àquele contexto.

Os índices da figura 5.5 representam as seguintes ações:

1. A aplicação envia uma mensagem ao núcleo contendo a identificação do objeto a ser removido do contexto associado a ela no núcleo.

2. O tratador de eventos do núcleo reconhece esta mensagem como um evento válido.

3. Através de uma consulta ao catálogo de aplicações, é identificado o contexto ao qual a aplicação está associada, que passa a ser o *contexto corrente*.

4. O objeto é removido do catálogo de objetos geométricos do núcleo.

5. O catálogo de visualizadores é consultado para obtenção do conjunto de visualizadores associados ao contexto corrente.

6. O núcleo prepara uma mensagem para divulgar a remoção do objeto do núcleo a todos os visualizadores associados ao contexto corrente.

7. Essa mensagem é enviada aos visualizadores, que deixam de mostrar o objeto em questão.

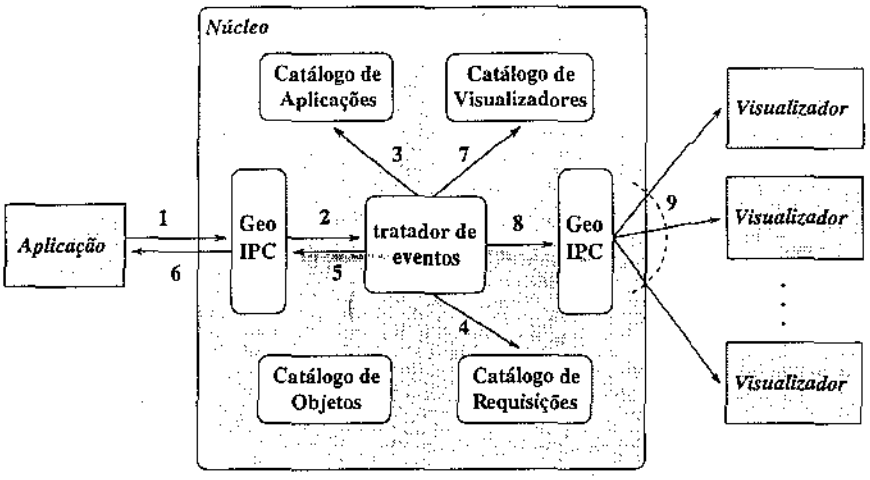

Figura 5.6: A operação de requisição de objetos geométricos

Requisição de objetos geométricos

Este evento permite que uma aplicação faça a requisição de algum objeto de entrada. Essa requisição será enviada aos visualizadores, o que permitirá que o usuário forneça uma entrada específica para a aplicação utilizando o visualizador que achar mais conveniente.

Os índices da figura 5.6 representam as seguintes ações:

1. A aplicação envia uma mensagem ao núcleo contendo a descrição do objeto desejado e a descrição do pedido.
2. O tratador de eventos do núcleo reconhece esta mensagem como um evento válido.
3. Através de uma consulta ao catálogo de aplicações, é identificado o contexto ao qual a aplicação está associada, que passa a ser o *contexto corrente*.
4. A requisição recebe um identificador único e é inserida no catálogo de requisições do núcleo associada ao contexto corrente.
5. O núcleo prepara uma mensagem de resposta à aplicação contendo o identificador único atribuído à requisição.
6. A aplicação recebe esse identificador e associa-o à requisição.
7. O catálogo de visualizadores é consultado para obtenção do conjunto de visualizadores associados ao contexto corrente.
8. O núcleo prepara uma mensagem para divulgar a nova requisição a todos os visualizadores associados ao contexto corrente.

9. Essa mensagem é enviada aos visualizadores que encarregar-se-ão de divulgar a requisição ao(s) usuário(s). Maiores detalhes sobre o funcionamento dos visualizadores serão vistos na seção 5.5.

Descadastramento de aplicações

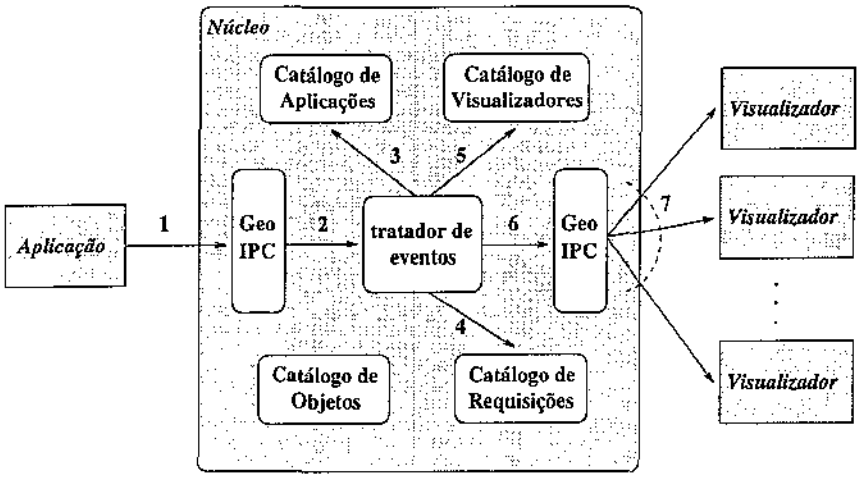

Figura 5.7: A operação de descadastramento de uma aplicação

Este evento representa a operação de descadastramento de uma aplicação associada a um determinado contexto do núcleo.

Os índices da figura 5.7 representam as seguintes ações:

1. A aplicação envia uma mensagem de descadastramento para o núcleo.

2. O tratador de eventos do núcleo reconhece esta mensagem como um evento válido.

3. O núcleo descadastra esta aplicação, que deixa de dispor dos recursos do ambiente GeoPrO. Além disso, o contexto ao qual esta aplicação estava associada passa a ser o *contexto corrente.*

4. O conjunto de requisições associadas à aplicação é removido do catálogo de requisições.

5. O catálogo de visualizadores é consultado para obtenção do conjunto de visualizadores associados ao contexto corrente.

6. O núcleo prepara uma mensagem para divulgar a remoção deste conjunto de requisições aos visualizadores.

7. Essa mensagem é enviada aos visualizadores, que deixam de apresentar esse conjunto de requisições como pendentes.

Cadastramento de visualizadores

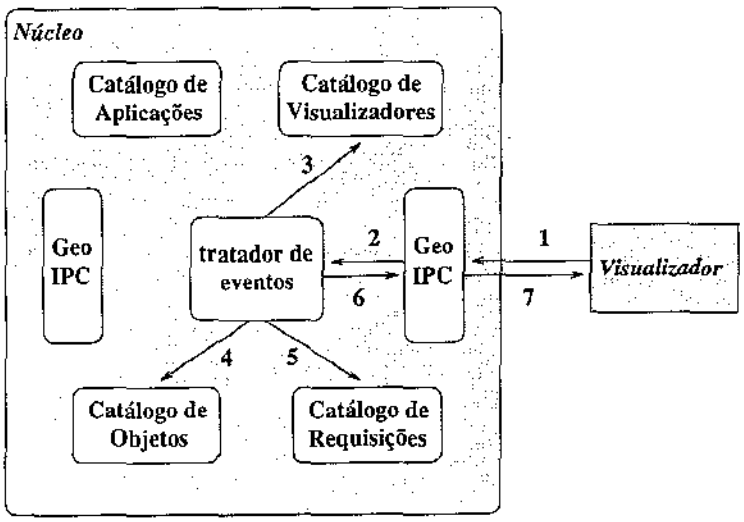

Figura 5.8: A operação de cadastramento de um visualizador

Este evento representa a operação de cadastramento de um visualizador a um determinado contexto do núcleo. No momento em que um novo visualizador cadastra-se ao núcleo ocorre uma fase de *sincronismo* fazendo com que o visualizador receba todos os objetos e as requisições pendentes do contexto associado a ele.

Os índices da figura 5.8 representam as seguintes ações:

1. O visualizador envia uma mensagem de cadastramento para o núcleo, especificando o seu *nome* e o *contexto* a que deseja conectar-se.

2. O tratador de eventos do núcleo reconhece esta mensagem como um evento válido.

3. O núcleo cadastra este visualizador no catálogo de visualizadores. Se o contexto ao qual este visualizador está associado ainda não existe, ele é criado. Este contexto passa a ser o *contexto corrente.*

4. O núcleo obtém todos os objetos do contexto corrente consultando o catálogo de objetos geométricos.

5. O núcleo obtém todas as requisições pendentes para o contexto corrente.

6. O núcleo prepara uma mensagem contendo as requisições e os objetos do contexto.

7. O visualizador recebe essa mensagem, estando, então, pronto para entrar em operação.

Resposta à requisição de objetos geométricos

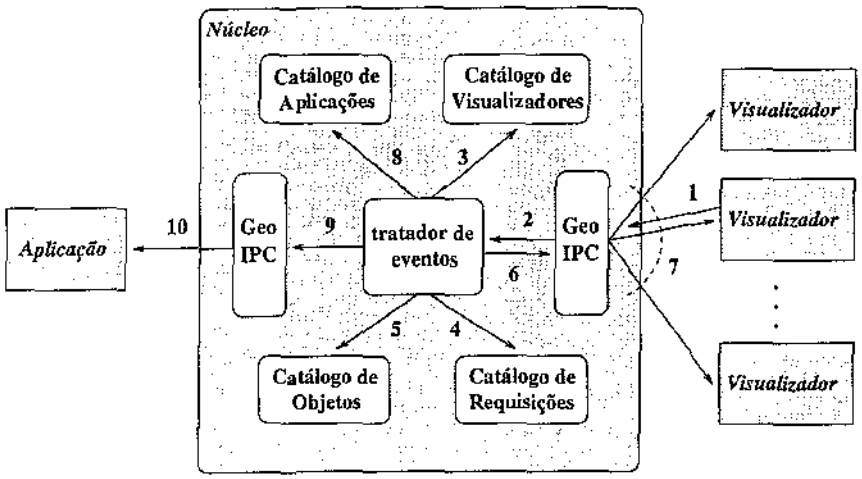

Figura 5.9: A operação de resposta à requisição de objetos geométricos

Este evento representa a resposta de algum visualizador a uma requisição de objeto feita por alguma aplicação.

Os índices da figura 5.9 representam as seguintes ações:

1. O visualizador envia uma mensagem ao núcleo contendo o objeto a ser enviado como resposta a uma determinada requisição.

2. O tratador de eventos do núcleo reconhece esta mensagem como um evento válido.

3. Através de uma consulta ao catálogo de visualizadores, é identificado o contexto ao qual este visualizador está associado, que passa a ser o *contexto corrente*. Além disso, é obtido o conjunto de visualizadores associados a este contexto.

4. A requisição é removida do catálogo de requisições.

5. O objeto recebe um identificador único e é inserido no catálogo de objetos geométricos do núcleo associado ao contexto corrente.

6. O núcleo prepara uma mensagem para divulgar o novo objeto a todos os visualizadores associados ao contexto corrente, inclusive o fornecedor do objeto.

7. Essa mensagem é enviada aos visualizadores que passam a mostrar o novo objeto.
8. É feita uma consulta ao catálogo de aplicações para identificar a aplicação que fez a requisição do objeto.
9. O núcleo prepara uma mensagem contendo o objeto a ser enviado a esta aplicação.
10. A aplicação recebe esse objeto.

Descadastramento de visualizadores

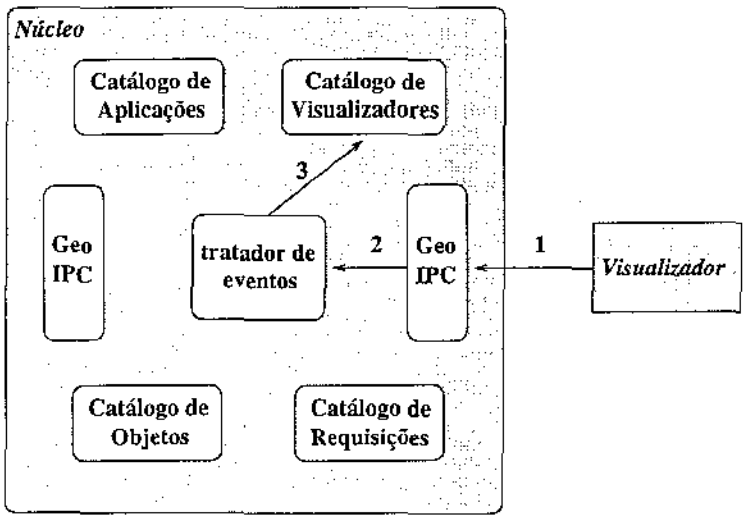

Figura 5.10: A operação de descadastramento de um visualizador

Este evento representa a operação de descadastramento de um visualizador de um determinado contexto do núcleo.

Os índices da figura 5.10 representam as seguintes ações:

1. O visualizador envia uma mensagem de descadastramento para o núcleo.
2. O tratador de eventos do núcleo reconhece esta mensagem como um evento válido.
3. O núcleo descadastra este visualizador, atualizando o cadastro de visualizadores.

5.4 Aplicações do usuário

Uma vez conectadas ao núcleo do GeoPrO, as aplicações do usuário estão habilitadas a inserir objetos geométricos nos contextos do núcleo para que sejam mostrados pelos visualizadores ativos ou remover objetos destes contextos para que sejam apagados dos visualizadores. Finalmente, as aplicações podem requisitar alguma entrada específica do usuário, que pode escolher o visualizador mais conveniente para prover os objetos requisitados. A seção 5.4.4 apresenta exemplos de aplicações que foram integradas ao ambiente GeoPrO.

A classe *Application*

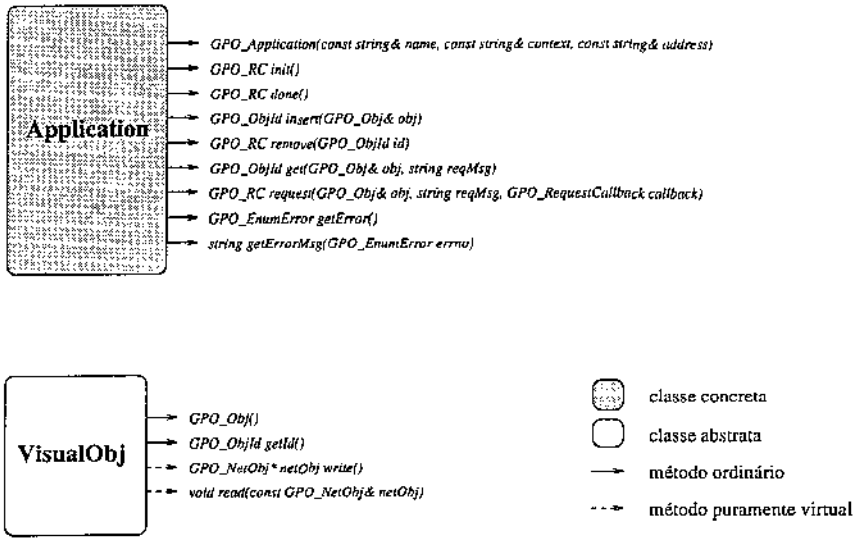

Figura 5.11: As interfaces públicas das classes *Application* e *VisualObj*

Nesta seção, apresentamos a interface entre a aplicação e o núcleo no nível da linguagem de programação. Essa interface está disponível para a linguagem C++ e, em breve, estará disponível para as linguagens Modula-3 e Java.

O objetivo desta seção é apresentar a interface pública da classe *Application* (veja figura 5.11). Detalhes sobre a implementação dessa classe podem ser obtidos no apêndice B.

A interface entre uma aplicação e o ambiente GeoPrO é feita através da classe *Application*. Uma instância dessa classe representa uma aplicação e está sempre relacionada a um determinado contexto de um núcleo.

A classe genérica *VisualObj* representa a entidade de integração entre os objetos geométricos da aplicação e o ambiente GeoPrO. Qualquer instância de uma subclasse da

classe *VisualObj* poderá, por alomorfismo, representar um objeto no ambiente GeoPrO. A classe *VisualObj* e a metodologia de integração entre os objetos da aplicação e o ambiente são descritas na seção 5.4.1.

Ao criar uma instância da classe *Application*, o usuário define o núcleo e o contexto ao qual ele deseja conectar-se. A definição do núcleo é feita através do nome do endereço *Internet* da máquina onde o núcleo do GeoPrO está rodando. Se este parâmetro for omitido, é assumido que o núcleo é local. O contexto também pode ser omitido e neste caso é assumido um contexto especial denominado "*default*". Usando a interface C++, a criação de uma instância é feita através da utilização do seguinte construtor:

```
GPO_Application(const string& name,
                const string& context = DEFAULT_CONTEXT,
                const string& address = DEFAULT_KERNEL_ADDRESS)
```

Uma vez criada a instância da classe, a aplicação deve conectar-se ao núcleo através do método *init*. Quando este método é evocado, a aplicação registra-se como um novo cliente junto ao núcleo do GeoPrO, podendo então usar os recursos do ambiente. Através da interface C++, essa iniciação é feita utilizando-se o seguinte método da classe *Application*:

```
GPO_RC init()
```

Para finalizar as suas operações com o núcleo do GeoPrO, a aplicação utiliza o método *done*, disponível na classe *Application*. Na interface C++ temos:

```
GPO_RC done()
```

Para inserir um objeto no contexto corrente do núcleo, a aplicação deve utilizar o método *insert*. Quando um objeto é adicionado a um contexto do núcleo, todos os visualizadores registrados àquele contexto receberão o novo objeto. Esse novo objeto, que agora faz parte de um contexto do núcleo, passa a ser *persistente*, recebendo um identificador (*id*) que servirá para identificá-lo univocamente perante o núcleo, as aplicações e os visualizadores. Em C++ temos:

```
GPO_ObjId insert(GPO_VisualObj& obj)
```

Para remover um objeto do núcleo, e conseqüentemente de todos os visualizadores, basta utilizar o método *remove*, passando como argumento o *id* do objeto a ser removido do contexto corrente do núcleo. Na interface C++ temos:

```
GPO_RC remove(GPO_ObjId id)
```

Quando uma aplicação deseja receber um objeto como entrada, ela pode enviar dois tipos de requisição para o núcleo: síncrona ou assíncrona. Na primeira, a aplicação ficará bloqueada até que a entrada desejada seja provida por algum visualizador conectado ao núcleo. Na segunda, a aplicação não ficará bloqueada e receberá o objeto requisitado de forma assíncrona através da evocação de uma *callback*. Na interface C++, estes serviços estão disponíveis através dos seguintes métodos:

```
GPO_ObjId get(GPO_VisualObj& obj,
              const string reqMsg)

GPO_RC request(GPO_VisualObj& obj,
               const string reqMsg,
               GPO_RequestCallback callback)
```

Onde o tipo *GPO_RequestCallback* é definido como:

```
typedef void (GPO_RequestCallback)(GPO_VisualObj& obj)
```

A verificação de erro é feita através dos métodos *getError* e *getErrorMsg*. O primeiro contém sempre um código de erro referente à última chamada à interface da classe *Application*. O segundo retorna uma mensagem descritiva para um determinado erro. Na interface C++, estes serviços estão disponíveis através dos seguintes métodos:

```
GPO_EnumError getError()

string getErrorMsg(GPO_EnumError errno)
```

5.4.1 Visualização de algoritmos usando GeoPrO

Esta seção tem por objetivo descrever a metodologia de utilização do ambiente GeoPrO para visualização e requisição de objetos geométricos.

Uma *classe geométrica* representa um específico objeto geométrico dotado de um conjunto de estruturas de dados e funcionalidades que são utilizadas na construção de algoritmos geométricos. Uma *classe geométrica visual* é uma extensão de uma classe geométrica que contém uma componente funcional responsável pela integração entre esta classe geométrica e o ambiente GeoPrO.

Uma classe geométrica visual é construída por derivação múltipla a partir de uma classe geométrica e da classe genérica *VisualObj*, conforme ilustra a figura 5.12. A classe

resultante herda todas as características geométricas da classe original, além da especificação funcional dos dois métodos de visualização oriundos da superclasse *VisualObj*: *read* e *write*. Em última análise, a implementação destes métodos segundo o protocolo de descrição geométrica (PDG) (descrito na seção 5.2.1) constitui o trabalho de integração entre uma classe geométrica qualquer e o ambiente GeoPrO.

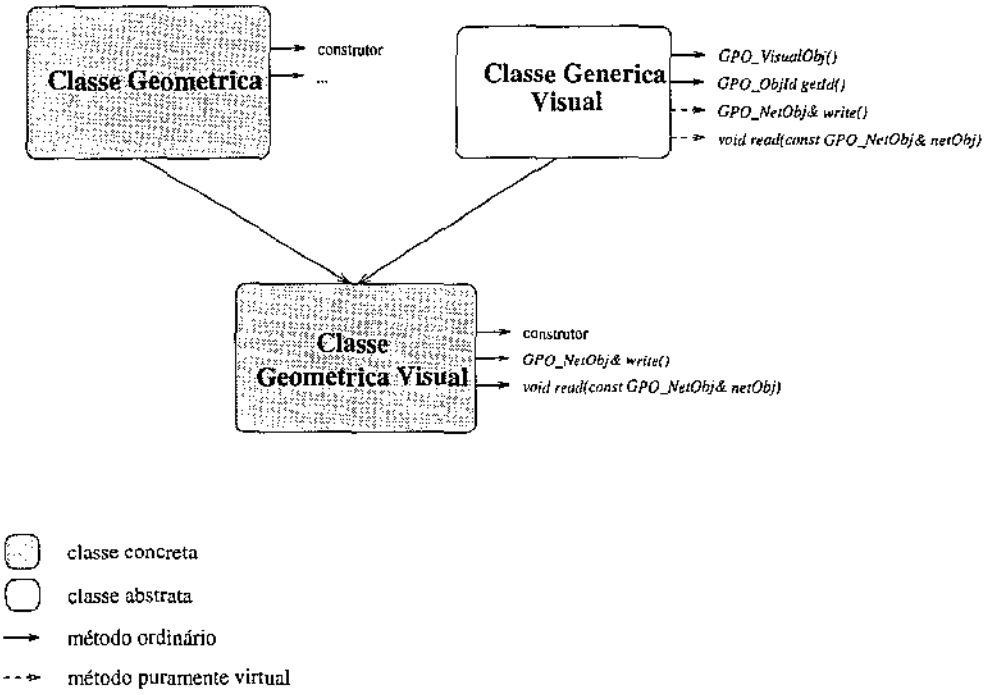

Figura 5.12: Construção de uma classe geométrica visual

5.4.2 Implementação dos métodos de visualização

O método *read* é utilizado pelo ambiente GeoPrO para informar ao objeto a descrição geométrica da entidade que ele deve representar, segundo o PDG. Assim, esse método será evocado quando uma requisição de entrada de dados para este objeto for atendida.

O método *write* é utilizado pelo ambiente para obter uma descrição geométrica do objeto de acordo com o PDG. Esse método é utilizado pelo ambiente GeoPrO quando um objeto é inserido em um contexto do núcleo ou quando é feita uma requisição de entrada de dados.

5.4.3 Um exemplo

Como exemplo de utilização dessa metodologia de integração, é mostrada abaixo a implementação da classe *VisualPolygon*, resultado da extensão da classe *polygon*, disponível na biblioteca LEDA, para incorporar os métodos de visualização, responsáveis pela integração com o ambiente GeoPrO.

No exemplo abaixo[4], a classe *VisualPolygon* é construída utilizando-se o mecanismo de herança múltipla da linguagem C++.

```
class VisualPolygon : public polygon, public GPO_VisualObj
{
  public:

    VisualPolygon()
      : polygon(), GPO_VisualObj(), pPolygon(0) {}

    VisualPolygon(const list<point>& pl)
      : polygon(pl), GPO_VisualObj(), pPolygon(0) {}

    virtual ~VisualPolygon() {
        if (pPolygon)
          delete pPolygon;
    }

    virtual GPO_NetObj& write() {
        pPolygon = new GPO_NetPolygon;
        if (!empty()) {
            const list<point>& points = vertices();
            point p;
            forall(p,points)
              pPolygon→add(GPO_Coord(p.xcoord(),p.ycoord(),1));
        }
        return *pPolygon;
    }
```

[4]A utilização de métodos *in-line* no exemplo constitui-se apenas num recurso para compactar a descrição, sendo, evidentemente, não recomendada para este caso.

```
                                    }
    virtual void read(const GPO_NetObj& netObj) {
        GPO_NetPolygon& poly = (GPO_NetPolygon&)netObj;
        GPO_Coord v;
        list<point> pl;
        forall(v,poly)
          pl.append(point(v.xcoord(),v.ycoord()));
        *this = pl;
    }

  protected:

    GPO_NetPolygon* pPolygon;
};
```

São implementados dois construtores que evocam os construtores das superclasses *polygon* e *VisualObj*. Além disso, o membro *pPolgyon* é iniciado com valor zero.

Um destrutor também é implementado unicamente para garantir a liberação da memória eventualmente alocada em *pPolygon*.

O método *write* retorna a descrição do polígono através de uma instância da classe *NetPolygon*. Essa instância é construída através de alocação dinâmica (via operador *new*) para que o seu escopo não fique restrito ao escopo do método. Uma referência a esta instância é armazenada em *pPolygon*.

No método *read*, o polígono é construído a partir de uma instância da classe *NetPolygon*, o que basicamente implica na conversão dos vértices, representados por pontos cartesianos da biblioteca LEDA (classe *point*), para pontos em coordenadas homogêneas com sinal (classe *Coord*), utilizados no GeoPrO.

Note que a iteração nos objetos compostos do GeoPrO (p.ex., *NetPolygon*), é feita de maneira análoga aos objetos compostos da biblioteca LEDA (p.ex., *polygon*), utilizando-se a macro de iteração *forall*, o que homogeniza a notação e aumenta a legibilidade do código resultante.

5.4.4 Aplicações usando o ambiente GeoPrO

Envelope de um conjunto de retas

Dado um conjunto de n retas no plano, o *envelope* do conjunto é definido pelo polígono cuja fronteira consiste das arestas limitadas de todas as regiões ilimitadas na subdivisão induzida pelo conjunto de retas (veja figura 5.13).

Como exemplo de utilização do ambiente GeoPrO e da biblioteca GeoPrOLib, foi implementado o algoritmo para determinação do envelope de um conjunto de retas apresentado em [12].

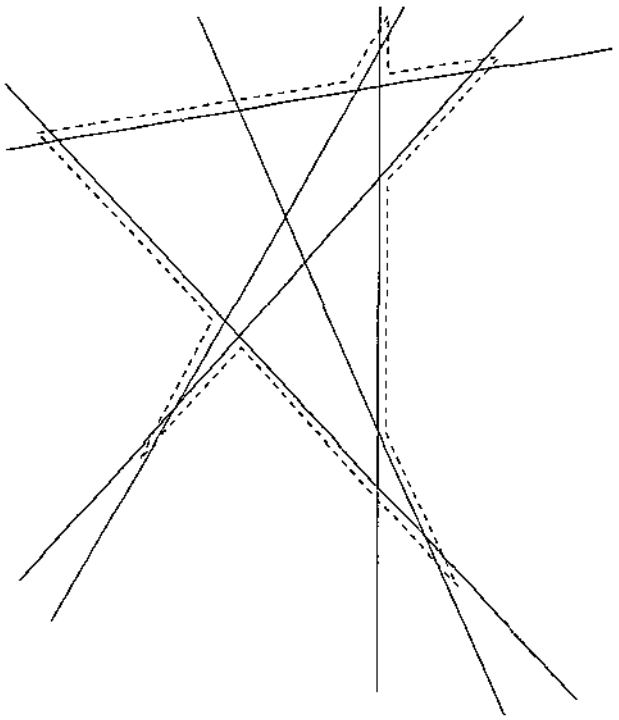

Figura 5.13: O envelope de um conjunto de retas

Nesta implementação, retas verticais e retas paralelas não precisaram ser considaderas como casos especiais, graças à nossa extensão de SoS para geometria projetiva orientada.

Detalhes sobre esta implementação podem ser encontrados em [9].

A biblioteca *Visual-LEDA*

A biblioteca *Visual-LEDA* é o resultado da extensão da coleção de tipos de dados e algoritmos geométricos disponíveis na biblioteca LEDA para que possam ser visualizados através do ambiente GeoPrO.

Essa extensão foi realizada segundo a metodologia de integração descrita na seção 5.4.1. A biblioteca *Visual-LEDA* possui os tipos apresentados na tabela abaixo.

Tipos básicos estendidos *Visual-LEDA*	Tipos básicos LEDA	Descrição
VisualPoint	*point*	um ponto no plano euclidiano
VisualRatPoint	*rat_point*	um ponto no plano em coordenadas racionais
VisualSegment	*segment*	um segmento de reta no plano euclidiano
VisualRatSegment	*rat_segment*	um segmento de reta no plano em coordenadas racionais
VisualLine	*line*	uma reta no plano euclidiano
VisualPolygon	*polygon*	um polígono simples no plano euclidiano
VisualCircle	*circle*	um círculo no plano euclidiano
VisualPoints	*list<point>*	uma lista de pontos no plano euclidiano
VisualRatPoints	*list<rat_point>*	uma lista de pontos no plano em coordenadas racionais
VisualSegments	*list<segments>*	uma lista de segmentos no plano euclidiano
VisualRatSegments	*list<rat_segments>*	uma lista de segmentos no plano em coordenadas racionais
VisualGraph<etype>	*GRAPH<point,etype>*	um grafo planar parametrizado nas arestas

Utilizando estes tipos estendidos, os seguintes algoritmos geométricos, disponíveis na biblioteca LEDA, foram facilmente integrados ao ambiente GeoPrO:

- *Triangulate Points.* Cálculo de uma *triangularização* de um conjunto de pontos no plano.
- *Delaunay Triangulation.* Cálculo da *Triangularização de Delaunay* de um conjunto de pontos no plano.

- *Sweep Segments.* Cálculo do grafo induzido por um conjunto de segmentos no plano. Os nós do grafo são todas as extremidades dos segmentos e todas as interseções próprias entre os segmentos. As arestas do grafo são representadas pelo conjunto maximal de subsegmentos abertos dos segmentos que não contêm nenhum nó do grafo.

- *Segment Intersection.* Cálculo de todas as interseções entre um conjunto de segmentos no plano.

- *Convex Hull.* Dado um conjunto de pontos no plano, calcula a envoltória convexa desse conjunto.

- *Closest Pair.* Dado um conjunto de pontos no plano, calcula o par de pontos com menor distância euclidiana entre eles.

- *Voronoi Diagram.* Cálculo do *Diagrama de Voronoi* de um conjunto de pontos no plano.

Finalmente, cabe ressaltar que a facilidade com que esta extensão foi realizada evidencia a eficiência da utilização da metodologia proposta para integração entre algoritmos e objetos geométricos existentes e o ambiente GeoPrO.

5.5 Visualizadores geométricos

Os *visualizadores geométricos* provêem um modo fácil de permitir a interação entre os usuários e as aplicações. Um visualizador é um processo que, uma vez conectado ao núcleo, é capaz de desenhar os objetos geométricos primitivos de acordo com um específico modelo geométrico. Ele também provê uma interface gráfica para entrada de objetos como resposta a requisições dos algoritmos (ou seja, das aplicações clientes conectadas ao núcleo). De maneira análoga às aplicações, um visualizador está sempre associado a um único contexto do núcleo.

Em adição a alguns visualizadores que o usuário tem à disposição (já implementados no ambiente GeoPrO), ele pode implementar um novo especialmente projetado para a sua aplicação. Tal implementação é facilitada se for feita através da derivação de um novo visualizador a partir de um pré-existente, segundo uma abordagem de programação orientada a objetos. A seção 5.5.1 apresenta a descrição dos visualizadores já implementados para o ambiente.

A classe *Visualizer*

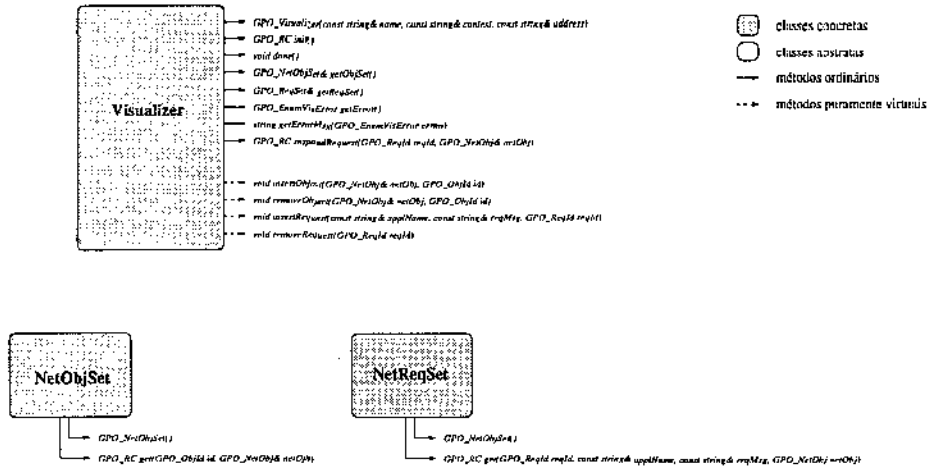

Figura 5.14: A classe *Visualizer*

Nesta seção, apresentamos a interface entre os visualizadores e o núcleo no nível da linguagem de programação. Essa interface está disponível para a linguagem C++ e, em breve, estará disponível para as linguagens Modula-3 e Java.

O objetivo desta seção é apresentar a interface pública da classe *Visualizer*, o que deve interessar aos usuários-programadores que desejem construir novos visualizadores para o

ambiente. Detalhes sobre a implementação dessa classe podem ser obtidos no apêndice B.

A interface entre um visualizador e o núcleo é feita através da classe *Visualizer*. Todos os visualizadores do ambiente GeoPrO são instâncias de subclasses da classe *Visualizer* (veja figura 5.14).

A criação de uma instância da classe *Visualizer* define o núcleo e o contexto associados ao visualizador. De modo análogo ao que ocorre com as aplicações, a definição do núcleo é feita através do endereço *Internet* da máquina onde ele está rodando. Se este parâmetro for omitido, é assumido que o núcleo é local. A omissão do contexto implica na utilização do contexto "*default*". O contexto ao qual um visualizador está associado será referenciado como *contexto corrente*. Usando a interface C++, a criação de uma instância é feita através da utilização do seguinte construtor:

```
GPO_Visualizer(const string& name,
               const string& context = DEFAULT_CONTEXT,
               const string& address = DEFAULT_KERNEL_ADDRESS)
```

Uma vez criada a instância da classe, o visualizador conecta-se ao núcleo através do método *init*, que, quando evocado, registra o visualizador como um novo cliente junto ao núcleo. Além disso, é neste instante que o visualizador entra em "sincronismo" com o núcleo, recebendo deste o conjunto de objetos e requisições do contexto corrente. Essa sincronização entre o núcleo e os visualizadores é necessária para permitir que um visualizador possa conectar-se ao núcleo a qualquer instante e não apenas no início das operações do núcleo.

Através da interface C++, esta iniciação é feita utilizando-se o seguinte método da classe *Visualizer*:

```
GPO_RC init()
```

Para finalizar as suas operações com o núcleo, o visualizador utiliza o método *done* disponível na classe *Visualizer*. Na interface C++ temos:

```
GPO_RC done()
```

A superclasse *Visualizer* oferece o método *getObjSet* para obtenção do conjunto de objetos do contexto corrente. Na interface C++ temos:

```
GPO_NetObjSet& getObjSet()
```

A superclasse *Visualizer* também oferece o método *getReqSet* para obtenção do conjunto das requisições do contexto corrente. Na interface C++ temos:

```
GPO_NetReqSet& getReqSet()
```

Quando a ação de um usuário sobre um visualizador implica na resposta à requisição de um objeto, o método *respondRequest* deve ser evocado para informar ao núcleo e, conseqüentemente, à aplicação que fez a requisição. Na interface C++ temos:

```
GPO_RC respondRequest(GPO_ReqId reqId, GPO_NetObj& netObj)
```

Quando um novo objeto é inserido em um contexto do núcleo, os visualizadores associados a este contexto são informados através da evocação do método *insertObject*. Um tratamento normal para esse método nas subclasses da classe *Visualizer* (ou seja, nos visualizadores derivados) é o desenho do novo objeto. Na interface C++ temos:

```
virtual void insertObject(GPO_ObjId id, GPO_NetObj& netObj)
```

Se um objeto é removido de um contexto do núcleo, os visualizadores associados a este contexto são informados através da evocação do método *removeObject*. Um tratamento normal para esse método nas subclasses da classe *Visualizer* é a omissão visual deste objeto. Na interface C++ temos:

```
virtual void removeObject(GPO_ObjId id, GPO_NetObj& netObj)
```

Assim que uma nova requisição de objeto é inserida em um contexto do núcleo, os visualizadores associados a este contexto são informados através da evocação do método *insertRequest*. Um tratamento normal para esse método nas subclasses da classe *Visualizer* é avisar ao usuário sobre a nova requisição e disponibilizar uma forma de atendimento a esta requisição. Na interface C++ temos:

```
virtual void insertRequest(GPO_ReqId reqId,
                           const string& applName,
                           const string& reqMsg)
```

Se uma requisição de objeto é respondida por algum visualizador ou quando ela deixa de ser válida devido ao descadastramento da aplicação que fez o pedido, os visualizadores associados ao contexto em questão precisam ser informados. Isto é feito através da evocação do método *removeRequest*. Um tratamento normal para esse método nas subclasses da classe *Visualizer* é a retirada dessa requisição do conjunto de requisições que o usuário pode atender. Na interface C++ temos:

virtual void removeRequest(GPO_ReqId reqId)

A verificação de erro é feita através dos métodos *getError* e *getErrorMsg*. O primeiro contém sempre um código de erro referente à última chamada à interface da classe *Visualizer*. O segundo retorna uma mensagem descritiva para um determinado erro. Na interface C++, estes serviços estão disponíveis através dos seguintes métodos:

GPO_EnumVisError getError()

string getErrorMsg(GPO_EnumVisError errno)

5.5.1 Visualizadores implementados

O modelo esférico do plano projetivo orientado $\mathbf{T}^2$, apresentado na seção 3.2, facilita a visualização da sua topologia e de suas propriedades geométricas, principalmente em relação aos pontos no infinito. Além disso, é uma ferramenta visual para auxiliar na interpretação de problemas e na derivação de algoritmos.

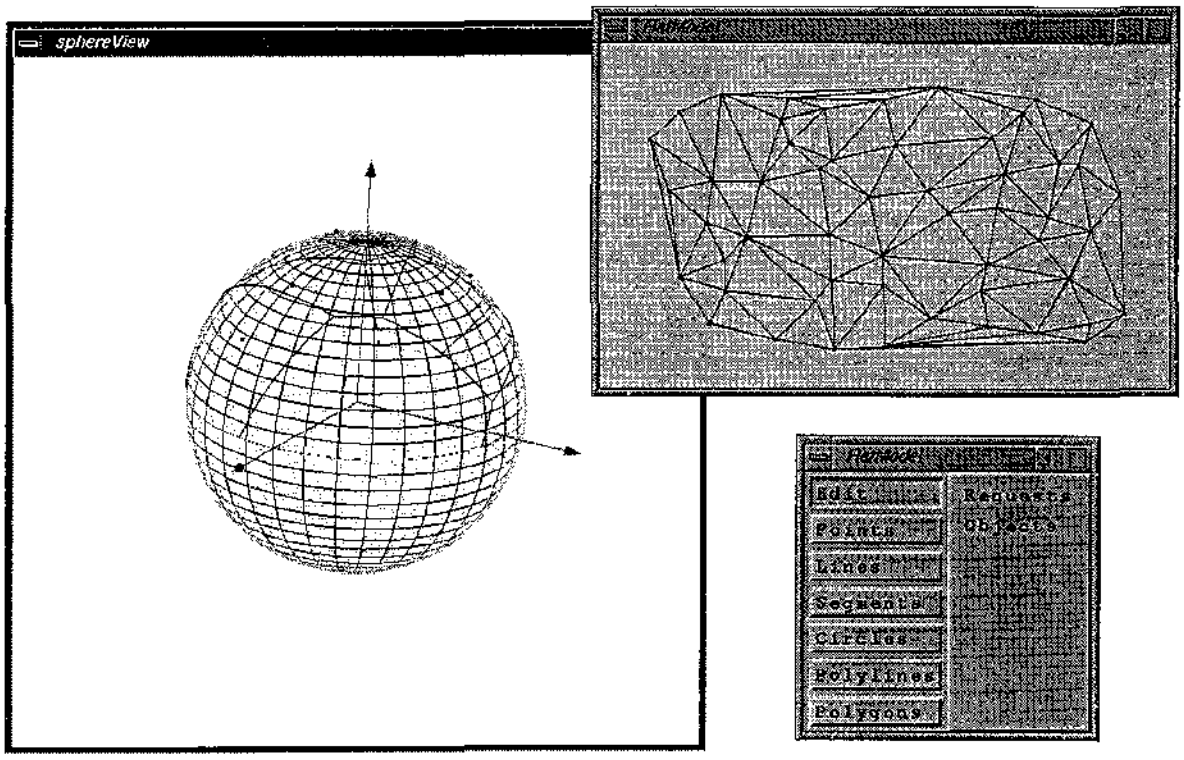

Figura 5.15: Visualizadores para o ambiente GeoPrO

Um visualizador para este modelo foi implementado (veja figura 5.15, lado esquerdo), possuindo uma interface gráfica que permite a visualização dos objetos básicos tratados pelos algoritmos geométricos conectados ao núcleo do ambiente GeoPrO. A implementação foi realizada sobre a plataforma IRIS Silicon Graphics, podendo ser portada para qualquer plataforma que suporte *OpenGL* e *X Window*.

Além do visualizador esférico, dois visualizadores planares foram implementados, um sobre *X Window* e outro sobre *Motif+OpenGL* (veja figura 5.15, lado direito).

Finalmente, está em andamento o desenvolvimento de um visualizador planar em Java que permitirá a visualização de algoritmos via *Web*, o que deverá ser de grande valia para a comunidade científica na divulgação de trabalhos relacionados ao desenvolvimento de algoritmos geométricos.

5.6 O projeto GeoPrO

5.6.1 Características do GeoPrO

Suporte a execução distribuída

O ambiente GeoPrO suporta múltiplas aplicações e múltiplos visualizadores executando simultaneamente em máquinas homogêneas ou heterogêneas distribuídas. Esse tipo de operação será referenciado como *execução distribuída.*

O suporte a execução distribuída tem como requisito básico a solução do seguinte problema: como as aplicações e os visualizadores identificam o núcleo ao qual desejam conectar-se?

Como a comunicação entre os processos é baseada no protocolo TCP/IP, a solução adotada foi a padronização de duas portas TCP/IP *Internet*, uma para requisição de conexão de aplicações e outra de visualizadores. Deste modo, um cliente (aplicação ou visualizador) identifica o núcleo ao qual irá se conectar apenas especificando o endereço da máquina onde este está sendo executado. Este esquema limita a execução de no máximo um núcleo por máquina, o que não é restritivo devido ao suporte a múltiplos contextos, descrito nesta seção.

As classes que implementam o suporte de comunicação à execução distribuída estão implementadas na biblioteca GeoPrO IPC, descrita no apêndice B.1.

Persistência dos objetos de visualização

Quando um objeto é inserido em um contexto do núcleo ele se torna *persistente*, recebendo um identificador único em relação a todos os objetos de todos os contextos do núcleo.

A criação de um objeto persistente pode ocorrer por meio de dois eventos distintos:

- a inserção de um objeto em um contexto do núcleo feita por uma aplicação, ou,
- a resposta de um visualizador a uma requisição de entrada de objeto feita por alguma aplicação.

Um objeto persistente está somente associado ao contexto ao qual ele foi adicionado e não à aplicação que realizou a inserção ou ao visualizador que respondeu a uma requisição.

Mesmo que o programa (aplicação ou visualizador) que gerou o evento que inseriu o objeto no núcleo termine, este permanecerá no ambiente, sendo apresentado nos visualizadores e disponível para servir de matriz para objetos enviados em resposta a pedidos

de entrada feitos pelas aplicações. Este objeto existirá até que alguma aplicação remova-o do contexto do núcleo ou até um eventual encerramento das operações do núcleo.

Uma extensão natural das funcionalidades do núcleo GeoPrO será a capacidade de armazenamento e recuperação dos contextos em uma base de dados. Isso expandirá a capacidade de armazenamento de informação do núcleo e permitirá que a existência dos contextos não fique restrita ao período de execução do núcleo.

Persistência de objetos é um requerimento indispensável para que, em um ambiente de programação distribuída, as diversas aplicações possam compartilhar resultados.

Suporte a múltiplos contextos

Um contexto no núcleo tem associado a ele um grupo de aplicações, um grupo de visualizadores, um conjunto de objetos geométricos e um conjunto de requisições de objetos. Cada aplicação, visualizador, objeto ou requisição está associada a um único contexto. A utilização de *múltiplos contextos* permite que diversos grupos de aplicações e visualizadores estejam associados a contextos distintos, manipulando somente conjuntos de objetos e requisições disponíveis no contexto a que estão associados.

O suporte a múltiplos contextos é uma característica fundamental no GeoPrO para permitir que diversos usuários compartilhem os recursos do ambiente de maneira independente.

Extensibilidade e escalabilidade

A abordagem orientada a objetos utilizada no projeto dos componentes do GeoPrO, a arquitetura multi-plataforma e o suporte a execução distribuída permitem que o ambiente seja estendido de acordo com as necessidades do usuário. Objetos cada vez mais complexos e visualizadores dotados de novos e sofisticados recursos de visualização e entrada de dados podem ser facilmente integrados ao ambiente.

Além disso, a flexibilidade de utilização de visualizadores para diferentes plataformas, independentemente das plataformas que hospedam as aplicações, permite que o usuário dimensione a utilização dos recursos computacionais disponíveis de acordo com as características de suas aplicações. Assim, se um *terminal X* deixa de ser suficiente para a demanda de visualização de uma determinada aplicação, o usuário poderá optar por utilizar um visualizador em uma plataforma com recursos gráficos mais sofisticados (p.ex., uma estação Silicon Graphics) que lhe esteja disponível.

5.7 Implementação

O pacote básico do ambiente GeoPrO compreende:

1. O núcleo do GeoPrO, apresentado na seção 5.3.
2. A interface *Application* para integração de aplicações ao ambiente, apresentada na seção 5.4.
3. A interface *Visualizer* para construção de novos visualizadores, apresentada na seção 5.5.

O pacote básico do ambiente GeoPrO é implementado por uma família de classes C++, podendo ser utilizada praticamente com qualquer compilador C++ (p.ex., GNU G++, Sun C++, AIX CSET++ e HP C++).

Na versão atual, toda a comunicação entre processos é feita através do protocolo TCP/IP, o que estabelece uma dependência seguramente não restritiva devido a elevada disponibilidade deste protocolo nas diferentes plataformas de *hardware/software.*

A compilação do ambiente depende ainda da disponibilidade da biblioteca LEDA [16], também altamente disponível para as mais diversas plataformas.

O núcleo do GeoPrO e as interfaces *Application* e *Visualizer* estão disponíveis para as seguintes plataformas:

- IRIS Silicon Graphic
- SunOS/Solaris - Sun Microsystems
- Linux

e, em breve, também estarão disponíveis para:

- Unix AIX - Risc IBM 6000 - IBM
- HP-UX - Risc HP 9000 - HP
- Windows NT/Windows 95 - Microsoft

5.8 Documentação

Dois manuais estão disponíveis para o ambiente GeoPrO:

- *Manual do Usuário*, com informações sobre instalação e utilização do ambiente GeoPrO.
- *Manual do Programador*, contendo a documentação necessária à implementação/integração de aplicações e novos visualizadores para o ambiente.

Capítulo 6

Extensões e trabalhos futuros

Esta seção apresenta uma breve descrição de desenvolvimentos em andamento e de possíveis extensões do presente trabalho.

Gerenciador para o ambiente GeoPrO Uma extensão imediata para o ambiente GeoPrO é o desenvolvimento de um módulo gerenciador responsável por atividades tais como: (1) configuração de parâmetros do ambiente; (2) fornecimento de informações sobre as aplicações e visualizadores conectados ao núcleo; (3) permitir a ação de um usuário-administrador sobre os contextos do núcleo, manipulando objetos e requisições; (4) cadastro de aplicações e visualizadores disponíveis e suporte a execução local ou remota dos mesmos; e (5) apresentação de dados estatísticos sobre a utilização do ambiente. A ação do usuário sobre o gerenciador se daria por meio de uma interface gráfica, o que estabelece uma dependência de plataforma para a implementação. Já está em estudo a implementação de dois gerenciadores, um para a plataforma SGI e outro em linguagem Java, o que permitirá que a administração do ambiente seja realizada via *Web*.

Interfaces para Modula-3 e Java Já estão em desenvolvimento as interfaces para que aplicações escritas nas linguagens Modula-3 e Java possam utilizar o ambiente GeoPrO.

Visualizadores *Web* Está em andamento o desenvolvimento de um visualizador planar em Java que permitirá a visualização de algoritmos via *Web*, o que deverá ser de grande valia para a comunidade científica na divulgação de trabalhos relacionados ao desenvolvimento de algoritmos geométricos.

Contextos perenes Uma extensão natural das funcionalidades do núcleo GeoPrO seria a capacidade de armazenamento e recuperação dos contextos em uma base de dados.

Isso expandiria a capacidade de armazenamento de informação do núcleo e permitiria que a existência dos contextos não ficasse restrita ao período de execução do núcleo.

Otimização do ambiente GeoPrO A identificação de pontos de congestionamento e o desenvolvimento de mecanismos para melhoria de desempenho do ambiente GeoPrO constituem atividades de extensão do presente trabalho.

Mecanismos de segurança O projeto atual do ambiente GeoPrO não incorpora mecanismos de segurança que podem ser necessários dependendo do tipo de utilização desejada. A criação de tais mecanismos constitui uma importante extensão do presente trabalho.

Sincronização entre vários núcleos Mecanismos de sincronização entre diferentes núcleos no ambiente GeoPrO poderiam permitir operações como replicação de contextos, migração de objetos e requisições, etc.

Extensão do protocolo de descrição geométrica Dependendo do tipo de utilização do ambiente GeoPrO, pode ser necessário estender o protocolo de descrição geométrica (descrito na seção 5.2.1) para suportar uma gama maior de objetos.

Extensão para tratamento de objetos 3D A extensão do ambiente GeoPrO para tratamento de objetos 3D constitui uma importante e desafiadora tarefa.

Porte do núcleo para Java Uma forma de aumentar a portabilidade do ambiente seria o porte do núcleo GeoPrO para a linguagem Java.

Extensão da biblioteca GeoPrOLib A implementação atual da biblioteca GeoPrOLib é limitada ao plano projetivo orientado e a um conjunto reduzido de primitivas geométricas. Esta biblioteca poderia ser estendida para dimensões arbitrárias e para um maior número de primitivas geométricas.

Capítulo 7

Conclusões

As principais contribuições deste trabalho são:

1. A extensão de perturbação simbólica para tratamento de degenerações em geometria projetiva orientada.

2. Uma biblioteca de primitivas geométricas para produção de aplicações sobre o plano projetivo orientado utilizando perturbação simbólica.

3. Um ambiente distribuído para visualização de algoritmos geométricos.

A geometria projetiva clássica, como extensão da geometria euclidiana, introduz uma série de benefícios, como a simplificação de fórmulas, a redução do número de casos particulares, a unificação e extensão de conceitos e a utilização de dualização como ferramenta poderosa para projeto de algoritmos na área de geometria computacional.

A geometria projetiva orientada ratifica e amplia estas vantagens através do tratamento consistente de linhas e planos orientados, ângulos com sinal, segmentos, direções, conjuntos convexos e muitos outros conceitos que a geometria projetiva clássica não suporta em toda generalidade.

A geometria projetiva orientada permite a definição do conceito de orientação em toda a sua generalidade, sem a necessidade de qualquer tratamento especial para pontos no infinito e conseqüentemente, permite a utilização de perturbação simbólica para tratamento de degeneração para todos os pontos do espaço projetivo orientado $\mathbb{T}^d$.

GeoPrOLib é uma biblioteca de primitivas geométricas para produção de aplicações sobre o plano projetivo orientado, utilizando perturbação simbólica para tratamento geral dos casos degenerados e aritmética exata para garantir robutez às operações efetuadas.

GeoPrO é um ambiente distribuído para visualização de algoritmos geométricos que suporta múltiplas aplicações e múltiplos visualizadores executando simultaneamente em máquinas heterogêneas distribuídas.

A utilização de uma abordagem cliente-servidor facilita a utilização do ambiente pelas aplicações, independentemente da linguagem de programação ou da plataforma de origem das mesmas.

A metodologia de integração entre classes geométricas existentes e o ambiente GeoPrO permite um alto nível de abstração e facilita a reutilização de código, um requisito fundamental para um ambiente de visualização geométrica.

Persistência de objetos é um requerimento indispensável para que, em um ambiente de programação distribuída, as diversas aplicações possam compartilhar resultados.

O suporte a múltiplos contextos permite que diversos usuários compartilhem os recursos do ambiente de maneira independente.

A flexibilidade de utilização de visualizadores para diferentes plataformas, independentemente das plataformas que hospedam as aplicações, permite que o usuário dimensione a utilização dos recursos computacionais disponíveis de acordo com as características de suas aplicações.

Finalmente, a abordagem orientada a objetos utilizada no projeto dos componentes do GeoPrO, a arquitetura multi-plataforma e o suporte a execução distribuída permitem que o ambiente seja estendido de acordo com as necessidades do usuário. Objetos cada vez mais complexos e visualizadores dotados de novos e sofisticados recursos de visualização e entrada de dados podem ser facilmente integrados ao ambiente.

Apêndice A

Convenções

A.1 Definições de *Classe*, *Objeto*, *Superclasse* e *Subclasse*

Aparecem na literatura diversas interpretações para os termos *classe*, *objeto*, *superclasse*, *subclasse*, etc. No presente texto, adotamos as convenções abaixo.

Uma declaração de *classe* introduz um novo tipo de dado. Se uma classe é derivada de uma outra classe, esta última é denominada *superclasse imediata* daquela. Diz-se que uma classe *estende* sua superclasse imediata porque ela pode prover detalhes adicionais em sua implementação.

Todo *objeto* é uma *instância* de alguma classe.

Uma classe A é uma *superclasse* da classe C se, e somente se, uma das seguintes sentenças é verdadeira:

- A é a mesma classe que C,
- A é uma superclasse imediata de C,
- existe alguma classe B tal que A é uma superclasse de B e B é uma superclasse de C.

Uma classe A é uma *superclasse própria* de uma classe B se, e somente se, A é uma superclasse de B mas não é B.

Uma classe B é uma *subclasse imediata* de A se, e somente se, a classe A é uma superclasse imediata de B. Uma classe B é uma *subclasse* de A se, e somente se, A é uma superclasse da classe B. Uma classe B é uma *subclasse própria* da classe A se, e somente se, A é uma superclasse própria da classe B.

Uma classe é *abstrata* quando ela possui uma especificação completa de sua interface, mas não possui a implementação de todos os métodos presentes em sua especificação.

Uma classe é *concreta* quando ela não é abstrata.

A.2 Fontes dos símbolos

- Nomes de classes e métodos são sempre apresentados utilizando-se a fonte *slanted* e sem o prefixo *GPO_*, que é utilizado em todos os símbolos públicos da implementação para evitar conflito de nomes. Por exemplo: *NetObj*, *request*.
- Nomes de entidades aparecem em itálico, normalmente seguidos pelos nomes das classes que as implementam. Por exemplo: *Domínio Internet* (*InternetDomain*).
- Palavras reservadas de linguagens de programação aparecem em negrito. Por exemplo: **const**.
- Palavras estrangeiras aparecem em itálico. Por exemplo: *multiprecision arithmetic*.

Apêndice B

Detalhes de implementação

B.1 GeoPrO IPC

A biblioteca GeoPrO IPC contém as classes que implementam o suporte de comunicação à execução distribuída no ambiente GeoPrO. GeoPrO IPC é baseada nos serviços de comunicação entre processos providos por implementações do protocolo TCP/IP para oferecer uma interface de comunicação e sincronização entre os componentes do ambiente GeoPrO: núcleo, aplicações e visualizadores.

GeoPrO IPC é implementada pela hierarquia de classes apresentada na figura B.1.

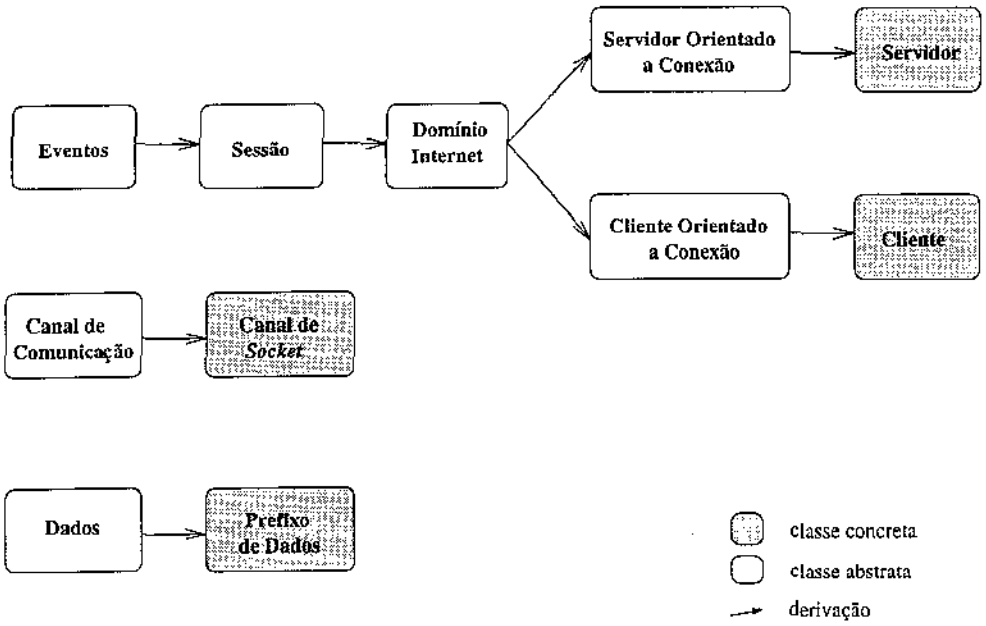

Figura B.1: Hierarquia das classes de comunicação

B.1.1 Eventos, sessões e conexões

A superclasse abstrata *Eventos* (*EventSource*) representa a especificação do modelo de comunicação orientado a eventos.

A classe abstrata *Sessão* (*CommSession*) é uma subclasse da classe *Eventos*, que representa a especialização do modelo de comunicação orientado a eventos implementando-o através do estabelecimento de sessões de comunicação entre os processos envolvidos. É nesse nível que é estabelecido que a comunicação entre os processos será feita através da utilização de conexões TCP/IP.

A subclasse *Domínio Internet* (*Internet Domain*) é uma especialização da classe *Sessão* que define o domínio da conexão TCP/IP com sendo *Internet*.

A partir da classe *Domínio Internet*, são derivadas duas subclasses (ainda abstratas):

- *Servidor Orientado a Conexão* (*ConnOrientedServer*), responsável pelo gerenciamento de conexões com um ou mais clientes. O servidor atende pedidos de conexão em uma porta TCP/IP *Internet* e estabele uma nova conexão para cada novo cliente.

- *Cliente Orientado a Conexão* (*ConnOrientedClient*), responsável pelo pedido de conexão com um servidor, identificado pelo *host* e por uma porta de serviço.

As classes concretas *Servidor* (*Server*) e *Cliente* (*Client*) são derivações das classes *Servidor Orientado a Conexão* e *Cliente Orientado a Conexão*, respectivamente. As instâncias destas classes permitem a troca de mensagens entre processos segundo o modelo cliente-servidor.

B.1.2 Mensagens

Uma mensagem é representada na biblioteca como sendo uma instância de uma classe derivada da superclasse abstrata *Dados* (*Data*) e é, alomorficamente, utilizada por instâncias das classes *Servidor* e *Cliente*.

A classe *Dados* representa uma especificação consistente para o formato das mensagens que são trocadas entre os processos. Além disso, ela encapsula a codificação/decodificação dos dados através de operações XDR[1][17], garantindo a consistência das mensagens trocadas entre processos executados em máquinas com representações de dados distintas. A descrição das mensagens trocadas no ambiente GeoPrO são apresentadas na seção B.1.4.

[1] *External Data Representation Serialization Routines.*

B.1.3 Canais de comunicação e *socket*

A classe abstrata *Canal de Comunicação* (*CommChannel*) é a especificação de uma classe responsável pela manipulação de diversas instâncias de *Eventos*. A figura B.2 ilustra o relacionamento entre a classe *Canal de Comunicação* e a classe *Eventos*.

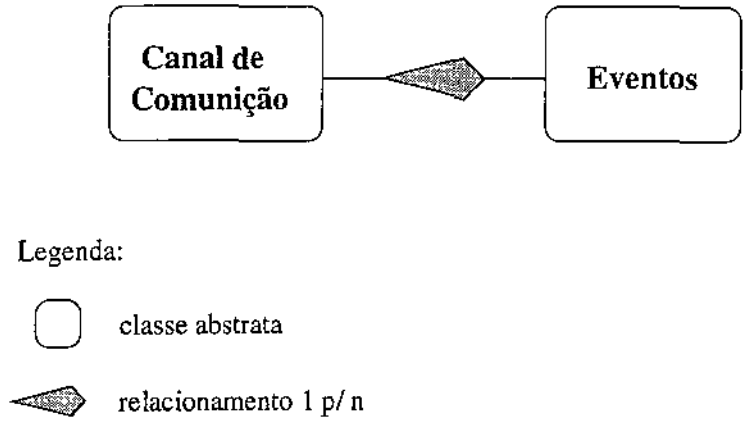

Figura B.2: Relacionamento entre a classe *Canal de Comunicação* e a classe *Eventos*

A classe concreta *Canal de Socket* (*SocketChannel*) é uma derivação que especializa a classe *Canal de Comunicação* para manipular diversas sessões TCP/IP, representadas por instâncias de subclasses da classe *Sessão*, ou seja, servidores e clientes.

B.1.4 O uso da biblioteca GeoPrO IPC no ambiente GeoPrO

Toda a comunicação entre as aplicações e o núcleo e entre este e os visualizadores é feita através da biblioteca GeoPrO IPC.

Cada núcleo do GeoPrO possui um *Canal de Socket* com dois *Servidores*, um responsável pela comunicação com as aplicações e outro responsável pela comunicação com os visualizadores.

Cada instância da classe *Application*, representando a interface entre uma aplicação e o núcleo, possui um *Canal de Socket* contendo um *Cliente* conectado ao núcleo. Detalhes sobre a classe *Application* são apresentados na seção B.3.

Simetricamente, uma instância da classe *Visualizer*, representando a interface entre um visualizador e o núcleo, possui um *Canal de Socket* contendo um *Cliente* conectado ao núcleo. Detalhes sobre a classe *Visualizer* são apresentados na seção B.4.

Finalmente, a figura B.3 mostra a hierarquia de classes de todas as mensagens utilizadas no ambiente GeoPrO.

Figura B.3: Hierarquia das mensagem no ambiente GeoPrO

B.2 O núcleo

As figuras B.4 e B.5 ilustram o relacionamento entre as diferentes classes que compõem a implementação do núcleo do GeoPrO.

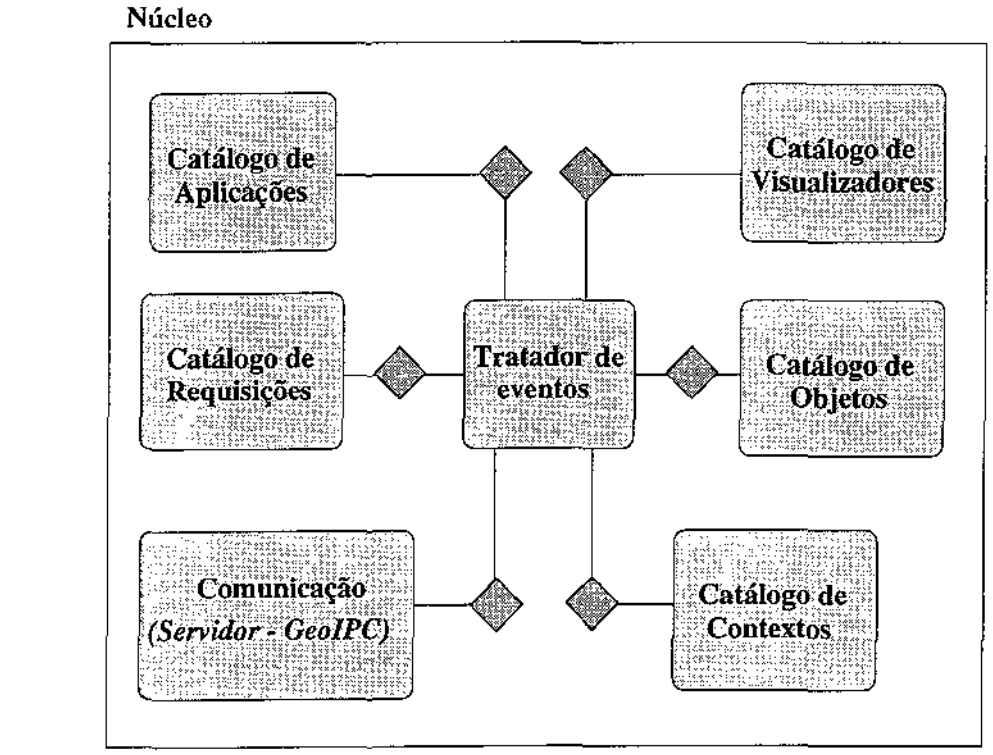

Figura B.4: O relacionamento entre as classes no núcleo - I

O núcleo é o resultado da interação entre instâncias das seguintes classes:

- *Tratador de Eventos* (*KernelHandler*), responsável pelo tratamentos de todos os eventos que chegam ao núcleo gerados pelas aplicações e pelos visualizadores. Esses eventos são apresentados na seção 5.3.
- *Comunicação* (*Server*). Duas instâncias da classe *Server* são responsáveis pela comunicação entre o núcleo e as aplicações e entre o núcleo e os visualizadores. As classes de comunicação são descritas na seção B.1.

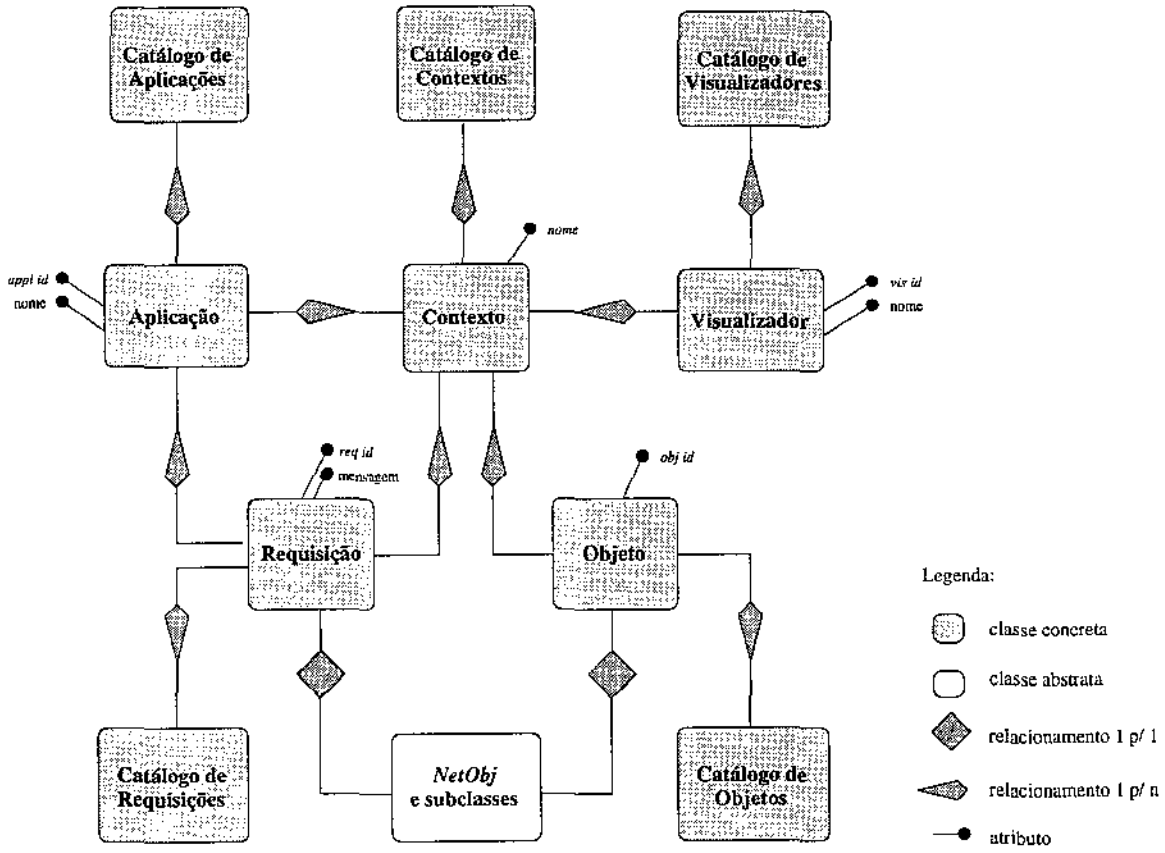

Figura B.5: O relacionamento entre as classes no núcleo - II

- *Catálogo de Aplicações* (*ApplCatalog*), responsável pelo controle das informações relacionadas às aplicações cadastradas no núcleo. Uma instância da classe *ApplCatalog* contém um conjunto de instâncias da subclasse *ApplCatalog::Info* representando cada uma das aplicações. Esses objetos são armazenados em uma estrutura de *hashing*, com um mapeamento injetivo do tipo *ApplId* para um conjunto de instâncias da classe *ApplCatalog::Info*.

- *Catálogo de Visualizadores* (*VisCatalog*), responsável pelo controle das informações relacionadas aos visualizadores cadastrados no núcleo. Uma instância desta classe contém um conjunto de instâncias da subclasse *VisCatalog::Info* representando cada umas das aplicações. Esses objetos são armazenados em uma estrutura de *hashing*, com um mapeamento injetivo do tipo *VisId* para um conjunto de instâncias da classe *VisCatalog::Info*.

- *Catálogo de Objetos geométricos* (*ObjCatalog*), responsável pelo controle dos objetos geométricos no núcleo, representados por instâncias da subclasse *ObjCatalog::Info* e armazenados em uma estrutura de *hashing*, com um mapeamento injetivo do tipo *ObjId* para um conjunto de instâncias da classe *ObjCatalog::Info*.

- *Catálogo de Requisições* (*ReqCatalog*), responsável pelo armazenamento das requisições de objetos feitas pelas aplicações, representadas por instâncias da subclasse *ReqCatalog::Info* e armazenados em uma estrutura de *hashing*, com um mapeamento injetivo do tipo *ReqId* para um conjunto de instâncias da classe *ReqCatalog::Info*.

- *Catálogo de Contextos* (*ContextCatalog*), responsável pelo controle dos diversos contextos que podem existir no núcleo, representados por instâncias da subclasse *ContextCatalog::Info*. Os diversos contextos são armazenados através de uma estrutura de *hashing*, com um mapeamento injetivo da classe *string* (o nome do contexto) para um conjunto de instâncias da classe *ContextCatalog::Info*.

- *Contexto* (*ContextCatalog::Info*), classe que representa um contexto do núcleo, ao qual estão associados um nome (*string*, um conjunto de referências para aplicações (*ApplCatalog::Info*), um conjuntos de referências para visualizadores (*VisCatalog::Info*), um conjunto de referências para requisições (*ReqCatalog::Info*) e um conjunto de referências para objetos geométricos (*ObjCatalog::Info*). Esses conjuntos são armazenados em estruturas de listas não ordenadas.

- *Aplicação* (*AppCatalog::Info*), classe que representa uma aplicação cadastrada no núcleo. Uma aplicação contém uma identificação (*ApplId*), um nome (*string*), uma

referência para o contexto da aplicação (*ContextCatalog::Info*) e um conjunto de referências para requisições (*ReqCatalog::Info*) feitas pela aplicação. Estas requisições são armazenadas através de uma estrutura de *hashing*, com um mapeamento injetivo do tipo *ReqId* para um conjunto de instâncias da classe *ReqCatalog::Info*.

- *Visualizador* (*VisCatalog::Info*), classe que representa um visualizador cadastrado no núcleo. Um visualizador contém uma identificação (*VisId*), um nome (*string*) e uma referência para o contexto do visualizador (*ContextCatalog::Info*).

- *Objeto geométrico* (*ObjCatalog::Info*), classe que representa um objeto geométrico inserido no núcleo, contendo uma identificação (*ObjId*), uma referência para o contexto ao qual pertence (*ContextCatalog::Info*) e uma instância de uma subclasse da classe *NetObj*, contendo a descrição geométrica do objeto.

- *Requisição* (*ReqCatalog::Info*), classe que representa uma requisição de objeto geométrico feita por alguma aplicação, contendo uma identificação (*ReqId*), uma referência para o contexto ao qual pertence (*ContextCatalog::Info*) e uma instância de uma subclasse da classe *NetObj*, contendo a descrição geométrica do objeto requisitado.

B.3 A interface *Application*

A figura B.6 ilustra o relacionamento entre as classes que compõem a implementação da interface *Application*, responsável pela integração entre as aplicações e o núcleo.

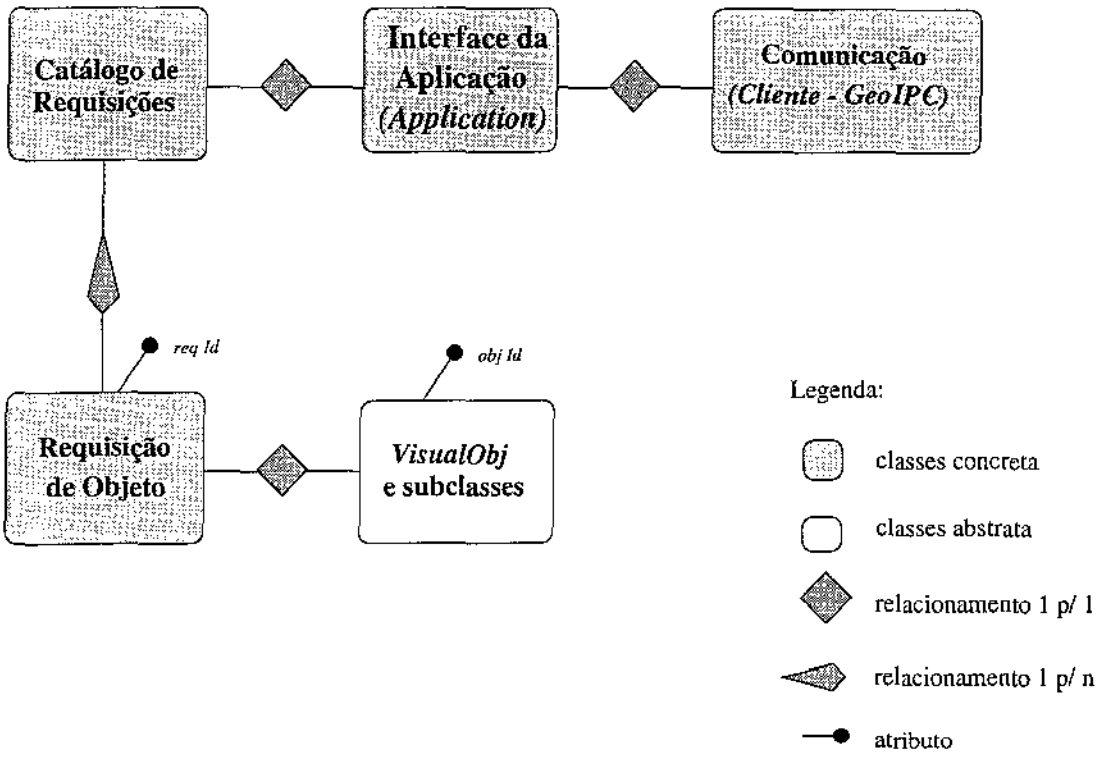

Figura B.6: O relacionamento entre as classes da interface das aplicações com o núcleo

Esta interface é o resultado da interação entre instâncias das seguintes classes:

- *Interface da Aplicação* (*Application*), responsável pelo interfaceamento entre a aplicação e o núcleo. A classe *Application* é apresentada na seção 5.4.
- *Comunicação* (*Client*). Uma instância da classe *Client* é responsável pela comunicação entre a aplicação e o núcleo. As classes de comunicação são descritas na seção B.1.
- *Catálogo de Requisições* (*ApplReqCatalog*), responsável pelo armazenamento das requisições de objetos feitas pelas aplicações, representadas por instâncias da subclasse *ApplReqCatalog::Info* e armazenados em uma estrutura de *hashing*, com um mapeamento injetivo do tipo *ReqId* para um conjunto de instâncias da classe *ApplReqCatalog::Info*.

- *Requisição* (*ApplReqCatalog::Info*), classe que representa uma requisição de objeto geométrico feita pela aplicação, contendo uma identificação (*ReqId*) e uma instância de uma subclasse da classe *VisualObj*, representando o objeto requisitado.

B.4 A interface *Visualizer*

A figura B.7 ilustra o relacionamento entre as classes que compõem a implementação da interface *Visualizer*, responsável pela integração entre os visualizadores e o núcleo.

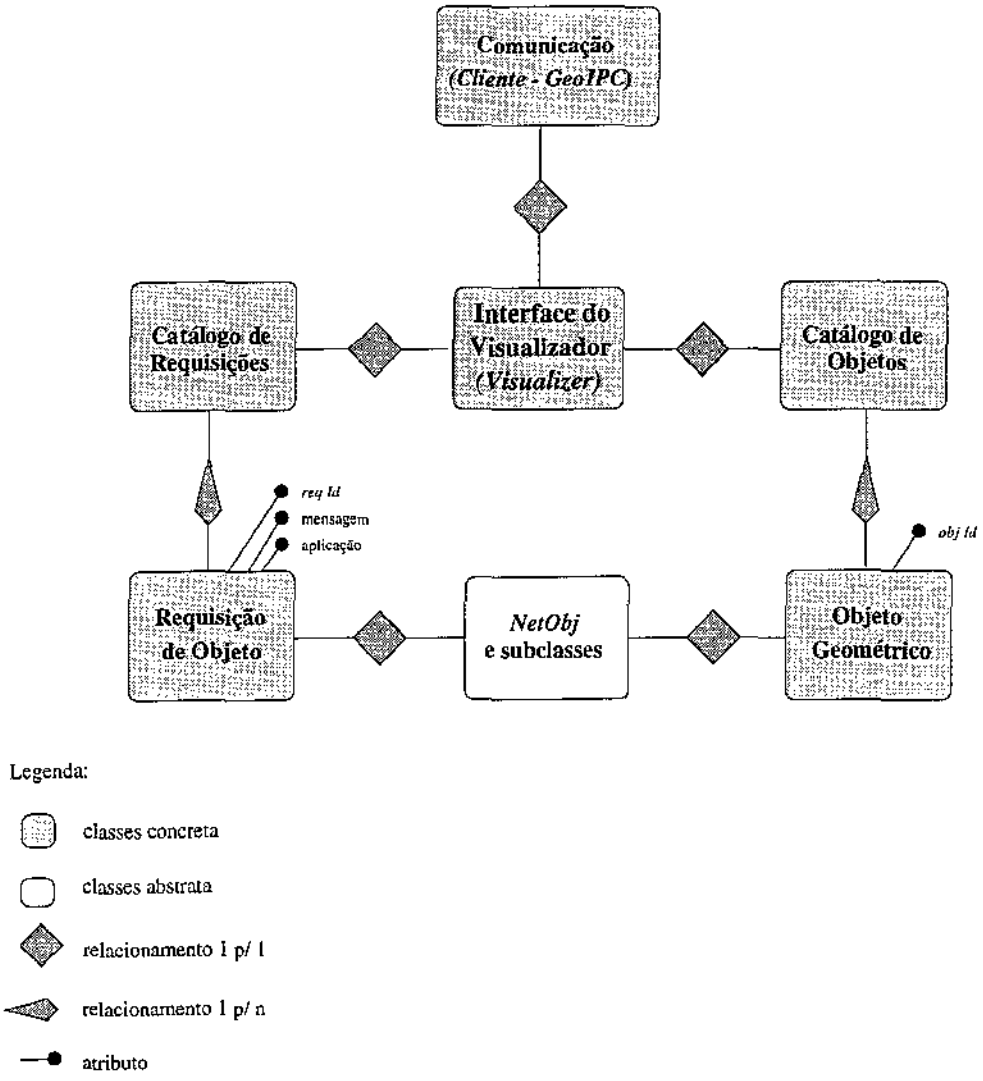

Figura B.7: O relacionamento entre as classes da interface dos visualizadores com o núcleo

Esta interface é o resultado da interação entre instâncias das seguintes classes:

- *Interface do Visualizador* (*Visualizer*) responsável pelo interfaceamento entre o visualizador e o núcleo. A classe *Visualizer* é apresentada na seção 5.5.
- *Comunicação* (*Client*). Uma instância da classe *Client* é responsável pela comunicação entre o visualizador e o núcleo. As classes de comunicação são descritas na

seção B.1.

- *Catálogo de Requisições* (*VisReqCatalog*), responsável pelo armazenamento das requisições de objetos feitas pelas aplicações, representadas por instâncias da subclasse *VisReqCatalog::Info* e armazenados em uma estrutura de *hashing*, com um mapeamento injetivo do tipo *ReqId* para um conjunto de instâncias da classe *VisReqCatalog::Info*.

- *Requisição* (*VisReqCatalog::Info*), classe que representa uma requisição de objeto geométrico feita por alguma aplicação, contendo uma identificação (*ReqId*) e uma instância de uma subclasse da classe *NetObj*, contendo a descrição geométrica do objeto requisitado.

- *Catálogo de Objetos* (*VisObjCatalog*), responsável pelo controle dos objetos geométricos visíveis no visualizador, representados por instâncias da subclasse *VisObjCatalog::Info* e armazenados em uma estrutura de *hashing*, com um mapeamento injetivo do tipo *ObjId* para um conjunto de instâncias da classe *VisObjCatalog::Info*.

- *Objeto Geométrico* (*VisObjCatalog::Info*), classe que representa um objeto geométrico visível no visualizador, contendo uma identificação (*ObjId*) e uma instância de uma subclasse da classe *NetObj*, contendo a descrição geométrica do objeto.

Bibliografia

[1] P. J. de Rezende and W. R. Jacometti. GeoLab: An environment for development of algorithms in computational geometry. In *Proc. 5th Canad. Conf. Comput. Geom.*, pages 175–180, Waterloo, Canada, 1993.

[2] P. J. de Rezende and J. Stolfi. *Fundamentos de Geometria Computacional.* IX Escola de Computação, 1994.

[3] P. J. de Rezende e C. N. Gon. GeoPrO: Geometria projetiva orientada com tratamento de degenerações. In *VII Simpósio Brasileiro de Computação Gráfica e Processamento de Imagens*, pages 315–316, 1995.

[4] H. Edelsbrunner and E. P. Mücke. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics*, 9(1):66–104, 1990.

[5] I. Z. Emiris and J. F. Canny. An efficient approach to removing geometric degeneracies. *Proc. 32nd Annual IEEE FOCS - San Juan*, pages 405–413, 1991.

[6] I. Z. Emiris and J. F. Canny. An efficient approach to removing geometric degeneracies. *Proc. 8th ACM Symp. Computational Geometry*, pages 74–82, June 1992.

[7] C. I. T. Force. Application challenges of computation geometry. Report TR-521-96, Princeton University, 1996.

[8] S. Fortune and C. V. Wyk. Efficient exact arithmetic for computational geometry. *ACM Symp. on Computational Geometry*, 9:163–172, 1993.

[9] C. N. Gon. "Envelope de um conjunto de retas", http://www.dcc.unicamp.br/~rezende/geopro/envelope.

[10] L. Guibas, D. Salesin, and J. Stolfi. Epsilon geometry: Building robust algorithms from imprecise computations. *Proc. of th 5th ACM Symp. on Computational Geometry*, pages 208–217, 1989.

[11] W. R. Jacometti. GeoLab – um ambiente para desenvolvimento de algoritmos em geometria computacional. Master's thesis, DCC - IMECC - UNICAMP, 1992.

[12] M. Keil. A simple algorithm for determining the envelope of a set of lines. *Inform. Process. Lett.*, 39:121–124, 1991.

[13] D. T. Lee, S.-M. Sheu, and C.-F. Shen. Geosheet: A distributed visualization tool for geometric algorithms. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, 1995.

[14] E. P. Mücke. private communication. 1995.

[15] T. Munzner, S. Levy, and M. Philips. Geomview: A system for geometric visualization. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages C12–C13, 1995.

[16] S. Naeher and K. Mehlhorn. LEDA: A library of efficient data types and algorithms. In *Proc. Internat. Colloq. Automata Lang. Program.*, pages 1–5, 1990.

[17] R. Palkovic. SunPro: The Sun programming environment - a Sun technical report. Technical report, Sun Microsystems, 1987.

[18] F. P. Preparata and M. I. Shamos. *Computation Geometry.* Springer-Verlag, 1985.

[19] P. Schorn. An object-oriented workbench for experimental geometric computation. In *Proc. 2nd Canad. Conf. Comput. Geom.*, pages 172–175, 1990.

[20] P. Schorn. Implementing the XYZ GeoBench: A programming environment for geometric algorithms. In *Proc. Internat. Workshop Comput. Geom. CG '91*, volume 553 of *LNCS*, pages 187–202. 1991.

[21] R. Seidel. The nature and meaning of perturbation in geometric computing. *Preprint*, 1995.

[22] J. Stolfi. *Oriented projective geometry: a framework for geometric computations.* Academic Press, Inc., 1st. edition, 1991.

[23] C. Yap. Towards exact geometric computation. *Fifth Canadian Conference on Computational Geometry*, pages 405–419, August 1993.

[24] C. Yap. The exact computation paradigm. *Pre-print*, August 1994.