MINISTÉRIO DA CIÊNCIA E TECNOLOGIA
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

ALGORITMOS GEOMÉTRICOS PARA BANCOS DE DADOS GEOGRÁFICOS: DA TEORIA À PRÁTICA NA TERRALIB

Gilberto Ribeiro de Queiroz

Dissertação de Mestrado em Computação Aplicada, orientada pelo Dr. Gilberto Câmara e pelo Dr. João Argemiro C. Paiva, aprovada em 22 de dezembro de 2003

INPE
São José dos Campos
2004

XX.XXX.XX(XXX.X)
QUEIROZ, G. R.
ALGORITMOS GEOMÉTRICOS PARA BANCOS DE DADOS GEOGRÁFICOS: DA TEORIA À PRÁTICA NA TERRALIB/ G. R. QUEIROZ. - São José dos Campos: INPE, 2003.
147p. - (INPE-XXXX-TDI/XXX).

1. TERRALIB. 2. TOPOLOGY (TOPOLOGIA). 3. ORDBMS (SGBDOR). 4. PLANE SWEEP (VARREDURA DO PLANO). 5. COMPUTATIONAL GEOMETRY (GEOMETRIA COMPUTACIONAL). 6. INTERSECTION (INTERSEÇÃO). 7. GIS (SIG).

FOLHA DE APROVAÇÃO

“Dealing with the finite nature of actual computers is an art that requires infinite patience”.

K. MULMULEY

A meus pais,
Gilberto e
Telma, e a minha querida Cássia.

AGRADECIMENTOS

Primeiramente, gostaria de expressar minha admiração e meus sinceros agradecimentos aos orientadores Dr. Gilberto Câmara e Dr. João Argemiro, pelos ensinamentos, paciência e atenção com que me orientaram neste trabalho. Também gostaria de agradecer a oportunidade que me foi dada de participar da equipe inicial de desenvolvimento da TerraLib.

Ao Dr. Antônio Miguel, pelas inúmeras colaborações com o trabalho, pelas conversas e experiências pessoais que contribuíram para minha formação pessoal e acadêmica.

Ao Dr. Clodoveu pelos livros e pelas discussões sobre Geometria Computacional.

Ao meu grande amigo Paulo Lima, que me apresentou a essa equipe fantástica da DPI/INPE.

Ao meu irmão Eduardo, pela ajuda.

À minha grande amiga Piedade, pelo apoio durante o mestrado.

Ao amigo Fernando Gibotti, que me deu a oportunidade de ministrar a primeira aula a alunos de graduação em informática no curso de técnicas de programação em C++ (FAIME/AGOSTO-2002).

Ao Dihel, pela disponibilidade de arrumar as bagunças do Windows e pelas dicas do Linux.

Ao amigo Júlio D'Alge pelas inúmeras explicações e discussões.

À amiga Hilcea Maria, pela ajuda.

Ao Juan, Lauro, Lúbia, Karine, Cartaxo e demais membros da equipe TerraLib, pelo apoio.

Aos amigos MHP, Bruno, Tiago, Cláudio, Reinaldo, ao palmeirense Glauco, Kaçapa, CAE, Brieba, Sidão e Tiago, pela amizade e pela boa convivência na República.

Aos amigos de sala: Cláudia Almeida, Paulina, Flávia, Rodrigo, Missae e Taciana.

À Ana Paula, pela avaliação dos operadores espaciais de conjunto e área.

Dinha, Helen, Terezinha e Janete por sempre me ajudarem com muita eficiência e carinho.

E a todos que de alguma forma contribuíram para a realização deste trabalho.

RESUMO

A variedade de algoritmos de geometria computacional, com seus diferentes compromissos de desempenho versus complexidade de implementação, representa um substancial desafio de engenharia de sistemas para os projetistas de sistemas de informação geográfica. Esse trabalho apresenta o estudo realizado sobre esses algoritmos, mostrando as técnicas empregadas e as decisões tomadas no desenvolvimento de operadores espaciais integrados ao ambiente TerraLib, que consiste em uma biblioteca para a construção de aplicativos geográficos. O papel dessas operações desenvolvidas é fornecer à biblioteca as funcionalidades não disponíveis em sistemas gerenciadores de bancos de dados sem o suporte espacial.

GEOMTETRIC ALGORITHMS FOR GEOGRAPHIC DATABASES: FROM THEORY TO PRACTICE IN TERRALIB

ABSTRACT

One of the greatest challenges of system engineering that a geographic information system designer has to face is the variety of computational geometry algorithms with their different performance commitments and complexities. This work describes the research on these algorithms showing techniques employed and the decisions taken when developing spatial operators integrated to the TerraLib environment, which is a library for development of geographic applications. The main goal of these operators is to provide, through this library, the functionalities that are not available in database management systems that do not include spatial support.

SUMÁRIO

Pág.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SÍMBOLOS

$\Re$ - Conjunto dos números reais

$\Re^2$ - Espaço bidimensional

π - Número PI

$\varnothing$ - Vazio

$\neg\varnothing$ - Não Vazio

∂O - Fronteira do objeto O

$O°$ - Interior do objeto O

O^- - Exterior do objeto O

LISTA DE SIGLAS E ABREVIATURAS

API	- Interface de Programação de Aplicações
BDG	- Banco de Dados Geográfico
BLOB	- Campo Binário Longo
GC	- Geometria Computacional
GIS	- Sistemas de Informação Geográfica
MBR	- Retângulo Envolvente Mínimo
OGIS	- Open GIS
SBDE	- Sistema de Bancos de Dados Espaciais
SDT	- Tipo de Dado Espacial
SFSSQL	- Simple Feature Specification For SQL
SGBD	- Sistema Gerenciador de Bancos de Dados
SGBD-OR	- Sistema Gerenciador de Bancos de Dados Objeto-Relacional
SGBD-R	- Sistema Gerenciador de Bancos de Dados Relacional

CAPÍTULO 1

INTRODUÇÃO

Sistemas de Informação Geográfica (Geographic Information Systems, GIS) são sistemas utilizados para representação computacional e análise de informações localizadas no espaço, possuindo aplicações em diversas áreas, como meio ambiente, saúde, gestão municipal e governamental (Câmara et al, 1996). De uma forma geral, esses sistemas compõem-se de três grandes subsistemas:

- Um banco de dados espaço-temporal, capaz de arquivar e recuperar dados provenientes de diferentes fontes (mapas temáticos, dados censitários e de cadastro urbano e rural, imagens de satélite e modelos numéricos de terreno);
- Um conjunto de técnicas de manipulação e análise de dados espaço-temporais, que incluem ferramentas de estatística espacial, consulta a objetos espaciais, processamento de imagens e geração de modelos numéricos de terreno;
- Um subsistema de interface e visualização, que oferece facilidades para apresentar os diferentes tipos de dados espaciais e suas representações.

Todos esses subsistemas fazem uso de algoritmos básicos de Geometria Computacional (GC) (Preparata e Shamos, 1985), que realizam operações atômicas sobre representações geométricas de dados espaço-temporais. Nesta dissertação, estaremos examinando um conjunto destes algoritmos que operam sobre representações vetoriais de dados geográficos (pontos, linhas e polígonos) e cuja ampla utilização e grande complexidade requerem cuidados especiais em seu projeto e programação.

Os problemas de GC vêm atraindo a atenção dos pesquisadores desde os primórdios da tecnologia de GIS (Nordbeck e Rystedt, 1967). Problemas como “pertinência de um ponto a um polígono”, “fecho convexo de um conjunto de pontos”, e “intersecção de linhas poligonais” têm sido objeto de grande número de estudos e propostas de algoritmos. De uma forma geral, estes algoritmos procuram otimizar seu desempenho

na situação de pior caso e são classificados conforme sua complexidade computacional utilizando a notação do "big O" (Cormen et al, 1990).

A necessidade de algoritmos adequados de GC em ambientes GIS pode ser facilmente verificada comparando-se o desempenho de algoritmos desenvolvidos sem qualquer refinamento, os chamados algoritmos de "força bruta", com aqueles projetados com técnicas mais sofisticadas. Para citar apenas um exemplo, um algoritmo de força bruta para interseção entre linhas poligonais apresenta complexidade de pior caso $O(n^2)$, enquanto a proposta de Bentley e Ottmann (1979) apresenta complexidade $O(n \log n + k \log n)$. Além disso, existem algoritmos com complexidades de pior caso semelhantes ao de força bruta, mas que no entanto utilizam técnicas que propiciam desempenhos práticos melhores.

A variedade de algoritmos de GC, com seus diferentes compromissos de desempenho versus complexidade de implementação, representa um substancial desafio de engenharia de sistemas para os projetistas de GIS. Se temos tantas alternativas, qual o melhor compromisso a adotar em cada caso? Será que dois algoritmos, de desempenhos teóricos semelhantes, tem o mesmo comportamento prático em termos de estabilidade e de desempenho em casos realistas de dados geográficos? Como se comporta, na prática do GIS, a teoria de Geometria Computacional?

Esta dissertação é uma tentativa de responder a estas perguntas. Partindo do estado-da-arte da teoria, nosso objetivo foi desenvolver algoritmos operacionais de GC para uma biblioteca de GIS de software livre (TerraLib) (Câmara et al, 2000), com requisitos de bom desempenho, fácil legibilidade e potencial de extensibilidade. Examinando as propostas da teoria de GC, verificamos que alguns algoritmos se mostraram mais adequados ao ambiente da TerraLib que outros. Em muitos casos, foi necessário fazer ajustes às propostas teóricas, incluindo adaptações às estruturas de dados da TerraLib e processamentos adicionais não indicados pelos autores.

Os algoritmos estudados neste trabalho foram:

- Cálculo da orientação entre três pontos;
- Cálculo de área de polígonos;

- Orientação dos vértices de um polígono;
- Teste de convexidade;
- Determinação do envoltório convexo de um conjunto de pontos;
- Ponto em polígono;
- Interseção entre dois segmentos;
- Interseção entre linhas poligonais;
- Operações de conjunto entre polígonos: união, interseção e diferença;
- Geração de mapas de distância (*buffer zones*);
- Determinação do relacionamento topológico.

Situações típicas em que estes algoritmos são utilizados nos GIS incluem:

- Seleção por apontamento, onde um usuário seleciona um determinado objeto através da interface gráfica;
- Consultas espaciais que envolvem restrições espaciais entre os objetos, como determinar os municípios vizinhos de uma determinada cidade;
- Criação de mapas de distância (*buffer zones*) ao redor dos objetos para determinar regiões de influência;
- Sobreposição de polígonos para extração de informações, como por exemplo, a porcentagem da área de um município coberta por um determinado tipo de vegetação;

Como resultado, este trabalho mostra os compromissos de projeto que são necessários para adaptar propostas teóricas a um ambiente computacional operacional. Nossos resultados indicam que é necessário considerável esforço de revisão conceitual e de espírito crítico para fazer com que a teoria funcione na prática. Uma motivação adicional é estar contribuindo para o desenvolvimento da biblioteca TerraLib, que

facilitará o desenvolvimento de aplicativos GIS, dando um grande incentivo à disseminação dessa tecnologia, baseada em software livre.

1.1 Organização da Dissertação

Inicialmente é feita, no Capítulo 2, uma apresentação da tecnologia de bancos de dados geográficos, incluindo arquitetura de GIS, topologia, indexação espacial, extensões espaciais dos SGBDs comerciais e gratuitos e uma fundamentação teórica da biblioteca TerraLib, que é o ambiente utilizado no desenvolvimento do trabalho.

A seguir, o Capítulo 3 apresenta as técnicas e algoritmos de Geometria Computacional que foram utilizados no desenvolvimento dos operadores espaciais da TerraLib. Nesse capítulo é feito o levantamento dos problemas subjacentes às operações espaciais, mostrando as cotas inferiores para resolução desses problemas e a complexidade de alguns algoritmos existentes para a resolução dos mesmos. As decisões tomadas na escolha dos algoritmos e o projeto dos operadores integrados na biblioteca TerraLib são apresentados neste capítulo, que ainda inclui um algoritmo para determinação do relacionamento topológico integrando os algoritmos geométricos nele apresentados.

No Capítulo 4 é apresentado o uso prático de alguns dos operadores dentro de um aplicativo geográfico construído sobre a TerraLib. E, finalmente, o Capítulo 5 apresenta as conclusões do trabalho e direções para trabalhos futuros.

O Apêndice A contém um resumo sobre a especificação do Consórcio Open GIS para a construção de extensões geográficas.

O Apêndice B contém códigos fontes ilustrativos do mecanismo de extensibilidade do PostgreSQL apresentado no Capítulo 2.

O Apêndice C contém alguns trechos de códigos ilustrativos, selecionados dos algoritmos geométricos, e a semântica definida para os operadores topológicos da TerraLib.

CAPÍTULO 2

BANCOS DE DADOS GEOGRÁFICOS

Nos últimos anos, as pesquisas na área de bancos de dados têm, cada vez mais, dado enfoque ao desenvolvimento de suporte a aplicações conhecidas como não tradicionais (Schneider, 1997), dentre elas sobressaem as aplicações conhecidas como GIS. A integração dos Sistemas Gerenciadores de Bancos de Dados (SGBD) com os GIS tem o potencial de mudar o panorama da maioria dos atuais sistemas disponíveis no mercado, que são baseados em estruturas proprietárias e que implementam várias das funcionalidades que um SGBD possui.

Os SGBD podem desempenhar um papel fundamental no desenvolvimento de novos GIS, assim como desempenham um papel fundamental em aplicações corporativas. Neste último tipo de aplicação, eles liberam os programadores para se preocuparem em resolver os problemas do domínio do negócio em si e não no desenvolvimento de funcionalidades básicas de armazenamento, tais como recuperação, controle de integridade e concorrência sobre os dados. Eles podem beneficiar os desenvolvedores de GIS, permitindo o enfoque em funcionalidades como ferramentas para análise espacial, visualização gráfica dos dados e entrada de dados.

Este capítulo apresenta uma revisão sobre os principais tópicos envolvidos no gerenciamento de dados geográficos através de um SGBD, as extensões geográficas disponíveis atualmente e os fundamentos da biblioteca TerraLib.

2.1 Sistemas de Bancos de Dados

Atualmente, os SGBD disponíveis no mercado pertencem a duas famílias: os Relacionais (SGBD-R) (Date, 1990) e os Objeto-Relacionais (SGBD-OR) (Silberschatz et al, 1999).

O modelo de dados dos SGBD-R consiste de uma coleção de relações (tabelas), cada qual com atributos (colunas) de um tipo específico (domínio). Nos sistemas comerciais atuais esses tipos incluem números inteiros, de ponto flutuante, cadeias de caracteres, datas e campos binários longos (BLOBs). Para esses tipos encontram-se disponíveis uma variedade de operadores (com exceção ao BLOB), como operações aritméticas, de conversão, de manipulação textual e operações com data. Outra característica dos SGBD-R é que eles fornecem eficientes mecanismos de armazenamento, indexação de dados unidimensionais através de B-Tree (Garcia-Molina et al, 2001), controle de concorrência e integridade dos dados armazenados.

Essa categoria de SGBD é fortemente voltada para aplicações corporativas (ou de automação comercial), manipulando grandes volumes de dados convencionais, isto é, dados que não levam em consideração, por exemplo, o componente espacial de um dado geográfico. Reconhece-se que tanto o modelo de dados quanto os operadores e métodos de acesso disponíveis nestes SGBD não são suficientes para atender a todos os tipos de aplicações (Frank, 1984), como no caso dos GIS, onde o componente espacial de um dado geográfico deve ser indexada através de mecanismos especiais (multidimensionais), como a R-Tree (Guttman, 1984).

Stonebraker (1996) aponta que a simulação de tipos de dados através de um esquema de tabelas em SGBD-R pode ter efeitos colaterais como a queda de desempenho, a dificuldade de codificação e a posterior manutenção do código da aplicação. Ele cita o caso de uma aplicação GIS que, ao tentar simular os tipos de dados, apresenta uma série de problemas. A fim de obter uma maior integração entre aplicações não-tradicionais (caso dos GIS) e os SGBDs, é necessário utilizar uma outra tecnologia de bancos de dados emergente, os SGBD-OR.

Os SGBD-OR estendem o modelo relacional, e apresentam entre as suas principais características a capacidade de fornecer um sistema de tipos de dados mais rico e passível de extensão com tipos e operadores que podem ser utilizados na linguagem de consulta. Além disso, possibilitam a extensão dos mecanismos de indexação sobre os novos tipos. Essas características eliminam os problemas ocorridos na simulação de

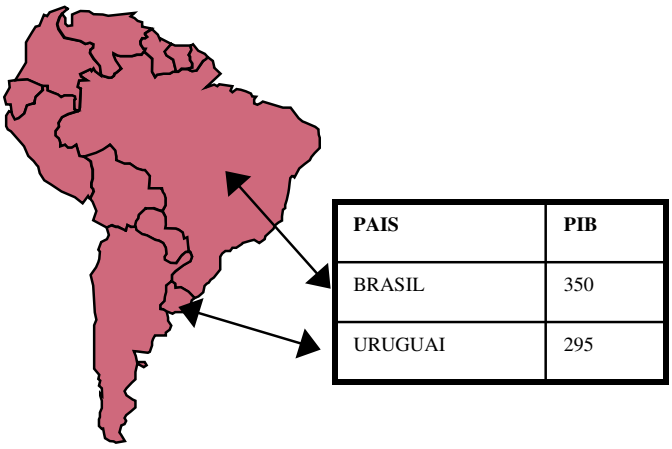
tipos de dados pelos SGBD-R, o que torna os SGBD-OR uma solução atrativa na integração com os GIS.

2.2 Arquitetura de GIS

Os dados que um Sistema de Informação Geográfica (GIS) manipula representam objetos e fenômenos, cuja localização geográfica é uma das principais características utilizadas em sua análise. Esses dados possuem dois componentes principais:

- **Atributos não espaciais:** também chamados de atributos alfanuméricos, que são um conjunto de atributos usados para descrever o dado geográfico de estudo, como por exemplo, nome e população de um país (Tabela na Figura 2.1);

- **Atributo espacial ou componente espacial:** especifica a localização geográfica e a forma (geometria) do objeto em estudo. Por exemplo, o território de um país pode ser representado por um polígono no espaço bi-dimensional (Mapa da Figura 2.1).



PAIS	PIB
BRASIL	350
URUGUAI	295

FIGURA 2.1 – Exemplo de dado geográfico.

FONTE: adaptada de (DPI/INPE, 2002).

A seguir são apresentadas as arquiteturas utilizadas pelos GIS, no que diz respeito à sua integração com os SGBDs (Davis e Câmara, 2001). Conforme será visto, elas diferem-

se, principalmente, na maneira e nos recursos utilizados para armazenar e recuperar dados geográficos.

2.2.1 Arquitetura Dual

A arquitetura dual caracteriza-se por utilizar um SGBD-R no gerenciamento da parte não espacial dos dados geográficos e formatos proprietários para armazenar os atributos espaciais, usando um identificador comum na interligação dos dois tipos de atributos. A Figura 2.2 ilustra esta arquitetura.

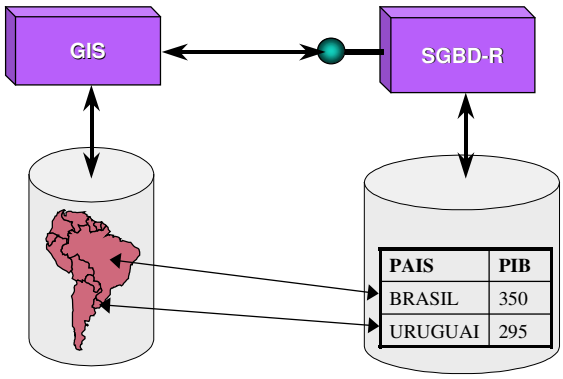


FIGURA 2.2 – Esquema da Arquitetura Dual.

FONTE: adaptada de (Davis e Câmara, 2001).

A principal vantagem dessa arquitetura é a atribuição de parte do gerenciamento dos dados aos SGBDs, pois os mecanismos de transação, recuperação a falhas, concorrência, integridade dos dados e indexação podem ser usados para a parte não espacial dos dados. No entanto, ela apresenta a falha de não integrar totalmente o dado geográfico, o que torna necessário para a aplicação desenvolver mecanismos que façam o controle de integridade, com a finalidade de evitar problemas de inconsistência, e também mecanismos para a indexação da parte espacial. Outro problema apresentado diz respeito às consultas diretamente em SQL, que ficam restritas à parte alfanumérica. O SPRING (DPI/INPE, 2002) é um exemplo de sistema que utiliza este tipo de arquitetura.

2.2.2 Arquitetura Baseada em Campos Longos

Esta arquitetura caracteriza-se por armazenar no banco de dados tanto os atributos espaciais quanto os alfanuméricos. Os atributos espaciais são armazenados utilizado-se de campos binários longos (BLOBs). A Figura 2.3 ilustra esta arquitetura.

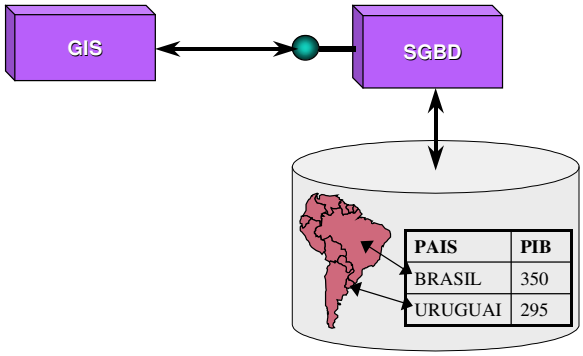


FIGURA 2.3 – Esquema da Arquitetura Baseada em Campos Longos.

FONTE: adaptada de (Davis e Câmara, 2001).

Uma das vantagens dessa arquitetura é a integração dos dados em um único ambiente, o que possibilita a utilização dos mecanismos de controle de integridade, transação e controle de concorrência, eliminando, assim, possíveis problemas de inconsistência. Por outro lado, essa estratégia fica limitada pelo pouco conhecimento que o SGBD possui sobre o dado armazenado como BLOB.

2.2.3 Arquitetura Baseada em Extensões

Uma forma de contornar os problemas de dados armazenados como BLOBs é através do uso de extensões espaciais dos SGBDs, como as extensões da Oracle e da IBM, ou através da definição de novos tipos e operadores, no caso de SGBDs extensíveis como o PostgreSQL. Entre as vantagens dessa arquitetura pode-se citar a existência de excelentes mecanismos de indexação, tanto de atributos convencionais quanto de atributos espaciais, e a existência de funções já definidas para tratamento dos atributos espaciais (operadores topológicos e métricos). Essas extensões são discutidas em maiores detalhes na Seção 2.6.

2.2.4 Arquitetura Combinada

No caso de aplicativos GIS que manipulam dados representados tanto na forma matricial quanto vetorial, é possível a utilização de uma arquitetura formada pela combinação das duas últimas. Neste caso, a parte vetorial é armazenada utilizando-se os recursos oferecidos pelas extensões, e a parte matricial é armazenada em BLOBs.

A TerraLib é um exemplo de biblioteca geográfica que utiliza essa arquitetura para integrar o dado geográfico em um único ambiente. As funcionalidades para manipulação de dados no formato matricial são fornecidas pela biblioteca, de modo a complementar os recursos ausentes, até o momento, nos SGBD.

2.3 Componentes de Sistemas de Bancos de Dados Espaciais

Um Sistema de Bancos de Dados Espacial (SBDE) é um sistema de bancos de dados com capacidades adicionais que permitem representar, consultar e manipular dados cuja localização espacial é uma das principais características. Entre as pesquisas nesta área, pode-se citar Shekar et al (1999), Schneider (1997), Medeiros e Pires (1994) e Güting (1994), que apontam os principais requisitos, técnicas e linhas de pesquisa.

Esses trabalhos apontam que os requisitos e funcionalidades de um SBDE são bastante influenciados pelos GIS, servindo como tecnologia de suporte à construção desses sistemas. Um SBDE fornece apenas algumas funcionalidades básicas de gerenciamento de dados, não sendo considerado um GIS completo. Funcionalidades como a de visualização, entrada de dados e análise espacial são consideradas de responsabilidade de implementação do GIS e não do SBDE.

Entre as principais características desses sistemas tem-se:

- Modelo de dados com suporte a tipos de dados espaciais (Spatial Data Types, SDT), como ponto, linha e polígono;

- Funções e operadores espaciais que permitem que esses tipos de dados sejam manipulados, assim como os tipos alfanuméricos básicos.

- Métodos de indexação espacial;
- Linguagem de consulta estendida para permitir a manipulação dos SDTs.

A fim de estabelecer um conjunto de funcionalidades básicas que todo SBDE deveria incorporar, o consórcio Open GIS (OGIS), apoiado pelos principais líderes do mercado mundial de banco de dados (Oracle, IBM e Informix), desenvolveu uma especificação conhecida como *Simple Features Specification For SQL – SFSSQL* (OGIS, 1995).

Essa especificação apresenta a proposta do modelo dos tipos geométricos, dos operadores topológicos, métricos e dos que geram novas geometrias que todo SBDE deveria apresentar, a fim de se ter um mínimo denominador comum entre eles. Além disso, essa especificação apresenta uma proposta de um esquema de metadados para as tabelas que contenham tipos geométricos.

Para as operações topológicas, a SFSSQL adotou o modelo baseado na matriz de 9-interseções dimensionalmente estendida (Clementini e Di Felice, 1995), aqui abreviadamente denominada matriz 9-interseções DE. A Seção 2.4 é dedicada à apresentação desse modelo, que também é adotado no projeto dos operadores topológicos da TerraLib.

A indexação espacial não discutida nessa especificação é apresentada em maiores detalhes na Seção 2.5, servindo de apoio para a revisão dos atuais SBDE existentes no mercado e no "mundo" do software livre, apresentados na Seção 2.6. O Apêndice A contém maiores informações sobre a especificação do OGIS.

2.4 Relacionamentos Topológicos

Freqüentemente, os usuários de um GIS utilizam consultas espaciais, que são restrições baseadas em relacionamentos espaciais entre os objetos, para analisar informações. Existem vários tipos de relacionamentos espaciais: métricos, direcionais e topológicos. O entendimento formal destes é fundamental para a análise de dados nos GIS, sendo importante utilizar uma ferramenta que permita descrevê-los (Abler, 1987). No caso dos

topológicos, existem várias propostas de modelos com o objetivo de descrever os possíveis relacionamentos entre dois objetos.

O modelo apresentado por Egenhofer e Franzosa (1991), conhecido como Matriz de 4-Interseções, baseia-se na determinação das interseções entre interiores ($O°$) e fronteiras (∂O) de dois objetos (A e B). Essas interseções podem ser representadas numa matriz 2×2 e, considerando os valores vazio ($\varnothing$) e não vazio ($\neg\varnothing$) entre as interseções, pode-se distinguir 16 relações. Para duas regiões simples (fronteiras conectadas) em $\Re^2$, apenas 8 delas podem ser realizadas, sendo duas simétricas (*Covers/Covered By* e *Inside/Contains*) . A Figura 2.4 ilustra esse modelo.

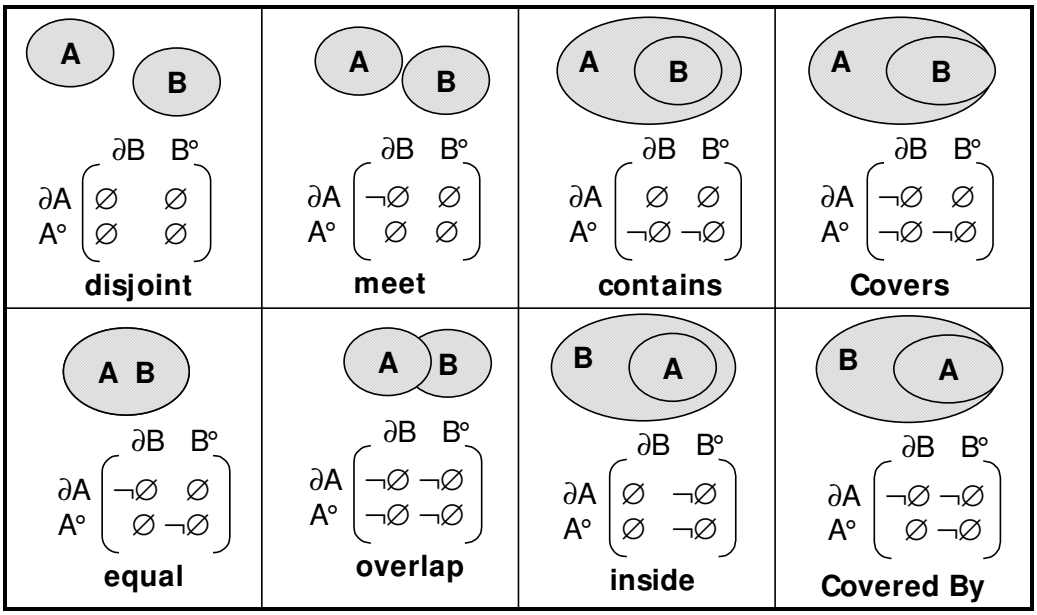


FIGURA 2.4 – Matriz de 4-Interseções para relações entre duas regiões.

FONTE: adaptada de (Egenhofer et al, 1994).

O modelo acima foi estendido de forma a considerar as interseções com os exteriores dos objetos (O^-), sendo conhecido como Matriz de 9-Interseções (Egenhofer e Herring, 1991). Com essa modificação é possível distinguir algumas configurações topológicas diferentes, que não são detectadas no primeiro modelo citado, principalmente entre linhas e entre linhas e regiões. Nessa abordagem há uma combinação de 512

relacionamentos, dos quais em $\Re^2$ podem ser realizados 8 entre regiões simples, 33 entre linhas e 19 entre regiões simples e linhas, 2 entre pontos e 3 entre pontos e linhas e regiões. A Figura 2.5 ilustra esse modelo para regiões.

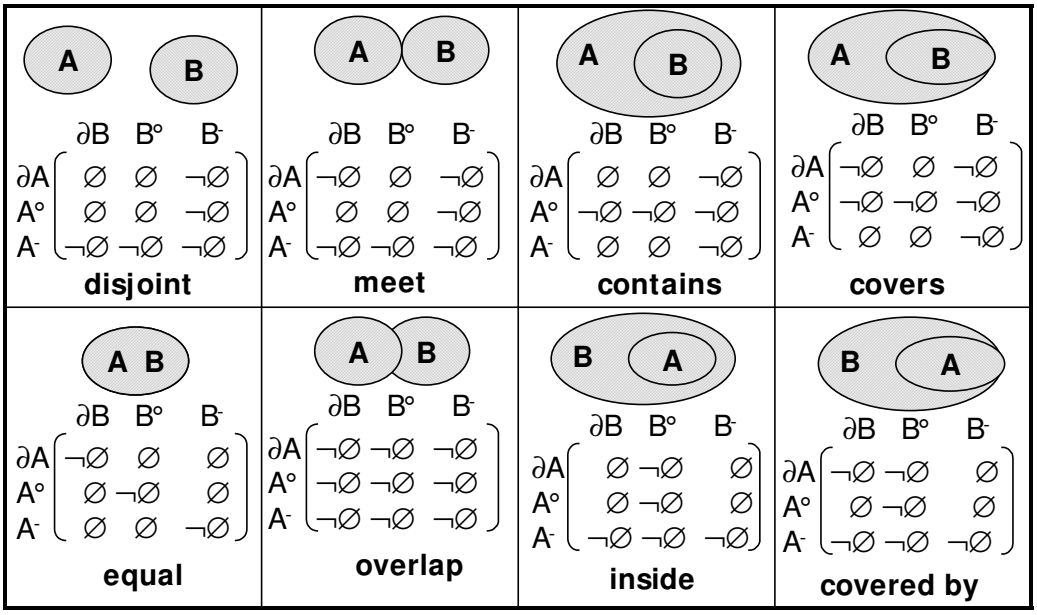


FIGURA 2.5 – Matriz de 9-Interseções para relações entre duas regiões.

FONTE: adaptado de (Egenhofer e Herring, 1991).

Clementini et al (1993) estenderam a abordagem da Matriz de 4-Interseções de forma a incluir a informação da dimensão da interseção. No espaço bidimensional a dimensão da interseção pode ser vazia, 0D (ponto), 1D (linha) ou 2D (região). O resultado dessa extensão é um conjunto de 52 relacionamentos, que é uma combinação muito grande para um usuário utilizar. A fim de realizar o mapeamento entre conceitos de um nível geométrico para um nível mais alto (dirigido ao usuário), os relacionamentos foram agrupados no menor número possível, resultando em relacionamentos topológicos um pouco mais gerais: *touch*, *in*, *cross*, *overlap*, *disjoint*. Esses cinco relacionamentos são conceitos sobrecarregados no sentido de que eles podem ser usados para ponto, linha e área.

O OGIS adotou a evolução desse modelo, chamado de Matriz de 9-Interseções DE (Clementini e Di Felice, 1995). Os relacionamentos são definidos da seguinte forma:

- **Touch:** aplica-se a região/região, linha/linha, linha/região, ponto/região e ponto/linha.

$$\langle \lambda_1, touch, \lambda_2 \rangle \Leftrightarrow (\lambda_1^o \cap \lambda_2^o = \emptyset) \wedge$$
$$((\partial\lambda_1 \cap \lambda_2^o \neq \emptyset) \vee (\lambda_1^o \cap \partial\lambda_2 \neq \emptyset) \vee (\partial\lambda_1 \cap \partial\lambda_2 \neq \emptyset))$$

- **In:** aplica-se a todas as combinações.

$$\langle \lambda_1, in, \lambda_2 \rangle \Leftrightarrow (\lambda_1^o \cap \lambda_2^o \neq \emptyset) \wedge (\lambda_1^o \cap \lambda_2^- = \emptyset) \wedge (\partial\lambda_1 \cap \lambda_2^- = \emptyset)$$

- **Cross:** aplica-se a linha/linha, linha/região e linha/região.

No caso linha/região:

$$\langle L, cross, R \rangle \Leftrightarrow (L^o \cap R^o \neq \emptyset) \wedge (L^o \cap R^- \neq \emptyset)$$

No caso linha/linha:

$$\langle L_1, cross, L_2 \rangle \Leftrightarrow \dim(L_1^o \cap L_2^o) = 0$$

- **Overlap:** aplica-se a região/região e linha/linha.

No caso região/região:

$$\langle A_1, overlap, A_2 \rangle \Leftrightarrow (A_1^o \cap A_2^o \neq \emptyset) \wedge (A_1^o \cap A_2^- \neq \emptyset) \wedge (A_1^- \cap A_2^o \neq \emptyset)$$

No caso linha/linha:

$$\langle L_1, overlap, L_2 \rangle \Leftrightarrow (\dim(L_1^o \cap L_2^o) = 1) \wedge (L_1^o \cap L_2^- \neq \emptyset) \wedge (L_1^- \cap L_2^o \neq \emptyset)$$

- **Disjoint:** aplica-se a todas as combinações.

$$\langle \lambda_1, disjo\,\mathrm{int}, \lambda_2 \rangle \Leftrightarrow (\lambda_1^o \cap \lambda_2^o = \emptyset) \wedge (\partial\lambda_1 \cap \lambda_2^o = \emptyset) \wedge$$
$$(\lambda_1^o \cap \partial\lambda_2 = \emptyset) \wedge (\partial\lambda_1 \cap \partial\lambda_2 = \emptyset)$$

2.5 Indexação Espacial

Os métodos de indexação espacial têm por objetivo acelerar o processamento das consultas espaciais, como as que envolvem operações topológicas, organizando os registros de tal maneira que objetos que estão mais próximos um dos outros no espaço correspondam a registros próximos um do outro no índice. Vários métodos são apresentados na literatura (Gaede e Günther, 1998) e, entre eles, os sistemas comerciais atuais empregam índices das seguintes famílias:

- **R-Tree:** este método é uma extensão da B-Tree para o espaço multidimensional, onde os objetos são aproximados através de seus retângulos envolventes mínimos (MBR) (Guttman, 1984). Conforme pode ser visto na Figura 2.6 os nós internos contém entradas da forma <MBR, ptr-nó>, onde MBR é o retângulo que envolve todos os retângulos do nó filho (apontado por ptr-nó). As entradas dos nós folha contêm o MBR do objeto e um ponteiro para ele. A busca nessa árvore difere da B-Tree pela possibilidade de ser necessário pesquisar pelo dado desejado em mais de uma sub-árvore, no caso do retângulo de pesquisa sobrepor mais de um retângulo em cada nível. Ciferri e Salgado (2001) apresentam um estudo comparativo de algoritmos dessa família que, geralmente, se diferenciam na estratégia adotada para manter o balanceamento da árvore no caso de divisão de um nó.

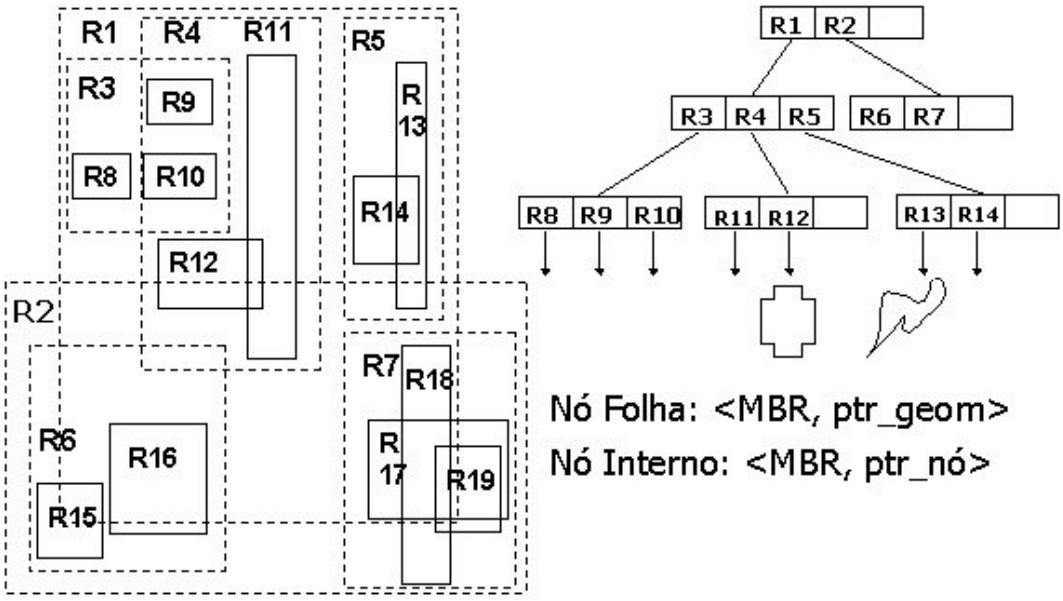


FIGURA 2.6 – Esquema da R-Tree.

- **Fixed Grid** (Worboys, 1995)**:** neste método de indexação, o espaço é particionado em uma grade retangular, onde cada célula é associada a uma página do disco. Essa associação é feita através de uma matriz bidimensional, conhecida como diretório, onde os elementos possuem o endereço de uma página. A Figura 2.7 ilustra a representação deste tipo de índice utilizada para o caso de pontos. Para objetos mais complexos, tais como polígonos, é utilizada a aproximação pelo MBR, sendo o objeto associado às páginas das células com as quais o seu MBR intercepta. A resolução da grade é um ponto de grande importância para este tipo de indexação.

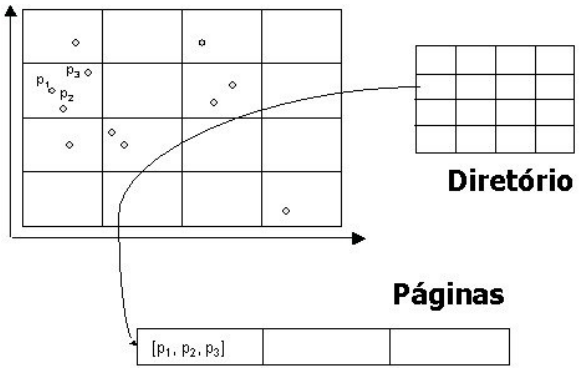


FIGURA 2.7 – Esquema do *Fixed Grid*.

- **QuadTree:** a idéia básica deste método é subdividir o espaço em quadrantes. A árvore é formada por nós que possuem quatro descendentes que representam os quatro quadrantes nos quais o original do nó foi subdividido. Quadrantes que não são mais subdivididos são armazenados em nós folha. Esse método pode ser aplicado a dados no formato matricial, pontos e objetos mais complexos como polígonos (Samet, 1990). A Figura 2.8 ilustra esse método de indexação.

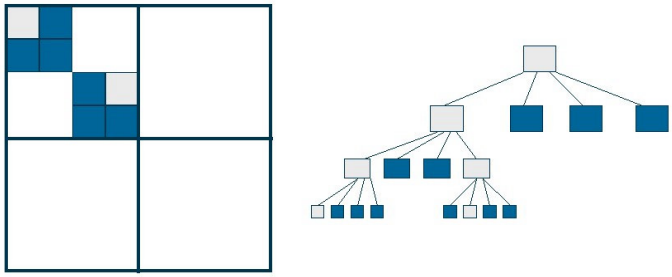


FIGURA 2.8 – Esquema da QuadTree.

2.6 SGBDs com Extensões Geográficas

Atualmente, existem basicamente três extensões comerciais disponíveis no mercado para tratar de dados geográficos no formato vetorial: Oracle Spatial (Ravada e Sharma, 1999), IBM DB2 Spatial Extender (IBM, 2002) e Informix Spatial e Geodetic Datablade (IBM, 2003). No universo do software de código fonte aberto e gratuito existem dois projetos para a criação de extensões espaciais, uma baseada no SGBD PostgreSQL (Stonebraker et al, 1990) chamada de PostGIS (Ramsey, 2002) e outra baseada no MySQL (Yarger et al, 1999).

Todas essas extensões são baseadas nas especificações da SFSSQL do OpenGIS (OGIS, 1995). A seguir, são apresentadas as características e funcionalidades das extensões da Oracle, do PostgreSQL e do MySQL. No final da seção encontra-se um quadro resumo com a comparação das características de cada uma delas e algumas considerações sobre a forma adotada por essas extensões para representar as geometrias.

2.6.1 Oracle Spatial

O Oracle Spatial é uma extensão espacial do SGBD Oracle, que utiliza seu modelo objeto-relacional. Esta extensão contém um conjunto de funcionalidades e procedimentos que permitem armazenar, acessar e analisar dados espaciais em um banco de dados Oracle. Seu modelo de dados consiste em uma estrutura hierárquica de elementos, geometrias e camadas de informação[1]. No nível mais baixo desse modelo tem-se os elementos, que são tipos geométricos primitivos (Figura 2.9).

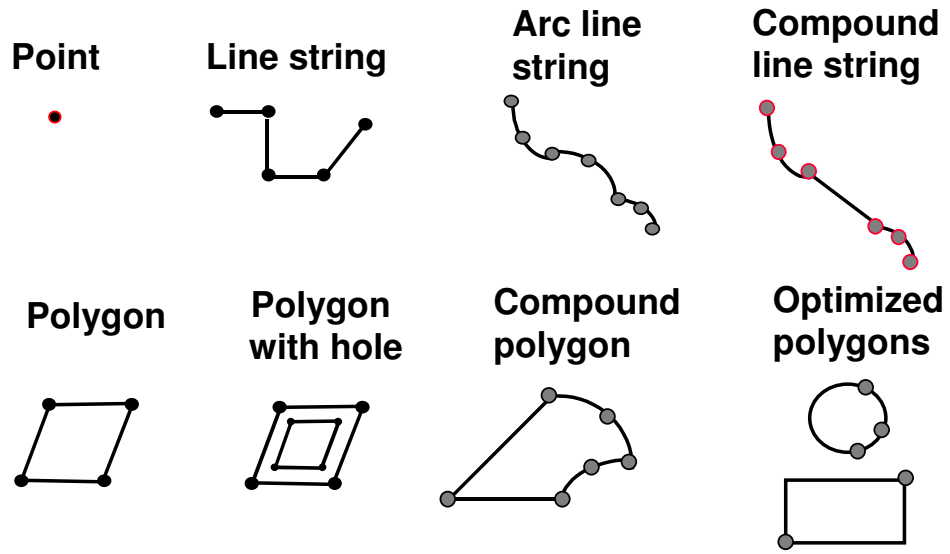


FIGURA 2.9 – Tipos de Elementos Primitivos do Oracle Spatial.

FONTE: adaptada de (Oracle, 2003).

As geometrias são compostas pelos elementos primitivos, podendo ser de um dos tipos ilustrado na Figura 2.10. E, finalmente, uma camada é formada por um conjunto de geometrias que possuem os mesmos atributos (tabela com uma coluna geométrica).

[1] Neste trabalho optamos pela expressão *camada de informação* no lugar de *plano de informação* ou *layer*.

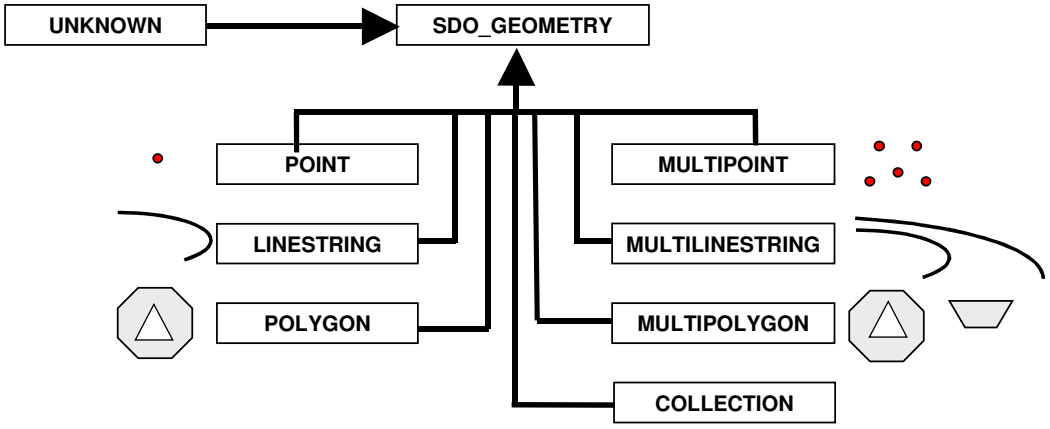


FIGURA 2.10 – Modelo de Geometrias do Oracle Spatial.

FONTE: adaptada de (Oracle, 2003).

Em uma tabela espacial, os atributos alfanuméricos da geometria são definidos como colunas de tipos básicos (VARCHAR2, NUMBER, etc) e a geometria, como uma coluna do tipo SDO_GEOMETRY. Esse tipo possui a seguinte estrutura:

```
SDO_GEOMETRY
{
     SDO_GTYPE      NUMBER
     SDO_SRID       NUMBER
     SDO_POINT      SDO_POINT_TYPE
     SDO_ELEM_INFO  SDO_ELEM_INFO_ARRAY
     SDO_ORDINATES  SDO_ORDINATE_ARRAY
}
```

Um objeto deste tipo, que armazena uma geometria, contém a informação do tipo da geometria (SDO_GTYPE), o sistema de projeção (SDO_SRID) das coordenadas da geometria (SDO_ORDINATES) e os tipos dos elementos que a formam (SDO_ELEM_INFO). Geometrias do tipo ponto podem ser armazenadas diretamente em SDO_POINT.

Além dos tipos geométricos, esta extensão fornece um conjunto de operadores e funções espaciais que são utilizados juntamente com a linguagem SQL para dar suporte às consultas espaciais. Para consultar relações topológicas entre duas geometrias é utilizado um operador chamado SDO_RELATE, que é implementado segundo a Matriz

de 9-interseções. Quanto à indexação espacial, esta extensão fornece dois tipos de índices: a R-Tree e a QuadTree (Kothuri et al, 2002).

A Figura 2.11 ilustra uma parte do mapa das cidades brasileiras, destacando a cidade de Araxá e os municípios vizinhos a ela.

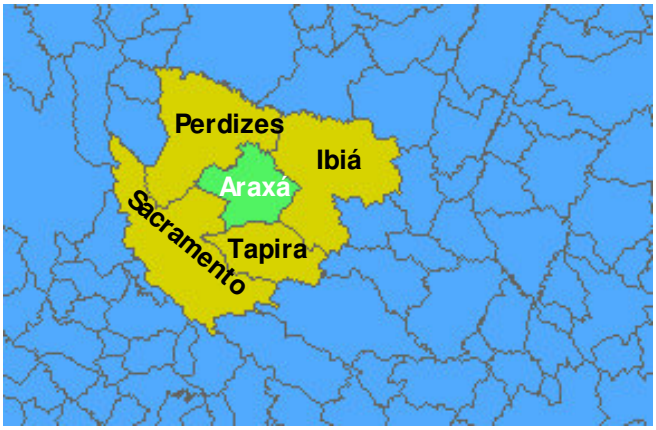


FIGURA 2.11 – Exemplo de Consulta Espacial Envolvendo Operadores Topológicos.

Uma possível maneira de representar as informações sobre esse mapa poderia ser através da tabela abaixo:

Municípios

Nome Atributo	**Tipo**
cod_municipio	NUMBER
Nome	VARCHAR(40)
Geometria	SDO_GEOMETRY

Uma consulta espacial para descobrir as cidades vizinhas a Araxá poderia ser montada da seguinte forma:

```
SELECT M2.nome
  FROM municipios M1, municipios M2
 WHERE M2.nome <> 'ARAXÁ'
   AND SDO_RELATE(M1.geometria, M2.geometria, 'MASK=TOUCH
```

```
    QUERYTYPE=WINDOW') = 'TRUE'
AND M1.nome = 'ARAXÁ'
```

Considerando que um índice espacial esteja definido sobre a coluna geométrica da tabela "municípios", o processamento da consulta acima poderia utilizá-lo, filtrando as cidades vizinhas pelo MBR das geometrias. E, a partir das cidades pré-selecionadas pelo índice, a função SDO_RELATE computaria o relacionamento topológico baseado na geometria exata, retornando as cidades que realmente são vizinhas a Araxá.

2.6.2 PostgreSQL

O PostgreSQL é um sistema gerenciador de banco de dados objeto-relacional, gratuito e de código fonte aberto, que desenvolveu-se a partir do projeto Postgres, iniciado em 1986, na Universidade da Califórnia (Berkeley), sob a liderança do professor Michael Stonebraker. Em 1995 foi adicionado suporte a SQL e o código fonte foi disponibilizado na Web (http://www.postgresql.org). Desde então, um grupo de desenvolvedores vem mantendo e aperfeiçoando o código fonte sob o nome de PostgreSQL.

Um dos recursos mais importantes desse SGBD é o seu mecanismo de extensibilidade, que permite a criação de tipos de dados, funções e operadores definidos pelo usuário. Esta funcionalidade e mais a capacidade de gerenciar grandes volumes de dados, tornam o PostgreSQL uma solução atraente na integração com os GIS, como veremos nas seções que discutem o seu mecanismo de extensibilidade e o projeto PostGIS, respectivamente.

Em sua versão de distribuição ele apresenta tipos de dados geométricos como ilustrado na Figura 2.12, uma R-Tree implementada da forma original proposta por Guttman que permite indexar as tabelas com esses tipos geométricos, e operadores espaciais simples.

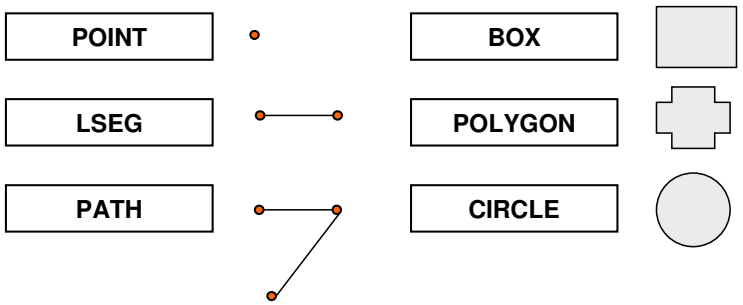


FIGURA 2.12 – Tipos Geométricos do PostgreSQL.

A implementação da R-Tree do PostgreSQL possui uma limitação quanto ao tamanho do dado da coluna: máximo de 8 Kbytes. Na prática, é muito comum, um GIS manipular dados representados, por exemplo, por polígonos maiores do que 8Kbytes cada, o que torna o uso desse índice inviável. Uma solução alternativa para esse conflito é o uso da R-Tree construída no topo do mecanismo de indexação GiST (Hellerstein et al, 1995), que também está disponível para esses tipos geométricos. Esse método de indexação foi introduzido por Hellerstein et al (1995) e implementado no PostgreSQL. Atualmente, ele é mantido por Teodore Signaev e Oleg Bartunov (http://www.sai.msu.su/~megera/postgres/gist) e não possui restrições de tamanho do dado a ser indexado. GiST é uma abreviação de *Generalized Search Trees*, consistindo em um índice (árvore balanceada) que pode fornecer tanto as funcionalidades de uma B-Tree quanto de uma R-Tree e suas variantes. A principal vantagem desse índice é possibilidade de definição do seu "comportamento". Ou seja, uma R-Tree pode ser implementada sobre ele.

A integração de um GIS através dos tipos geométricos padrão, disponíveis no PostgreSQL, não é muito vantajosa. Os poucos operadores espaciais existentes realizam a computação apenas sob o retângulo envolvente das geometrias e não diretamente nelas. E os tipos de dados são simples, como polígonos que não permitem a representação de buracos, não existindo também geometrias que permitam representar objetos mais complexos como os formados por conjuntos de polígonos. As primeiras versões da TerraLib utilizavam esses tipos na construção do *driver* TePostgreSQL (Ferreira et al, 2002), mas era necessária a construção de um modelo de dados mais

elaborado para suprir a ausência de tipos geométricos mais complexos (polígonos com buracos, coleções homogêneas de polígonos, linhas e pontos).

2.6.2.1 Mecanismo de Extensibilidade

Esse mecanismo permite incorporar capacidades adicionais ao PostgreSQL, de forma a torná-lo mais flexível para o gerenciamento de dados para cada classe de aplicação. No caso dos GIS, isso significa a possibilidade do desenvolvimento de uma extensão geográfica capaz de armazenar, recuperar e analisar dados espaciais. Os passos envolvidos na criação de uma extensão são:

- **Extensão dos tipos de dados:** consiste na definição dos novos tipos de dados a serem acrescentados ao sistema de tipos do SGBD. Esses tipos são representados por estruturas de dados definidas em linguagem C (Kernighan e Ritchie, 1990). Para que seja possível utilizar esse novo tipo é necessário fornecer o nome do tipo, as informações sobre armazenamento e as rotinas de conversão da representação do dado em memória para uma forma textual (ASCII) e vice-versa. No Apêndice B, os trechos de código B.1 a B.4 ilustram a definição de um tipo em C, chamado TeLinearRing, que representa uma linha fechada composta por um conjunto de coordenadas (TeCoord2D).

- **Extensão das funções e operadores:** além da flexibilidade do sistema de tipos, o PostgreSQL permite a criação de métodos que operem sobre as instâncias dos tipos definidos. Essas funções podem ser integradas ao servidor informando o nome da função, a lista de argumentos e o tipo de retorno. Essas informações são registradas no catálogo do sistema. Os trechos de código de B.5 a B.7 ilustram a definição de uma função cuja tarefa é calcular a área do tipo de dado TeLinearRing, considerando-o como um polígono simples.

- **Extensão do mecanismo de indexação:** conforme dito anteriormente, o PostgreSQL possui quatro mecanismos de indexação (B-Tree, R-Tree, GiST e HASH, sem considerar o índice funcional que é definido sobre a B-Tree). Esses índices podem ser usados para os tipos de dados e funções definidos pelo

usuário, bastando para isso registrar no catálogo do sistema as informações das operações necessárias a cada índice. Por exemplo, para um tipo que deseje ser indexado segundo a B-Tree, é necessário indicar ao sistema as operações de comparação “<”, “<=”, “=”, “>=” e “>” para que o índice saiba como realizar buscas.

2.6.2.2 PostGIS

A fim de servir como uma alternativa a produtos comerciais de custos elevados como o Oracle Spatial, a comunidade de software livre vêm desenvolvendo a extensão espacial PostGIS, construída sobre o PostgreSQL. Atualmente, a empresa Refractions Research Inc (http://postgis.refractions.net) mantém o time de desenvolvimento dessa extensão, que segue as especificações da SFSSQL. Este projeto explora vários dos recursos oferecidos pelo PostgreSQL e, atualmente, apresenta as seguintes funcionalidades:

- Tipos de objetos geométricos que podem ser representados e armazenados como um subconjunto do nível 3 da Tabela A.1 (Apêndice A):

  Point: (0 0 0)

  LineString: (0 0, 1 1, 1 2)

  Polygon: ((0 0 0, 4 0 0, 4 4 0, 0 4 0, 0 0 0), (1 1 0, ...), ...)

  MultiPoint: (0 0 0, 1 2 1)

  MultiLineString: ((0 0 0, 1 1 0, 1 2 1), (2 3 1, 3 2 1, 5 4 1))

  MultiPolygon: (((0 0 0, 4 0 0, 4 4 0, 0 4 0, 0 0 0), (...), ...), ...)

  GeometryCollection: (POINT(2 3 9), LINESTRING((2 3 4, 3 4 5))

- Esquema para definição de metadados conforme a SFSSQL. As tabelas espaciais são criadas em duas etapas. Na primeira, é criada a tabela apenas com os tipos alfanuméricos e depois, utilizando a função

AddGeometryColumn('dbname', 'tablename', 'columngeomname', srid, 'postgis_geom_type', ndimension), os atributos geométricos são adicionados à tabela. Essa função, recomendada pelo OGIS, é utilizada para preencher automaticamente a tabela de metadados das colunas geométricas.

- Indexação através de R-Tree definida sobre o GiST.

- Suporte aos operadores espaciais, fornecido através da integração do PostGIS com a biblioteca GEOS (Geometry Engine Open Source) (Refractions, 2003). Essa biblioteca, desenvolvida pela empresa Refractions em parceria com a Vivid Solutions, é uma tradução da API Java JTS (Java Topology Suíte) (Vivid Solutions, 2003) para a linguagem C++. A JTS é uma implementação de operadores espaciais que seguem as especificações da SFSSQL.

- Suporte para conversão de sistemas de coordenadas através da biblioteca Proj4 (Evenden, 2003).

Como pode ser visto, essa extensão fornece ao PostgreSQL mecanismos importantes para o gerenciamentos de dados representados de forma vetorial. Atualmente, essa extensão é utilizada pelo *driver* TePostgreSQL da TerraLib.

2.6.3 MySQL

Outro SGBD que entra no mercado de software livre para GIS como resposta às extensões comerciais atuais é o MySQL. Atualmente, a extensão espacial desse SGBD encontra-se em construção (MySQL AB, 2003). O seu projeto segue as especificações da SFSSQL. Os únicos recursos disponibilizados até o momento são os tipos de dados espaciais semelhantes aos fornecidos pelo PostGIS (Point, LineString, Polygon, MultiLineString, MultiPolygon, MultiPoint e GeometryCollection) e o mecanismo de indexação através de R-Tree.

Entre as características ausentes nessa extensão e que constam na especificação do OGIS, encontram-se as tabelas de metadados das colunas geométricas, funções de

criação e remoção das colunas geométricas e operadores espaciais. Essa extensão também não fornece nenhum recurso para conversão entre sistemas de coordenadas.

Essa extensão apresenta duas desvantagens em relação ao PostGIS. Primeiro, as colunas geométricas devem ser criadas com a restrição de não conter valores nulos (NOT NULL) para poderem ser indexadas através do índice espacial. Segundo, a extensão só está disponível para o tipo de tabela MyISAM, o que significa a ausência de recursos de transação.

2.6.4 Resumo das Extensões Espaciais

O quadro abaixo apresenta um resumo comparativo entre os SGBDs com suporte a tipos espaciais:

TABELA 2.1 – Quadro Comparativo entre os SGBDs com Suporte Espacial.

Recurso	**Oracle Spatial**	**PostgreSQL com Tipos Geométricos**	**PostgreSQL com PostGIS**	**MySQL**
Tipos Espaciais	SFSSQL	Tipos Simples	SFSSQL	SFSSQL
Indexação Espacial	R-Tree e QuadTree	R-Tree Nativa ou R-Tree sobre GiST	R-Tree sobre GiST	R-Tree
Operadores Topológicos	Matriz 9-Interseções	Não	Matriz 9-Interseções DE (GEOS)	Em Desenvolvimento
Operadores Conjunto	Sim	Não	Sim (GEOS)	Em Desenvolvimento
Operador de *Buffer Region*	Sim	Não	Sim (GEOS)	Em Desenvolvimento
Transformação entre Sistemas de Coordenadas	Sim	Não	Sim (proj4)	Não
Tabelas de Metadados das colunas geométricas	Sim (Diferente do OGIS)	Não	Sim (Conforme OGIS)	Não

Além das características resumidas no quadro acima, todos os SGBDs possuem em comum a forma de representar as geometrias. Em GIS tradicionais, como o SPRING, é comum o uso do modelo topológico completo. Já nos SGBD com extensões espaciais, as regiões são armazenadas separadamente, conforme o modelo *spaghetti* (Rigaux et al, 2002).

Nesse modelo, a geometria de qualquer objeto espacial em uma linha da tabela é descrita independentemente dos demais, podendo assumir qualquer posição relativa no plano, inclusive interceptando de forma arbitrária. No caso de polígonos com fronteiras comuns há uma redundância de representação, pois a fronteira comum a cada polígono é representada independentemente. Além disso, a topologia não é armazenada explicitamente, sendo necessário computá-la sob demanda. Como exemplo, cada um dos polígonos ilustrados na Figura 2.11 como adjacentes à cidade de Araxá, possui sua própria lista de vértices, de forma independente da cidade de Araxá e de suas respectivas vizinhas.

Os operadores topológicos nesse modelo ganham grande importância por dois motivos:

- O primeiro, porque como esses operadores realizam o cálculo do relacionamento topológico sob demanda, eles devem ser implementados de forma a serem eficientes. Isso evita que os usuários fiquem aguardando por muito tempo às respostas de suas consultas espaciais.

- O segundo motivo e também de grande importância, é o fato de que eles podem ser utilizados para estabelecer restrições de integridade (Davis et al, 2001) aos dados armazenados no banco de dados. Como exemplo, a modelagem de uma aplicação ilustrada na Figura 2.11 poderia definir algumas regras para restringir o armazenamento dos dados. Uma possível regra seria estabelecer que toda cidade ao ser inserida só possa estabelecer o relacionamento de "toca" com uma outra cidade já armazenada, evitando assim, a inserção de cidades cuja geometria sobreponha a de outras.

2.7 TerraLib

A TerraLib[2] (Câmara et al, 2000) é uma biblioteca de classes em linguagem de programação C++ (Stroustrup, 1997) projetada com o objetivo de fornecer um ambiente de desenvolvimento de pesquisas em GIS. Essa biblioteca se apóia nos avanços tecnológicos dos sistemas de bancos de dados, especialmente os espaciais, realizando a

completa integração dos tipos de dados espaciais dentro dos SGBDs. Para realizar essa tarefa, ela fornece uma arquitetura que dá acesso direto aos dados, que podem ser armazenados em diversos SGBDs, como Oracle Spatial, PostgreSQL, SQL Server (Ramalho, 1999) e MySQL (Yarger et al, 1999), através de uma interface comum (API). Essa interface permite a criação e a manipulação de bancos de dados geográficos sem se ter a preocupação com os detalhes de cada SGBD. Essa arquitetura é apresentada em (Ferreira et al, 2002).

Outro forte pilar da TerraLib é o emprego de técnicas modernas de programação, como o catálogo de padrões apresentados por Gamma et al (2000) e a programação genérica (Austern, 1999). Câmara et al (2001) apresenta o emprego de padrões na TerraLib e em Queiroz et al (2002) é apresentado o uso do paradigma de programação genérica no desenvolvimento de algoritmos de agrupamentos. A seguir serão apresentados os principais componentes dessa biblioteca.

2.7.1 Modelo de Geometrias

A Figura 2.13 mostra o diagrama UML (Page-Jones, 2000) das principais classes do *kernel*[3] da TerraLib utilizadas nesse trabalho. Essas classes servem para manipulação em memória da componente espacial dos dados geográficos representados no formato vetorial. No diagrama, foi adotado o uso de estereótipos no relacionamento de herança para indicar o tipo esperado pela classe base.

[2] Biblioteca disponível na Web, no endereço http://www.terralib.org
[3] O conjunto das classes fundamentais da TerraLib é denominado de *kernel*

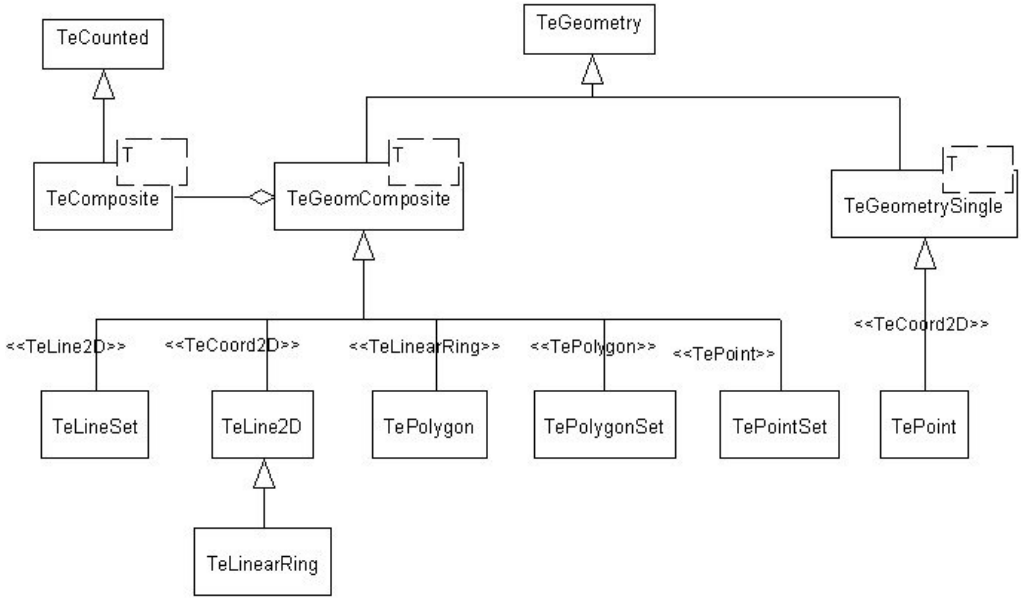


FIGURA 2.13 – Diagrama das classes geométricas da TerraLib.

A classe TeCounted fornece o suporte ao projeto de classes que utilizem o padrão *Bridge*. Este padrão é utilizado no projeto das classes TeComposite (*body*) e TeGeomComposite (*handle*). A classe TeComposite é uma especialização do padrão *Composite*. O emprego desses padrões em conjunto com *templates* torna o projeto das classes de geometria extremamente simples e claro.

As classes derivadas de TeGeometrySingle são compostas de um único elemento. Por exemplo, a classe TePoint representa um ponto formado por uma única coordenada (classe TeCoord2D) com atributos de abscissa e ordenada.

As classes derivadas de TeGeomComposite são formadas por um conjunto (vetor) de outras geometrias. Por exemplo, a classe TePointSet representa um conjunto de pontos. A classe TeLine2D representa uma linha poligonal formada por um conjunto de coordenadas. A classe TeLinearRing representa uma linha poligonal fechada (primeiro ponto da linha igual ao último) que também é chamada de anel. Um polígono é representado pela classe TePolygon, que consiste em um conjunto de anéis. A mesma idéia se aplica ao caso das classes TePolygonSet e TeLineSet.

2.7.2 Modelo de Dados

A TerraLib define um conjunto especifico de tabelas com o objetivo de facilitar a representação dos dados geográficos nos SGBD. Nesse esquema, as informações sobre um determinado objeto ou fenômeno geográfico são representadas por uma camada de informação. Essas camadas, representadas pela tabela te_layer, podem ser compostas por mais de um tipo de representação (te_representation) (vetorial ou matricial), estando ou não associadas a uma ou mais tabelas de atributos não espaciais. A Figura 2.14 mostra as tabelas responsáveis pelo esquema de armazenamento das geometrias. Alguns campos dessas tabelas foram omitidos por não comprometerem o entendimento.

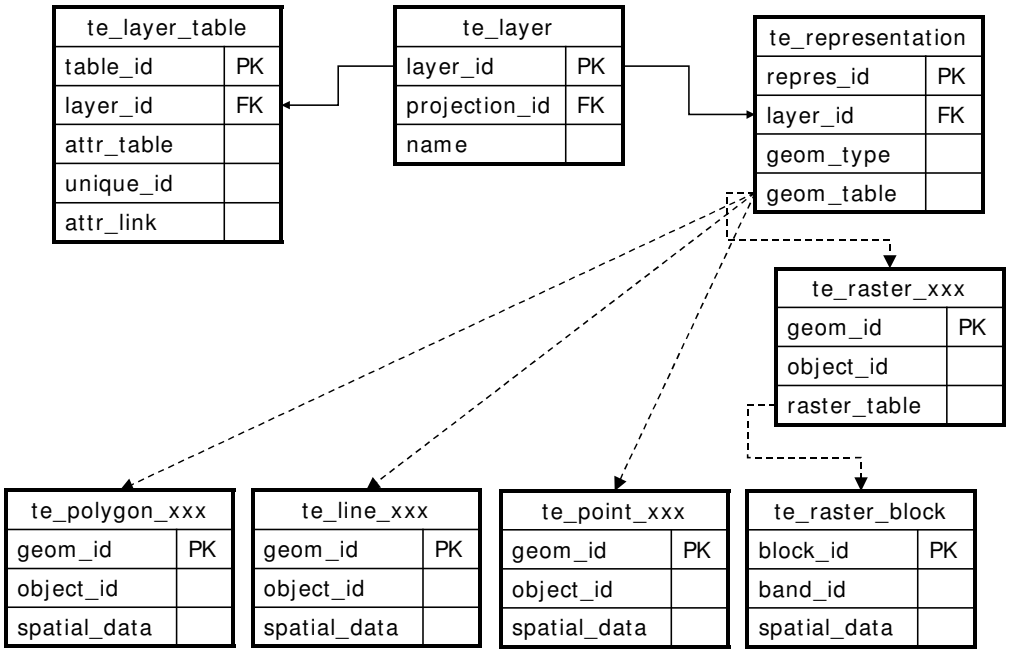


FIGURA 2.14 – Modelo de dados da TerraLib: Representação do Dado Geográfico.

Na representação vetorial, um dado pode ser mapeado para três tipos de tabelas com colunas capazes de representar polígonos (te_polygon_xxx)[4], linhas (te_line_xxx) ou pontos (te_point_xxx). O tipo da coluna spatial_data varia de acordo com o SGBD

[4] Os nomes das tabelas com tipos geométricos são gerados substituindo o valor "XXX" pelo número do PI.

empregado e o campo object_id é utilizado para fazer a ligação com os atributos alfanuméricos da(s) tabela(s) de atributos.

Entidades geográficas representadas por conjuntos de polígonos (TePolygonSet), ou por conjunto de linhas (TeLineSet) ou por conjunto de pontos (TePointSet) são divididos em várias linhas, todas com o mesmo valor de object_id para possibilitar a identificação das geometrias que formam o objeto.

As imagens (representação matricial) são particionadas em blocos (te_raster_block_xxx), que são armazenados em uma tabela com uma coluna do tipo BLOB. Vinhas et al (2003) apresenta os detalhes das estratégias de gerenciamento e armazenamento de dados matriciais na TerraLib.

Além das tabelas para armazenamento dos atributos espaciais, a TerraLib define outro conjunto de tabelas (Figura 2.15) para facilitar a manipulação do dado geográfico. Um PI pode estar associado a temas (te_theme) que representam uma seleção dos objetos do PI. A tabela te_theme armazena as restrições espaciais, temporais ou convencionais que dão origem aos temas. Como exemplo, dado um PI que contenha todos os estados do Brasil com seus atributos área, população e taxa de analfabetismo, uma consulta sobre esse PI como: “selecione todos os estados onde a população seja maior que 5 milhões”, resultaria em um novo tema associado a este PI, sendo a restrição “população maior que 5 milhões” armazenada na tabela te_theme.

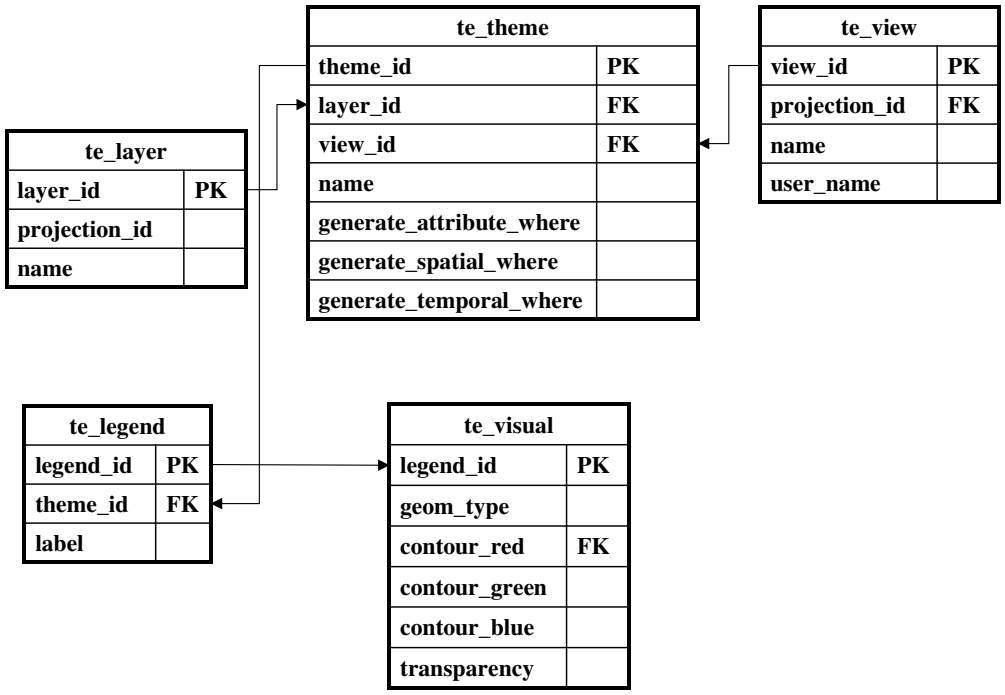


FIGURA 2.15 – Modelo de Dados da TerraLib: Vistas e Temas[5].

Os temas podem estar associados a legendas, as quais são armazenadas na tabela (te_legend). As legendas representam resultados de classificação e apontam para as informações de apresentação visual (te_visual).

Para facilitar a organização dos dados no SGBD, este modelo implementa o conceito de vistas (te_view). A vista está associada a uma visão particular de um usuário sobre os dados armazenados no SGBD pela TerraLib. Uma descrição mais detalhada de todas as tabelas e colunas do modelo de dados encontra-se disponível no endereço http://www.terralib.org.

2.7.3 API para Sistemas de Bancos de Dados

A TerraLib possui uma API comum para trabalhar com diversos sistemas de bancos de dados. Ela utiliza a estratégia de *drivers* de bancos de dados empregada na ODBC

[5] Alguns campos das tabelas mostradas na figura foram omitidos por questões de espaço. O modelo completo encontra-se disponível em http://www.terralib.org

(Figura 2.16). Para cada SGBD é criado um *driver* específico que implementa as funções da API para manipulação e definição do esquema da base geográfica.

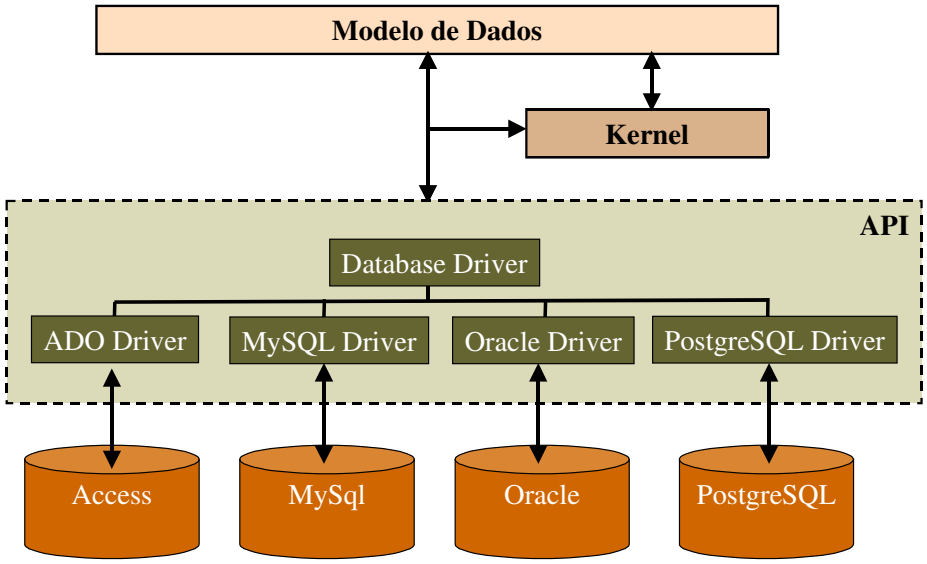


FIGURA 2.16 – Arquitetura de Construção de Bancos de Dados Geográficos da TerraLib.

Esses *drivers* realizam o mapeamento entre os tipos de dados dos SGBDs para os tipos de dados presentes no *kernel* da biblioteca. Além disso, ele traduz as consultas espaciais para o dialeto SQL de cada SGBD, delegando a este sua execução. Eles devem explorar ao máximo os recursos oferecidos por cada SGBD, principalmente, os que apresentam extensões espaciais com mecanismos de indexação espacial, tipos de dados e operadores espaciais.

A interface comum desses *drivers* é definida por duas classes abstratas, TeDatabase e TeDatabasePortal, que, entre outras coisas, fornecem métodos para:

- Estabelecimento de conexão com o servidor de banco de dados;
- Execução de comandos SQL (DDL e DML);
- Criação de novas tabelas;

- Criação do modelo de dados da TerraLib;
- Definição dos índices espaciais;
- Criação de integridade referencial e índice secundários;
- Execução de consultas SQL que retornem um conjunto de resultados;
- Manipular o conjunto de resultados da execução de uma consulta;
- Inserção, atualização e recuperação das geometrias segundo o modelo de dados;
- Execução de consultas espaciais;
- Armazenamento e recuperação de dados matriciais.

CAPÍTULO 3

GEOMETRIA COMPUTACIONAL E BANCOS DE DADOS GEOGRÁFICOS

A Geometria Computacional (Preparata e Shamos, 1985) é uma área da Ciência da Computação que considera a análise e projeto de algoritmos e de estruturas de dados para problemas geométricos, sendo o GIS considerado um dos principais campos de sua aplicação. Essa recente área de pesquisa ganhou atenção por volta de 1975 a partir dos trabalhos de Shamos (1975) e Shamos e Hoey (1975). Ela busca por algoritmos simples, robustos e eficientes, estudando problemas práticos como consultas de proximidades entre pontos ou interseções entre conjuntos de segmentos, aplicando métodos que utilizam os conhecimentos desenvolvidos na área de pesquisa em projeto e análise de algoritmos e estruturas de dados (Knuth, 1973).

Os algoritmos desenvolvidos por esse campo da computação aplicam-se a dados representados na forma vetorial, sendo os representados na forma matricial estudados por áreas como Processamento Digital de Imagens (Gonzalez e Woods, 2000). Entre as principais operações vetoriais de um GIS pode-se citar: consultas de ponto em polígono, determinação do envoltório convexo, operações entre conjuntos de geometrias (união, interseção e diferença), geração de mapas de distância e operações topológicas. Essas operações são construídas a partir de outras operações, chamadas de elementares ou básicas, como área de triângulos, interseção entre dois segmentos, interseção entre linhas poligonais e área de um polígono.

Em particular, as operações de consulta de ponto em polígono e de interseção entre linhas poligonais desempenham um papel fundamental na construção de outros operadores, como no caso dos topológicos. Nesses operadores, as duas operações que consomem o maior tempo de processamento são, respectivamente, a operação que determina os pontos de interseção entre as linhas poligonais e os testes de ponto em polígono. Portanto, a escolha de um algoritmo eficiente que possa ser integrado a um GIS é de fundamental importância.

Uma forma de medir o desempenho de um algoritmo é através da análise de complexidade, considerando-se o caso mais desfavorável (pior caso). Esse é um dos enfoques no estudo dos problemas geométricos por parte da Geometria Computacional. Os algoritmos para solução desses problemas são desenvolvidos com o objetivo de apresentarem um desempenho ótimo, e, geralmente, assumem um modelo de computação teórico conhecido como *real* RAM (Preparata e Shamos, 1985).

No entanto, como será mostrado ao longo desse capítulo, no caso de aplicações GIS, que trabalham com dados do mundo real, nem sempre a escolha de um algoritmo levando-se em consideração apenas a sua análise de complexidade de pior caso é a forma mais adequada. Outros fatores precisam ser considerados, como a característica dos dados de entrada, que podem favorecer algoritmos com maior complexidade de pior caso, e as constantes escondidas nessas análises. Além disso, o modelo de computação assumido acaba acarretando sérias dificuldades durante a implementação dos algoritmos.

Este capítulo trata dos aspectos relacionados ao processamento do componente espacial do dado geográfico, com ênfase na aplicação de técnicas da Geometria Computacional, apresentando os limites inferiores dos problemas subjacentes às operações geométricas e alguns dos algoritmos existentes. O principal objetivo aqui é apresentar o projeto dos operadores espaciais desenvolvidos para a TerraLib e as decisões tomadas na escolha dos algoritmos integrados nessa biblioteca. Para isso, os problemas e algoritmos serão apresentados iniciando-se pelos mais elementares, que servem como bloco de construção para os mais complexos.

3.1 Área de um Triângulo

A determinação da área de um triângulo é uma das operações mais básicas empregadas por outros algoritmos. Ela é calculada como a metade da área de um paralelogramo (Figura 3.1) (Figueiredo e Carvalho, 1991). O produto vetorial dos vetores A e B determina a área (S) do paralelogramo com os lados A e B, e pode ser computado a partir do seguinte determinante (no caso bidimensional):

$$S = \begin{vmatrix} 1 & 1 & 1 \\ X_A & Y_A & 0 \\ X_B & Y_B & 0 \end{vmatrix} = (x_b - x_a) \times (y_c - y_a) - (x_c - x_a) \times (y_b - y_a) \quad (3.1)$$

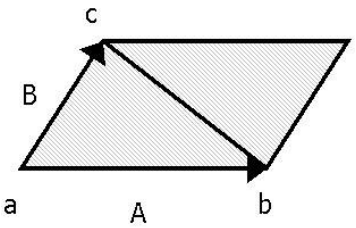


FIGURA 3.1 – Área do Triângulo abc.

A equação 3.1 fornece, além da área, uma informação muito útil para os algoritmos de geometria computacional: a orientação dos três pontos que formam o triângulo. Caso a área seja negativa os pontos a, b e c encontram-se no sentido horário, se positiva, os pontos encontram-se no sentido anti-horário e se ela for zero, indica que os três pontos são colineares (estão alinhados).

Na TerraLib, o operador que fornece a orientação foi desenvolvido com base na equação acima. Esse operador denominado de TeCCW (Counter Clockwise Test) retorna os valores TeCOUNTERCLOCKWISE, TeCLOCKWISE e TeNOTURN se o sentido definido entre os pontos for, respectivamente, anti-horário, horário ou se os três pontos estiverem alinhados. Abaixo, é ilustrada a assinatura desse operador, sendo seu código mostrado no Apêndice C em C.1.

```
short

TeCCW(const TeCoord2D& c1, const TeCoord2D& c2, const TeCoord2D& c3)
```

Um cuidado importante na implementação desse operador diz respeito aos erros numéricos de arredondamento. Esses erros, introduzidos pelo tipo de representação em ponto flutuante (double) das coordenadas dos pontos, podem gerar problemas de instabilidade numérica no operador e, conseqüentemente, problemas para as operações construídas com base nele. A Seção 3.11 é dedicada a esse assunto, mostrando como esses erros foram tratados na TerraLib.

3.2 Teste de Convexidade

O operador definido na seção anterior pode ser utilizado para determinar se um polígono é ou não convexo. Se o sentido definido por três pontos consecutivos da fronteira do polígono for sempre o mesmo então o polígono é convexo. Em polígonos côncavos pelo menos um dos testes indicará sentido contrário aos demais. Considerando o teste de orientação com complexidade constante, $O(1)$, o algoritmo para determinar a convexidade é $O(n)$ onde n é o número de vértices do polígono sendo testado.

Na TerraLib, o operador TeIsConvex foi desenvolvido utilizando essa estratégia, sendo sua assinatura mostrada abaixo e o código em C.2.

```
bool TeIsConvex(const TeLinearRing& ring)
```

3.3 Área de Polígonos

A técnica de cálculo da área de um polígono simples com vértices $v_0, v_1, ..., v_{n-1}$ é baseada no cálculo da área de triângulos (O'Rourke, 1998). A seguinte fórmula fornece a área de um polígono:

$$A(P) = \frac{1}{2} \times \sum_{i=0}^{i=n-1} (x_i + x_{i+1}) \times (y_{i+1} - y_i) \tag{3.2}$$

Um algoritmo que utilize essa fórmula para o cálculo da área apresenta complexidade $O(n)$, onde n é o número de vértices do polígono. Outra informação importante que essa fórmula fornece é a orientação dos vértices do polígono. Caso o resultado seja positivo os vértices estão ordenados no sentido anti-horário e caso seja negativo os vértices encontram-se no sentido horário (Bourke, 2002a). Esse teste é empregado pelo algoritmo que realiza as operações de união, interseção e diferença, que será discutido mais adiante.

O operador Te2Area definido na TerraLib com base nessa fórmula, retorna um valor que corresponde a duas vezes a área de um polígono simples (sem buracos), representado por um anel (TeLinearRing). O resultado dessa função pode ser um valor positivo ou negativo, dependendo da orientação do polígono.

A partir da função Te2Area, duas outras operações são construídas. A primeira é o cálculo de área, válido para retângulos (TeBox), polígonos (TePolygon) e conjuntos homogêneos de polígonos (TePolygonSet), através da função TeGeometryArea. Para as outras geometrias da biblioteca o resultado retornado por essa função é sempre zero. A outra operação, chamada de TeOrientation, retorna TeCLOCKWISE caso a fronteira do anel que forme o polígono esteja orientada no sentido horário (Te2Area retorna um valor negativo) ou TeCOUNTERCLOCKWISE caso a fronteira esteja no sentido anti-horário (Te2Area retorna um valor positivo). Abaixo, são apresentadas as assinaturas dessas funções. Os códigos podem ser vistos em C.3 e C.4 respectivamente.

```
double Te2Area(const TeLinearRing& r)

template<class T> double TeGeometryArea(const T& geom)

template<> double TeGeometryArea(const TePolygon& p)

template<> double TeGeometryArea(const TePolygonSet& ps)

template<> double TeGeometryArea(const TeBox& b)

short TeOrientation(const TeLinearRing& r)
```

3.4 Envoltório Convexo

Neste trabalho será adotada a nomenclatura de Rezende e Stolfi (1994) para o termo *Convex Hull*, que é denominado de Envoltório Convexo. O envoltório convexo de um conjunto de pontos S é o menor polígono convexo P que contém S. Esse envoltório fornece informações sobre o formato de um conjunto de pontos (como no caso de pontos de queimada), serve como uma primeira aproximação de um objeto mais complexo e tem aplicações na área de robótica.

Existe na literatura um grande número de algoritmos para a resolução desse problema. Preparata e Shamos (1985), de Berg et al (1997) e O'Rouke (1998) apresentam discussões sobre esses algoritmos, pautadas na análise de complexidade e o espaço dimensional para o qual o algoritmo é válido. O algoritmo escolhido neste trabalho é conhecido como varredura de Graham (de Berg et al, 1997), com as modificações de Andrew (1979). A apresentação do algoritmo será dada a seguir e a justificativa da sua escolha encontra-se no final.

Dado um conjunto de pontos *P* no plano (Figura 3.2a), o primeiro passo consiste em ordenar esses pontos ao longo do eixo "*x*" (da esquerda para a direita), obtendo uma seqüência ordenada de pontos $p_1, p_2, ..., p_n$. O envoltório é então construído em duas etapas, na primeira, são computados os pontos pertencentes à parte superior do envoltório, que são os pontos que vão do ponto mais à esquerda (p_1) ao mais à direita (p_n) no sentido horário (Figura 3.2b). Na segunda etapa, são computados os pontos pertencentes à parte inferior do envoltório, que são os pontos que vão do ponto p_n ao ponto p_1, andando da direita para a esquerda no sentido horário (Figura 3.2c). O envoltório final é constituído da união das duas partes (Figura 3.2d).

Tanto a parte superior quanto a inferior pode ser computada a partir da observação que em um polígono convexo orientado no sentido horário, quando se anda ao longo de sua fronteira, é feita uma rotação à direita em cada vértice ou não se faz nenhuma, caso o vértice forme um ângulo de 180^0, onde os três pontos que formam o vértice são colineares.

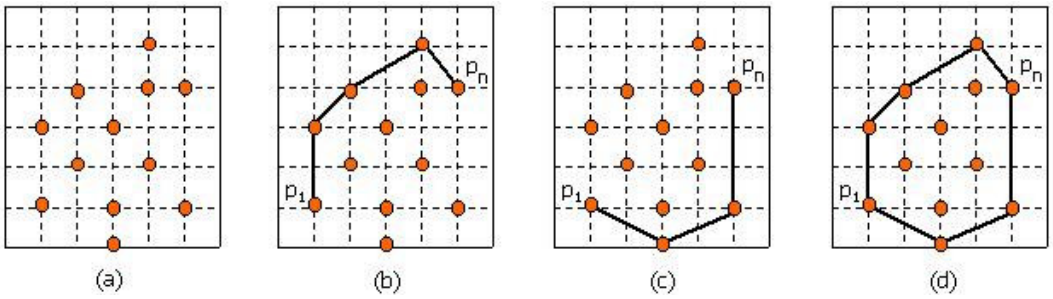


FIGURA 3.2 – (a) Conjunto de pontos *P*, (b) Parte superior do envoltório, (c) Parte superior do envoltório e (d) Envoltório convexo de *P*.

Seja *Lsup* uma lista que conterá os pontos da parte superior do envoltório ordenada da esquerda para a direita. Essa lista inicia-se com os pontos *p1* e *p2* do conjunto *P*. Depois, cada ponto p_i, com $2 < i \leq n$, é adicionado à lista, checando sempre o requisito de que os três últimos pontos devem fazer uma rotação à direita ou serem colineares. Caso o ponto adicionado satisfaça o requisito, adicionamos o próximo. Caso contrário, o ponto do meio dos três últimos é removido de *Lsup* e novamente é checado se os três últimos pontos satisfazem o requisito. Esse procedimento é feito até que os últimos três pontos em *Lsup* satisfaçam o requisito ou que *Lsup* contenha apenas dois pontos.

Depois de ser processado o último ponto p_n, a parte superior do envoltório estará computada. A parte inferior é computada de maneira semelhante, com a única diferença que se deve partir do ponto p_n para o ponto p_1.

Esse algoritmo possui complexidade $O(n \log n)$, onde n é o número de pontos do conjunto de entrada P. Essa é a complexidade do primeiro passo, onde o conjunto de pontos P é ordenado da esquerda para a direita, e que domina o algoritmo. O passo que constrói as partes superiores e inferiores possui complexidade $O(n)$, pois cada ponto de P é adicionado à sua respectiva lista (superior ou inferior) no máximo uma vez. A complexidade desse algoritmo é ótima para a resolução do problema de se computar o envoltório convexo de um conjunto de pontos.

O operador TeConvexHull foi desenvolvido com base nesse algoritmo. Sua escolha se justifica pela complexidade ótima, que é superior à de algoritmos triviais, e pela simplicidade quando comparado a outros algoritmos da literatura, que podem ser aplicados a casos tridimensionais ou superiores. Como observado, esse algoritmo não pode ser estendido para trabalhar com o espaço tridimensional ou superiores, mas no caso da TerraLib, isso não traz nenhum transtorno pois o espaço adotado é o bidimensional. A implementação utiliza o algoritmo *sort* e a estrutura de dados *vector* da STL (Austern, 1999). Abaixo são apresentadas as assinaturas desse operador e seu código fonte pode ser visto em C.5.

```
template<class T> TePolygon TeConvexHull(const T& coordSet);

template<> TePolygon TeConvexHull(const TePolygon& p);

template<> TePolygon TeConvexHull(const TePolygonSet& ps);

template<> TePolygon TeConvexHull(const TePointSet& ps);
```

3.5 Ponto em Polígono

Uma das operações mais comuns em um GIS é determinar se um ponto está no interior de um polígono. As técnicas são divididas por duas classes de polígonos, os convexos e os simples[6] (convexos ou não).

A técnica que se aplica a polígonos convexos inclui um algoritmo com tempo logarítmico ($O(\log n)$) em relação ao número de vértices. Ele se baseia na propriedade de que os vértices ocorrem em uma ordem angular ao redor de qualquer ponto interno (Preparata e Shamos, 1985). Um ponto interno z (Figura 3.3) pode ser encontrado em tempo constante ($O(1)$), por exemplo, determinando-se o centróide de um dos triângulos formado por três vértices consecutivos do polígono. O ponto interno é usado para particionar o polígono. Colocando os vértices em uma estrutura de dados que possibilite pesquisa binária, basta encontrar a partição e então testar se o ponto q é interno ou não, através do teste de orientação de três pontos.

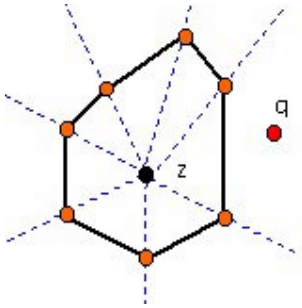


FIGURA 3.3 – Técnica aplicada a polígonos convexos.

FONTE: adaptada de (Preparata e Shamos, 1985).

A detecção de ponto em polígonos simples possui algoritmos de complexidades lineares ($O(n)$) em relação ao número de vértices e técnicas mais sofisticadas que após uma etapa de pré-processamento de complexidade $O(n \log n)$ determina se o ponto é interno ou não em tempo logarítmico ($O(\log n)$). Para esta última abordagem Narkhede e Manocha (2003) apresentam uma implementação do algoritmo de Seidel (1991) que

[6] Isto é polígonos em que os segmentos que formam a sua fronteira não possuem auto-interseções

pode ser usada em consultas de ponto em polígono. A aplicação desse tipo de algoritmo é interessante quando várias consultas são feitas sobre um mesmo polígono, pois as consultas posteriores à primeira serão respondidas em tempo muito menor.

Entre os algoritmos lineares o mais comum é o teste do número de cruzamentos de uma semi-reta horizontal ou vertical (chamado de raio), que parte do ponto testado, com os segmentos que formam a fronteira do polígono (Haines, 1994), (Bourke, 2002b), e (Taylor, 1994). Se o número de cruzamentos for par o ponto encontra-se fora, e se for ímpar, dentro. A Figura 3.4 ilustra a idéia desse teste. Conforme pode ser observado, o raio que parte do ponto p_1 que está fora do polígono, cruza os segmentos da fronteira um número par de vezes (2 vezes), enquanto os raios que partem dos pontos p_2 e p_3 que estão localizados no interior, cruzam um número ímpar de vezes (1 e 3 respectivamente).

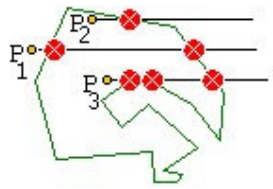


FIGURA 3.4 – Teste do número de cruzamentos do raio.

Huang e Shih (1997) apresentam a análise de outros algoritmos e Haines (1994) aponta que, embora alguns algoritmos tenham a mesma complexidade linear do teste de cruzamento, na prática eles possuem um desempenho pior por fazerem uso de operações mais custosas computacionalmente (arco-cosseno, raiz quadrada).

A operação de teste de ponto em polígono na TerraLib foi implementada utilizando o algoritmo do número de cruzamentos do raio. A escolha desse algoritmo se justifica pelo tipo de consulta espacial mais freqüente a ser respondida, que é a seleção por apontamento. Nesse tipo de consulta é mais apropriada a aplicação de um algoritmo linear do que um que precise realizar um pré-processamento da ordem de $O(n \log n)$ para então responder em tempo logarítmico.

Baseado nisso, a implementação foi adaptada de Haines (1994), que por sua vez utiliza a estratégia de Samosky (1993). O motivo de ter sido dada preferência a essa implementação é por ela evitar a realização de divisões no teste de cruzamento. Samosky transforma esse teste em uma comparação que realiza apenas multiplicações e subtrações, e os testes de Haines mostram que essa transformação pode gerar uma melhora de desempenho em arquiteturas de hardware onde a divisão seja mais custosa computacionalmente. Abaixo é apresentada a assinatura do operador desenvolvido. O código pode ser visto em C.6.

```
bool TePointInPoly(const TeCoord2D& c, const TeLinearRing& r);
```

Esse operador é capaz de responder se um determinado ponto encontra-se no interior ou no exterior do polígono. No entanto, caso o ponto esteja localizado sobre a fronteira, o teste é indeterminado, ou seja, para alguns objetos o teste pode resultar em verdadeiro e em outros, falso. Conforme veremos na seção que trata das operações topológicas e de conjunto, o teste para ver se um determinado ponto encontra-se sobre a fronteira de uma linha poligonal é de grande relevância. Por isso, foi desenvolvido outro operador na TerraLib que fornece essa informação. Esse operador, denominado de TeIsOnBorder, checa para cada segmento que forma uma linha poligonal se o ponto encontra-se sobre ele. A operação na qual ele se apóia é o teste de orientação. A assinatura dessa função é mostrada abaixo e o código encontra-se em C.7.

```
bool TeIsOnBorder(const TeCoord2D& c, const TeLine2D& l)
```

3.6 Interseção de Segmentos

Dados dois segmentos a e b, formados pelos pontos p_1p_2 e p_3p_4 (Figura 3.5), respectivamente, verificar se eles se interceptam consiste em testar se os pontos p_1 e p_2 estão de lados opostos do segmento formado por p_3p_4 e também se p_3 e p_4 estão de lados opostos do segmento formado por p_1p_2. Este problema se conecta com o problema da área de triângulo, pois, determinar se p_3 está do lado oposto de p_4 em relação ao segmento a, consiste em avaliar o sinal da área dos triângulos formados por $p_1p_2p_3$ e $p_1p_2p_4$. Se os sinais forem contrários, significa que os pontos estão de lados opostos. Se

o mesmo for verdadeiro para os triângulos $p_3p_4p_1$ e $p_3p_4p_2$, então, com certeza podemos afirmar que as retas que passam pelos segmentos *a* e *b* se interceptam em algum ponto, embora não se possa afirmar ainda que os segmentos têm interseção.

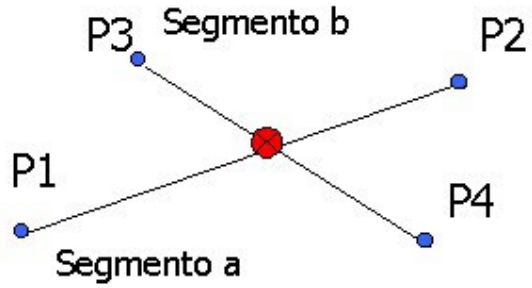


FIGURA 3.5 – Segmentos que se interceptam.

Em Saalfeld (1987) e em Bourke (2002c) é discutida uma forma de determinar o ponto de interseção entre dois segmentos baseada na representação paramétrica dos segmentos[7]. Dados dois segmentos formados pelos pontos p_1p_2 e p_3p_4, respectivamente, e com $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$, $p_3 = (x_3, y_3)$ e $p_4 = (x_4, y_4)$, o ponto de interseção entre eles é dado por:

$$p_1 + u(p_2 - p_1) = p_3 + v(p_4 - p_3) \tag{3.3}$$

Esta igualdade dá origem a um sistema com duas equações e duas incógnitas (*u* e *v*):

$$\begin{cases} x_1 + u(x_2 - x_1) = x_3 + v(x_4 - x_3) \\ y_1 + u(y_2 - y_1) = y_3 + v(y_4 - y_3) \end{cases} \tag{3.4}$$

Desenvolvendo o sistema temos:

$$u = \frac{(x_4 - x_3)(y_1 - y_3) - (y_4 - y_3)(x_1 - x_3)}{(y_4 - y_3)(x_2 - x_1) - (x_4 - x_3)(y_2 - y_1)} \tag{3.5}$$

$$v = \frac{(x_2 - x_1)(y_1 - y_3) - (y_2 - y_1)(x_1 - x_3)}{(y_4 - y_3)(x_2 - x_1) - (x_4 - x_3)(y_2 - y_1)} \tag{3.6}$$

[7] A equação paramétrica para um segmento de coordenadas p_1 e p_2 é dada por: $p = p_1 + u(p_2 - p_1)$, onde se $0 < u < 1$ define um ponto localizado entre p_1 e p_2.

Calculados os parâmetros u e v, podemos determinar o ponto de interseção:

$$\begin{cases} x_{interseccao} = x_1 + u(x_2 - x_1) \text{ ou } x_{interseccao} = x_3 + v(x_4 - x_3) \\ y_{interseccao} = y_1 + u(y_2 - y_1) \text{ ou } y_{interseccao} = y_3 + v(y_4 - y_3) \end{cases} \quad (3.7)$$

As expressões de *u* e *v*, respectivamente 3.5 e 3.6, possuem interpretações importantes. Os denominadores são os mesmos, e, portanto, numa implementação computacional eles deverão ser calculados uma única vez. Se o denominador (3.5 e 3.6) for zero, as duas linhas são paralelas. Se além do denominador os numeradores de ambos os parâmetros também forem zero, então as duas linhas são coincidentes. Na verdade, as equações paramétricas aplicam-se a linhas e, portanto, só haverá interseção entre os dois segmentos em um ponto localizado sobre ambos, o que significa valores de u e v ambos no intervalo [0,1]. Conforme colocado por Saalfeld (1987), a representação paramétrica evita que casos especiais, como o de segmentos verticais, tenham que ser tratados de forma separada.

Na TerraLib, o operador TeIntersection para segmentos de reta, representados pelas suas coordenadas extremas, foi desenvolvido com base nessa técnica. Com o intuito de agilizar os testes de interseção, foi empregada a estratégia de eliminação de possíveis interseções falsas através do teste de interseção entre os MBRs dos segmentos. A assinatura do operador de interseção entre segmentos é mostrada abaixo e o código completo pode ser visto em C.8.

```
bool
TeIntersection(const TeCoord2D& a, const TeCoord2D& b,
               const TeCoord2D& c, const TeCoord2D& d,
               TeIntersCoordsVec& coords,
               TeSegmentIntersectionType& intersectionType)
```

O parâmetro *coords* dessa função consiste num vetor de coordenadas, que pode conter entre zero e duas coordenadas, dependendo do tipo de interseção, que pode ser classificada como própria (TeProperIntersection) e imprópria (TeImproperIntersection). Uma interseção é dita própria quando a sua coordenada é interior aos dois segmentos testados, que é o caso em que eles se cruzam. As interseções impróprias ocorrem quando suas coordenadas correspondem a pontos extremos dos segmentos. Este é o caso, por exemplo, de segmentos que se tocam ou que se sobrepõem.

Em Geometria Computacional, essa operação é conhecida como primitiva de construção geométrica, pois ela pode gerar novos valores de coordenadas. Um dos problemas associados a essa primitiva são os erros de arredondamento devido ao cálculo da coordenada de interseção. Os algoritmos que sofrem os maiores impactos com esses erros são os baseados no *plane sweep*, que necessitam de tratar as novas coordenadas geradas pela interseção. Conforme será visto na Seção 3.11, não há na literatura uma maneira universal de lhe dar com esses erros.

3.7 Interseção de Conjunto de Segmentos

O problema de se determinar os pontos de interseção entre um conjunto de segmentos é um dos mais importantes no caso de um GIS que trabalha com representação vetorial. Isso porque operações como união, interseção, diferença e as operações que avaliam o relacionamento topológico necessitam determinar esses pontos como uma das primeiras etapas de seus processamentos, sendo esta a de maior consumo de processamento. Portanto, essa operação deve ser realizada no menor tempo possível, o que implica no uso de um algoritmo bem projetado para esse fim. No que segue, é mostrado o estudo comparativo realizado entre as técnicas e algoritmos para a resolução desse problema que serviu de suporte para a elaboração de operadores eficientes na TerraLib.

3.7.1 Algoritmos de Interseção: Caso I – Força Bruta

Dados dois polígonos simples A e B, formados por *n* e *m* segmentos, respectivamente, os pontos de interseção entre os segmentos que formam as suas fronteiras podem ser determinados a partir do seguinte algoritmo:

```
algoritmo Força Bruta
|   para i ← 1 até n
|   |     para j ← 1 até m
|   |     |    se A[i] ∩ B[j]
|   |     |    |  então REPORTAR(P_INTER)
|   |     |    fim se
|   |     fim para
|   fim para
fim algoritmo
```

onde A[i] é o i-ésimo segmento que forma a fronteira de A e B[j] é o j-ésimo segmento que forma a fronteira de B.

Tomando $n = m$, esse algoritmo apresenta claramente a complexidade $O(n^2)$. Para o caso de n com valores pequenos, esse algoritmo funciona bem. No entanto à medida que n e m crescem, ele se torna lento. A Figura 3.6a ilustra o caso onde existem n x m interseções. Nesse caso, qualquer algoritmo terá complexidade $O(n^2)$, que é o limite superior imposto por esse problema. No entanto, em casos práticos como o ilustrado na Figura 3.6b é esperado que um número muito menor de interseções existam.

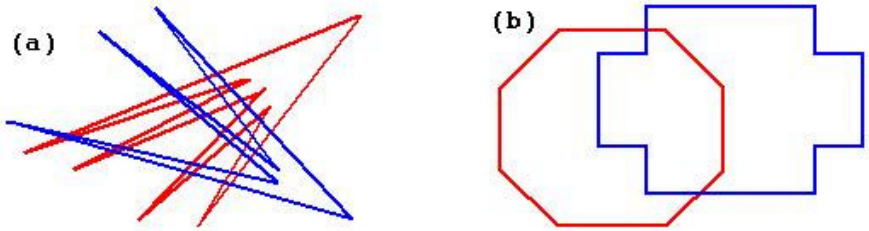


FIGURA 3.6 – (a) N x M interseções e (b) Caso prático.

Uma primeira tentativa de melhorar esse algoritmo seria descartar dos testes de interseção os segmentos da primeira linha poligonal cujo MBR não sobreponha o retângulo formado pela interseção dos MBRs das duas linhas (ver Figura 3.7).

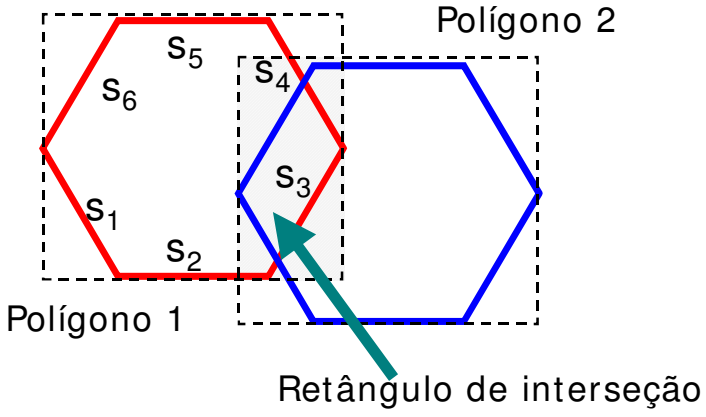


FIGURA 3.7 – Estratégia de eliminação de testes de interseção.

No entanto isso não evita que os segmentos que sobreponham o retângulo de interseção (s_3 e s_4) sejam testados contra todos os outros segmentos da outra linha. Dessa forma, à medida que a área do retângulo de interseção dos MBRs das duas linhas aumenta, a eficiência de execução desse algoritmo diminui. Mesmo com esta alteração a complexidade do algoritmo permanece $O(n^2)$.

O tempo de execução para este algoritmo é inaceitável para os casos onde o dado manipulado envolve linhas poligonais com muitos segmentos. Nesses casos, um usuário de um aplicativo como o TerraView teria que aguardar um longo intervalo de tempo para o processamento de uma consulta espacial. O Apêndice C em C.9 apresenta o trecho de código desse algoritmo implementado na TerraLib. Os algoritmos apresentados nas próximas seções representam soluções alternativas para o problema de desempenho citado.

3.7.2 Algoritmos de Interseção: Caso II – Plane Sweep

Shamos e Hoey (1976) apresentam um dos primeiros trabalhos discutindo o problema de interseção entre objetos com base na análise de complexidade. Eles fornecem um algoritmo para um problema similar que é determinar se, em um conjunto de n segmentos, há pelo menos um par que se intercepte. A solução dada possui complexidade $O(n \log n)$, que é igual ao limite inferior para a resolução deste problema. A técnica empregada por eles no desenvolvimento deste algoritmo é conhecida como paradigma da linha de varredura (Rezende e Stolfi, 1994) ou *plane sweep* (Nievergelt e Preparata, 1982).

Essa técnica se baseia no fato de que dois segmentos podem ser comparados se existir uma linha vertical, em uma determinada abscissa, que intercepte os dois segmentos. Na Figura 3.8, os segmentos D e B podem ser comparados, e, portanto, uma relação de ordem entre eles pode ser definida. Na abscissa u, o segmento B encontra-se acima do segmento D, pois a sua interseção com a linha vertical (em u) encontra-se acima da interseção de D com u. A notação usada para estas relações é: $B >_u D$, que significa que B encontra-se acima de D em u.

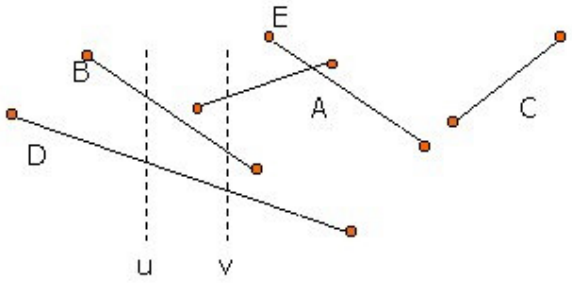


FIGURA 3.8 – *Sweep Line*.

FONTE: adaptada de (Shamos e Hoey, 1976).

O *plane sweep* consiste em deslizar uma linha vertical *l*, conhecida como *sweep line*, da esquerda para a direita, mantendo em cada ponto a ordem dos segmentos que a interceptam. A ordem mantida em *l* se altera quando um ponto final esquerdo de um dos segmentos é atingido, e nesse caso o segmento deve entrar na ordem definida em *l*. Quando *l* desliza para a direita de um ponto final direito de um dos segmentos, esse segmento deve deixar a ordem, pois a linha vertical atinge uma abscissa na qual este segmento não pode ser mais comparado com os demais segmentos presentes na ordem definida por *l*. A ordem em *l* também se altera quando se passa por um ponto de interseção entre dois segmentos, caso em que eles alternam as suas posições na ordem. Uma observação importante é que para dois segmentos se interceptarem é necessário que em algum momento eles sejam adjacentes na ordem vertical. Essa observação é fundamental, pois o teste de interseção fica resumido a testar os segmentos consecutivos na ordem em *l*. Em resumo, segmentos mais próximos um do outro são os melhores candidatos a terem interseção, enquanto segmentos mais distantes não. Essa idéia de proximidade baseia-se no fato de que, à medida que a *sweep line* anda no eixo *x*, somente os segmentos cujos intervalos se sobrepõem é que podem se interceptar (Figura 3.9a). E a ordem *y* definida em *l* permite que o teste seja realizado somente entre segmentos adjacentes nesta ordem, evitando assim que todos os segmentos tenham que ser testados (Figura 3.9b).

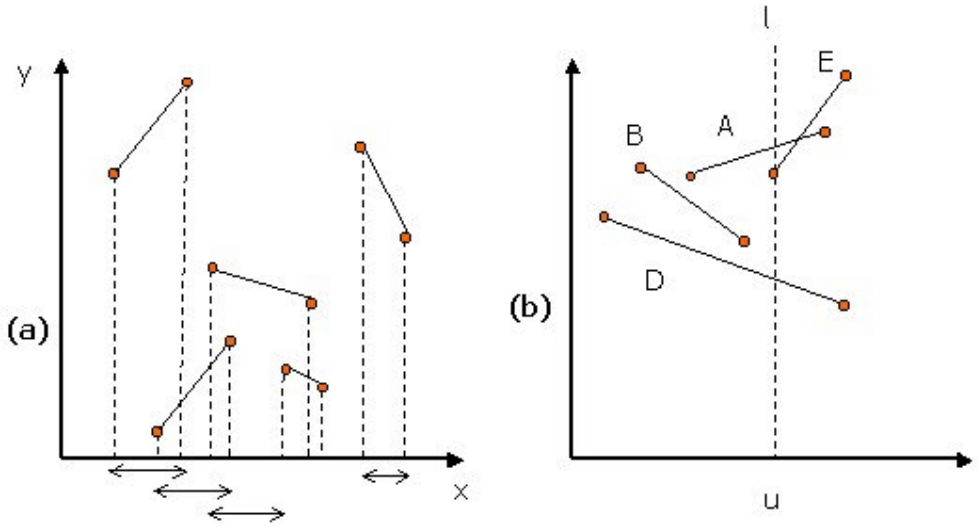


FIGURA 3.9 – *Plane Sweep*.

O algoritmo baseado no *plane sweep* para detecção de interseção, apresentado por Shamos e Hoey (1976), mantém duas estruturas de dados:

- Fila de eventos (*Q*): a linha *l* na realidade não desliza continuamente. Ela é movida somente pelos pontos onde a ordem mantida por ela pode ser alterada. Esses pontos são os pontos finais direito e esquerdo dos segmentos, que são denominados de eventos. A fila de eventos (*Q*) contém os pontos finais de cada segmento ordenado da esquerda para a direita na ordem *x*. Uma possível estrutura de dados utilizada para armazenar esses eventos é um vetor, pois essa fila não se altera.

- Relação de Ordem em *l* (*T*): estrutura utilizada para manter a ordem dos segmentos que interceptam *l*. A fim de manter essa ordem, a estrutura deve suportar as seguintes operações:

    - INSERE(s, T): insere o segmento *s* na ordem definida em *T*.

    - REMOVE(s, T): remove o segmento *s* na ordem *T*.

    - ACIMA(s, T): retorna o segmento imediatamente acima do segmento *s* na ordem *T*.

- ABAIXO(s, T): retorna o segmento imediatamente abaixo do segmento *s* na ordem *T*.

Uma estrutura de dados que pode ser utilizada na implementação de *T* é uma árvore balanceada, como por exemplo, uma *Red Black Tree* (Wood, 1993), que permite que as operações acima sejam realizadas em tempo $O(\log n)$.

Abaixo é mostrado o algoritmo de Shamos e Hoey:

```
algoritmo DetectaInterseção(S {conjunto de segmentos})
|    Q ← 2n pontos finais dos segmentos em S;
|    Ordenar(Q);
|    T ← ∅;
|    para i ← até 2n faça
|    |    p ← Q[i]; {p é um ponto final do segmento s}
|    |    se(p for o ponto final esquerdo de s)
|    |    |    então
|    |    |    |    INSERE(s, T);
|    |    |    |    a ← ACIMA(s, T);   b ← ABAIXO(s, T);
|    |    |    |    se(a intercepta s) então retorne(a, s);
|    |    |    |    se(b intercepta s) então retorne(b, s);
|    |    |    fim então
|    |    |    senão {p é um ponto final direito}
|    |    |    |    a ← ACIMA(s, T);   b ← ABAIXO(s, T);
|    |    |    |    se(a intercepta b) então retorne(a, b)
|    |    |    |    REMOVE(s, T);
|    |    |    fim senão
|    |    fim se
|    fim para
fim algoritmo
```

Esse algoritmo pode ser aplicado, com algumas modificações, a dois problemas práticos: verificar se uma dada linha poligonal é simples e determinar se há ou não interseção entre duas linhas poligonais. Com o intuito de investigar a técnica da varredura mais de perto, foram desenvolvidos dois operadores na TerraLib que tratam dos problemas mencionados utilizando essa técnica.

As assinaturas dos operadores denominados de TeIntersects são mostradas abaixo e o código fonte encontra-se nos arquivos TeIntersector.h e TeIntersector.cpp do *kernel* da TerraLib.

```
bool
TeIntersects(const TeLine2D& line, const TeObjectIndex& lineIndex)

bool
TeIntersects(const TeLine2D& redLine, const TeLine2D& blueLine,
       const TeObjectIndex& redIndex, const TeObjectIndex& blueIndex)
```

Como um dos objetivos do trabalho era encontrar um algoritmo eficiente que pudesse ser usado na TerraLib, um conjunto de classes auxiliares foi projetado para facilitar o desenvolvimento destes. O projeto dessas classes é apresentado no Apêndice C em C.10, e serve de auxílio para o entendimento das implementações descritas no restante dessa seção.

A primeira variação do algoritmo (Figura 3.10), implementada pela primeira função, determina se há interseção entre os segmentos de uma dada linha. Os segmentos que formam a linha (classe TeSegment no Apêndice C) são gerados e associados aos respectivos eventos (classe TeEvent), que são então armazenados no vetor de eventos (classe TeEventVector). Depois de totalmente preenchido, esse vetor é ordenado, através do algoritmo *sort* da STL, em ordem lexicográfica ("xy"). O processamento feito até este ponto é conhecido como pré-processamento. Em seguida os eventos são processados de forma similar ao algoritmo original (etapa de processamento), mantendo a estrutura com a relação de ordem dos segmentos (classe TeStatusTreeBO). Essa estrutura é implementada como uma árvore balanceada cujos nós contém no máximo um segmento (armazenado no campo redSegment_). O processamento é feito considerando o seguinte:

- Primeiro: interseções entre as extremidades dos segmentos não são computadas como interseções válidas, isto é, conta-se apenas as interseções entre segmentos que se cruzam ou as interseções que sejam em um ponto interior a um dos segmentos.

- Segundo: os segmentos são considerados como abertos, ou seja, dentro de um mesmo evento são processados primeiramente os segmentos que possuem a coordenada direita sobre o evento e, depois, os com a coordenada esquerda.

- As interseções em extremidades de segmentos que não são consecutivos na linha são detectadas durante a etapa de processamento dos eventos. Eventos consecutivos com a mesma coordenada representam interseções entre os extremos de segmentos os quais não são consecutivos.

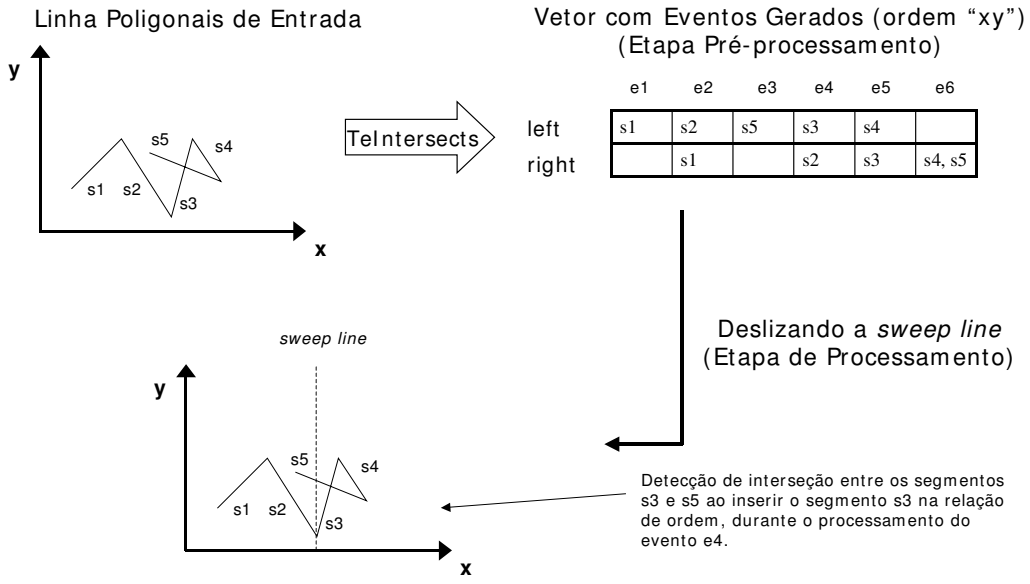


FIGURA 3.10 – Detecção de auto interseção entre segmentos de uma linha poligonal.

A segunda variação (Figura 3.11), implementada pelo segundo operador TeIntersects, testa a interseção entre duas linhas poligonais. Ele é construído sobre as mesmas classes do operador anterior. A grande diferença é que foi optado por manter duas listas de eventos: uma para eventos gerados pela primeira linha poligonal (eventos vermelhos) e outra para a segunda (eventos azuis). Essa decisão foi tomada partindo da observação de que a etapa de pré-processamento representa uma boa parte do tempo de execução dos algoritmos baseados no *plane sweep*. E, além disso, é muito comum um mesmo objeto formado por uma linha poligonal estar envolvido na comparação com vários outros. Armazenando os eventos de cada linha de forma independente é dada a possibilidade de reaproveitar o pré-processamento da linha em futuros testes, diminuindo assim o tempo de execução deles para objetos que já tenham sido testados pelo menos uma vez.

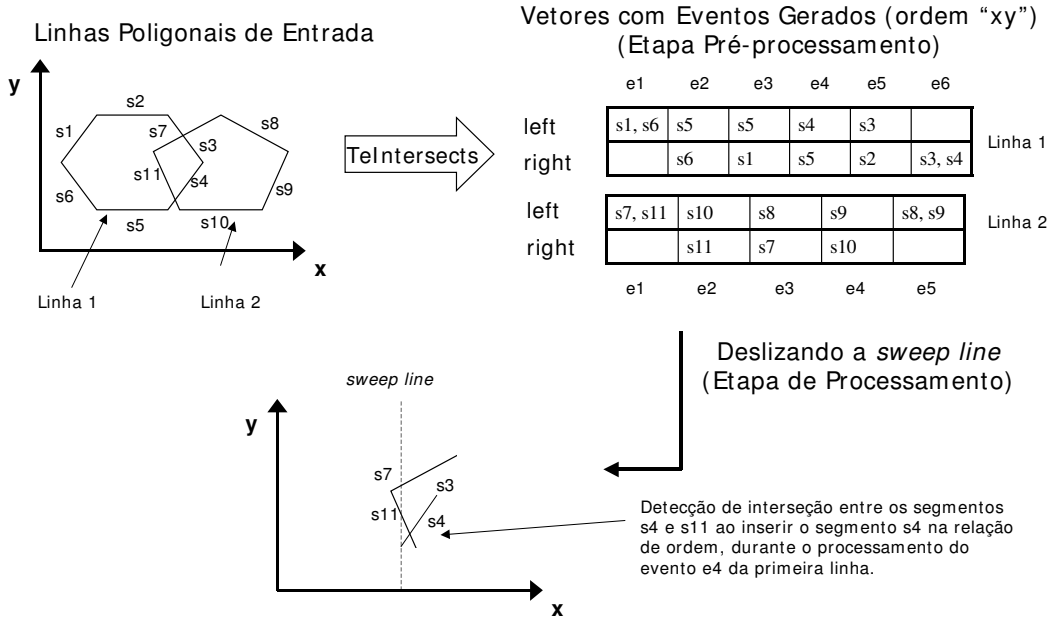


FIGURA 3.11 – Detecção de interseção entre duas linhas poligonais.

Para isso, foi criada a noção de índice de vetores de evento (Figura 3.12, classe TeEventsIndex). Os vetores com os eventos são armazenados nesse índice e recuperados quando necessário. Quando os dados do pré-processamento não são mais úteis, eles são então descartados. O projeto do índice utiliza o padrão Singleton[8] (Gamma et al, 2000).

[8] Esta classe já pertence à TerraLib, e encontra-se em TeSingleton.h

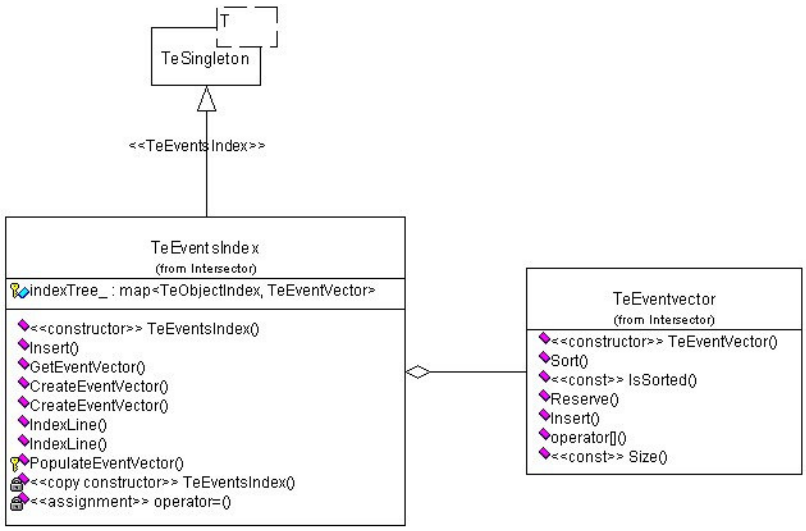


FIGURA 3.12 – Diagrama das classes de índice dos eventos.

O algoritmo apresentado por Shamos e Hoey possui complexidade comprovada de $O(n \log n)$, mas ele somente detecta se há ou não uma interseção. Bentley e Ottmann (1979) apresentam a extensão desse algoritmo para o problema de reportar os pontos de interseção entre um conjunto de n segmentos. O algoritmo original é alterado de forma a manter a ordem vertical dos segmentos que interceptam l mesmo após encontrar um ponto de interseção. Dois segmentos que se interceptem à direita do ponto onde l se encontra estarão na ordem correta até que l passe pelo ponto de interseção. Deste ponto em diante os segmentos alteram sua ordem.

No algoritmo de Bentley e Ottmann um ponto de interseção entre dois segmentos também é considerado um evento. Portanto, as estruturas de dados são modificadas de forma que a fila de eventos (Q) possa ser atualizada com os pontos de interseção, que são inseridos obedecendo a ordem horizontal (x). Essa nova fila de eventos pode ser representada, por exemplo, por uma árvore balanceada. E a estrutura que mantém a relação de ordem (T) é modificada para suportar a operação de inversão de posição de dois segmentos.

O algoritmo possui complexidade sub-ótima, $O(n \log n + k \log n)$, onde k é o número de interseções. Um dos motivos para que ele não atinja o limite inferior de $n \log n + k$ é que os pontos de interseção são reportados na ordem x, que é a ordem na qual eles são

inseridos na fila de eventos. A complexidade desse algoritmo depende não só do número de segmentos de entrada, mas também do número de interseções reportadas. Esse algoritmo pertence a uma classe conhecida como algoritmos sensíveis à saída. Outra análise importante do algoritmo diz respeito à complexidade do espaço de armazenamento requerido por ele, $O(n+k)$ (Bentley e Ottmann, 1979).

Para o problema de reportar o pontos de interseção entre n segmentos, Chazelle e Edelsbrunner (1992) apresentam um algoritmo ótimo, $O(n\log+k)$, com espaço $O(n+k)$. No caso dos operadores espaciais (topológicos e conjunto), esse problema de interseção pode ser refinado para um problema mais específico, conhecido como problema de interseção vermelho e azul (*red/blue segment intersection problem*): dado um conjunto disjunto de segmentos de linha vermelho (R) e um conjunto disjunto de segmentos azuis (B) no plano, determinar as interseções entre os segmentos desses dois conjuntos.

Para esse problema, Chan (1994) apresenta um algoritmo com tempo $O(n\log n+k)$ e espaço $O(n)$, também baseado no paradigma da varredura, conhecido como Trapezoid Sweep. Esse algoritmo se fundamenta na idéia da decomposição do plano em trapézios, chamados de trapézios "azuis" (Figura 3.13), que são os trapézios obtidos a partir dos segmentos azuis e pelas extensões verticais de cada ponto final dos segmentos (vermelhos ou azuis) até os segmentos azuis imediatamente acima e abaixo desses pontos. Essas linhas verticais delimitam as arestas laterais (esquerda e direta) do trapézio. As arestas superiores e inferiores são compostas pelas partes dos segmentos azuis. O interior dos trapézios não contém nenhum ponto final dos segmentos (vermelho ou azul).

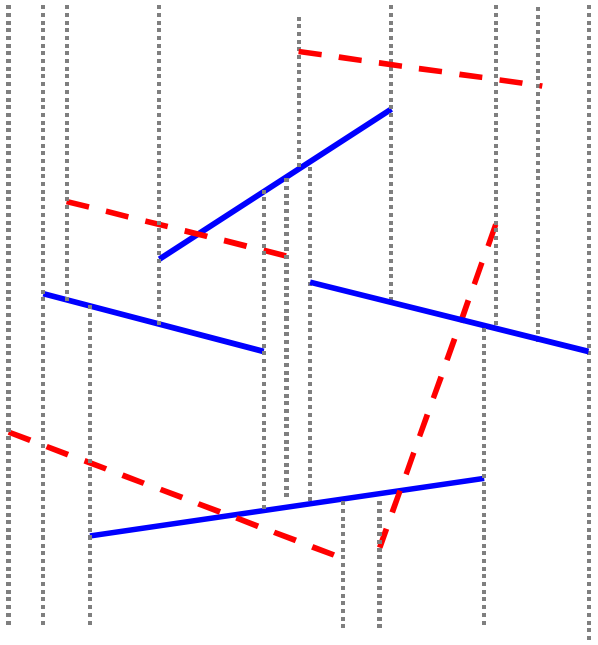


FIGURA 3.13 – Ilustração da decomposição do plano em trapézios.

FONTE: adaptada de (Chan, 1994).

O Algoritmo consiste em varrer os trapézios azuis, da esquerda para a direita, através de uma *sweep line* vertical, adicionando-os a uma *sweep front F* imaginária, toda vez que a parede direita de um trapézio é atingida. Ao término do processamento dos trapézios, todos os pontos de interseção são computados. Uma diferença básica em relação ao algoritmo de Bentley e Ottmann é o uso de duas estruturas para manter a ordem dos segmentos. Uma delas é usada para manter a ordem dos segmentos vermelhos e a outra para manter a ordem dos segmentos azuis, que interceptam a *sweep line*. Maiores detalhes sobre as estruturas de dados e o processamento desse algoritmo podem ser encontrados em (Chan, 1994).

Na TerraLib o paradigma da varredura foi utilizado na implementação do operador TeIntersections, cuja assinatura pode ser vista abaixo. O código fonte encontra-se nos arquivos TeIntersections.h e TeIntersections.cpp do *kernel* da TerraLib.

```
bool
TeIntersections(const TeLine2D& redLine, const TeLine2D& blueLine,
        const TeObjectIndex& redIndex, const TeObjectIndex& blueIndex,
        TeReportVector& report);
```

Esse operador é uma versão modificada do algoritmo de Bentley e Ottmann. O algoritmo de Chan foi considerado para implementação, mas, no entanto, da forma em que ele é descrito no artigo original, ele pressupõe que os segmentos em cada conjunto sejam disjuntos e que não haja interseções em pontos situados sobre as extremidades dos segmentos (caso dos segmentos que se sobrepõe ou que se tocam). Esses casos, descritos no artigo como degenerados, acontecem com freqüência em dados reais e o tratamento dos mesmos em nossa implementação não foi conseguida de forma satisfatória[9].

A versão implementada mantém duas listas distintas para os eventos gerados por cada linha. Além desses eventos, gerados pelas extremidades dos segmentos, há os gerados pelos pontos de interseção (classe TeIntersectionEvent). A Figura 3.14 ilustra as classes projetadas para dar o suporte à manipulação desse último tipo de evento.

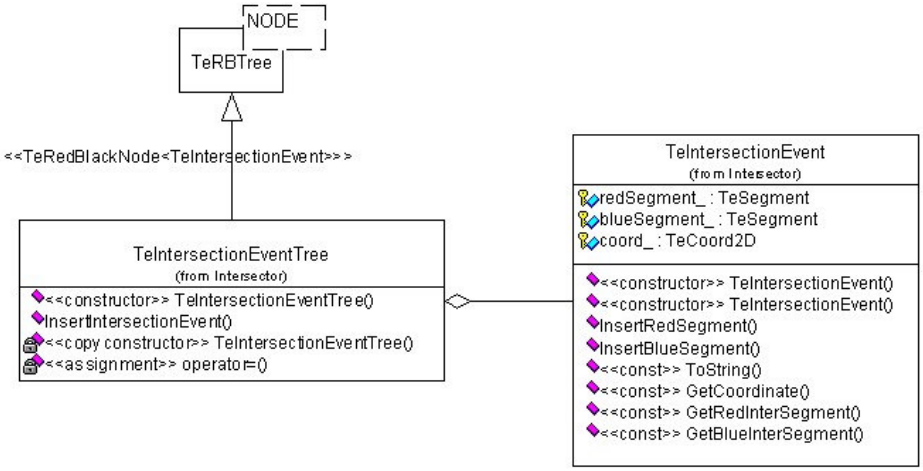


FIGURA 3.14 – Diagrama das classes de tratamento dos eventos de interseção.

Os eventos de interseção são ordenados de acordo com a ordem lexicográfica “xy” dos pontos de interseção. O processamento dos eventos é semelhante ao dos operadores de teste de interseção, com a exceção de que, quando um evento de interseção está sendo processado, a ordem dos segmentos na relação de ordem é alterada. Esta operação é

[9] Em contatos pessoais com o autor do artigo descobrimos que não há uma forma geral para o tratamento desses casos.

realizada removendo os segmentos que se interceptam da relação de ordem e re-inserindo-os novamente na ordem alternada.

3.7.3 Algoritmos de Interseção: Caso III – Partição do Espaço

Os algoritmos mostrados até aqui apresentam complexidades menores que a do algoritmo de força bruta. No entanto, outros fatores ainda devem ser considerados na resolução do problema de reportar os pontos de interseção, como por exemplo, a característica dos dados de entrada.

Andrews et al (1994) e Andrews e Snoeyink (1995) apresentam uma comparação entre métodos advindos da Geometria Computacional, e métodos desenvolvidos pela comunidade GIS para resolver esse problema. Os algoritmos testados por eles foram agrupados em duas categorias: algoritmos por partição espacial e algoritmos por ordenação espacial.

Nos algoritmos da primeira classe, o espaço é subdivido em regiões, e os segmentos são atribuídos às regiões interceptas por cada um deles. As interseções são computadas entre os segmentos de cada região, normalmente empregando um algoritmo de força bruta. A idéia é aplicar heurísticas que realizem filtros, diminuindo o número de segmentos a serem testados.

Os algoritmos agrupados na segunda classe são os baseados em estratégias de Geometria Computacional, onde a preocupação é com o desenvolvimento de algoritmos onde a análise de complexidade de pior caso possua uma boa complexidade. Os algoritmos do Plane Sweep e Trapezoid Sweep podem ser classificados nesta categoria.

Os testes realizados no trabalho deles mostram que embora os algoritmos da primeira classe não garantam uma eficiência no pior caso como os da Geometria Computacional, eles acabam tirando proveito da característica dos dados de um GIS: segmentos curtos, espaçados, com poucas interseções por segmento e uniformemente distribuídos no plano. Dessa forma, eles acabam sendo mais eficientes (velozes) do que os da segunda classe.

Outro trabalho que realiza testes semelhantes é apresentado por Pullar (1990), onde é mostrado que uma técnica baseada no *Fixed Grid*, também pertencente à categoria dos algoritmos por partição, é bastante competitiva em relação aos algoritmos baseados no *plane sweep*, apesar da complexidade de pior caso ser bem maior, $O(n^2)$.

Com o intuito de investigar as conclusões desses trabalhos mais de perto, consideramos o desenvolvimento de um algoritmo que pode ser classificado na categoria dos algoritmos por partição, que é uma adaptação dos trabalhos de (Akman et al, 1989), (Franklin et al, 1988) e (Franklin et al, 1989) baseado no *Fixed Grid*.

Dadas duas linhas poligonais, uma vermelha e outra azul, os segmentos da primeira linha são cobertos por uma grade retangular fixa (*Fixed Grid*) (Figura 3.15). Cada segmento vermelho é associado às células da grade por onde ele passa. Na Figura 3.16 a classe TeGrid é utilizada para representar a grade retangular sobreposta a uma linha poligonal. A associação é mantida em um *multimap* da STL (Austern, 1999). Os pontos de interseção são determinados procurando para cada segmento azul a lista de células por onde ele passa e então utilizando um algoritmo de força bruta esses pontos são determinados.

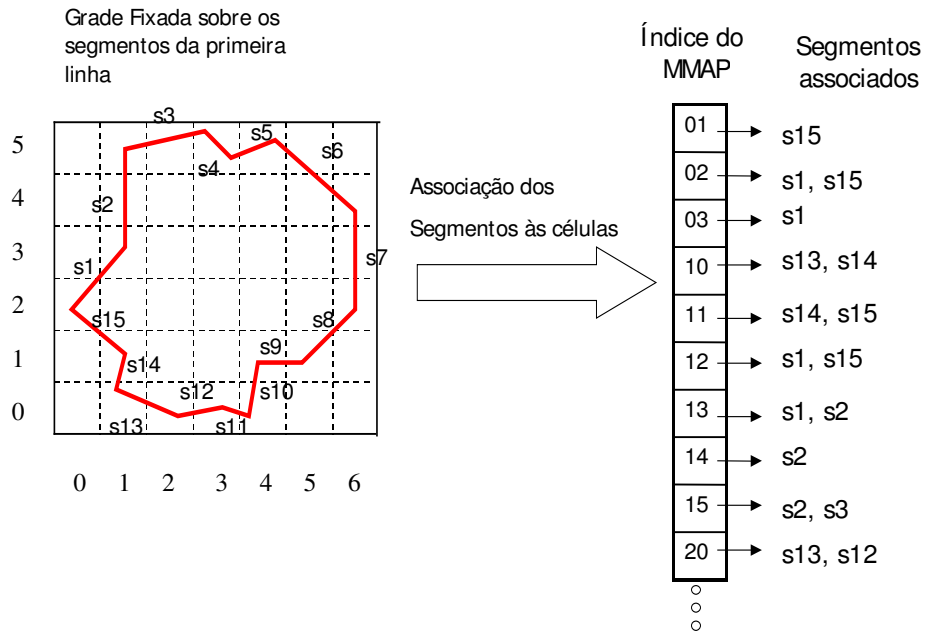


FIGURA 3.15 – Fixed Grid.

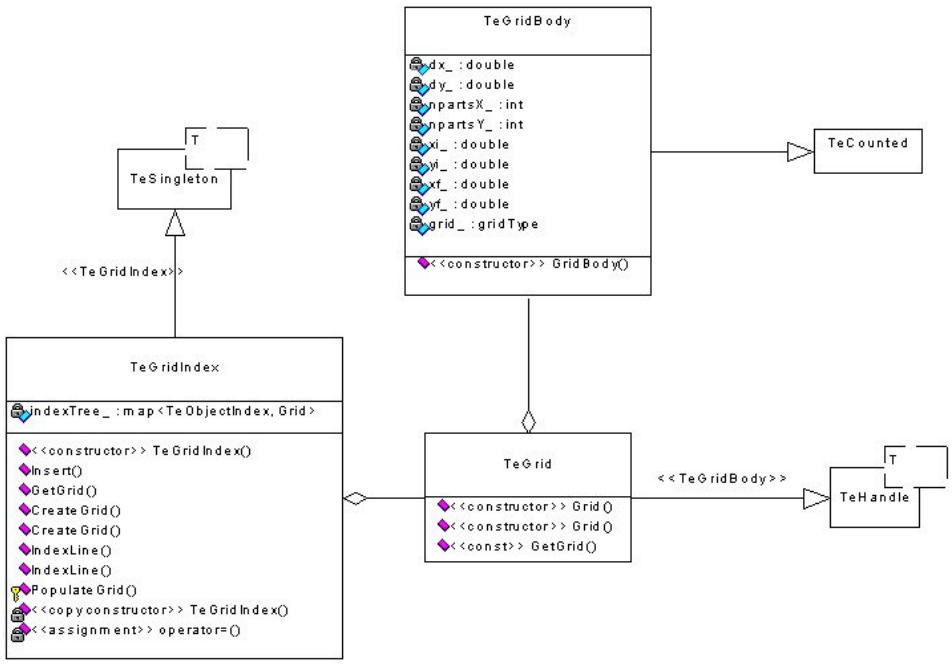


FIGURA 3.16 – Diagrama das classes de suporte do algoritmo do *Fixed Grid*.

Um dos pontos chaves desse algoritmo é a determinação da resolução da grade. Neste trabalho, ela é determinada a partir da média do comprimento dos segmentos. Franklin et al (1988) mostra que utilizando-se de parâmetros estatísticos, como a média do comprimento dos segmentos, a grade se adapta bem aos dados de entrada.

No projeto desse algoritmo, como a etapa onde é feita a associação dos segmentos às células da grade pode ser considerada como um pré-processamento, foi adotada uma estratégia semelhante à usada na Seção 3.7.2 para guardar temporariamente esse pré-processamento (classe TeGridIndex). Isso facilita novos testes com a mesma linha poligonal.

3.7.4 Comparação prática dos algoritmos

A fim de testar na prática a eficiência dos algoritmos acima, alguns testes foram realizados com dados reais. Os dados testados consistem em polígonos extraídos de mapas dos estados brasileiros, dos municípios brasileiros, e das unidades federativas dos Estados Unidos (EUA). As implementações foram executadas em uma máquina PC

Pentium Celeron 2.4 GHz, com 512 Mbytes RAM, utilizando Windows XP. Os tempos coletados para os algoritmos são dados em milisegundos. A Tabela 3.1 apresenta para cada experimento o número de arestas dos polígonos sendo testados (valor de entrada) contra uma própria versão (ligeiramente deslocada) e o tempo de processamento para cada implementação dos algoritmos tomado como a média de 1000 execuções.

TABELA 3.1 – Tempos em milisegundos para o processamento dos algoritmos de interseção.

Segmentos de Entrada	**Força Bruta**	**Plane Sweep**	**Fixed Grid**
25 x 25	0,15	0,313	0,141
51 x 51	0,31	0,625	0,281
128 x 128	1,88	1,609	0,844
219 x 219	5,63	2,922	1,282
410 x 410	18,75	5,968	2,703
801 x 801	68,44	11,360	4,656
1103 x 1103	132,19	15,141	5,094
2431 x 2431	636,87	39,047	12,156
3331 x 3331	1.199,53	55,250	17,469
4356 x 4356	2.020,63	70,750	20,015
8161 x 8161	6.859,40	137,19	36,41
11245 x 11245	12.762,50	195,62	54,06
52422 x 52422	288.792,20	1.035,78	256,57

Os experimentos acima mostram que o *fixed grid* é o que obteve o melhor resultado no geral. Além disso, outros testes com dados reais, como o caso de uso de polígonos adjacentes, mostraram que ele também leva uma considerável vantagem em relação aos demais. Em outras situações práticas, observadas com análises visuais através do software TerraView (DPI, 2003), constatamos que essa técnica apresenta um desempenho superior em quase todas as situações.

Nossos testes corroboram as conclusões de (Andrews et al, 1994), (Andrews e Snoeyink, 1995) e (Pullar, 1990) de que os algoritmos por partição são mais competitivos no caso dos dados de um GIS. Esse desempenho deve-se a dois motivos, primeiro é claro pela característica do dado, e depois, por que apesar da complexidade de pior caso do algoritmo baseado no *Fixed Grid* ser quadrática, para os casos práticos ela é bem menor. Enquanto isso, o algoritmo baseado no *plane sweep* esconde, por trás da complexidade de pior caso, as constantes das operações sobre a árvore balanceada, que não podem ser negligenciadas.

Com tudo, acreditamos que o algoritmo baseado no *plane sweep* teria melhor desempenho no caso de dados que contenham segmentos longos e que estejam distribuídos aleatoriamente. Esses seriam os casos em que o algoritmo do *fixed grid* teria um comportamento assintótico quadrático.

Uma das utilidades dos algoritmos de interseção é durante o processamento de consultas espaciais, como as que envolvem, por exemplo, a restrição de vizinhança (*touches*) entre polígonos: "selecionar os municípios vizinhos à cidade de Araxá". Nessas consultas, os algoritmos do *plane sweep* e do *fixed grid* apresentam grande vantagem sobre o algoritmo força bruta. Isso porque eles nos permitem empregar a técnica desenvolvida neste trabalho de armazenar temporariamente os seus pré-processamentos para posterior uso, enquanto que no força bruta nenhum melhoramento pode ser realizado.

No caso do algoritmo do *plane sweep* esse reaproveitamento significa que, após a primeira cidade ser testada como vizinha a Araxá, o pré-processamento dos eventos gerados pelas linhas poligonais de ambas as cidades serão armazenadas

temporariamente. Quando a segunda cidade for comparada com Araxá, apenas os eventos da linha poligonal da segunda cidade serão criados, os de Araxá serão reaproveitados. Conseqüentemente, teremos um ganho de desempenho. Mas é importante frisar que mesmo nesses casos, onde os pré-processamentos já estão disponíveis, as duas listas de eventos terão que ser percorridas.

No caso do algoritmo do *fixed grid* a idéia é semelhante, no entanto, somente a grade fixada sobre a primeira linha poligonal é armazenada. No caso do teste com uma segunda cidade, é necessário apenas percorrer os segmentos que formam a linha poligonal dela, procurando pelos segmentos da cidade de Araxá já associados às células da grade. Dessa forma, teremos novamente um ganho no processamento deste tipo de consulta.

Essa idéia de reaproveitar o pré-processamento ganha grande importância na geração de matrizes de proximidade entre os objetos constituintes de uma camada de informação, onde é necessário determinar a vizinhança para todos os objetos.

Outra conclusão que tiramos durante o uso do algoritmo do *fixed grid*, é que ele se ajusta melhor a consultas do tipo: "quais os rios que passam dentro do estado de São Paulo". Nesse caso, testado o primeiro rio contra o polígono de São Paulo, o índice armazena a grade com os segmentos do polígono do estado, e na consulta ao próximo rio, apenas as arestas deste serão percorridas. No caso do algoritmo do *plane sweep*, é necessário percorrer os eventos das duas linhas poligonais.

Além disso, em testes futuros, analisaremos a possibilidade de combinação do algoritmo do *fixed grid* com o algoritmo de teste de ponto em polígono usando cruzamento do raio, em situações onde seja necessário testar vários pontos contra um mesmo polígono. Esperamos obter um ganho, pois possivelmente depois do primeiro teste não será mais necessário cruzar o raio contra todas as arestas do polígono.

3.8 Determinação do Relacionamento Topológico entre dois Objetos

Apesar do grande número de modelos existentes para descrever os relacionamentos topológicos entre dois objetos (Seção 2.4), não há na literatura um algoritmo específico para a determinação desses relacionamentos na prática. Este trabalho propõe uma abordagem simples para a determinação desses relacionamentos utilizando os algoritmos geométricos apresentados neste capítulo para determinar as interseções entre interiores, exteriores e fronteiras de dois objetos.

A semântica do conjunto básico de operadores topológicos projetados para a TerraLib segue o modelo da Matriz de 9-Interseções DE e pode ser visto no Apêndice C em C.11. A adoção desse modelo permite que a TerraLib forneça uma interface compatível com a dos SGBDs com extensões espaciais e com as especificações do OGIS.

Os operadores disponibilizados são os mesmos da SFSSQL, com a única diferença, a introdução dos relacionamentos Covers e CoveredBy. Isso permite a distinção de algumas configurações específicas e não impede que seja mantida a compatibilidade com o OGIS. Por exemplo, uma operação chamada TeWithinOrCoveredBy equivalente ao relacionamento *Within* do OGIS poderia ser montada utilizando o operador TeRelation[10], bastando para isso montar uma expressão formada por um "E" lógico do resultado desse operador. A seguir, serão apresentadas as etapas envolvidas na determinação dos relacionamentos.

3.8.1 Exploração do retângulo envolvente

A determinação dos relacionamentos topológicos com base na geometria exata dos objetos envolve processamentos computacionalmente caros. Uma das estratégias comumente empregadas para tentar diminuir esse processamento é a exploração do retângulo envolvente das geometrias (MBR). Nos operadores topológicos da TerraLib, foi empregada uma estratégia adaptada do trabalho apresentado por Clementini et al (1994). Basicamente ela consiste no estabelecimento de um mapeamento entre os

[10] Esse operador informa qual o relacionamento topológico entre dois objetos

relacionamentos topológicos dos MBRs e das geometrias. Abaixo é ilustrado o exemplo para o operador TeContains:

- **TeContains(X, Y)**

```
se TeContains(MBR(X), MBR(Y))
|  então avaliar o rel. entre as geometrias exatas.
|  senão retorna não contém.
fim se
```

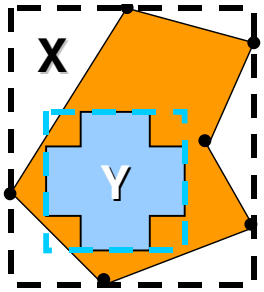


FIGURA 3.17 – Exemplo de exploração do MBR pelos operadores topológicos.

3.8.2 Determinação dos relacionamentos topológicos

A idéia apresentada a seguir, e ilustrada na Figura 3.18, aplica-se à determinação do relacionamento topológico entre dois polígonos simples. No entanto, ela pode ser generalizada para o caso de testes entre pontos e linhas, pontos e polígonos, linhas e polígonos e entre linhas.

Algoritmo Determina Relacionamento Topológico entre Dois Polígonos

Entrada: POLÍGONO R (VERMELHO) e POLÍGONO B (AZUL)
Saída: RELACIONAMENTO TOPOLÓGICO

Passo 0: Comparar os MBR.
Se houve interseção prosseguir no Passo 1.
Senão houve, retorna DISJUNTO(DISJOINT).

Passo 1: Determinar os pontos de interseção entre os dois polígonos. Isso pode ser feito utilizando algum dos algoritmos apresentados na Seção 3.7.
Esta etapa informa se há ou não interseção entre as fronteiras dos objetos.

Passo 2: **Se não houve interseção então**
Testar qualquer ponto do polígono A, num teste de ponto em polígono (Seção 3.5), com B para determinar a localização de A em relação a B. Este teste precisa ser feito também com um ponto de B contra A.

- Se o ponto do polígono A estiver dentro de B então retorne **DENTRO(WITHIN).**
- Se o ponto de B estiver dentro de A então retorne **CONTÉM(CONTAINS).**
- Se os dois testes indicarem que os pontos testados estão fora, então retorne **DISJUNTO(DISJOINT).**

senão

Realizar a fragmentação da fronteira de A, em relação aos pontos de interseção. Esta etapa deve tentar gerar o menor número de fragmentos possíveis.
Testar a localização de cada um dos fragmentos em relação ao polígono azul. Este teste pode ser feito com o algoritmo de ponto em polígono (Seção 3.5).
Se obtivermos uma configuração em que há pelo menos um fragmento dentro e um fora, esta etapa pode ser encerrada.
Com base na localização dos fragmentos, as interseções entre fronteiras, interiores e exteriores podem ser inferidas:

- se houve fragmentos dentro e fora do polígono azul então retorne **SOBREPÕE(OVERLAPS).**
- se houve fragmentos somente dentro e na fronteira do polígono azul então retorne **COBERTO POR(COVERED BY).**
- se houve fragmentos somente fora e na fronteira do polígono azul decidir entre TOCA(TOUCHES) e COBRE(COVERS). Isso pode ser feito fragmentando a fronteira do polígono B e testando a localização dos fragmentos em relação a A.
  se houver fragmentos dentro de A, então retorne **COBRE(COVERS)** senão retorne **TOCA(TOUCHES).**
- se todos os fragmentos estavam na fronteira então retorne **EQUALS.**

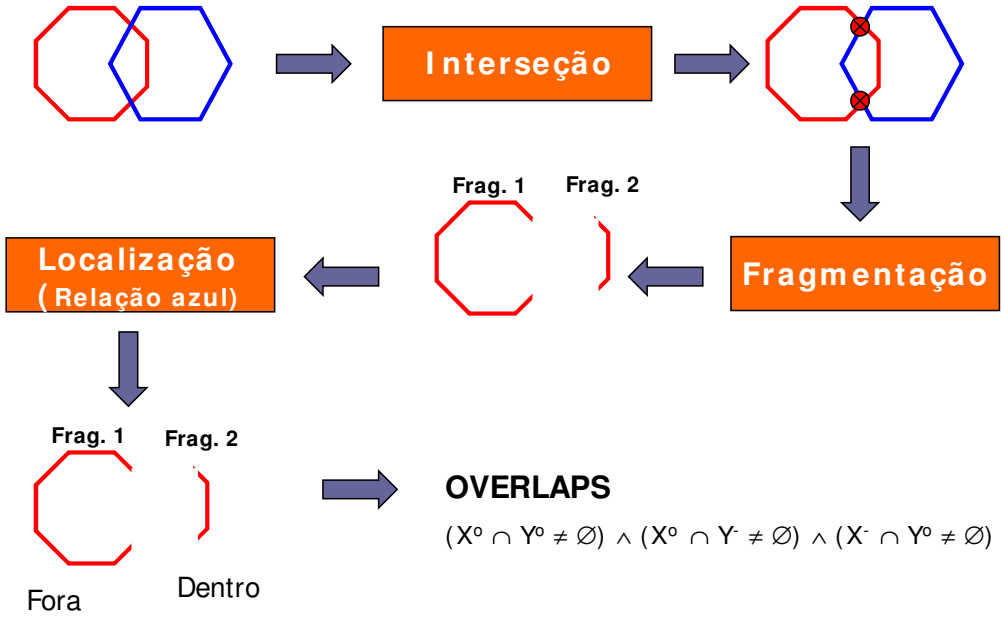


FIGURA 3.18 – Exemplo de Determinação do Relacionamento Topológico.

O teste de relacionamento topológico para o caso de pontos e polígonos foi feito utilizando-se o teste de ponto em polígono (TePointInPoly) e o teste de ponto sob a linha de fronteira do polígono (TeIsOnLine).

Para o caso de pontos e linhas basta utilizar o teste TeIsOnLine para determinar se o ponto encontra-se sobre a linha. Para o caso de linhas com linhas, utiliza-se o algoritmo de interseção e o fragmentador, para determinar o relacionamento entre elas. O teste entre linhas e polígonos segue a mesma idéia do algoritmo para teste entre polígonos.

A eficiência do algoritmo colocado acima está ligada aos algoritmos de determinação dos pontos de interseção e de fragmentação das fronteiras. De posse dos pontos de interseção, outra etapa fundamental é gerar os fragmentos da fronteira do polígono. Esta etapa é de suma importância, pois evita que todos os vértices de cada polígono tenham que ser testados em um teste de ponto em polígono. Ao invés disso, apenas um vértice de cada um dos fragmentos que possuam mais que dois vértices precisam ser testados. Para fragmentos formados apenas por dois vértices, testa-se um ponto interno, como, por exemplo, o ponto médio.

O fragmentador assume verdadeira importância no caso de polígonos adjacentes (que se tocam), neste caso, a fronteira em comum poderia exigir que vários testes de ponto em polígono fossem realizados, mas com a reconstrução da fronteira apenas um teste para toda a fronteira é necessário. O operador que realiza a fragmentação pode ser visto no arquivo TeFragmentation.h e TeFragmentation.cpp no *kernel* da TerraLib. A implementação dos operadores topológicos encontra-se disponível nos arquivos TeGeometryAlgorithms.h e TeGeometryAlgorithms.cpp.

3.9 Operações de Conjunto

As operações de conjunto: união, interseção e diferença (Figura 3.19), foram implementadas na TerraLib utilizando o algoritmo apresentado por Margalit e Knott (1989). Esse algoritmo, simples e prático, pode ser aplicado a polígonos com ou sem buracos. Ele recebe como entrada dois polígonos (A e B), cada um formado por um único anel, podendo representar uma ilha ou um buraco, e a informação da operação desejada (união, interseção ou diferença).

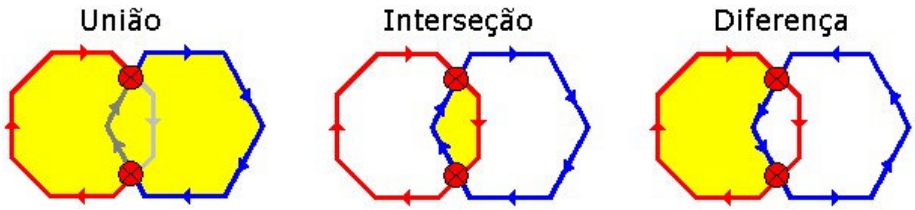


FIGURA 3.19 – Ilustração das Operações de Conjunto.

Esse algoritmo é descrito originalmente em seis passos. Alguns deles foram adaptados durante a implementação na tentativa de melhorar o desempenho global do processamento. As etapas descritas abaixo correspondem à forma como ele foi implementado:

1. Primeiro, as orientações dos polígonos são normalizadas de acordo com a Tabela 3.2. Aqui, o operador TeOrientation é aplicado para descobrir a orientação de cada um dos polígonos. Caso a orientação não seja satisfeita, ela é então alterada. Essa etapa é equivalente ao primeiro passo do algoritmo original.

TABELA 3.2 – Regras de Orientação.

Tipo de Polígono		**Operação**		
A	**B**	**União**	**Interseção**	**Diferença**
ilha	ilha	mesmas orientações	mesmas orientações	orientações opostas
ilha	buraco	orientações opostas	orientações opostas	mesmas orientações
buraco	ilha	orientações opostas	orientações opostas	mesmas orientações
buraco	buraco	mesma orientação	mesmas orientações	orientações opostas

2. Os pontos de interseção entre os polígonos são determinados utilizando o operador TeIntersections baseado no *Fixed Grid*. Essa etapa equivale ao terceiro passo do algoritmo original, com exceção do tipo de técnica usada para determinar os pontos.

3. A terceira etapa consiste em fragmentar as fronteiras dos dois polígonos, utilizando o operador de fragmentação (TeFragmentation) mencionado na seção anterior. O objetivo dessa etapa é evitar que todos os vértices de cada polígono tenham que ser testados contra o outro no momento da classificação. *Essa é a principal diferença da implementação na TerraLib do algoritmo original*. É esperado que em casos práticos, nem todos os vértices precisem ser checados para realizar a sua classificação. Ela pode ser inferida através do fragmento ao qual o vértice pertence.

4. A quarta etapa consiste em classificar os fragmentos da fronteira de cada um dos polígonos em relação ao outro, em "fora", "dentro" ou na "fronteira". Esta classificação é feita utilizando o operador TePointInPoly e o operador TeIsOnline. Os fragmentos classificados são mantidos em duas árvores balanceadas (T_1 para os fragmentos vermelhos e T_2 para os fragmentos azuis) indexadas pela coordenada inicial dos fragmentos. Essas árvores são representadas pelo contêiner *map* da STL. Apenas os segmentos correspondentes

ao tipo da operação a ser realizada são mantidos (Tabela 3.3). As estruturas utilizadas representam outra alteração em relação ao algoritmo original. Essa etapa é equivalente aos passos 2, 4 e 5 do algoritmo original.

TABELA 3.3 – Tipo de Fragmento Selecionado x Operação.

Tipo de Polígono		**Operação**					
		União		**Interseção**		**Diferença**	
A	**B**	**A**	**B**	**A**	**B**	**A**	**B**
ilha	Ilha	fora	fora	Dentro	Dentro	fora	dentro
ilha	buraco	dentro	fora	Fora	Dentro	dentro	dentro
buraco	Ilha	fora	dentro	dentro	Fora	fora	fora
buraco	buraco	fora	fora	Fora	Fora	dentro	fora

5. Os polígonos de saída são construídos procurando para cada fragmento, contido na estrutura de dados da etapa anterior, a sua continuação. Essa etapa equivale ao sexto passo do algoritmo original.

O algoritmo acima encontra-se implementado na TerraLib sob o nome de TeOperation. Uma de suas aplicações prática é a construção da operação de *buffer* discutida a seguir.

3.10 Mapas de Distância

Outra operação importante para um GIS é a construção de mapas de distância ou *buffer zones*, que são áreas construídas ao redor de objetos mantendo uma certa distância. A Figura 3.20 ilustra a idéia dessas operações para pontos, linhas e polígonos respectivamente.

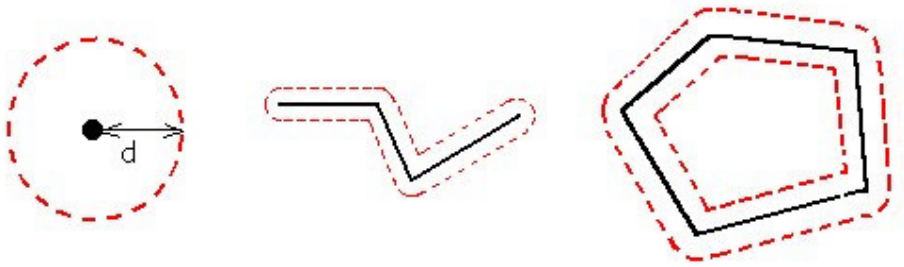


FIGURA 3.20 – *Buffer Zones* de ponto, linha e polígono.

A determinação do *buffer* ao redor de um ponto é feita de forma direta, como uma circunferência de raio *d*. O *buffer* ao redor de uma linha é formada pela união de *buffers* elementares definidos para cada segmento da linha. Esses *buffers* elementares são formados a partir de semicircunferências traçadas nas extremidades dos segmentos (uma em cada extremidade). E utilizando o operador de união, esses *buffers* são combinados até formar o resultado final da linha. O *buffer* de polígonos é semelhante ao de linha, com a diferença de que é possível gerar *buffers* negativos (voltados para o interior do polígono).

3.11 Robustez e Casos Degenerados

Os algoritmos geométricos apresentados até aqui são, geralmente, projetados e provados corretos, assumindo os dados de entrada na configuração geral e também a computação exata sobre números reais (van Kreveld et al, 1997). Porém, na implementação desses algoritmos, a aritmética exata é trocada pela aritmética finita dos computadores atuais, e os dados do mundo real podem encontrar-se em posições especiais (não somente na configuração geral). Isso acarreta sérios problemas, pois as provas de correção dos algoritmos não se estendem às implementações. As configurações de dados de entrada em posições especiais são conhecidas como casos degenerados. E os problemas advindos desses casos e da aritmética finita são conhecidos como problemas de robustez.

No algoritmo de Bentley e Ottmann, por exemplo, segmentos verticais, segmentos que se sobrepõem e segmentos que se interceptam apenas nas extremidades são considerados como casos degenerados. No algoritmo de teste de ponto em polígono, que

utiliza o número de cruzamentos do raio, os casos degenerados são os de pontos que se localizam sobre a fronteira ou cujo raio cruze vértices do polígono. Um dos efeitos desses casos para os algoritmos mencionados é o de aumentar o número de tratamentos especiais na implementação, o que pode resultar em problemas de ineficiência e de robustez, pois as implementações tornam-se muito complexas.

Nos algoritmos implementados na TerraLib esses casos foram devidamente tratados, como no caso do operador de ponto em polígono, que foi complementado com o operador que checa se o ponto encontra-se sobre a fronteira. No caso do operador construído baseado no algoritmo de Bentley e Ottmann, os casos degenerados foram tratados durante as operações de inserção e de troca de posição dos segmentos da estrutura que mantém a relação de ordem.

Já o problema de robustez numérica está muito ligado a avaliação de predicados geométricos, como o de teste da orientação de um ponto em relação a outros dois (Seção 3.1) e de operadores de construção de novas geometrias, como o cálculo da interseção entre dois segmentos (Seção 3.6). Na literatura, há basicamente três abordagens possíveis para a substituição da hipótese de aritmética exata sobre números reais:

- A primeira consiste na substituição pela aritmética de ponto flutuante, com respectivo tratamento dos erros de arredondamento (Chazelle et al, 1996). A forma mais comum do tratamento desses erros é o uso de tolerâncias. Neste caso, os testes condicionais são comparados com um valor ε, estabelecido com base em experimentos ou por algum critério. Caso o valor comparado seja menor que ε, ele é tratado como zero. Nesse modo de computação, é assumido tempo constante ($O(1)$) para as operações aritméticas.

- A segunda técnica consiste no uso de aritmética exata (Yap e Dubé, 1995). Esta técnica substitui a aritmética real, por um subconjunto deste, tipicamente números inteiros, racionais ou de ponto flutuante de alta precisão. Yap e Dubé (1995) apresentam uma revisão de vários trabalhos realizados nesta área. O uso de números inteiros para representar as coordenadas pode apresentar problemas

de *overflow* durante a realização das operações geométricas. As outras técnicas, números racionais e de precisão estendida, apresentam problemas de desempenho, pois as operações passam a ser realizadas em software, ou seja, não se pode assumir que as operações sejam executadas em tempo $O(1)$.

- Schneider et al (1998) descreve uma abordagem, chamada de Geometria Computacional de Precisão Finita, cuja idéia é levar o problema do domínio contínuo para o domínio discreto. A discretização ocorre na forma de uma grade uniforme de inteiros, onde as operações são executadas. Pontos, pontos extremos de segmentos de linha e vértices de polígonos são representados por coordenadas inteiras ajustadas sobre pontos da grade.

Apesar da grande quantidade de pesquisas feitas na última década, não há um consenso sobre qual abordagem é a melhor para o tratamento da robustez numérica (Tamassia et al, 1996). Cada uma possui inconveniências, tipicamente problemas de limitação em sua aplicação ou ineficiência.

As primitivas geométricas da TerraLib foram construídas utilizando-se de tolerâncias (ε) para tentar contornar os problemas de precisão dos cálculos com números em ponto flutuante. A classe TeGeometryAlgorithmsPrecision ilustrada no diagrama UML da Figura 3.21 foi projetada com o intuito de gerenciar o valor ε e as operações de comparação que envolvam este valor. Essa classe evita que vários parâmetros de tolerância tenham que ser espalhados pelo código dos algoritmos geométricos.

Como regra geral, o valor ε pode ser ajustado de acordo com a precisão do dado de entrada. Por exemplo, para um dado com coordenadas que possuam três dígitos decimais de precisão, usa-se um valor ε de 0.0005. O Apêndice C (C.12) apresenta a implementação desta classe.

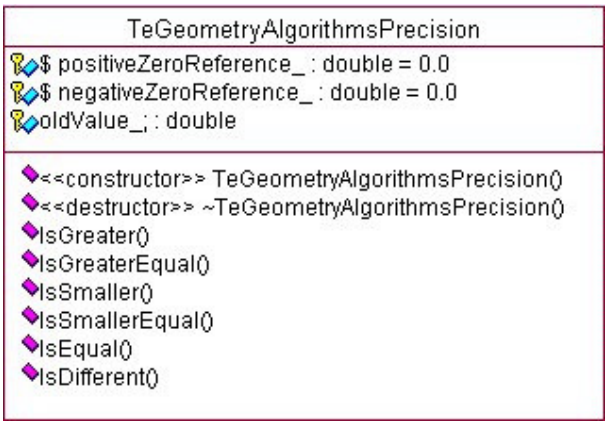


FIGURA 3.21 – Classe de gerenciamento da tolerância ε.

CAPÍTULO 4

EXEMPLO DE APLICATIVO GEOGRÁFICO

Os operadores espaciais desenvolvidos neste trabalho foram integrados na TerraLib e o resultado da aplicação deles pode ser visualizado através do aplicativo geográfico TerraView (DPI, 2003) construído sobre essa biblioteca. Esses operadores são utilizados pela biblioteca para dar o suporte aos SGBD que não possuem extensões espaciais ou nos quais as operações ainda não foram completadas, como o MySQL, PostgreSQL, SQL Server e Access.

A Figura 4.1 ilustra duas consultas espaciais realizadas no aplicativo. A primeira é uma consulta de seleção por apontamento, onde a cidade de Araxá foi selecionada e destacada utilizando o operador topológico TeWithin. A segunda consulta consiste em determinar as cidades vizinhas à Araxá com o auxílio do operador TeTouches.

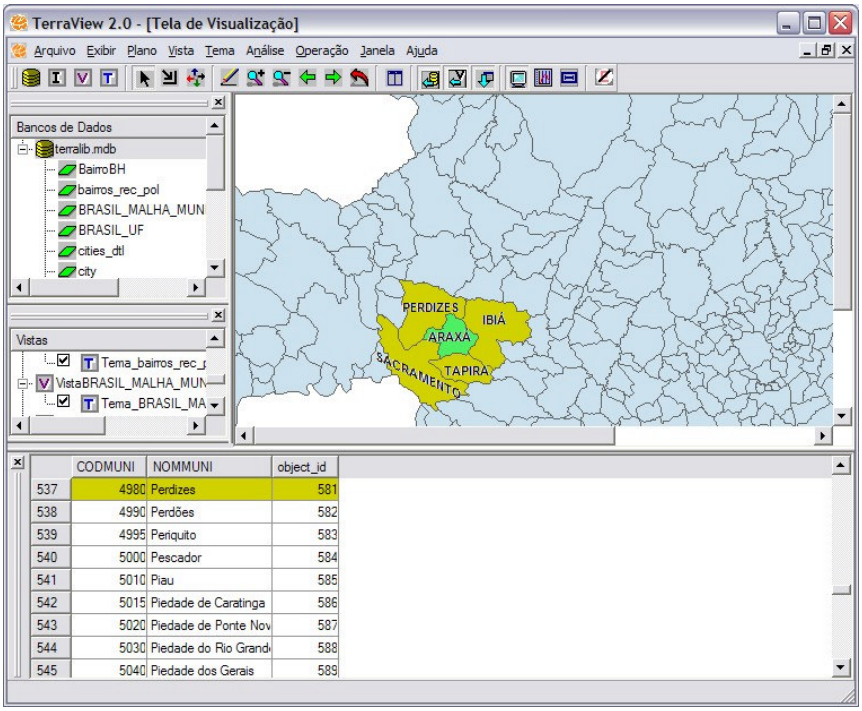


FIGURA 4.1 – Uso do operador TeTouches para determinar os vizinhos da cidade Araxá.

O aplicativo disponibiliza a interface para consultas espaciais mostrada na Figura 4.2. Como exemplo, é mostrada algumas telas com os resultados de consultas espaciais que utilizam os operadores TeBufferRegion, TeConvexHull e TeUnion (Figuras 4.3, 4.4 e 4.5 respectivamente).

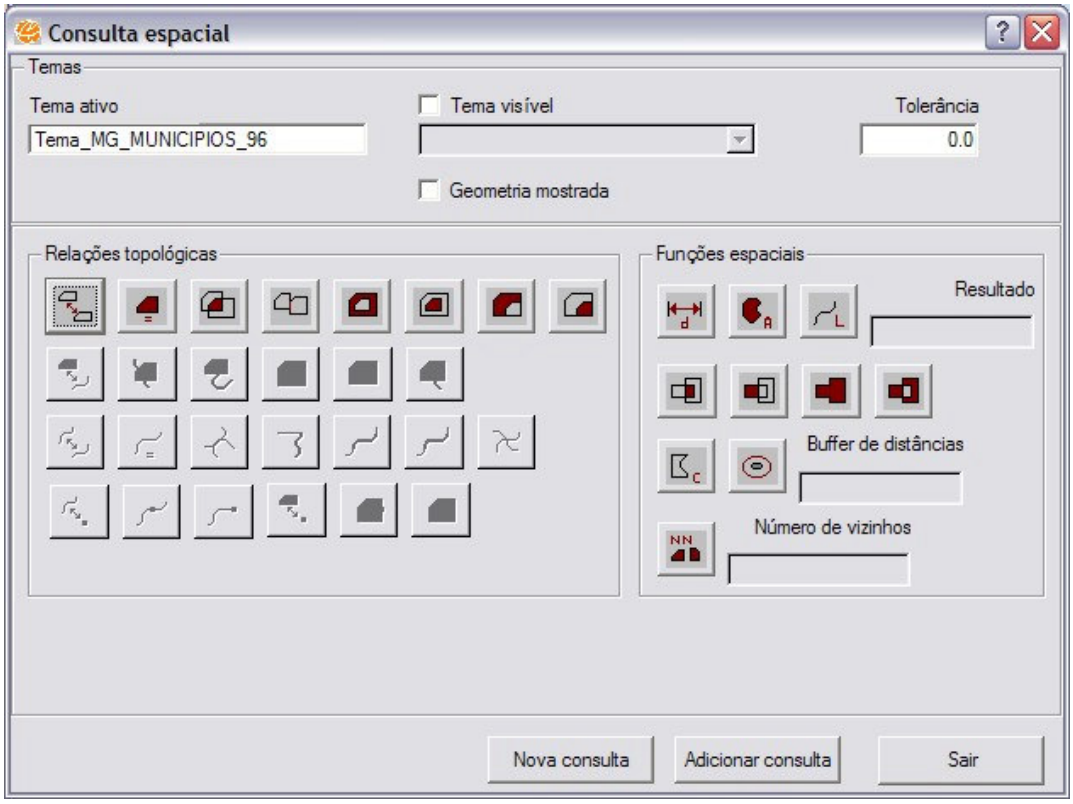


FIGURA 4.2 – Interface para consultas espaciais.

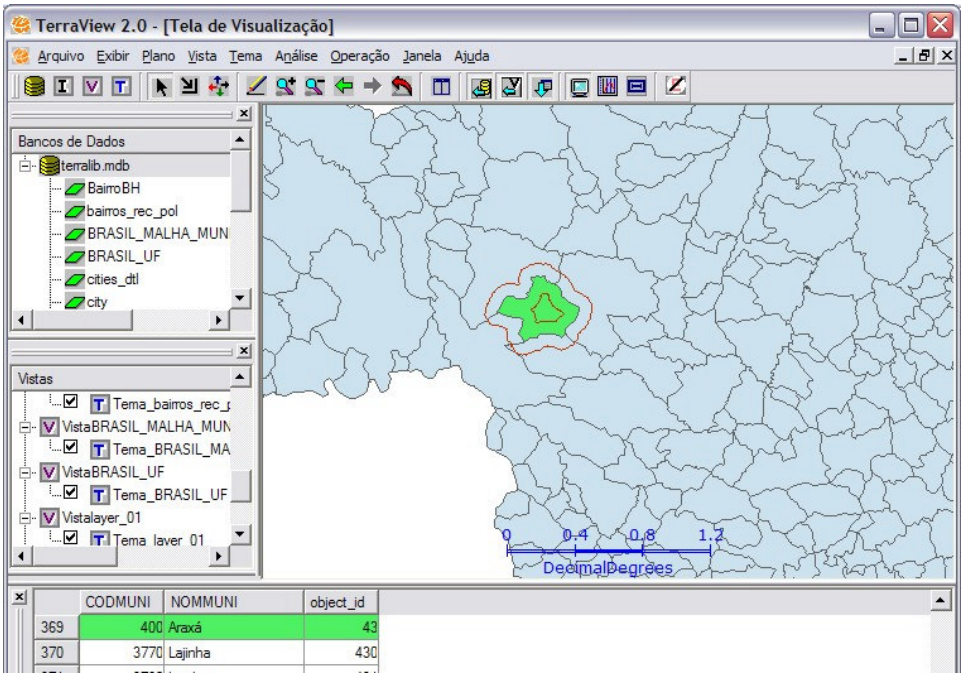


FIGURA 4.3 – Resultado do operador TeBuffer.

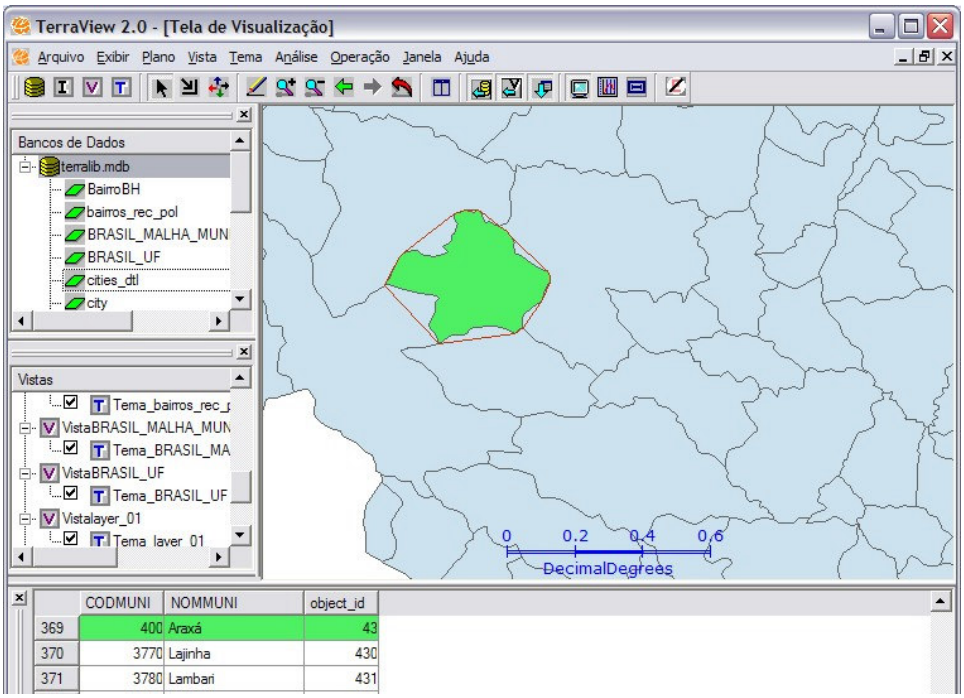


FIGURA 4.4 – Resultado da aplicação do operador TeConvexHull.

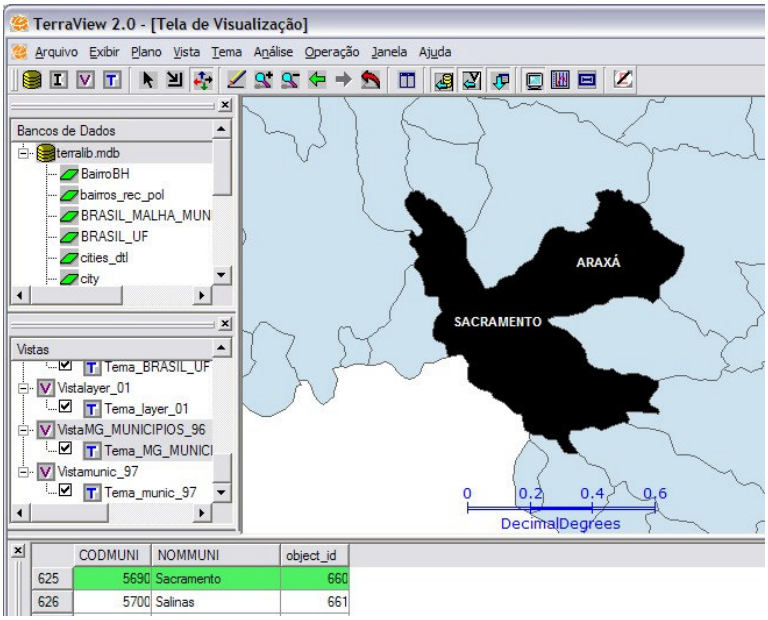


FIGURA 4.5 – União dos polígonos das cidade de Araxá e Sacramento.

Além de serem usados na janela de consulta espacial os operadores topológicos são empregados em várias partes da biblioteca, como nas rotinas de importação de dados, na criação de camadas celulares (Figura 4.6) e algoritmos de análise espacial.

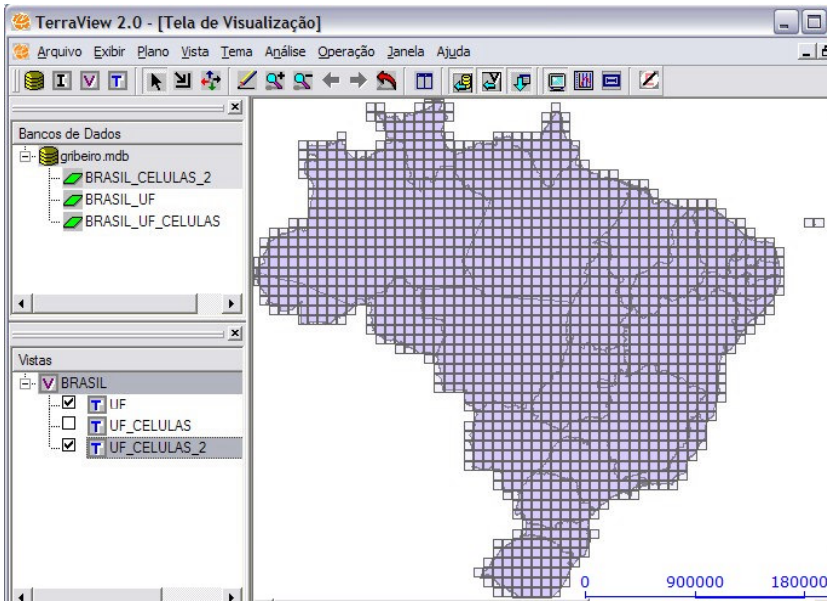


FIGURA 4.6 – Recobrimento do plano de polígonos por células.

CAPÍTULO 5

CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho apresentou o desenvolvimento de operadores espaciais para banco de dados geográficos implementados na TerraLib. Esses operadores permitem que a biblioteca mantenha uma interface de programação de aplicações (API) comum, suprindo a demanda de operações espaciais em SGBD que não as possuem. Além disso, o conjunto de operadores topológicos disponíveis na biblioteca permite que seja mantida compatibilidade com as especificações propostas no OGIS.

Conforme visto na pesquisa por algoritmos eficientes para computação dos pontos de interseção entre linhas poligonais, podemos observar que o critério de escolha desses algoritmos no caso de aplicações GIS não pode simplesmente se pautar na análise de complexidade de pior caso do algoritmo. Nossos testes mostraram que algoritmos com complexidade de pior caso maior que a de outros, como o caso dos algoritmos do *Fixed Grid* e do *Plane Sweep*, na prática podem ter um desempenho superior. Com os algoritmos empregados nesse trabalho, os usuários podem realizar consultas espaciais sem ter que esperar por longos períodos de tempo para o seu processamento.

Uma lição importante aprendida com o emprego de tolerâncias no tratamento de erros numéricos é que ela não soluciona definitivamente o problema de robustez numérica. Ela apenas adia o tratamento dessas questões. Mas seu emprego neste trabalho se justifica por ele estar voltado para o uso de algoritmos geométricos eficientes na resolução de problemas em bancos de dados geográficos. Contudo, a idéia é começar a levantar questões e discussões a respeito deste assunto para que, no futuro, novas formas de computação numérica sejam avaliadas e então incorporadas à TerraLib.

Abaixo, são citados alguns trabalhos que serão explorados em trabalhos futuros:

- Exploração de técnicas para realizar eliminação de testes geométricos através de aproximações do interior dos objetos, como realizado atualmente pelo Oracle Spatial (Kothuri e Ravada, 2001).

- Avaliação da integração dos trabalhos de Narkhede e Manocha (2003) juntamente à TerraLib, para explorar consultas espaciais de ponto em polígono onde um número muito grande de pontos seja consultado para o mesmo polígono.

- Avaliação da integração do algoritmo do *Fixed Grid* com o teste de ponto em polígono e comparação com a implementação de Narkhede e Manocha (2003).

- Desenvolvimento e comparação com outros operadores de interseção para linhas poligonais, como o X-Order e o Tiling apresentados por Pullar (1990), o uso de QuadTrees (Samet, 1990) e BSP Trees (van Oosterom, 1990).

- Desenvolvimento de estrutura para representação de sub-divisões planares como a DCEL (Preparata e Shamos, 1985).

- Utilização de técnicas de computação exata ou representação de coordenadas através de inteiros com respectivo tratamento de *overflow*.

REFERÊNCIAS BIBLIOGRÁFICAS

Abler, R. F. The National Science Foundation National Center for Geographic Information and Analysis. **International Journal of Geographical Information Systems**, v. 1, n. 4, p. 303-326, 1987.

Akman, V.; Franklin, W. R.; Kankanhalli, M.; Narayanaswami, C. Geometric computing and uniform grid technique. **Computer-Aided Design**, v. 21, n. 7, p. 410-420, set. 1989.

Andrew, A. M. Another efficient algorithm for convex hulls in two dimensions. **Information Processing Letters**, n. 9, p. 216-219, 1979.

Andrews, D. S.; Snoeyink, J. Geometry in GIS is not combinatorial: segment intersection for polygon overlay. In: Annual Symposium on Computational Geometry, 11., 1995, Vancouver. **Proceedings…** Canada: British Columbia, 1995, p. 424-425.

Andrews, D. S.; Snoeyink, J.; Boritz, J. Chan, T.; Denham, G.; Harrison, J.; Zhu, C. Further comparison of algorithms for geometric intersection problems. In: International Symposium on Spatial Data Handling, 6., 1994, Edinburgh. **Proceedings…** Taylor and Francis, 1994, p. 709-724.

Austern, M. H. **Generic programming and the STL**. Massachusetts: Addison-Wesley, 1999.

Bentley, J. L.; Ottmann, T. A. Algorithms for reporting and counting geometric intersections. **IEEE Transactions on Computers**, v. C-28, n. 9, p. 643-647, set. 1979.

Bourke, P. **Determining if a point lies on the interior of a polygon**. Disponível em: <http://www.astronomy.swin.edu.au/~pbourke>. Acesso em: maio 2002.

Bourke, P. **Intersection point of two lines (2 dimensions)**. Disponível em: <http://astronomy.swin.edu.au/~pbourke/geometry/lineline2d>. Acesso em: maio 2002.

Bourke, P. **Calculating the area and centroid of a polygon**. Disponível em: <http://astronomy.swin.edu.au/~pbourke/geometry/polyarea>. Acesso em: maio 2002.

Câmara, G.; Casanova, M. A.; Hemerly, A. S.; Magalhães, G. C.; Medeiros, C. M. B. **Anatomia de sistemas de informação geográfica**. Campinas: Unicamp, 1996.

Câmara, G.; Souza, R. C. M.; Pedrosa, B. M.; Vinhas, L.; Monteiro, A. M. V.; Paiva, J. A. Paiva; Carvalho, M. T.; Gatass, M. Terralib: technology in support of GIS innovation. In: Workshop Brasileiro de Geoinformática, 2., 2000, São Paulo. **Anais...** São Paulo: INPE, 2000, p. 126-133.

Câmara, G.; Vinhas, L.; Souza, R. C. M.; Paiva, J. A.; Monteiro, A. M. V.; Carvalho, M. T.; Raoult, B. Design patterns in gis development: The terralib experience. In: Workshop Brasileiro de Geoinformática, 3., 2001, Rio de Janeiro. **Anais...** Rio de Janeiro: IME, 2001, p. 95-101.

Chan, T. M. A simple trapezoid sweep algorithm for reporting red/blue segment intersections. **6th Can. Conf. Comp. Geom.** Saskatoon, Saskatchewan, 1994.

Chazelle, B.; Edelsbrunner, H. An optimal algorithm for intersecting line segments in the plane. **JACM**, v. 39, n. 1, p. 1-54, 1992.

Chazelle, B.; et al. **Application challenges to computational geometry: computational geometry task force report**. Princeton University, dez. 1996. Disponível em: <http://www.cs.duke.edu/;jeffe/compgeom/taskforce.html>. Acesso em: maio 2003.

Ciferri, R. R.; Salgado, A. C. Análise de Eficiência de Métodos de Acesso Espaciais em Termos da Distribuição Espacial dos Dados. In: Workshop Brasileiro de Geoinformática, 3., 2001, Rio de Janeiro. **Anais…** Rio de Janeiro: IME, 2001, p. 95-101.

Clementini, E.; Di Felice, P. A comparison of methods for representing topological relationships. **Information Sciences**, n. 3, p. 149-178, 1995.

Clementini, E.; Di Felice, P.; van Oosterom, P. A small set of formal topological relationships for end-user interection. In: Abel, D.; Ooi, B. C. (Org.). Advances in Spatial Databases – Third International Symposium, SSD'93. **Lecture Notes in Computer Science LNCS 692**. Singapore: Springer-Verlag, 1993, pp. 277-295.

Clementini, E.; Sharma, J.; Egenhofer, M. Modelling topological spatial relations: strategies for query processing. **Computers & Graphics**, v. 18, n. 6, p. 815-822, 1994.

Cormen, T. H.; Leiserson, C. E.; Rivest, R. L. **Introduction to algorithms**. E.U.A.: McGraw-Hill, 1990.

Date, C. J. **Introdução a sistemas de bancos de dados**. Rio de Janeiro: Campus, 1990.

Davis, C. A.; Borges, K. A. V.; Laender, A. H. F. Restrições de integridade em bancos de dados geográficos. In: Workshop Brasileiro de Geoinformática, 3., 2001, Rio de Janeiro. **Anais...** Rio de Janeiro: IME, 2001, p. 95-101.

Davis, C.; Câmara, G. Arquitetura de Sistemas de Informação Geográfica. In: Câmara, G.; Davis, C.; Monteiro, A. M. V. (Org.). **Introdução à ciência da geoinformação**. São José dos Campos: INPE, out. 2001. cap. 3.

de Berg, M.; van Kreveld, M.; Overmars, M.; Schwarzkopf, O. **Computational geometry: algorithms and applications**. Berlin Heidelberg: Springer-Verlag, 1997.

Doyon, J. F. **MapFile reference - MapServer 4.0**. Disponível em: <http://mapserver.gis.umn.edu/>. Acesso em: nov. 2003.

DPI/INPE. **Manual do Spring**. Disponível em: <http://www.dpi.inpe.br/spring>. Acesso em: jan. 2002.

DPI; **Documentação TerraLib**. Disponível em: <http://www.terralib.org.br>. Acesso em: out. 2003.

Egenhofer, M. J.; Clementini, E.; Di Felice, P. Topological relations between regions with holes. **International Journal of Geographical Information Systems**, v. 8, n. 2, p. 129-142, 1994.

Egenhofer, M. J.; Franzosa, R. D. Point-set topological spatial relations. **International Journal of Geographical Information Systems**, v. 5, n. 2, p. 161-174, 1991.

Egenhofer, M. J.; Herring, J. R. **Categorizing binary topological relations between regions, lines, and points in geographic databases**. Orono: University of Maine, 1991.

Evenden, G. I. **Cartographic projection procedures for the Unix environment - a user's manual**. Disponível em: <http://remotesensing.org/proj/>. Acesso em: nov. 2003.

Ferreira, K. R. **API Genérica**. 2003. Dissertação (Mestrado em Computação Aplicada) – Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2003.

Ferreira, K. R.; Queiroz, G. R.; Paiva, J. A. C.; Souza, R. C. M.; Câmara, G. Arquitetura de software para construção de bancos de dados geográficos com SGBD Objeto-Relacionais. In: Simpósio Brasileiro de Bancos de Dados**,** 18., out. 2002, Gramado, RS, pp. 57-67.

Figueiredo, L. H.; Carvalho, P. C. P. **Introdução à geometria computacional**. Rio de Janeiro: IMPA, 1991.

Frank, A. U. Requirements for database systems suitable to manage large spatial databases. In: International Symposium on Spatial Data Handling, 1., 1984, Zurich. **Proceedings…** Zurich, 1984, p. 38-60.

Franklin, W. R.; Chandrasekhar, N.; Kankanhalli, M.; Seshan, M.; Akman, V. Efficiency of Uniform Grids for Intersection Detection on Serial and Parallel Machines. In: Computer Graphics International, 1988, Geneva, Switzerland. **Proceedings…** Geneva, maio 1988, p. 51-62.

Franklin, W. R.; Chandrasekhar, N.; Kankanhalli, M.; Sun, D.; Zhou, M.; Wu, P. YF Uniform grids: a technique for intersection detection on serial and parallel machines. In: Auto Carto 9: International Symposium on Computer-Assisted Cartography, 9., abril 1989, Baltimore, Mariland. **Proceedings…** The Society The Congress, 1989, p. 100-109.

Gaede, V.; Günther, O. Multidimensional access methods. **ACM Computing Surveys**, v. 30, p. 170-231, 1998.

Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. **Padrões de projeto: soluções reutilizáveis de software orientado a objetos**. Porto Alegre: Bookman, 2000.

Garcia-Molina, H.; Ullman, J. D.; Widom, J. **Implementação de sistemas de bancos de Dados**. Rio de Janeiro: Campus, 2001.

Gonzalez, R. C.; Woods, R. E. **Processamento de imagens digitais**. São Paulo: Edgard Blücher, 2000.

Güting, R. H.; An introduction to spatial database systems. **VLDB Journal**, v. 3, n. 4, p. 357-399, out. 1994.

Guttman, A. R-trees: a dynamic index structure for spatial search. **ACM SIGMOD**, p. 47-57, Jun. 1984.

Haines, E. Point in polygon strategies. In: Heckbert, P. S. (Org.). **Graphics Gems IV**. Boston, E.U.A.: Academic Press, 1994. p. 24-46.

Hellerstein, J. M.; Naughton, J. F.; Pfeffer, A. Generalized search trees for databases systems. In: international Conference in VLDB, 21., set. 1995, Zurich, Switzerland. **Proceedings…** Morgan Kaufman, 1995, 562-573.

Huang, C.; Shih, T. On the complexity of point-in-polygon algorithms. **Computers & Geosciences**. v. 23, n. 1, p. 109-118, 1997.

IBM. **IBM DB2 Spatial Extender: user's guide and reference**. Disponível em: <http://www.ibm.com.br>. Acesso em: jun. 2002.

IBM. **Working with the Geodetic and Spatial DataBlade modules**. Disponível em: <http://www.ibm.com>. Acesso em: out. 2003.

Kernighan, B. W.; Ritchie, D. M. **C: a linguagem de programação - padrão ANSI**. Brasil: Campus, 1990.

Knuth, D. E. **The art of computer programming, vol. 1**. MA: Addison-Wesley, 1973.

Kothuri, R. K. V.; Ravada, S.; Abugov, D. Quadtree and R-tree indexes in Oracle Spatial: a comparison using GIS data. **ACM SIGMOD**, p. 546-557, jun. 2002.

Kothuri, R. K.; Ravada, S. Efficient processing of large spatial queries using interior approximations. In: **LNCS 2121**. Berlin Heidelberg: Springer-Verlag, 2001, p. 404-421.

Margalit, A.; Knott, G. D. An algorithm for computing the union, intersection or difference of two polygons. **Computers & Graphics**, v. 13, n. 2, p. 167-183, 1989.

Medeiros, C. B.; Pires, F. Databases for GIS. **ACM SIGMOD**, p. 107-115, 1994.

MySQL AB. **MySQL reference manual**. Disponível em: <http://www.mysql.com>. Acesso em: nov. 2003.

Narkhede, A.; Manocha, D. **Fast polygon triangulation based on Seidel's algorithm**. Disponível em: <http://www.cs.unc.edu/~dm/CODE/GEM/chapter.html>. Acesso em: jul. 2003.

Nievergelt, J.; Preparata, F. P. Plane-Sweep algorithms for intersecting geometric figures. **ACM**, v. 25, n. 10, p. 739-747, Out. 1982.

Nordbeck, S.; Rystedt, B. Computer cartography point in polygon problems. **BIT**, n. 7, p. 39-64, 1967.

Open GIS Consortium. **Opengis simple features specification for SQL revision 1.1**. Disponível em: <http://www.ogis.org>. Acesso em: maio 1995.

Oracle Coporation. **Oracle Spatial guide**. Disponível em: <http://www.oracle.com>. Acesso em: maio 2003.

O'Rourke, Joseph. **Computacional geometry in C**. Cambridge: Cambridge University Press, 1998.

Page-Jones, M. **Fundamentals of object-oriented design in UML**. New York: Addison-Wesley, 2000.

Preparata , F. P.; Shamos , M. I. **Computational geometry an introduction**. New York: Springer-Verlag, 1985.

Pullar, D.; Comparative study of algorithms for reporting geometrical intersections. In: International Symposium on Spatial Data Handling, 4., 1990, Zurich. Proceedings… 1990, p. 66-76.

Queiroz, G. R.; Vinhas, L.; Ferreira, K. R.; Câmara, G.; Paiva, J. A. C. Programação genérica Aplicada a algoritmos de Agrupamento. In: Workshop dos Cursos de Computação Aplicada do INPE, 2., nov. 2002, São José dos Campos. **Anais...** São José dos Campos: INPE, nov. 2002, p. 31-47.

Ramalho, J. A. **Microsoft SQL Server 7 - iniciação e referência**. São Paulo: Makron Books, 1999.

Ramsey, P. **PostGIS manual**. Disponível em: <http://postgis.refractions.net/documentation>. Acesso em: jan. 2002.

Ravada, S.; Sharma, J. Oracle8i Spatial: experiences with extensible databases. In: International Symposium on Spatial Databases, 6., jul. 1999, Hong Kong, China. **Proceedings…** Berlin: Springer-Verlag, 1999, p. 355-359.

Refractions Inc. **GEOS API documentation**. Disponível em: <http://geos.refrcations.net>. Acesso em: nov. 2003.

Rezende, P. J.; Stolfi, J. **Fundamentos de geometria computacional**. Campinas: Unicamp Press, 1994.

Rigaux, P.; Scholl, M.; Voisard, A. **Spatial databases with application to GIS**. San Francisco: Morgan Kaufmann Publishers, 2002.

Saalfeld, A. It doesn't make me nearly as CROSS - some advantages of the point-vector representation of line segments in automated cartography. **International Journal of Geographical Information Systems**, v. 1, n. 4, p. 379-386, 1987.

Samet, H. **Application of spatial data structures: computer graphics, image processing, and GIS**. MA: Addison-Wesley, 1990.

Samosky, J.; **SectionView: a system for interactively specifying and visualizing sections through three-dimensional medical image data**. Dissertação (Mestrado) – Department of Electrical Engineering and Computer Science, MIT, 1993.

Schneider, M. **Spatial data types for database systems**. Berlin Hidelberg: Springer-Verlag, 1997.

Schneider, M.; Güting, R. H.; de Ridder, T. Computational Geometry on the Grid: Traversal and Plane-Sweep Algorithms for Spatial Applications. In: Canadian Conf. on Computational Geometry, 10., ago. 1998, Montreal, Canada. **Proceedings…** Montreal: McGill University, 1998. Disonível em: <http://cgm.cs.mcgill.ca/cccg98/proceedings>. Acesso em: fev. 2003.

Seidel, R. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. **Computacional Geometry: Theory and Applications**, n. 1, p. 51-64, 1991.

Shamos, M. I. Geometric complexity. In: Annual ACM Symposium on Theory of Computing, 17., 1975, Albuquerque, EUA. **Proceedings…** New York: ACM Press, 1975, p. 224-233.

Shamos, M. I.; Hoey, D. Closest-point problems. In: Annual IEEE Symposium on Foundations of Computer Science, 16., oct. 1975, University of California, Berkeley. **Proceedings…** IEEE, 1975, p. 151-162.

Shamos, M. I.; Hoey, D. Geometric intersection problems. In: Annual IEEE Symposium on Foundations of Computer Science, 17., oct. 1976, Houston, Texas. **Proceedings…** IEEE, 1976, p. 208-215.

Shekar, S.; Chawla, S.; Ravada, S.; Fetterer, A.; Liu, X.; Lu, C. Spatial databases: accomplishments and research needs. **IEEE Transactions on Knowledge and Data Engineering**, v. 11, n. 1, p. 45-55, jan. 1999.

Silberschatz, A.; Korth, H. F.; Sudarshan, S. **Sistemas de bancos de dados**. São Paulo: Makron Books, 1999.

Stonebraker, M. **Object-relational DBMSs: the next great wave**. San Francisco: Morgan Kaufmann, 1996.

Stonebraker, M.; Rowe, L. A.; Hirohama, M. The implementation of Postgres. **IEEE Transactions on Knowledge and Data Engineering**, v. 2, n. 1, p. 125-142, mar. 1990.

Stroustrup, B; **The C++ programming language**. Massachusetts: Addsison-Wesley, 1997.

Tamassia, R.; et al. Strategic Directions in Computational Geometry. **ACM Computing Surveys**, v. 28, n. 4, p. 591-606, Dez. 1996.

Taylor, G. E.; Point in Polygon Test. **Survey Review**, v. 32, n. 254, p. 479-484, 1994.

van Kreveld, M.; Nievergelt, J.; Roos, T.; Widmayer, P. **Algorithmic foundations of geographic information systems**. Germany: Springer-Verlag, 1997.

van Oosterom, P.; A modified binary space partitioning tree for geographic information systems. **International Journal of Geographical Information Systems**, v. 4, n. 2, p. 133-146, Abr. 1990.

Vinhas, L.; Souza, R. C. M.; Câmara, G. Image data handling in spatial databases. In: Simpósio Brasileiro de Geoinformática**,** 5., nov. 2003, Campos Jordão, Brasil. **Anais...** INPE, 2003. CD-ROM.

Vivid Solutions. **JTS technical specifications**. Disponível em: <http://www.vividsolutions.com/jts/jtshome.htm>. Acesso em: nov. 2003.

Wood, D. **Data structures, algorithms, and performance**. MA: Addisson-Wesley, 1993.

Worboys, M. F. **GIS a computing perspective**. London: Taylor & Francis, 1995.

Yap, C.; Dubé, T. **Exact computation paradigm**. World Scientific Press, pp. 452-492, 1995.

Yarger, R. J.; Reese, G.; King, T. **MySQL & mSQL**. EUA: O'Reilly, 1999.

APÊNDICE A

OpenGIS SFSSQL

A proposta da SFSSQL do OpenGIS (OGIS) consiste na especificação de um conjunto de geometrias, de operações topológicas, métricas e que geram novas geometrias, em tipos de dados vetoriais. Além disso, essa especificação contém um esquema de tabelas para metadados das informações espaciais. Ela introduz o conceito de "tabela com feições" para representação dos dados geográficos. Nesta tabela, os atributos não espaciais são mapeados para colunas de tipos disponíveis na SQL92, e a componente espacial para colunas cujo tipo de dados é baseado no conceito de "tipos de dados geométricos adicionais para SQL".

As colunas geométricas podem seguir dois modelos, o SQL92 e o SQL92 com Tipos Geométricos. O primeiro consiste na utilização de uma tabela para representar as colunas espaciais. A segunda, utiliza o conceito de tipos abstratos de dados, sendo a coluna representada por um tipo geométrico que estende os tipos da SQL.

Modelo de Dados das Geometrias

A Figura A.1 ilustra a hierarquia de tipos definida pela especificação do OGIS. Este diagrama é o mesmo tanto para o modelo da SQL92 quanto para o da SQL92 com Tipos Geométricos.

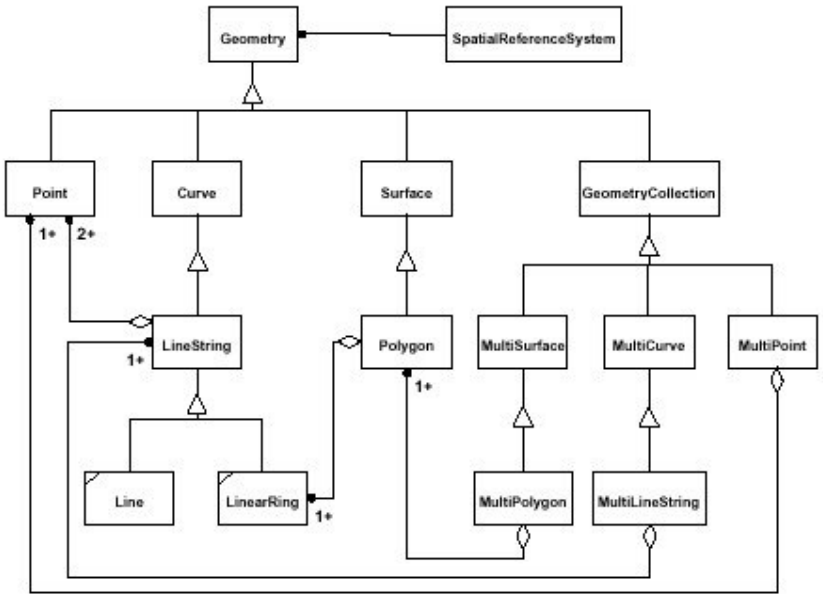


FIGURA A.1 – Hierarquia de geometrias da SFSSQL.

FONTE: adaptada de (OGIS, 1995).

O modelo acima representa objetos 0, 1 ou 2 dimensionais no espaço bidimensional. Algumas classes são abstratas como: Curve, Surface, MultiSurface e MultiCurve. Uma classe especial é a GeometryCollection, que pode ser composta por mais de um tipo de geometria (tipo heterogêneo). As outras classes são tipos básicos, como no caso de Point, LineString e Polygon, que podem formar coleções homogêneas como MultiPoint, MultiLineString e MultiPolygon, respectivamente. Cada uma dessas classes possui uma série de atributos, métodos e definições que são apresentadas na especificação.

Operadores para Relacionamentos Topológicos

Os operadores de relacionamento topológico são definidos como métodos da classe Geometry e, portanto, servem para testar os relacionamentos topológicos entre quaisquer objetos do modelo anterior. Os seguintes operadores estão definidos: Equals, Disjoint, Intersects, Touches, Crosses, Within, Contains, Overlaps. Todos são métodos unários que recebem uma outra geometria como parâmetro de entrada e retornam o valor inteiro 0 (FALSE) se o relacionamento não for satisfeito e 1 (TRUE) caso este seja satisfeito.

Além desses oito operadores, um outro operador, chamado Relate, é definido, recebendo a geometria a ser comparada e um segundo parâmetro que representa um padrão da matriz de interseção na forma de uma string de nove caracteres (teste entre interseção de fronteira, interior e exterior). Ele retorna o inteiro 1 (TRUE) se as geometrias forem relacionadas espacialmente de acordo com os valores especificados na matriz. O OGIS utiliza o modelo de 9-Interseções DE.

Outros Operadores Importantes

Além dos operadores topológicos, a especificação define vários outros métodos para a classe Geometry que são comumente empregados pelos GIS. Entre eles pode-se citar:

- Distance(outraGeometria:Geometry):Double = retorna a distância entre as geometrias;
- Buffer(distância:Double):Geometry = retorna uma geometria definida por um mapa de distância;
- ConvexHull():Geometry = retorna um polígono convexo que contém todos os pontos da geometria;
- Intersection(outraGeometria:Geometry):Geometry = retorna a geometria resultante da interseção das geometrias;
- Union(outraGeometria:Geometry):Geometry = retorna a geometria resultante da união de duas geometrias;
- Difference(outraGeometria:Geometry):Geometry : retorna a geometria resultante da diferença entre as geometrias.

A classe Surface e MultiSurface ainda apresentam especificamente:

- Area():Double = área de uma região;
- Centroid():Point = um ponto representando o centróide da geometria;

- PointOnSurface():Point = um ponto que esteja na superfície.

Arquitetura de Implementação das Tabelas com Feições

O OGIS propõe o seguinte esquema (Figura A.2) de metadados para representação dos dados geográficos para tabelas com atributos de Tipos Geométricos na SQL:

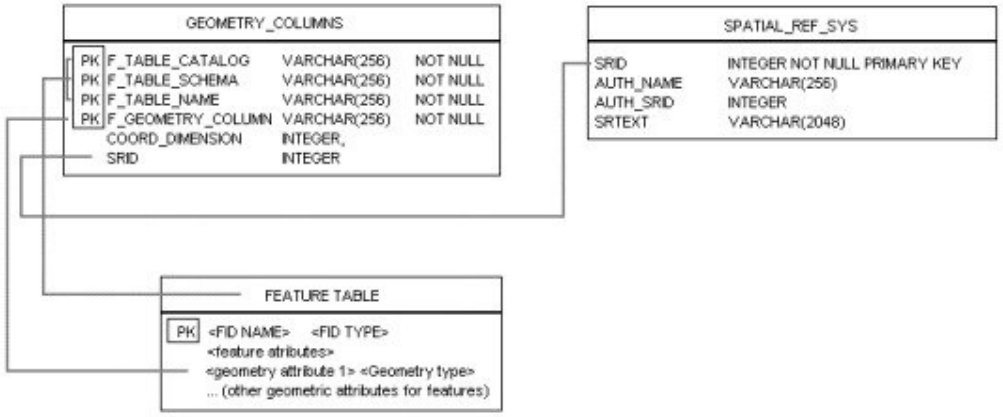


FIGURA A.2 – Esquema de Metadados da SFSSQL.

FONTE: adaptada de (OGIS, 1995).

A tabela SPATIAL_REF_SYS armazena as informações de cada sistema de coordenadas (SRS) utilizado no banco de dados. A tabela GEOMETRY_COLUMNS serve como tabela de metadado para as colunas geométricas das tabelas com feições. Nestas tabelas, cada instância de uma geometria deve ser associada com um SRS de forma a permitir associações de SRS a instâncias que ainda não estejam armazenadas no banco. Isso é útil para que as funções possam verificar a compatibilidade dos SRS de objetos. Além das tabelas, a especificação cita que a SQL deve suportar um subconjunto dos tipos mostrados no diagrama da Figura A.1. A Tabela \ref{tabnivel} mostra os possíveis tipos que uma implementação deva suportar.

TABELA A.1 – Possíveis Combinações dos Tipos Geométricos

Nível	**Tipos Disponíveis**	**Tipos Instanciáveis**
1	Geometry, Point, Curve, LineString,	Point, LineString, Polygon,

	Surface, Polygon, GeomCollection	GeometryCollection
2	Geometry, Point, Curve, LineString, Surface, Polygon, GeomCollection, MultiPoint, MultiCurve, MultiLineString, MultiSurface, MultiPolygon	Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon
3	Geometry, Point, Curve, LineString, Surface, Polygon, GeomCollection, MultiPoint, MultiCurve, MultiLineString, MultiSurface, MultiPolygon	Point, LineString, Polygon, GeomCollection, MultiPoint, MultiLineString, MultiPolygon

Uma coluna pode ser de qualquer um dos tipos disponíveis. Uma coluna declarada como sendo de um certo tipo pode armazenar quaisquer instâncias deste tipo além de instâncias de seus subtipos.

APÊNDICE B

Códigos Exemplo do Mecanismo de Extensibilidade do PostgreSQL

Código B.1 – Definição de um Tipo de Dados

A extensão de tipos de dados no PostgreSQL pode ser feita em linguagem C (Kernighan e Ritchie, 1990) através da definição de uma estrutura de dados, que é responsável pela representação do tipo em memória. O Trecho de código abaixo ilustra a definição de um tipo chamado TeLinearRing, que representa uma linha fechada composta por um conjunto de coordenadas (TeCoord2D). Os tipos definidos pelo usuário podem ser de tamanho fixo ou variável, como no caso do exemplo abaixo.

```
/*---------------------------------------------------------------------
 * TeLinearRing -> Provides support for a 2D linear ring.
 *            A linear ring is a 2D line (without self-intersections)
 *            whose first point is the same as the last point.
 *---------------------------------------------------------------------
 */
typedef struct
{
 int32 size;            //struct length, required for
                        //PostrgreSQL as varlena types
 int32 ncoords_;        //number of coords in the line
 TeCoord2D coords_[1];  //variable length array of Coords
} TeLinearRing;
```

Código B.2 – Rotina de Entrada para TeLinearRing

Além da estrutura de dados, é necessário definir outras duas rotinas, que fazem a conversão do tipo de acordo com a sua representação em memória para ASCII e vice-versa. Estas rotinas são conhecidas como funções de entrada e saída, e elas associam uma representação textual externa para o dado. Para o tipo TeLinearRing teríamos o seguinte formato de representação: [(X1, Y1), (X2, Y2), ..., (Xn, Yn)].

A função de entrada recebe uma *string* (na representação externa) como parâmetro e retorna a representação interna do tipo de dado. O trecho de código abaixo ilustra a implementação de uma rotina de entrada para o tipo TeLinearRing.

```
PG_FUNCTION_INFO_V1(TeLinearRing_in);
Datum TeLinearRing_in(PG_FUNCTION_ARGS)
{
 char *str = PG_GETARG_CSTRING(0);
 TeLinearRing *ring;
 int ncoords;
 int size;
 char *s;
 if((ncoords = elements_in_TeLine2D_or_SimpleSet(str,RDELIM_TeLine2D))
     <= 0)
    elog(ERROR, "Bad TeLinearRing external representation '%s'", str);

 size = offsetof(TeLinearRing, coords_[0]) + sizeof(ring->coords_[0])
        * ncoords;
 ring = (TeLinearRing *) palloc(size);
 MemSet((char *) ring, 0, size);
 ring->size = size;
 ring->ncoords_ = ncoords;
 if((!TeLine2D_SimpleSet_decode(ncoords, str, &s, &(ring->coords_[0]),
          LDELIM_TeLinearRing, RDELIM_TeLinearRing)) || (*s != '\0'))
      elog(ERROR, "Bad TeLinearRing external representation '%s'",
           str);
 if(!is_TeLinearRing(ring))
   {
    pfree(ring);
    ring = NULL;
    elog(ERROR, "In a TeLinearRing the first point must be the same as
                 the last point '%s'", str);
   }
 PG_RETURN_TeLine2D_P(ring);
}
```

Código B.3 – Rotina de Saída para TeLinearRing

A função de saída recebe o dado na representação interna e deve convertê-lo para a representação externa. O trecho de código abaixo ilustrada a função de saída para o tipo TeLinearRing.

```
PG_FUNCTION_INFO_V1(TeLinearRing_out);
Datum TeLinearRing_out(PG_FUNCTION_ARGS)
{
 TeLinearRing *ring = (TeLinearRing *)PG_GETARG_TeLinearRing_P(0);
 char *str;
 str = palloc(ring->ncoords_ * (P_MAXLEN + 3) + 2);
 if(!TeLine2D_SimpleSet_encode(ring->ncoords_, ring->coords_, str,
```

```
                                LDELIM_TeLinearRing,
                                RDELIM_TeLinearRing,
                               "Unable to format TeLinearRing"))
   elog(ERROR, "Unable to format TeLinearRing");
 PG_RETURN_CSTRING(str);
}
```

Código B.4 – Registrando o Tipo TeLinearRing

Depois de criado o tipo, uma biblioteca compartilhada deve ser gerada para ser integrada dinamicamente ao servidor de banco de dados. Isto evita que o servidor tenha que ser interrompido. Depois, é necessário registrar o tipo através do comando SQL: **CREATE TYPE**, informando as rotinas de entrada e saída:

```
/*****************************************
  TeLinearRing SQL statements
 *****************************************/
CREATE FUNCTION telinearring_in(opaque)
    RETURNS telinearring
    AS '/opt/tepgdatatypes.so'
    LANGUAGE 'c';

CREATE FUNCTION telinearring_out(opaque)
    RETURNS opaque
    AS '/opt/tepgdatatypes.so'
    LANGUAGE 'c';

CREATE TYPE telinearring
(
 alignment = double,
 internallength = VARIABLE,
 input = telinearring_in,
 output = telinearring_out,
 storage = main
);
```

Código B.5 – Criação de uma Função

O trecho de código abaixo ilustra a definição de uma função para calcular a área de um polígono simples representado por um TeLinearRing.

```
/*---------------------------------------------------------------------
 * Area of a TeLinearRing
 *---------------------------------------------------------------------
 */
PG_FUNCTION_INFO_V1(TeLinearRingArea);
Datum TeLinearRingArea(PG_FUNCTION_ARGS)
```

```
{
 TeLinearRing *r = (TeLinearRing *)
                   PG_DETOAST_DATUM(PG_GETARG_DATUM(0));
 double area = 0.0;
 int    npoints = 0;
 int    loop;
 npoints = r->ncoords_;

 for(loop = 0; loop < (npoints - 1); ++loop)
 {
     area += ((r->coords_[loop].x_ * r->coords_[loop + 1].y_) -
              (r->coords_[loop + 1].x_ * r->coords_[loop].y_)) ;
 }
 area *= 0.5;
 PG_RETURN_FLOAT4(area);
}
```

Código B.6 – Registrando a Função de Cálculo de Área

As funções devem ser colocadas em uma biblioteca compartilhada para poderem ser integradas ao servidor de banco de dados. É necessário registrar a função através do comando SQL: **CREATE FUNCTION**, como mostrado abaixo.

```
CREATE FUNCTION area(telinearring)
RETURNS float4
AS '/opt/tepgfunctions.so', 'telinearringarea'
LANGUAGE 'c' with (isstrict);
```

Código B.7 – Definição de um operador

Uma funcionalidade importante oferecida pelo PostgreSQL é que os nomes das funções podem ser sobrecarregadas desde que os parâmetros sejam de tipos diferentes. Depois de criada uma função é possível definir um operador para ela através do comando SQL: **CREATE OPERATOR**. A importância da definição do operador está ligada ao otimizador de consultas que pode usar as informações contidas na definição do operador para decidir a melhor estratégia para a consulta (ou simplificação desta). Outra funcionalidade oferecida pelo PostgreSQL é a definição de funções agregadas, que podem ser construídas de forma análoga às funções sobre tipos, porém são registradas com o comando SQL: **CREATE AGGREGATE**.

```
CREATE OPERATOR op_name
(
```

```
 leftarg = tipo,
 rightarg = tipo,
 procedure = função,
 commutator = op_name
);
```

APÊNDICE C

Código dos Operadores Espaciais da TerraLib

Os operadores espaciais estão organizados segundo a seguinte estrutura no *kernel* da TerraLib:

- Os operadores topológicos, de teste de ponto em polígono, de cálculo de área, do cálculo do envoltório convexo, os que testam se um polígono é convexo e o teste de colinearidade foram organizados nos arquivos TeGeometryAlgorithms.h e TeGeometryAlgorithms.cpp.

- Os operadores de conjunto (interseção, união e diferença) foram organizados nos arquivos TeOverlay.h e TeOverlay.cpp.

- Os operadores de mapa de distância foram organizados nos arquivos TeBufferRegion.h e TeBufferRegion.cpp.

- Os operadores de interseção entre linhas poligonais e entre dois segmentos foram organizados em TeIntersector.h e TeIntersector.cpp.

- O operador de fragmentação de fronteiras foi organizado em TeFragmentation.h e TeFragmentation.cpp.

- O *framework* de uma árvore Red-Black encontra-se em TeRedBlackTree.h.

Abaixo, são apresentados alguns trechos de código utilizados para fins de entendimento do projeto dos operadores.

Código C.1 – Operação de orientação entre três pontos

```
short
TeCCW(const TeCoord2D& c1, const TeCoord2D& c2, const TeCoord2D& c3)
{
      double dx1 = c2.x() - c1.x();
      double dx2 = c3.x() - c1.x();
```

```
        double dy1 = c2.y() - c1.y();
        double dy2 = c3.y() - c1.y();

        double dx1y2 = dx1 * dy2;
        double dy1x2 = dy1 * dx2;

        // slope of the second line is greater than the first,
        // so counterclockwise.
        if(TeGeometryAlgorithmsPrecision<>::IsGreater(dx1y2, dy1x2))
              return TeCOUNTERCLOCKWISE;

        // slope of the first line is greater than the second,
        // so clockwise.
        if(TeGeometryAlgorithmsPrecision<>::IsSmaller(dx1y2, dy1x2))
              return TeCLOCKWISE;

        return TeNOTURN;
}
```

Código C.2 – Operação de verificação de convexidade

```
bool TeIsConvex(const TeLinearRing& ring)
{
        short orientation = TeCCW(*(ring.end() - 2), ring[0], ring[1]);

        const unsigned int nStep = ring.size() - 1;

        for(unsigned int i = 1; i <  nStep; ++i)
        {
              short this_orientation = TeCCW(ring[i-1], ring[i],
                                             ring[i+1]);

              if(this_orientation == TeNOTURN)
                    continue;

              if((orientation != TeNOTURN) &&
                 (orientation != this_orientation))
                    return false;

              orientation = this_orientation;
        }

        return true;
}
```

Código C.3 – Cálculo de área em polígonos

```
double Te2Area(const TeLinearRing& r)
{
        double S2 = 0.0;

        TeLinearRing::iterator it = r.begin();
        TeLinearRing::iterator end = r.end() - 1;

        while(it != end)
```

```
	{
		S2 += (it->x() + (it + 1)->x()) *
			((it + 1)->y() - it->y());
		++it;
	}

	return S2;
}

template<> double TeGeometryArea(const TePolygon& p)
{
	TePolygon::iterator it = p.begin();

	double area = fabs(Te2Area(*it));

	// subtract inner rings area.
	while((++it) != p.end())
		area -= fabs(Te2Area(*it));

	return (area / 2.0);
}

template<> double TeGeometryArea(const TePolygonSet& ps)
{
	TePolygonSet::iterator it = ps.begin();

	double area = 0.0;

	while(it != ps.end())
	{
		area += TeGeometryArea(*it);
		++it;
	}

	return (area / 2.0);
}

template<> double TeGeometryArea(const TeBox& b)
{
	return ((b.x2() - b.x1()) * (b.y2() - b.y1()));
}

template<class T> double TeGeometryArea(const T& geom)
{
	return 0.0;
}
```

Código C.4 – Orientação de um polígono simples

```
short TeOrientation(const TeLinearRing& r)
{
	double area = Te2Area(r);

	if(area > 0.0)
		return TeCOUNTERCLOCKWISE;
```

```
	if(area < 0.0)
		return TeCLOCKWISE;

	return TeNOTURN;  // Rise an exception in future
}
```

C.5 – Determinação do envoltório convexo

```
// Returns a linear ring that is the convex hull
// of a given list of coords
TePolygon ConvexHull(vector<TeCoord2D>& coordSet)
{
	// sorting the coords
	sort(coordSet.begin(), coordSet.end(), xOrder<TeCoord2D>());

	register unsigned int i = 0;
	register unsigned int n = coordSet.size();

	TeLine2D upperHull;
	TeLine2D lowerHull;

	lowerHull.add(coordSet[0]);
	lowerHull.add(coordSet[1]);

	unsigned int count = 2;

	for(i = 2; i < n; ++i)
	{
		lowerHull.add(coordSet[i]);

		++count;

		while(count > 2 &&
			TeCCW(lowerHull[count - 3], lowerHull[count - 2],
				lowerHull[count - 1]) <= TeNOTURN)
		{
			lowerHull.erase(count - 2);
			--count;
		}
	}

	upperHull.add(coordSet[n - 1]);
	upperHull.add(coordSet[n - 2]);

	count = 2;

	for(i = n - 2;  i > 0;  --i)
	{
		upperHull.add(coordSet[i - 1]);
		++count;

		while(count > 2 &&
			TeCCW(upperHull[count - 3], upperHull[count - 2],
			upperHull[count - 1]) <= TeNOTURN)
```

```
            {
                  upperHull.erase(count - 2);
                  --count;
            }
      }

      upperHull.erase(0);
      upperHull.erase(count - 1);

      lowerHull.copyElements(upperHull);
      lowerHull.add(lowerHull[0]);

      TeLinearRing aux_ring(lowerHull);
      TePolygon p;
      p.add(aux_ring);
      return p;
}

template<class T> inline TePolygon TeConvexHull(const T& coordSet)
{
      // creates an auxiliary line with the points of the ring
      vector<TeCoord2D> aux;

      T::iterator it = coordSet.begin();

      while(it != coordSet.end())
      {
            aux.add(*it);
            ++it;
      }

      // removes duplicated coords from structs like ring
      removeDuplicatedCoords(aux);

      return ConvexHull(aux);
}

template<> inline TePolygon TeConvexHull(const TePolygon& p)
{
      vector<TeCoord2D> coords;
      back_insert_iterator<vector<TeCoord2D> > it(coords);
      // Copy the first ring of each polygon without the
      // last point (that is equals to the first).

      copy(p[0].begin(), p[0].end() - 1, it);

      return ConvexHull(coords);
}

template<> inline TePolygon TeConvexHull(const TePolygonSet& ps)
{
      vector<TeCoord2D> coords;
      back_insert_iterator<vector<TeCoord2D> > it(coords);
      // Copy the first ring of each polygon without the
      // last point (that is equals to the first).
```

```
      TePolygonSet::iterator it_ps = ps.begin();
      while(it_ps != ps.end())
      {
            TeLinearRing r = (*it_ps)[0];
            copy(r.begin(), r.end() - 1, it);
            ++it_ps;
      }

      return ConvexHull(coords);
}

template<> inline TePolygon TeConvexHull(const TePointSet& ps)
{
      vector<TeCoord2D> coords;
      // Copy the first ring of each polygon without the
      // last point (that is equals to the first).

      TePointSet::iterator itr = ps.begin();

      while(itr != ps.end())
      {
            coords.push_back(itr->location());

            ++itr;
      }

      return ConvexHull(coords);
}

//! If we have the  two end point equals, so we remove it.
inline void removeDuplicatedCoords(vector<TeCoord2D>& coordSet)
{
      if(coordSet[0] == coordSet[coordSet.size() - 1])
            coordSet.erase(coordSet.end() - 1);

      return;

}
```

C.6 – Teste de ponto em polígono

```
bool TePointInPoly(const TeCoord2D& c, const TeLinearRing& r)
{
      if(r.size() < 4)
            return false;

      register double   ty, tx;

      register const unsigned int nVertices = r.size();

      register bool inside_flag = false;

      register int j, yflag0, yflag1;

      TeLinearRing::iterator vtx0, vtx1;
```

```
        tx = c.x();
        ty = c.y();

        vtx0 = r.end() - 2;

        yflag0 = (vtx0->y() >= ty);

        vtx1 = r.begin();

        for(j = nVertices; --j; )
        {
              yflag1 = (vtx1->y() >= ty);

              if(yflag0 != yflag1)
                    if((
                       (vtx1->y() - ty) * (vtx0->x() - vtx1->x()) >=
                       (vtx1->x() - tx) * (vtx0->y() - vtx1->y())
                       ) == yflag1)
                          inside_flag = !inside_flag ;

              yflag0 = yflag1 ;
              vtx0 = vtx1 ;
              vtx1++;
        }

        return inside_flag;

}
```

C.7 – Teste de ponto sobre a fronteira

```
bool TeIsOnBorder(const TeCoord2D& c, const TeLine2D& l)
{
      if(l.size() < 2)
            return false;

      if(TeDisjoint(c, l.box()))
            return false;

      TeLine2D::iterator it = l.begin();

      unsigned int nstep = l.size() - 1;

      for(unsigned int i = 0; i < nstep; ++i)
      {
            if(TeIsOnSegment(c, *(it), *(it + 1)))
                  return true;

            ++it;
      }

      return false;
}
```

C.8 – Interseção entre dois segmentos de reta

```
bool
TeIntersection(const TeCoord2D& a, const TeCoord2D& b,
               const TeCoord2D& c, const TeCoord2D& d,
               TeIntersCoordsVec& coords,
               TeSegmentIntersectionType& intersectionType)
{
 if(TeBoxIntersects(a, b, c, d))
 {
  if((TeEquals(a, c) || TeEquals(a, d)) &&
     (TeEquals(b, c) || TeEquals(b, d)))
  {
   intersectionType = TeImproperIntersection;

   coords.clear();

   coords.push_back(a);
   coords.push_back(b);

   return true;
  }

  bool between1 = false;
  short sign1 = TeCCW(a, b, c, between1);

  bool between2 = false;
  short sign2 = TeCCW(a, b, d, between2);

  bool between3 = false;
  short sign3 = TeCCW(c, d, a, between3);

  bool between4 = false;
  short sign4 = TeCCW(c, d, b, between4);

  // if there is an intersection
  if((sign1 * sign2) <= 0 && (sign3 * sign4 <= 0))
  {
    coords.clear();

    intersectionType = TeProperIntersection;

    if(between1)
    {
      intersectionType = TeImproperIntersection;
      coords.push_back(c);
    }

    if(between2)
    {
      intersectionType = TeImproperIntersection;
      coords.push_back(d);
    }

    if(between3 && !TeEquals(a, c) && !TeEquals(a, d))
    {
```

```
      intersectionType = TeImproperIntersection;
      coords.push_back(a);

      return true;
    }

    if(between4 && !TeEquals(b, c) && !TeEquals(b, d))
    {
      intersectionType = TeImproperIntersection;
      coords.push_back(b);

      return true;
    }

    if(intersectionType == TeImproperIntersection)
      return true;

    double denominator = (d.y() - c.y()) * (b.x() -  a.x()) -
                         (d.x() - c.x()) * (b.y() - a.y());


    //if(denominator == 0.0)  // parallel can not occur here any more!
    //  return false;         // I expect this is true!

    // parameters
    double Ua = ((d.x() - c.x()) * (a.y() - c.y()) -
                 (d.y() - c.y()) * (a.x() - c.x())) / denominator;
    double Ub = ((b.x() - a.x()) * (a.y() - c.y()) -
                 (b.y() - a.y()) * (a.x() - c.x())) / denominator;

    if(Ua > 0.0 && Ua < 1.0  && Ub > 0.0 && Ub < 1.0)
    {
      coords.push_back(TeCoord2D(a.x() + Ua * (b.x() - a.x()),
                                 a.y() + Ua * (b.y() - a.y())));
      return true;
    }

  }
 }

 intersectionType = TeImproperIntersection;

 return false;

}
```

C.9 – Interseção entre linhas poligonais utilizando o algoritmo Força Bruta

```
bool
TeIntersections(const TeLine2D& redLine, const TeLine2D& blueLine,
                TeReportVector& report, const unsigned int& redObjId,
                const unsigned int& blueObjId, const bool& sortReport)
{
      if(redLine.size() < 2 || blueLine.size() < 2)
            return false;
```

```
if(TeDisjoint(redLine.box(), blueLine.box()))
      return false;

register unsigned int i = 0;
register unsigned int j = 0;

TeBox interBox;

// Creates a intersection box from lines boxes.
TeIntersection(redLine.box(), blueLine.box(), interBox);

unsigned int nstep_redLine  = redLine.size() - 1;
unsigned int nstep_blueLine = blueLine.size() - 1;

TeIntersCoordsVec coords;

TeSegmentIntersectionType t = TeImproperIntersection;

for(i = 0; i < nstep_redLine; ++i)
{
  // Creates the segment box.
  TeBox red_box = makeBox(redLine[i].x(), redLine[i].y(),
                          redLine[i+1].x(), redLine[i+1].y());

  // See if red segment box intersects with the
  // intersection box.
  if(TeDisjoint(interBox, red_box))
      continue;   // If it doesn't intersect, go to another
                  // segment => skip comparasion beteween other
                  // "m" segments.

  TeSegment redSeg(redLine[i], redLine[i+1], i, redObjId);

  for(j = 0; j < nstep_blueLine; ++j)
  {
      // Check intersection.
      if(TeIntersection(redLine[i], redLine[i+1], blueLine[j],
                        blueLine[j+1], coords, t))
      {

        TeSegment blueSeg(blueLine[j], blueLine[j+1], j,
                          blueObjId);

        // Há pelo menos um ponto de interseção.
        TeIntersectionPoint ip1(redSeg, blueSeg, coords[0]);
        report.push_back(ip1);



        // Se houver outro.
        if(coords.size() == 2)
        {
            TeIntersectionPoint ip2(redSeg, blueSeg, coords[1]);
            report.push_back(ip2);
```

```
                }
            }
        }
    }

    if(sortReport)
        sort(report.begin(), report.end());

    return !report.empty();
}
```

C.10 – Classes auxiliares para os algoritmos baseados no plane sweep

A Figura C.1 ilustra o diagrama UML das classes auxiliares projetadas para dar suporte à implementação dos algoritmos de interseção. A classe TeSegment representa um segmento de reta que compõe a fronteira de um linha poligonal. Objetos dessa classe são representados por duas coordenadas extremas (TeCoord2D), sendo a primeira, a coordenada mais à esquerda e a segunda, a mais à direita. Essa ordem é utilizada nos algoritmos baseados no *plane sweep* que possuem uma linha de varredura vertical movendo-se da esquerda para a direita.

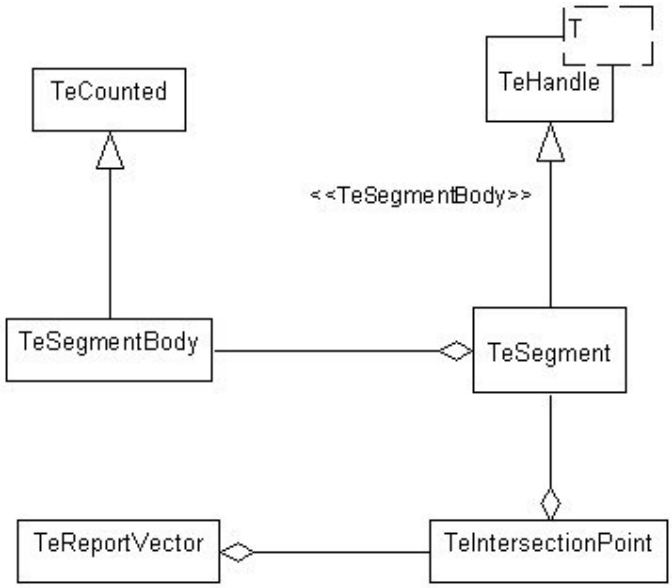


FIGURA C.1 – Diagrama de classes de suporte aos algoritmos de interseção.

A classe TeHandle é utilizada em conjunto com a classe TeCounted[11] para possibilitar o projeto de classes que utilizem o padrão *Bridge* (Gamma et al, 2000). O segmento é um exemplo que utiliza esse padrão, justificado pelo compartilhamento simultâneo das informações contidas em um segmento por várias estruturas de dados.

Os pontos de interseções são representados pela classe TeIntersectionPoint. Um ponto de interseção é formado pela coordenada de interseção e pelos dois segmentos que se interceptam no ponto. O conjunto dos pontos de interseção reportados pelos algoritmos são armazenados em um vetor (STL), representados pela classe TeReportVector. A organização dos pontos de interseção em um vetor dessa classe se dá pelo número do segmento em relação à linha poligonal e pela ocorrência do ponto de interseção no segmento. Essa ordem foi adotada a fim de facilitar o processamento do operador de fragmentação[12].

Os algoritmos baseados no *plane sweep* operam sobre duas estruturas de dados, uma para manipular os eventos e a outra para manter a relação de ordem. Os eventos foram modelados no diagrama da Figura C.2 pelas classes TeEventBody e TeEvent. Um evento é formado por uma coordenada (por onde a *sweep line* deslizará) e por dois vetores contendo os segmentos que deram origem a esse evento. Segmentos cujo ponto extremo esquerdo corresponda à coordenada do evento são armazenados em um vetor (beginEvents_) e os segmentos cujo ponto extremo direito corresponda à coordenada do evento são armazenados em outro (endEvents_). A lista de eventos (Q) é representada pela classe TeEventVector. Os eventos dentro desse vetor são ordenados de acordo com a ordem lexicográfica “xy”.

[11] Esta classe já pertence à TerraLib, e encontra-se em TeCounted.h
[12] Esse operador é explicado na seção que trata dos operadores topológicos

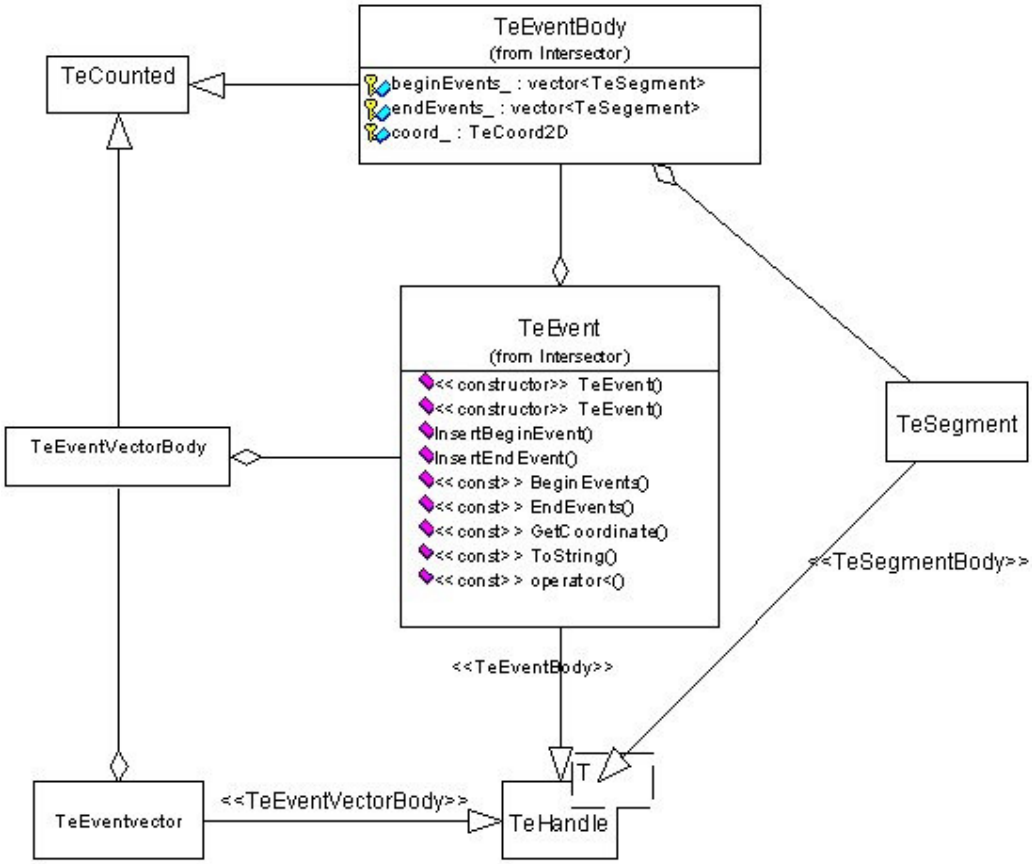


FIGURA C.2 – Diagrama das classes de eventos dos algoritmos *plane sweep*.

A estrutura de dados que permite manter a relação de ordem (T) dos segmentos na *sweep line* é modelada a partir de um *framework* de uma árvore Red-Black (Figura C.3).

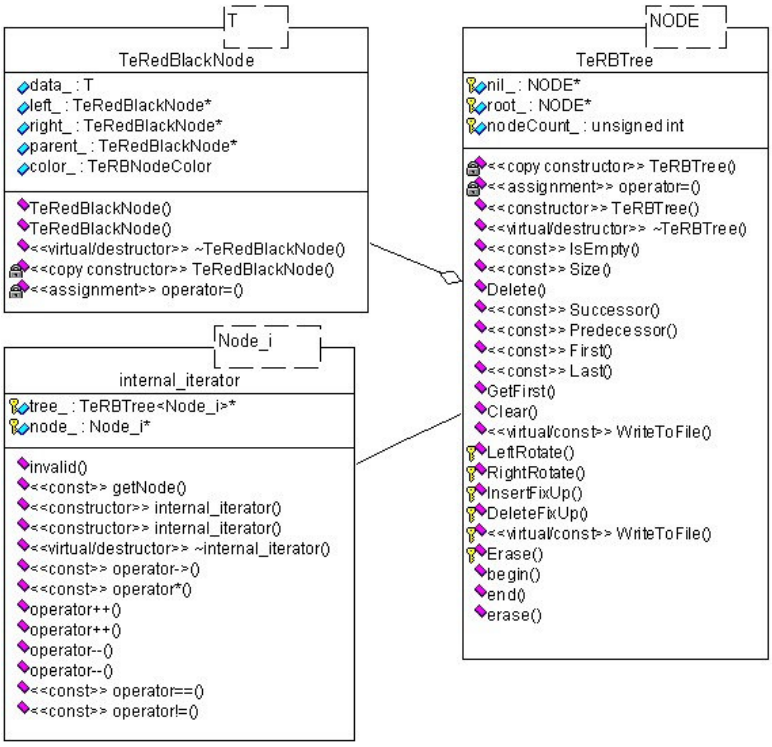


FIGURA C.3 – *Framework* da árvore Red-Black.

A implementação dessa árvore é baseada em (Cormen et al, 1990), com a diferença de que quando um nó sendo eliminado possui dois descendentes, o nó sucessor é religado em seu lugar ao invés de simplesmente ser copiado para ele. Dessa forma, os únicos iteradores (da classe internal_iterator) que são invalidados são os que se referem ao nó sendo removido.

A estrutura de dados que permite manter a relação de ordem (T) é modelada pela classe TeStatusTreeBO (Figura C.4). Esta classe possui métodos para inserir e remover um segmento da ordem, e para recuperar os segmentos imediatamente acima e abaixo de um dado segmento. Um nó (classe TeStatusNodeDataBO) pode conter até dois segmentos. Ele foi projetado dessa maneira para acomodar as variações dos algoritmos de detecção de interseção.

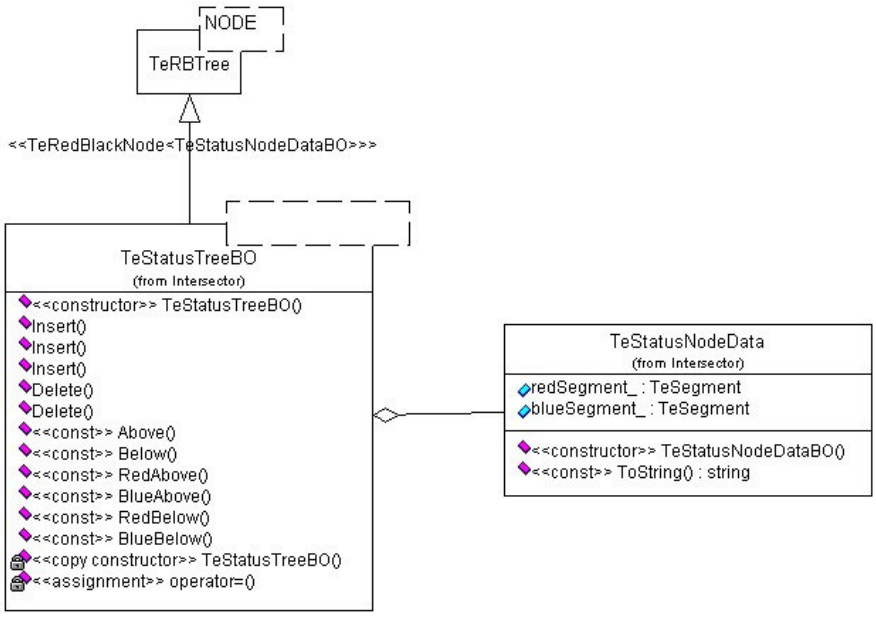


FIGURA C.4 – Diagrama das classes para manutenção da relação de ordem dos algoritmos baseados no *plane sweep*.

Maiores detalhes sobre as classes apresentadas nesta seção podem ser vistos no arquivo TeIntersector.h, TeIntersector.cpp e TeRedBlackTree.h contidos no *kernel* da TerraLib.

C.11 – Semântica dos operadores topológicos da TerraLib

A Figura C.5 ilustra a definição de interior e fronteira dos objetos considerada no projeto dos operadores topológicos. Objetos do tipo ponto possuem interior e exterior, sendo a fronteira vazia. Linhas abertas possuem as fronteiras representadas pelos dois pontos extremos e o interior representado pelos demais pontos pertencentes à linha. Linhas fechadas não possuem fronteiras. Objetos do tipo área possuem a fronteira representada pelos pontos que formam a linha de fronteira.

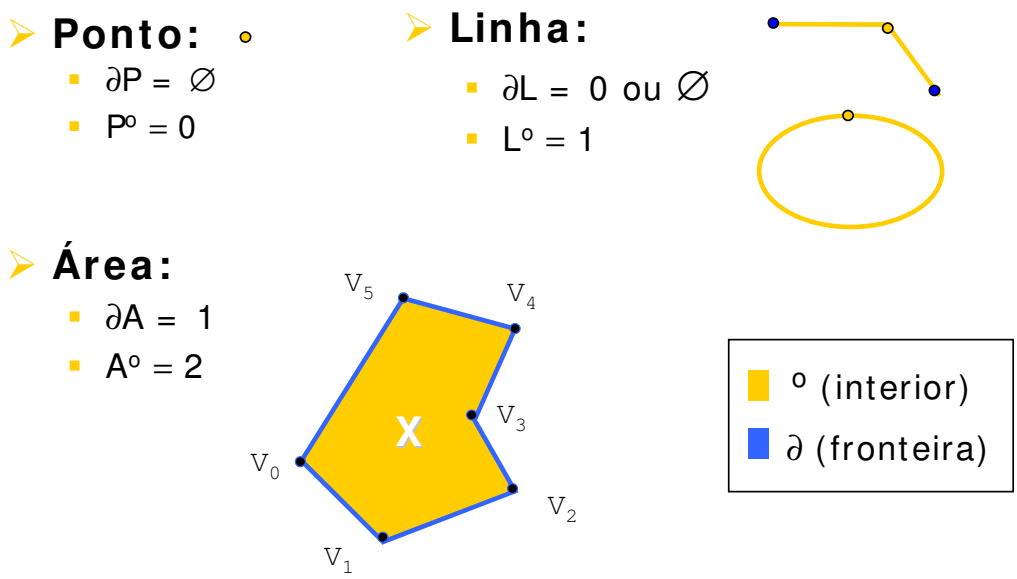


FIGURA C.5 – Interior e Fronteira das Geometrias.

Os operadores topológicos são definidos a partir das seguintes expressões:

- **TeDisjoint(λ_1, λ_2):** $\lambda_1 \cap \lambda_2 = \emptyset$

  ou seja, $(\lambda_1^o \cap \lambda_2^o = \emptyset) \wedge (\partial\lambda_1 \cap \lambda_2^o = \emptyset) \wedge (\lambda_1^o \cap \partial\lambda_2 = \emptyset) \wedge (\partial\lambda_1 \cap \partial\lambda_2 = \emptyset)$

- **TeTouches(λ_1, λ_2):** $(\lambda_1^o \cap \lambda_2^o = \emptyset) \wedge (\lambda_1 \cap \lambda_2 \neq \emptyset)$

  ou seja, $(\lambda_1^o \cap \lambda_2^o = \emptyset) \wedge ((\partial\lambda_1 \cap \lambda_2^o \neq \emptyset) \vee (\lambda_1^o \cap \partial\lambda_2 \neq \emptyset) \vee (\partial\lambda_1 \cap \partial\lambda_2 \neq \emptyset))$

- **TeWithin(λ_1, λ_2):** $(\lambda_1 \cap \lambda_2 = \lambda_1) \wedge (\lambda_1^o \cap \lambda_2^o \neq \emptyset)$

  - Caso de pontos e outras geometrias:

    $(\lambda_1^o \cap \lambda_2^o \neq \emptyset)$

  - Caso de áreas/áreas ou linhas/linhas:

    $(\lambda_1^o \cap \lambda_2^o \neq \emptyset) \wedge (\lambda_1^o \cap \lambda_2^- = \emptyset) \wedge (\partial\lambda_1 \cap \lambda_2^- = \emptyset) \wedge (\partial\lambda_1 \cap \partial\lambda_2 = \emptyset)$

  - Caso de linhas(λ_1) com áreas (λ_2):

$(\lambda_1^o \cap \lambda_2^o \neq \emptyset) \wedge (\lambda_1^o \cap \lambda_2^- = \emptyset) \wedge (\partial\lambda_1 \cap \lambda_2^- = \emptyset) \wedge (\partial\lambda_1 \cap \partial\lambda_2 = \emptyset) \wedge$
$(\lambda_1^o \cap \partial\lambda_2 = \emptyset)$

- **TeCoveredBy(λ_1, λ_2):** $(\lambda_1 \cap \lambda_2 = \lambda_1) \wedge (\lambda_1^o \cap \lambda_2^o \neq \emptyset)$

  - Caso de áreas/áreas ou linhas/linhas:

  $(\lambda_1^o \cap \lambda_2^o \neq \emptyset) \wedge (\lambda_1^o \cap \lambda_2^- = \emptyset) \wedge (\partial\lambda_1 \cap \lambda_2^- = \emptyset) \wedge (\partial\lambda_1 \cap \partial\lambda_2 \neq \emptyset)$

  - Caso de linhas(λ_1) com áreas (λ_2):

  $(\lambda_1^o \cap \lambda_2^o \neq \emptyset) \wedge (\lambda_1^o \cap \lambda_2^- = \emptyset) \wedge (\partial\lambda_1 \cap \lambda_2^- = \emptyset) \wedge (\partial\lambda_1 \cap \partial\lambda_2 = \emptyset) \wedge$
  $((\partial\lambda_1 \cap \partial\lambda_2 \neq \emptyset) \vee (\lambda_1^o \cap \partial\lambda_2 \neq \emptyset))$

- **TeCrosses(λ_1, λ_2):**

  $\dim(\lambda_1^o \cap \lambda_2^o) = (\max(\dim(\lambda_1^o), \dim(\lambda_2^o)) - 1) \wedge (\lambda_1 \cap \lambda_2 \neq \lambda_1) \wedge (\lambda_1 \cap \lambda_2 \neq \lambda_2)$

  - Caso de linhas(λ_1) com áreas (λ_2):

  $(\lambda_1^o \cap \lambda_2^o \neq \emptyset) \wedge (\lambda_1^o \cap \lambda_2^- \neq \emptyset)$

  - Caso de linhas/linhas:

  $\dim(L_1^o \cap L_2^o) = 0$

- **TeOverlaps(λ_1, λ_2):**

  $(\dim(\lambda_1^o) = \dim(\lambda_2^o) = \dim(\lambda_1^o \cap \lambda_2^o)) \wedge (\lambda_1 \cap \lambda_2 \neq \lambda_1) \wedge (\lambda_1 \cap \lambda_2 \neq \lambda_2)$

  - Caso de áreas/áreas:

  $(A_1^o \cap A_2^o \neq \emptyset) \wedge (A_1^o \cap A_2^- \neq \emptyset) \wedge (A_1^- \cap A_2^o \neq \emptyset)$

  - Caso de linhas/linhas:

$$(\dim(L_1^o \cap L_2^o) = 1) \wedge (L_1^o \cap L_2^- \neq \emptyset) \wedge (L_1^- \cap L_2^o \neq \emptyset)$$

- **TeIntersects(λ_1, λ_2):** esse operador corresponde à negação do operador TeDisjoint.

- **TeContains(λ_1, λ_2):** TeWithin(.λ_2, λ_1)

- **TeCovers(λ_1, λ_2):** TeCoveredBy(.λ_2, λ_1)

- **TeEquals(λ_1, λ_2):**. $(\lambda_1 \cap \lambda_2 = \lambda_1) \wedge (\lambda_1 \cap \lambda_2 = \lambda_2)$

  ou seja, $(\partial\lambda_1 \cap \lambda_2^o = \emptyset) \wedge (\partial\lambda_1 \cap \lambda_2^- = \emptyset)$

- **TeRelation(λ_1, λ_2):** retorna o relacionamento topológico entre os objetos.

C.12 – Operações que envolvem o uso de tolerâncias

```
//! This class implements the Epsilon-tweaking used in calculus.
template<class T = double> class TeGeometryAlgorithmsPrecision
{
 protected:

      //! Keeps the old value
      double oldValue_;

      //! Current tolerance factor.
      static double positiveZeroReference_;

      static double negativeZeroReference_;

 public:

      //! Constructor
      TeGeometryAlgorithmsPrecision(const double& ref)
      {
            oldValue_ = positiveZeroReference_;
            positiveZeroReference_ = ref;
            negativeZeroReference_ = -ref;
      }

      //! Destructor
      ~TeGeometryAlgorithmsPrecision()
      {
            positiveZeroReference_ = oldValue_;
            negativeZeroReference_ = -oldValue_;
      }
```

```
        //! Tells if d1 is greater than d2 according to
        //  tolerance factor.
        static bool IsGreater(const double& d1, const double& d2)
        {
              return ((d1 - d2) > positiveZeroReference_);
        }

        //! Tells if d1 is greater than or equal to d2 according to
        //  tolerance factor.
        static bool IsGreaterEqual(const double& d1, const double& d2)
        {
              return ((d1 - d2) >= negativeZeroReference_);
        }

        //! Tells if d1 is smaller than d2 according to
        //  a tolerance factor.
        static bool IsSmaller(const double& d1, const double& d2)
        {
              return ((d1 - d2) < negativeZeroReference_);
        }

        //! Tells if d1 is smaller than or equals to d2 according to
        //  a tolerance factor.
        static bool IsSmallerEqual(const double& d1, const double& d2)
        {
              return ((d1 - d2) <= positiveZeroReference_);
        }

        //! Tells if d1 is equals to d2 according to a tolerance factor.
        static bool IsEqual(const double& d1, const double& d2)
        {
              return (fabs(d1 - d2) <= positiveZeroReference_);
        }

        //! Tells if d1 is different from d2 according to
        //  a tolerance factor.
        static bool IsDifferent(const double& d1, const double& d2)
        {
              return (fabs(d1 - d2) > positiveZeroReference_);
        }
};

template<class T> double
      TeGeometryAlgorithmsPrecision<T>::positiveZeroReference_ = 0.0;
template<class T> double
      TeGeometryAlgorithmsPrecision<T>::negativeZeroReference_ = 0.0;
```