MINISTÉRIO DA CIÊNCIA E TECNOLOGIA
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

# INTERFACE PARA OPERAÇÕES ESPACIAIS EM BANCO DE DADOS GEOGRÁFICOS

Karine Reis Ferreira

Dissertação de Mestrado em Computação Aplicada, orientada pelo Dr. João Argemiro Carvalho Paiva E Dr. Gilberto Câmara

INPE
São José dos Campos
2003

Aprovada pela Banca Examinadora em cumprimento a requisito exigido para a obtenção do Título de **Mestre** em **Computação Aplicada.**

Dr. Antônio Miguel Vieira Monteiro
Presidente/INPE, SJCampos-SP

Dr. João Argemiro de Carvalho Paiva
Orientador/INPE, SJCampos-SP

Dr. Gilberto Câmara Neto
Membro da Banca/INPE, SJCampos-SP

Dr. Jugurta Lisboa Filho
Membro da Banca
Convidado UFV, Viçosa-MG

Drª Ana Carolina Brandão Salgado
Membro da Banca
Convidada UFPE, Recife-PE

Candidata: Karine Reis Ferreira

São José dos Campos, 16 de junho de 2003.

*"Cada um de nós compõe a sua história,*
*e cada ser em si, carrega o dom de ser capaz, e ser feliz"*

Renato Teixeira

*A meus pais,*
*Élcio Álves Ferreira e*
*Terezinha Reis Ferreira.*

## AGRADECIMENTOS

Chegou a hora de agradecer àquelas pessoas que, de alguma forma, contribuíram para a realização deste trabalho. Pessoas que me incentivaram de diferentes maneiras. A todos vocês o meu sincero "MUITO OBRIGADO"!

Ao INPE (Instituto Nacional de Pesquisas Espaciais) e FUNCATE (Fundação de Ciência, Aplicações e Tecnologia Espaciais), pela oportunidade de fazer esse mestrado.

Aos professores da Computação Aplicada (CAP) e da Divisão de Processamento de Imagens (DPI), pela arte de ensinar e atenção.

Ao orientador João Argemiro, pelo apoio, conhecimento passado e, principalmente, pela amizade durante esses meses.

Ao orientador Gilberto Câmara, pelo incentivo e exemplo.

Aos colegas da Divisão de Processamento de Imagens (DPI), que sempre me receberam de braços abertos. Em especial ao Antônio Miguel Monteiro, pela amizade e ajuda, principalmente, quando cheguei aqui. Estou muito feliz de trabalhar com vocês!

À equipe de desenvolvimento da TerraLib, especialmente a Lúbia Vinhas e Ricardo Cartaxo, pela atenção, ajuda e experiência passada.

Aos meus pais, Élcio e Terezinha, pelo amor e incentivo, sem os quais eu nunca teria chegado até aqui. Impossível expressar em palavras meu amor e gratidão por vocês!

Ao meu irmão Gustavo e toda família de Campo Belo, pelo carinho, preocupação e incentivo.

Ao Lú, meu namorado, pelo amor, apoio e compreensão nas horas de ausência. Você foi muito importante nessa etapa da minha vida!

À minha "família de São José", Elisa e Luciano, pelo carinho e incentivo.

Aos amigos Gilberto Ribeiro, Paulo Lima, Isabela, Mariza, Ana Paula, Caio e Alysson. Cada um de vocês contribuiu de uma forma especial para esse trabalho.

## RESUMO

Este trabalho apresenta uma interface de programação genérica, ou API (*Application Programming Interface*), para operações espaciais em banco de dados geográficos implementada no ambiente TerraLib, que consiste em uma biblioteca base para a construção de aplicativos geográficos de arquitetura integrada. Essa API fornece operações sobre dados geográficos armazenados em SGBDs relacionais (SGBDR) e objeto-relacionais (SGBDORs). No caso da nova geração de SGBDORs que possuem uma extensão espacial, como o Oracle Spatial, a API explora ao máximo seus recursos para tratar dados geográficos, como por exemplo, indexação espacial, operadores e funções para manipular e consultar esses dados através da linguagem de consulta SQL. As operações fornecidas pela API são agrupadas em: (1) operações sobre dados vetoriais, como por exemplo, consultas de relações topológicas e métricas, geração de uma nova geometria a partir de uma distância em torno de uma geometria específica (*buffer*) e operações de conjunto (interseção, união e diferença); e (2) operações sobre dados matriciais, como operações zonais e recorte a partir de uma máscara.

## ABSTRACT

This work presents a generic programming interface, or API (Application Programming Interface), for spatial operations in geographical database developed in the TerraLib environment - a base library for construction of geographical applications with integrated architecture. This API provides operations on geographical data stored in relational DBMS (RDBMS) and object-relational DBMS (ORDBMS). In the case of a new generation of ORDBMS which has a spatial extent, like Oracle Spatial, the API explores at most its resources to treat geographical data, for example, spatial indexes, operators and functions to manipulate and query these data through the query language SQL. The supplied operations of this API can be grouped as: (1) operations over vector data, for example, topological and metrical relation query, generation of a new geometry through a distance around an specific geometry (buffer), and set operations (intersection, union and difference); and (2) operations over raster data, as zonal operation and clipping based in a mask.

## SUMÁRIO

Pág.

## LISTA DE FIGURAS

## LISTA DE TABELAS

# CAPÍTULO 1

# INTRODUÇÃO

Um número cada vez mais crescente de sistemas de informação vem incluindo técnicas para tratamento computacional de dados geográficos. Estes sistemas, chamados de sistemas de informação geográfica (SIG), foram inicialmente desenvolvidos nas décadas de 80 e 90 como sistemas "stand-alone", sem a capacidade de compartilhar ou gerenciar dados de forma eficiente. Adicionalmente, os SIG foram desenvolvidos como ambientes monolíticos, constituídos de pacotes de uso genérico de centenas de funções, o que dificultava sobremaneira seu aprendizado e uso por não-especialistas.

Para resolver estas limitações, nos anos recentes os SIGs estão evoluindo de modo a satisfazer duas grandes classes de requisitos:

(a) Gerenciar os dados espaciais através de Sistemas Gerenciadores de Banco de Dados (SGBDs) (Rigaux et al, 2002). Com isso, os SIGs passam a utilizar os recursos oferecidos pelos SGBDs para controle e manipulação dos dados espaciais, como por exemplo, gerência de transações, controle de integridade e concorrência de acessos, ao invés de ter que implementá-los. A tecnologia de SIG baseada no uso de SGBD é chamada na literatura de "arquitetura integrada" (Câmara et al, 1996).

(b) Compor um ambiente modular e extensível que permita a transição dos atuais sistemas monolíticos, que contêm centenas de funcionalidades, para uma nova geração de aplicativos geográficos (*spatial information appliances*), que são sistemas dedicados para necessidades específicas (Egenhofer, 1999). Esta nova geração de SIGs deverá ser modular, extensível e capaz de suportar a incorporação de sistemas independentes e autônomos de maneira modular, adicionando ao aplicativo geográfico novas funcionalidades conforme os requisitos da aplicação (Voisard and Schweppe, 1997).

Os SIGs baseados na arquitetura integrada armazenam todos os tipos de dados geográfico, em suas componentes espacial e alfanumérica, em um SGBD usualmente baseado na tecnologia objeto-relacional (SGBDOR). Para atender à evolução dos SIGs, os SGBDs objeto-relacionais estão sendo estendidos para tratar tipos de dados geográficos, chamados de Sistemas de Banco de Dados Geográficos ou extensões espaciais. Essas extensões fornecem funcionalidades e procedimentos que permitem armazenar, acessar e analisar dados geográficos de formato vetorial (Güting, 1994). Existem hoje três extensões comerciais disponíveis: Oracle Spatial (Ravada e Sharma, 1999), IBM DB2 Spatial Extender (IBM, 2001) e Informix Spatial Datablade (IBM, 2002). Há ainda a extensão PostGIS (Ramsey, 2002) para o SGBD PostgreSQL, que é objeto-relacional, gratuito e de código fonte aberto.

Para a construção de aplicativos geográficos, é necessário dispor de um ambiente adaptável e modular, e uma das formas tradicionalmente utilizadas em Engenharia de Software para tal fim são as bibliotecas de funções. Como observa Stroustrup (1999), construir uma biblioteca de software eficiente é uma das tarefas mais difíceis da Computação, e das que mais contribuem para o avanço da tecnologia de informação.

Para atender ao duplo requisito de construir aplicativos geográficos modulares baseados no suporte de SGBD relacionais e objeto-relacionais, está sendo desenvolvida a TerraLib (Câmara et al, 2000), uma biblioteca de software livre base para a construção de uma nova geração de aplicativos geográficos baseados na arquitetura integrada. No entanto, o desenvolvimento de uma biblioteca eficiente e útil apresenta muitos desafios, alguns dos quais são discutidos e resolvidos neste trabalho, a saber:

- Como compatibilizar, numa única interface de programação (API), implementações de tipos de dados espaciais realizadas por diferentes fabricantes de SGBDOR?

- Como estender as facilidades disponíveis nos SGBDOR, para tipos de dados necessários para os aplicativos geográficos, mas não disponíveis nestes sistemas?

- Como oferecer uma interface de programação genérica que permita a realização de operações de consulta espacial sobre diferentes tipos de dados espaciais, de forma modular e eficiente para o programador de aplicações?

## 1.1 Objetivo

O presente trabalho tem como objetivo desenvolver uma interface de programação genérica, ou API (*Application Programming Interface*), para operações espaciais em banco de dados geográficos para a biblioteca TerraLib. Uma API é um conjunto de funções que facilitam a troca de mensagens ou dados entre dois ou mais sistemas distintos.

A API desenvolvida deve ser composta por funções que permitam realizar operações espaciais sobre dados geográficos armazenados em um SGBD. Essas operações podem ser realizadas sobre dados vetoriais (Rigaux et al, 2002), como consultas de relações topológicas e métricas, geração de uma nova geometria a partir de uma distância em torno de uma geometria específica (*buffer*) e operações de conjunto (interseção, união e diferença), ou sobre dados matriciais (Tomlin, 1990), como operações zonais e recorte a partir de uma máscara.

Outra característica importante da API é ser genérica no sentido de suportar diferentes SGBDs, relacionais e objeto-relacionais, que possuem interface com a TerraLib. No caso de SGBDORs que possuem extensões espaciais, como o Oracle Spatial, a API deve explorar ao máximo seus recursos para tratar dados geográficos, como indexação espacial e operadores e funções para manipular esses dados.

## 1.2 Metodologia

Para cumprir o objetivo do trabalho, a API foi implementada na TerraLib nas classes de interface com os SGBDs em dois níveis:

(1) Nível genérico: as operações foram implementadas na classe base da interface com os SGBDs, chamada `TeDatabase`, utilizando as estruturas de dados geográficos e as funções da TerraLib. Essas operações serão utilizadas por todos

os SGBDs que não possuem extensão espacial, como o Access, MySQL e PostgreSQL.

(2) Nível específico: as operações foram reimplementadas na classe de interface com o SGBD Oracle Spatial, chamada `TeOracleSpatial`, utilizando os recursos de sua extensão espacial. Essas operações são computadas pelo próprio Oracle Spatial através de operadores e funções espaciais que são usados juntamente com a linguagem SQL.

Como as extensões espaciais existentes não tratam dados geográficos de representação matricial, as operações sobre esses dados foram implementadas apenas no nível genérico, sendo utilizadas por todos os SGBDs, inclusive pelo Oracle Spatial.

Para testar a API, foram desenvolvidas duas interfaces gráficas para operações espaciais no TerraView (Ferreira et al, 2002), um aplicativo geográfico construído sobre a TerraLib. Essas interfaces têm como objetivo utilizar as operações implementadas na API e mostrar, graficamente, os resultados na área de desenho do TerraView, facilitando os testes.

### 1.3 Motivação e Contribuição

Operações espaciais são funcionalidades básicas que devem ser fornecidas por aplicativos geográficos para seleção, manipulação e análise de dados espaciais. A principal motivação deste trabalho é disponibilizar essas operações na TerraLib em um nível maior de abstração para os desenvolvedores desses aplicativos. Isso permitirá que o desenvolvedor utilize as operações da API sem precisar ter conhecimento de como elas são computadas, podendo assim, se dedicar à implementação de outras funcionalidades, como por exemplo, visualização dos dados, interface com o usuário e ferramentas para análise espacial.

Uma outra motivação é estar contribuindo para a implementação da TerraLib, a qual facilitará e dará um grande incentivo para o desenvolvimento de novos aplicativos geográficos de código fonte aberto e gratuito. E, além disso, explorar a nova geração de

sistemas gerenciadores de banco de dados objeto-relacionais que tratam dados geográficos.

### 1.4 Estrutura da Dissertação

Esta dissertação está dividida em 5 capítulos, sendo o primeiro uma introdução sobre o trabalho, trazendo seu objetivo, um resumo da metodologia utilizada para alcançar esse objetivo, as motivações e contribuições.

O segundo capítulo apresenta alguns conceitos sobre Sistemas de Banco de Dados Geográficos, incluindo a evolução de Sistemas de Informações Geográficas e as características das extensões espaciais. Faz ainda faz uma revisão sobre operações espaciais, sobre dados vetoriais e matriciais, e como essas operações são implementadas no Oracle Spatial. Estes capítulos fornecem uma base teórica para o desenvolvimento do trabalho.

O terceiro capítulo apresenta a API desenvolvida neste trabalho, dando uma introdução sobre a TerraLib e mostrando como as operações foram implementadas. O quarto capítulo mostra exemplos de utilização dessas operações, mostrando seus resultados através do TerraView. O quinto capítulo apresenta a conclusão do trabalho, mostrando propostas de trabalhos futuros.

O Anexo A traz os códigos em C++ das funções para geração de um arco a partir de três pontos.

O Anexo B mostra a implementação em C++ das estruturas de dados e funções utilizadas para o cálculo das estatísticas para a operação Zonal.

O Anexo C apresenta a implementação em C++ do iterador restrito a uma região do *raster* e suas estratégias de percurso, desenvolvido para as operações sobre dados matriciais.

# CAPÍTULO 2

# SISTEMAS DE BANCO DE DADOS GEOGRÁFICOS

## 2.1 Introdução

Dado espacial ou geográfico é um termo usado para representar fenômenos do mundo real através de duas componentes: (a) sua localização geográfica, ou seja, sua posição em um sistema de coordenadas conhecido; e (b) seus atributos descritivos, como por exemplo, cor, custo, pH, etc. A localização geográfica é representada por coordenadas em um sistema de coordenadas específico, onde uma coordenada é um número que representa uma posição relativa a um ponto de referência.

Um dado espacial pode ser representado por dois modelos de dados distintos: vetorial ou matricial (*raster*) (Burrough e Mcdonnell, 1998). O modelo de dados vetorial é utilizado para representar o espaço como um conjunto de entidades discretas (geo-objetos ou objetos geográficos) definidas por uma unidade (ponto, linha ou polígono) geograficamente referenciada e por seus atributos descritivos. Por exemplo, as bacias hidrográficas do Brasil podem ser representadas por objetos geográficos, onde cada bacia é representada por um polígono que define o seu limite, cada rio por uma linha e cada usina hidrelétrica por um ponto. Além da localização, cada objeto geográfico tem seus atributos descritivos, como a população de cada bacia, a potência gerada em cada usina, os nomes dos rios, etc.

O modelo de dados matricial é utilizado para representar o espaço geográfico como uma superfície contínua (geo-campo), sobre a qual variam os fenômenos a serem observados segundo diferentes distribuições (Goodchild, 1992). Este espaço é então representado por uma matriz P($m$,$n$), formada por $m$ linhas e $n$ colunas, onde cada célula possui a sua localização (um par de coordenadas que especifica seu endereço) e um valor correspondente ao atributo estudado. Como exemplo, podemos citar um mapa geoquímico, o qual associa o teor de um mineral a cada ponto.

Um conjunto de dados geográficos, um geo-campo ou um mapa de geo-objetos, que estão agrupados segundo algum contexto e que podem ser caracterizados por um mesmo conjunto de atributos, é chamado de plano de informação ou *layer*.

## 2.2 Sistemas de Informações Geográficas

Devido ao rápido desenvolvimento de novas tecnologias para coletar e digitalizar dados espaciais e ao grande aumento da demanda para processar e analisar esses dados, foram implementados sistemas dedicados chamados Sistemas de Informação Geográfica (SIGs). SIGs são sistemas que tratam computacionalmente dados espaciais através das funcionalidades (Rigaux et al, 2002):

- Entrada e validação de dados espaciais;
- Armazenamento e gerenciamento desses dados;
- Saída e apresentação visual desses dados;
- Transformação de dados espaciais;
- Interação com o usuário;
- Combinação de dados espaciais para criar novas representações do espaço geográfico;
- Ferramentas para análise espacial.

Ao longo dos anos, os SIGs foram implementados seguindo diferentes arquiteturas. Tais arquiteturas são distintas, principalmente, na maneira e nos recursos utilizados para armazenar e recuperar dados espaciais. Essas arquiteturas evoluíram de forma a deixar, cada vez mais, a responsabilidade de gerenciamento dos dados para os Sistemas Gerenciadores de Banco de Dados (SGBDs).

### 2.2.1 Sistemas Gerenciadores de Banco de Dados

Segundo Korth e Silberschatz (1994), um sistema de gerenciamento de banco de dados consiste em uma coleção de dados inter-relacionados, normalmente referenciada como banco de dados, e em um conjunto de programas para acessá-los. Em (Frank, 1998) são citados alguns recursos oferecidos pelos SGBDs:

- Recursos para aumentar a performance de armazenamento e recuperação de dados, utilizando técnicas oferecidas pelo sistema operacional, como agrupamento de dados (*clustering*) e gerenciamento de memória intermediária (*buffer*);
- Padronização de acesso aos dados e separação entre os dados armazenados e as funções de manipulação e recuperação destes dados;
- Interface entre banco de dados e programas de aplicação baseada na descrição lógica dos dados, tornando os detalhes sobre as estruturas físicas de armazenamento invisíveis para os programas de aplicação;
- Manutenção da consistência dos dados no banco através da utilização do conceito de transação;
- Gerenciamento do controle de concorrência, evitando perda de informações e mantendo a consistência dos dados;
- Mecanismos de recuperação de falhas capazes de detectar uma falha, como quedas de energia ou quebra de disco, e recuperar o banco de dados em seu estado consistente;
- Restrições de acesso a usuários não autorizados;
- Mecanismos para a definição de restrições de integridade.

Para manipular e acessar dados armazenados em bancos de dados relacionais, a IBM desenvolveu, na década de 70, uma linguagem chamada SQL (*Structured Query*

*Language)* (NIST, 1993). Em 1986, a SQL foi considerada pelo ANSI (*American National Standards Institute*) como a linguagem padrão para banco de dados relacionais, e desde então é utilizada pela maioria dos SGBDs relacionais comerciais. A SQL oferece recursos para definir estruturas de dados; consultar, inserir e modificar dados do banco de dados e especificar restrições de segurança. Considerando esses recursos, a linguagem SQL é formada por várias partes, dentre elas podem ser citadas:

- SQL DDL (*Data Definition Language*): Linguagem de definição de dados que fornece comandos para definição, modificação de esquemas e remoção de tabelas e criação de índices e de restrições de integridade.

- SQL DML (*Data Manipulation Language*): Linguagem de manipulação de dados que fornece comandos para busca de informações no banco de dados e inserção, remoção e modificação de dados do banco de dados.

### 2.2.2 Arquiteturas de SIGs

Há basicamente três diferentes arquiteturas de SIGs que utilizam os recursos de um SGBD: Dual, Integrada baseada em SGBDs relacionais e Integrada baseada em extensões espaciais sobre SGBDs objeto-relacionais (Câmara et al, 1996).

A arquitetura Dual, mostrada na Figura 2.1, armazena o dado espacial separadamente. A componente alfanumérica é armazenada em um SGBD relacional e a componente espacial é armazenada em arquivos proprietários. As principais desvantagens desta arquitetura são:

- Dificuldades no controle e manipulação dos dados espaciais;

- Dificuldade em manter a integridade entre a componente espacial e a componente alfanumérica;

- Consultas mais lentas, pois são processadas separadamente. A parte convencional da consulta é processada pelo SGBD separado da parte espacial, que é processada pelo aplicativo utilizando os arquivos proprietários;

- Falta de interoperabilidade entre os dados. Cada sistema produz seu próprio arquivo proprietário sem seguir um formato padrão, o que dificulta a integração destes dados.

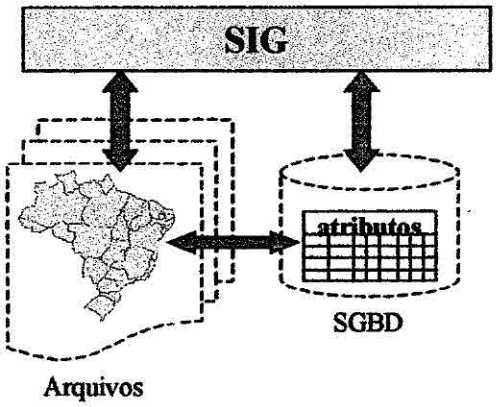



Figura 2.1- Arquitetura Dual

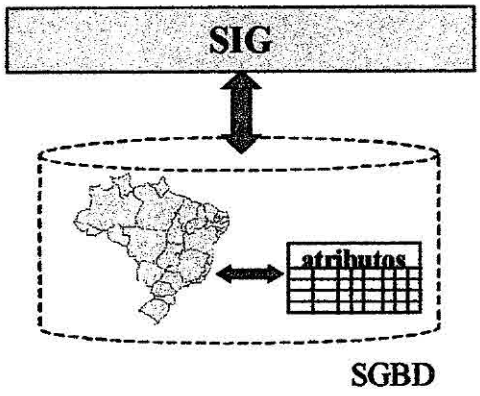



Figura 2.2 - Arquitetura Integrada

A arquitetura Integrada, mostrada na Figura 2.2, consiste em armazenar todo o dado espacial em um SGBD, sua componente espacial e alfanumérica. Sua principal vantagem é a utilização dos recursos de um SGBD para controle e manipulação de dados espaciais, como gerência de transações, controle de integridade e concorrência. Sendo assim, a manutenção de integridade entre a componente espacial e alfanumérica é feita pelo SGBD.

A arquitetura Integrada baseada em um SGBD relacional utiliza campos longos, chamados de BLOBs, para armazenar a componente espacial do dado. Suas principais desvantagens são:

- Não é capaz de capturar a semântica dos dados espaciais: como o SGBD trata o campo longo como uma cadeia binária, não é possível conhecer a semântica do seu conteúdo;

- Métodos de acesso espacial e otimizador de consultas devem ser implementados pelo SIG: como o SGBD trata os dados espaciais como uma cadeia binária, não possui mecanismos satisfatórios para o seu tratamento;

- Limitações da linguagem SQL para a manipulação dos dados espaciais: a SQL padrão oferece recursos limitados para o tratamento de campos longos.

O outro tipo de Arquitetura Integrada consiste em utilizar extensões espaciais desenvolvidas sobre SGBDs objeto-relacionais (SGBDOR). Estas extensões contêm funcionalidades e procedimentos que permitem armazenar, acessar e analisar dados espaciais de formato vetorial.

Como desvantagens dessa arquitetura podem ser citadas a falta de mecanismos de controle de integridade sobre os dados espaciais e a falta de padronização das extensões da linguagem SQL para tratar dados espaciais.

## 2.3 Extensões Espaciais

Os SGBDs objeto-relacionais, também chamados de SGBDs extensíveis, oferecem recursos para a definição de novos tipos de dados e de novos métodos ou operadores para manipular esses tipos, estendendo assim seu modelo de dados e sua linguagem de consulta. Por isso, um SGBDOR é mais adequado para tratar dados complexos, como dados geográficos, do que um SGBDR, o qual não oferece esses recursos.

Os tipos de dados definidos pelos usuários em um SGBD objeto-relacional são representados como objetos e são manipulados como qualquer outro tipo básico do SGBD. Para suportar essas características, foi necessário extender a linguagem SQL para criar, gerenciar e consultar objetos persistentes em um SGBD.

Nos últimos anos, o comitê de padronização da linguagem SQL vem adicionando novas especificações à SQL para suportar modelos de dados orientados a objetos. A atual versão da SQL, que está em processo de evolução para ser estendida, é chamada de SQL3 (Mattos, 1996).

Algumas características da SQL3 para suportar estruturas orientadas a objetos são:

- Suportar tipos de dados definidos pelo usuário (tipos abstratos de dados);
- Oferecer construtores para tipos de coleção, como conjuntos, listas e vetores;

- Suportar funções e procedimentos definidos pelo usuário;
- Suportar grandes objetos (BLOBs e CLOBs).

Um SGBDOR que possui uma extensão para tratar dados espaciais é chamado de Sistema de Banco de Dados Espaciais (Shekhar et al, 1999) e, segundo Güting (1994), deve:

- Fornecer tipos de dados espaciais (TDEs), como ponto, linha e região, em seu modelo de dados e manipulá-los assim como os tipos alfanuméricos básicos (inteiros, string, etc);
- Estender a linguagem de consulta SQL para suportar operações e consultas espaciais sobre TDEs;
- Adaptar outras funções de níveis mais internos para manipular TDEs eficientemente, tais como métodos de armazenamento e acesso (indexação espacial) e métodos de otimização de consultas (junção espacial).

Portanto, além dos TDEs, as extensões espaciais fornecem operadores e funções que são utilizados, juntamente com a linguagem de consulta do SGBD, para consultar relações espaciais e executar operações sobre TDEs. Além disso, fornecem métodos de acesso eficiente de TDEs através de estruturas de indexação, como R-tree (Guttman, 1984) e QuadTree (Samet, 1984).

Atualmente, essas extensões são limitadas em tratar somente dados espaciais de representação vetorial, não fornecendo suporte a dados matriciais. Portanto, um Sistema de Banco de Dados Espaciais não oferece tipos de dados específicos para armazenar dados matriciais e nem recursos para executar operações espaciais sobre esses dados. Geralmente os dados matriciais são armazenados em campos longos (BLOBs) como um conjunto de valores binários.

Todas as extensões existentes, como Oracle Spatial, DB2 Spatial Extender, Informix Spatial Datablade e PostGIS, baseiam-se nas especificações do OpenGIS (OGC, 1996).

Porém, possuem variações relevantes entre os modelos de dados, semântica dos operadores espaciais e mecanismos de indexação. O OpenGIS é uma associação formada por organizações públicas e privadas envolvidas com SIGs, dedicada à criação e gerenciamento de uma arquitetura padrão para geoprocessamento. Seu objetivo técnico é definir e manter:

- Um modelo universal de dados espaço-temporais e de processos, chamado modelo de dados geográficos OpenGIS;
- Uma especificação para cada uma das principais linguagens de consulta a banco de dados para implementar o modelo de dados OpenGIS;
- Uma especificação para cada um dos principais ambientes computacionais distribuídos para implementar o modelo de processo OpenGIS.

O OpenGIS define uma extensão da linguagem SQL, chamada SQL92 com tipos geométricos, para armazenar, recuperar, consultar e modificar dados espaciais em um SGBD. Nessa definição, o OpenGIS especifica um conjunto de tipos de dados geográficos, baseado em seu modelo de dados, e funções para manipular esses tipos através de SQL (OGC, 1999).

## 2.4 Operações Espaciais em Bancos de Dados Geográficos

### 2.4.1 Dados vetoriais

Engenhofer (1994) dividiu as operações espaciais sobre dados vetoriais em três classes: (1) operações unárias, que acessam uma propriedade espacial de uma geometria; (2) operações binárias, que calculam um valor entre duas geometrias; e (3) relações binárias, que determinam a relação espacial entre duas geometrias. Rigaux et al (2002) subdividiu essas três classes em sete classes com características mais específicas:

(1) Operações unárias com resultado booleano: testam um objeto geográfico segundo uma propriedade específica, como por exemplo, se é convexo (*Convex*) ou se está conectado (*Connected*).

(2) Operações unárias com resultado escalar: computam o comprimento ou perímetro (*Length*) e área (*Area*) de um objeto geográfico.

(3) Operações unárias com resultado espacial: geram uma nova geometria como resultado, como por exemplo, uma nova geometria a partir de uma distância em torno de uma geometria específica (*Buffer*), uma geometria convexa a partir do objeto geográfico (*ConvexHull*), operações que retornam o mínimo retângulo envolvente (*MBR*) ou o centróide (*Centroid*) do objeto geográfico.

(4) Operações *n*-árias com resultado espacial: recebem como parâmetro um conjunto de objetos geográficos e retornam uma única geometria como resultado. A operação *ConvexHull* pode pertencer a essa classe quando receber mais de um objeto geográfico como parâmetro de entrada.

(5) Operações binárias com resultado espacial: essa classe é formada basicamente pelas operações de conjunto, como, interseção (*Intersection*), União (*Union*) e diferença (*Difference*).

(6) Operações binárias com resultado booleano: também chamadas de predicados ou relações espaciais, podem ser divididas em:

   a. relações topológicas: relações invariantes a transformações topológicas como translação, rotação e mudança de escala, por exemplo, contém (*Contain*), disjunto (*Disjoint*), intercepta (*Intersects*), cruza (*Cross*), dentre outras;

   b. relações direcionais: relações que expressam noções de direção, por exemplo, acima de (*Above*), ao norte de (*NorthOf*), dentre outras;

   c. relações métricas: relações que consideram a distância, como por exemplo, todas geometrias que estão a uma determinada distância de outra geometria qualquer.

(7) Operações binárias com resultado escalar: computar a distância entre dois objetos geográficos (*Distance)* é a típica operação desta classe.

Primeiramente, foram definidas oito relações topológicas binárias baseadas nas quatro interseções entre fronteiras e interiores de dois objetos geográficos *n*-dimensionais (Egenhofer e Franzola, 1995). Este modelo é chamado de 4-Interseções e as interseções entre as fronteiras ($\partial A$) e interiores ($A^\circ$) dos objetos são analisadas de acordo com seu valor, vazio ($\varnothing$) ou não-vazio ($\neg\varnothing$). Na Figura 2.2.3 esse modelo é ilustrado.

Para suportar relações espaciais entre objetos geográficos com estruturas mais complexas, como regiões com ilhas e separações, foi necessário acrescentar o conceito de exterior ($A^-$) de um objeto geográfico ao modelo de 4-Interseções. Portanto, o modelo foi estendido para analisar o resultado da interseção entre as fronteiras, interiores e exteriores de dois objetos, resultando em um novo modelo chamado Modelo de 9-Interseções (Egenhofer e Herring, 1991). Mais informações sobre relações topológicas entre regiões com ilhas podem ser encontradas em Egenhofer et al (1994).

Nos modelos citados acima, os resultados das interseções são avaliados considerando os valores vazio ou não-vazio. Há várias situações em que é necessário considerar as dimensões das interseções não vazias. Por exemplo, um certo estado $X$ só considera um outro estado $Y$ como vizinho se eles têm pelo menos uma aresta em comum. Neste caso, para encontrar os vizinhos do estado $X$ não basta saber quais estados "tocam" ou são "adjacentes" a ele, e sim se o resultado da interseção entre eles é uma aresta.

Com isso, surgem novos modelos que consideram as dimensões dos resultados das interseções não vazias, como o Modelo para Relações Topológicas Binárias Detalhadas (Egenhofer et al, 1993), o qual é baseado no modelo de 4-Interseções, e o Modelo de 9-Interseções Estendido Dimensionalmente (DE-9IM), que é baseado no modelo de 9-Interseções (Paiva, 1998).

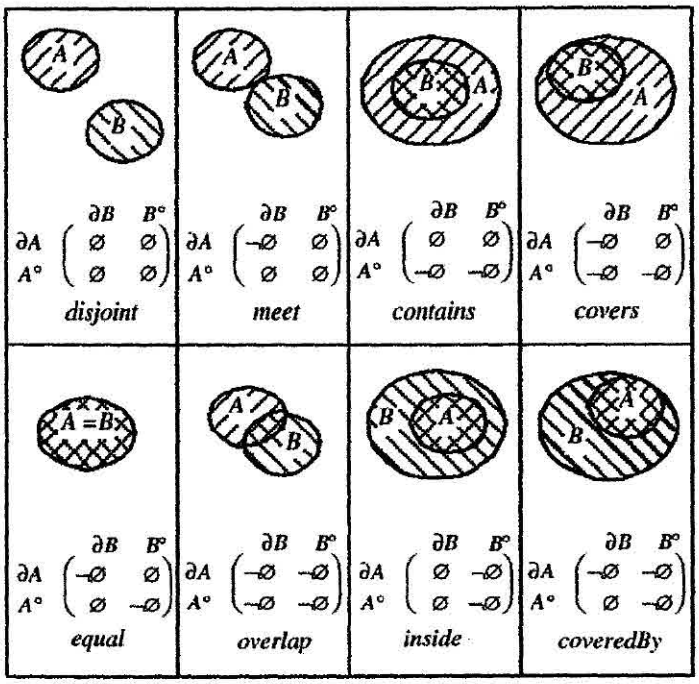



Figura 2.2.3 - Relações topológicas

### 2.4.2 Dados matriciais

Os geo-campos podem ser especializados em (Cordeiro et al, 1996):

- Temático: onde cada ponto do espaço, ou seja, cada célula do *raster* está associada a um tema, como por exemplo, um geo-campo de vegetação pode ser caracterizado pelo conjunto de temas: floresta densa, floresta aberta e cerrado;
- Numérico: onde cada ponto do espaço está associado a um valor real, como por exemplo, um mapa de campo magnético ou um mapa de altimetria;
- Dado de sensor remoto: é uma especialização do numérico obtida através da discretização da resposta recebida por um sensor de uma superfície terrestre.

Tomlin (1990) definiu três tipos de operações sobre dados matriciais que são expressas em termos de localizações individuais (local), de zonas ou regiões (zonal) ou de vizinhos das localizações (focal). Essas operações, também chamadas de álgebra de

mapas, estão implementadas na linguagem de álgebra de mapas LEGAL, a qual é associada ao SIG SPRING (Câmara, 1995).

As operações locais geram como saída um geo-campo onde o valor de cada localização é calculado em função dos valores dos geo-campos de entrada na localização correspondente. Na linguagem LEGAL estão implementadas as seguintes operações locais:

- Operações matemáticas: aritméticas (soma, subtração, multiplicação e divisão), trigonométricas (seno, co-seno, tangente, etc) e relacionais (maior que, menor que, igual, diferente, etc);
- Recorte (*Mask*): recorta um geo-campo a partir de outro geo-campo ou a partir de um polígono;
- Reclassificação (*Reclassify*): reclassifica um geo-campo temático segundo um conjunto de regras, que define como os temas serão mapeados para novos temas;
- Fatiamento (*Slice*): transforma um geo-campo numérico em temático, fazendo o mapeamento de cada intervalo de valores do numérico para um tema específico, através de um conjunto de regras;
- Ponderação (*Weight*): transforma um geo-campo temático em numérico, mapeando cada tema para um valor específico, definido pelas regras.

As operações focais ou de vizinhança geram como saída um geo-campo computado com base na dimensão e forma de uma vizinhança em torno de cada localização. Como exemplo dessas operações podem ser citados os cálculos de valores mínimo, máximo e médio para uma vizinhança em torno de uma localização e filtros para processamento de dados de sensores remotos.

As operações zonais geralmente utilizam um geo-campo temático como restrição espacial sobre um outro geo-campo numérico. Essas operações consistem em calcular estatísticas, como soma, média e variância, de um conjunto de valores de um geo-campo

numérico que estão contidos em uma determinada zona (região delimitada por um polígono). No caso de um geo-campo temático, cada tema pode ser definido como uma zona. As operações implementadas no LEGAL são: ZonalSum, ZonalMean, ZonalMax, ZonalMin, ZonalStdev e ZonalVariety.

## 2.5 Oracle Spatial

O Oracle Spatial (Murray, 2001) é uma extensão espacial desenvolvida sobre o modelo objeto-relacional do SGDB Oracle. Este modelo permite definir novos tipos de dados, através da linguagem de definição de dados SQL DDL, e implementar operações sobre esses novos tipos através da linguagem PL/SQL, que é uma extensão procedural da SQL (Lassen et al, 1998). Esta extensão é baseada nas especificações do OpenGIS e contém um conjunto de funcionalidades e procedimentos que permitem armazenar, acessar e consultar dados espaciais de representação vetorial em um banco de dados Oracle.

Seu modelo de dados consiste em uma estrutura hierárquica de elementos, geometrias e planos de informação (*layers*), onde planos são formados por um conjunto de geometrias, que por sua vez são formadas por um conjunto de elementos. Um elemento pode ser do tipo Ponto, Linha ou Polígono (com ou sem ilhas). Uma geometria pode ser formada por um único elemento ou por um conjunto homogêneo (MultiPontos, MultiLinhas ou MultiPolígonos) ou heterogêneo (Coleção) de elementos. Um plano de informação é formado por uma coleção de geometrias que possuem um mesmo conjunto de atributos.

Devido à utilização de um modelo objeto-relacional, cada geometria é armazenada em um objeto espacial chamado SDO_GEOMETRY. Este objeto contém a geometria em si, suas coordenadas, e informações sobre seu tipo e projeção. 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. Nesta tabela espacial cada instância da geometria é armazenada em uma linha e o conjunto de todas as geometrias de uma mesma tabela formam um plano. Abaixo é mostrado um exemplo da criação de uma tabela espacial em SQL:

```
create table Estados_do_Brasil
(
      nome_estado          VARCHAR2(100),
      nome_captial         VARCHAR2(100),
      populacao_03         NUMBER,
      fronteira            MDSYS.SDO_GEOMETRY
)
```

O objeto SDO_GEOMETRY é composto pelos seguintes atributos:

- SDO_GTYPE: formado por quatro números, onde os dois primeiros indicam a dimensão da geometria e os outros dois o seu tipo. Os tipos podem ser: 00 (não conhecido), 01 (ponto), 02 (linha ou curva), 03 (polígono), 04 (coleção), 05 (multipontos), 06 (multilinhas) e 07 (multipolígonos);

- SDO_SRID: utilizado para identificar o sistema de coordenadas, ou sistema de referência espacial, associado à geometria;

- SDO_POINT: é definido utilizando um objeto do tipo SDO_POINT_TYPE, que contém os atributos X, Y e Z para representar as coordenadas de um ponto. Somente é preenchido se a geometria for do tipo ponto, ou seja, se os dois últimos números do SDO_GTYPE forem iguais a "01";

- SDO_ELEM_INFO: é um vetor de tamanho variável que armazena as características dos elementos que compõem a geometria. Cada elemento tem suas coordenadas armazenadas no SDO_ORDINATES que são interpretadas por três números armazenados no SDO_ELEM_INFO:

    - SDO_STARTING_OFFSET: indica qual a posição da primeira coordenada do elemento no SDO_ORDINATES;

    - SDO_ETYPE: indica o tipo do elemento;

    - SDO_INTERPRETATION: indica como o elemento deve ser interpretado juntamente com o SDO_ETYPE.

- SDO_ORDINATES: é um vetor de tamanho variável que armazena as coordenadas da geometria.

O Oracle Spatial fornece dois tipos de indexação espacial, R-tree e Quadtree, podendo ser utilizados simultaneamente. Indexação espacial, como qualquer outro tipo de indexação, fornece mecanismos para limitar o conjunto de busca, aumentando assim a performance das consultas e da recuperação de dados geográficos.

Cada tipo de índice é apropriado para diferentes situações. A R-tree consiste em particionar o dado em conjuntos de objetos geográficos, baseando-se na distribuição dos mínimos retângulos envolventes (MBRs) desses objetos no espaço. A Quadtree particiona o espaço 2D, recursivamente, em quadrantes, independente da distribuição dos objetos geográficos no plano.

Além das estruturas de indexação, a extensão fornece funções para avaliar a performance dos índices criados (ANALYZE_RTREE) e para reconstruir um antigo índice depois da inserção de novas geometrias na tabela espacial (REBUILD).

O modelo global próprio do Oracle Spatial, chamado MDSYS, apresenta um conjunto de tabelas de metadados que são utilizadas por algumas funcionalidades internas da extensão, como por exemplo, nas consultas espaciais. Há, basicamente, dois tipos de tabelas de metadados:

- As que contêm informações sobre a indexação espacial de cada tabela e coluna espacial. Essas tabelas são: USER_SDO_INDEX_METADATA, ALL_SDO_INDEX_INFO e DBA_SDO_INDEX_INFO;
- As que contêm informações sobre as geometrias armazenadas em cada tabela e coluna espacial, como sua dimensão (mínimo retângulo envolvente) e a tolerância em cada dimensão. Essas tabelas são: USER_SDO_GEOM_METADATA, ALL_SDO_GEOM_METADATA e DBA_SDO_GEOM_METADATA.

### 2.5.1 Operações espaciais no Oracle Spatial

A extensão utiliza um modelo de consulta baseado em duas etapas, chamadas de primeiro e segundo filtro. O primeiro filtro considera as aproximações das geometrias,

pelo critério do mínimo retângulo envolvente (MBR), para reduzir a complexidade computacional. É considerado um filtro de baixo custo computacional e seleciona um subconjunto menor de geometrias candidatas que será passado para o segundo filtro. O segundo filtro trabalha com as geometrias exatas, por isso é computacionalmente mais caro e só é aplicado ao subconjunto resultante do primeiro filtro. Retorna o resultado exato da consulta.

Para suportar consultas e operações espaciais, o Oracle Spatial fornece um conjunto de operadores e funções que são utilizados juntamente com a linguagem SQL. Para consultar relações espaciais são utilizados os operadores que utilizam os dois filtros do modelo de consulta:

- SDO_RELATE: consulta relações topológicas entre objetos geográficos, com base no Modelo de 9-Interseções de Engenhofer e Herring (1991). Recebe como parâmetro o tipo da relação a ser computada, a qual pode ser: EQUAL, DISJOINT, TOUCH, INSIDE, OVERLAPBDYINTERSECT, OVERLAPBDYDISJOINT, ANYINTERACT, CONTAINS, ON, COVERS e COVERREDBY;

- SDO_WITHIN_DISTANCE: verifica se dois objetos geográficos estão a uma determinada distância, a qual é passada como parâmetro;

- SDO_NN: identifica *n* vizinhos mais próximos de um objeto geográfico.

Dentre as funções podem ser citadas: SDO_BUFFER, gera uma nova geometria a partir de uma distância em torno de uma geometria específica (*buffer*); SDO_AREA e SDO_LENGTH, calculam a área e o perímetro; SDO_DISTANCE, calcula a distância entre dois objetos; e, SDO_INTERSECTION, SDO_UNION e SDO_DIFFERENCE, geram uma nova geometria resultante da interseção, união e diferença, respectivamente, entre outras duas.

Como exemplo do uso desses operadores e funções juntamente com a linguagem SQL, considere a consulta "listar todas as zonas sensíveis dentro de um raio de 8 km de uma

área de lixo tóxico (area de risco)", que foi ilustrado na Figura 2.2.4. Supondo a existência de duas tabelas espaciais, uma para armazenar as zonas sensíveis que são representadas por polígonos e a outra para armazenar as áreas de risco ou de lixo tóxico, representadas por pontos. As tabelas são mostradas na Tabela 2.1.

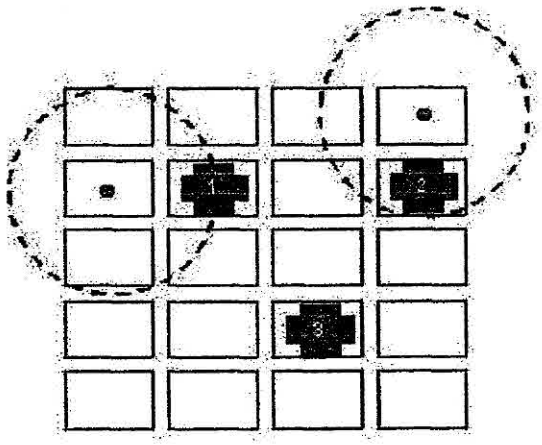


Figura 2.2.4 - Cenário da Consulta

TABELA 2.1 – TABELAS ESPACIAIS

| Zona_sensível | | area_risco | |
|---|---|---|---|
| **Nome Atributo** | **Tipo** | **Nome Atributo** | **Tipo** |
| nome | VARCHAR2(50) | nome | VARCHAR2(50) |
| zona | SDO_GEOMETRY | local | SDO_GEOMETRY |

A consulta em SQL utiliza o operador SDO_RELATE, juntamente com a função SDO_BUFFER, e é computada pelo SGBD Oracle:

```
SELECT ass.nome, ar.nome
FROM   AREA_RISCO ar, ZONA_SENSIVEL ass,
       USER_SDO_GEOM_METADATA m
WHERE  m.table_name = 'AREA_RISCO'
AND    (SDO_RELATE(ass.zona,
       SDO_GEOM.SDO_BUFFER(ar.local, m.diminfo,
       8000), 'mask=ANYINTERACT querytype=WINDOW') = 'TRUE');
```

Observa-se na consulta acima que, além das tabelas espaciais, a função SDO_BUFFER utiliza as informações da tabela de metadado do Oracle Spatial, chamada USER_SDO_GEOM_METADATA.

O operador SDO_RELATE é usado sempre na cláusula WHERE de uma consulta em SQL e recebe como parâmetros: a tabela e sua coluna espacial envolvida na consulta (`ass.zona`), a geometria resultande da função SDO_BUFFER, a relação topológica (`mask = ANYINTERACT`) e o tipo da consulta (`querytype`), que pode ser WINDOW ou JOIN. O tipo de consulta WINDOW é usado quando se quer comparar uma simples geometria com todas as geometrias de uma coluna espacial. E o tipo JOIN é usado quando se quer comparar todas as geometrias de uma coluna espacial com todas as geometrias de outra coluna.

# CAPÍTULO 3

# UMA INTERFACE GENÉRICA PARA OPERAÇÕES ESPACIAIS EM BANCOS DE DADOS GEOGRÁFICOS

## 3.1 Introdução

A interface para operações espaciais, ou API, desenvolvida neste trabalho foi implementada na TerraLib (Câmara et al, 2000), uma biblioteca de software livre que oferece suporte para a construção de uma nova geração de aplicativos geográficos, baseados na arquitetura integrada. A API é composta por um conjunto de funções que permitem realizar operações espacias sobre dados geográficos, vetoriais e matriciais, armazenados em diferentes SGBDs, relacionais ou objeto-relacionais.

Além de disponibilizar operações espaciais a um nível maior de abstração na TerraLib, a API desenvolvida explora uma nova geração de SGBDs objeto-relacionais que oferecem recursos para tratar dados geográficos, chamados de Sistemas de Banco de Dados Geográficos. Portanto, no caso de SGBDORs que possuem extensões espaciais, como Oracle Spatial, a API explora ao máximo seus recursos, como indexação espacial e operadores e funções para manipular esses dados. Para que esta API fosse aplicada ao pacote espacial da Oracle, foi também desenvolvido neste trabalho o *driver* para o Oracle Spatial (`TeOracleSpatial`) na TerraLib, utilizando a biblioteca OCI (*Oracle Call Interface*) (Locke, 1999).

A Figura 3.1 citua a API no contexto TerraLib, Aplicativos Geográficos e SGBDs. Os aplicativos geográficos desenvolvidos sobre a TerraLib utilizam as funções da API para relizarem operações espaciais sobre os dados geográficos armazenados nos SGBDs. A API é responsável pela interface com os diferentes SGBDs e pela computação dessas operações.

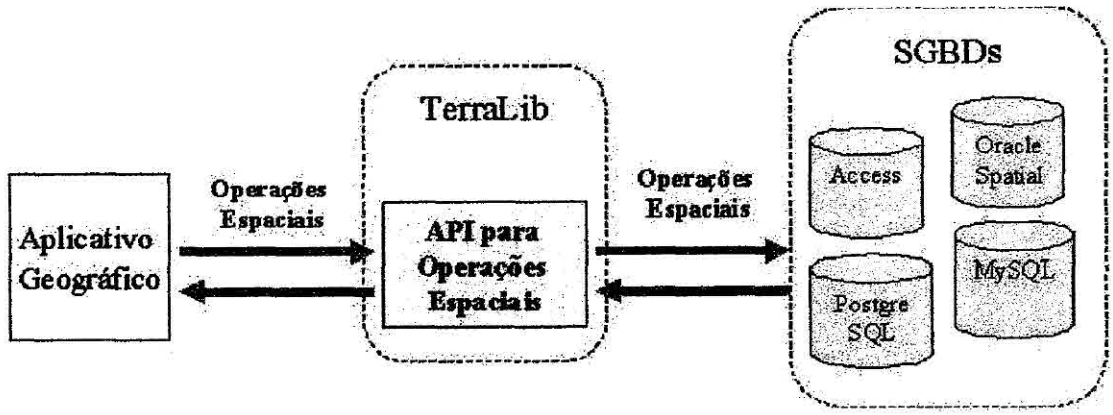



Figura 3.1 - API para operações espaciais

As operações sobre dados vetoriais, implementadas na API, foram baseadas nas operações definidas em Rigaux et al (2002) e nas relações topológicas especificadas por Engenhofer (1994). E as operações sobre dados matriciais foram baseadas nas especificações de Tomlin (1990). A Figura 3.2 apresenta todas as funções da API.

| | Booleano | Escalar | Espacial |
|---|---|---|---|
| Unárias (DV) | - | `calculateLength` `calculateArea` | `Buffer` `ConvexHull` |
| Binárias (DV + DV) | `SpatialRelation` `NearestNeighbors` | `calculateDistance` | `Intersection` `Union` `Difference` `XOr` |
| Binárias (DV + DM) | - | `Zonal` | `Mask` |

**DV: Dado Vetorial** **DM: Dado Matricial**

Figura 3.2 - Operações espaciais da API

As funções disponíveis na API podem ser divididas em sete classes considerando o número e o tipo dos argumentos de cada função e o tipo do dado resultante:

- Funções unárias sobre dados vetoriais com resultado escalar:

- `calculateArea`: calcula a área de uma região;
- `calculateLength`: calcula o comprimento ou o perímetro;

- Funções unárias sobre dados vetoriais com resultado espacial:
  - `ConvexHull`: gera uma nova geometria convexa a partir de uma outra geometria;
  - `Buffer`: gera uma nova geometria a partir de uma distância em torno de uma geometria específica;
- Funções binárias sobre dados vetoriais com resultado booleano:
  - `SpatialRelation`: consulta relações topológicas entre geometrias
  - `NearestNeighbors`: calcula os os *n* vizinhos mais próximos de uma geometria específica;
- Funções binárias sobre dados vetoriais com resultado escalar:
  - `calculateDistance`: calcula a distância entre duas geometrias;
- Funções binárias sobre dados vetoriais com resultado espacial:
  - `Intersection`, `Union`, `Difference` e `Xor`: geram novas geometrias através de operações de conjunto, respectivamente, interseção, união, diferença e diferença simétrica, entre duas geometrias;
- Funções binárias entre um dado matricial e outro vetorial com resultado escalar:
  - `Zonal`: calcula as estatísticas de uma região ou zona de um dado matricial delimitada por um polígono
- Funções binárias entre um dado matricial e outro vetorial com resultado espacial:

- `Mask`: recorta um dado matricial a partir de uma máscara que é definida por um polígono.

As próximas seções deste capítulo mostrarão como essas operações foram implementadas, apresentando antes uma breve descrição da TerraLib.

### 3.2 Breve Descrição da TerraLib

A TerraLib é uma biblioteca de software livre que oferece suporte para a construção de uma nova geração de aplicativos geográficos, baseados na arquitetura integrada (Câmara et al, 2000). É uma biblioteca de classes desenvolvida na linguagem C++ (Stroustrup, 1999), seguindo os paradigmas de orientação a objetos (Booch, 1994), programação genérica (Austern,1998) e *design patterns* (Gamma et al, 1995). É organizada em três partes principais:

- *Kernel*: composto por classes básicas (estruturas de dados) para representação dos dados geográficos, tanto no formato vetorial quanto matricial, e algoritmos sobre esses dados, classes de sistemas de projeção cartográfica e uma classe base que define uma interface comum para todos os *drivers*;

- *Drivers*: para cada SGBD existe um *driver*, ou seja, uma classe específica que implementa todas as funcionalidades da classe base conforme as particularidades e características de cada SGBD. Dentre essas funcionalidades estão inserção, recuperação e remoção de dados geográficos;

- *Functions*: contém os algoritmos que utilizam as estruturas básicas do *Kernel*, incluindo análise espacial, linguagens de consulta e simulação e funções para conversão de dados.

As classes do *Kernel* para representação de dados geográficos em memória são mostradas na Figura 3.3.3. Existem classes para representar dados vetoriais, como por exemplo, `TeCoord2D` para representar coordenadas de duas dimensões, `TeLine2D` para linhas e `TePolygon` para polígonos, e classes para representar dados matriciais, como a classe `TeRaster`. Além de classes para representar dados geográficos, a

TerraLib possui classes que se comportam como conteiners de objetos geográficos, como a classe `TeLayer`, que representa um plano de informação ou *layer*.

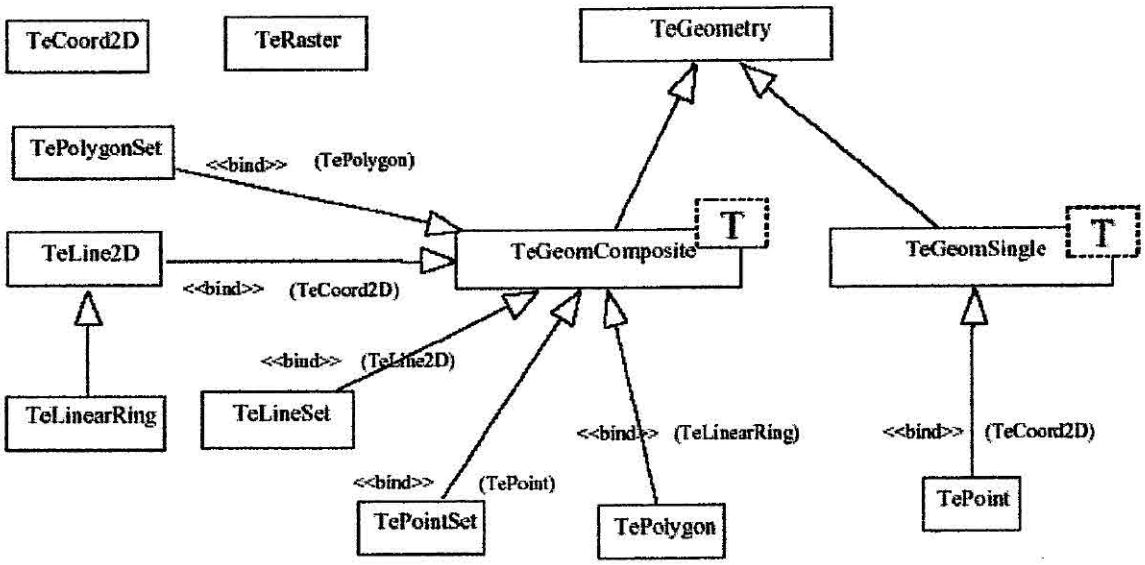



Figura 3.3.3 - Classes do *Kernel* da TerraLib

Os *drivers* possuem uma interface comum que é definida por duas classes bases: `TeDatabase` e `TeDatabasePortal`. Dentre as funcionalidades fornecidas por essa interface podem ser citadas: estabelecimento de conexão com o servidor de banco de dados; criação do modelo de dados da TerraLib; execução de comandos em SQL DDL e DML; execução de consultas em SQL que retornam um conjunto de resultados e manipulação desses resultados; definição de índices espaciais; e, armazenamento e manipulação de dados vetoriais e matriciais (Ferreira et al, 2002).

Cada *driver* fornece uma interface entre o *Kernel* da TerraLib e um SGBD específico, permitindo o armazenamento e manipulação dos dados geográficos diretamente no SGBD, sem a necessidade de utilização de estruturas proprietárias. Esses *drivers* exploram ao máximo os recursos oferecidos por cada SGBD, principalmente os que apresentam extensões espaciais que fornecem mecanismos de indexação espacial, tipos de dados geográficos e operadores espaciais.

Atualmente, existem quatro *drivers* na TerraLib: `TeOracleSpatial` (Oracle Spatial), `TePostgreSQL` (PostgreSQL), `TeMySQL` (MySQL) e `TeADO` (acessar SGBDs via tecnologia ADO). A Figura 3.3.4 mostra a estruturação das classes que representam a interface comum e os *drivers*.

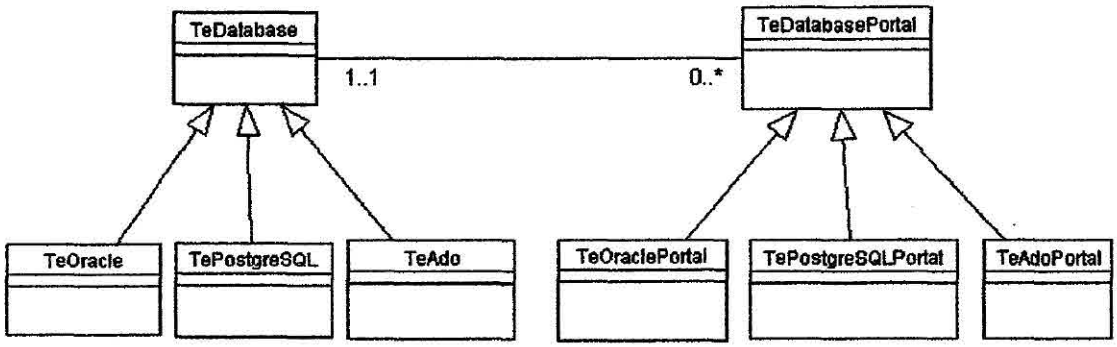



Figura 3.3.4 - Interface com SGBDs

A TerraLib possui um modelo de dados específico que é criado por cada *driver* no respectivo SGBD. Esse modelo é composto por esquemas de tabelas espaciais para armazenar e manipular os dados geográficos vetoriais e matriciais (TerraLib, 2003). Nos SGBDs que não possuem extensão espacial para tratar dados geográficos, esses são armazenados em campos longos binários, chamados de BLOB. Já nos SGBDs que possuem extensão espacial, como o Oracle Spatial, os dados geográficos vetoriais são armazenados em estruturas de dados específicas fornecidas pela extensão. Mas, em todos os SGBDs, os dados matriciais são armazenados em campos longos binários, pois nenhuma extensão espacial oferece recursos para tratar esse tipo de dado.

Como exemplo, as figuras a seguir mostram o esquema da tabela de polígonos do modelo de dados da TerraLib representada em um SGBD relacional (Figura 3.3.5) e no Oracle Spatial (Figura 3.3.6). Essa tabela no SGBD relacional armazena o polígono em um campo do tipo longo binário (*spatial_data*), além de guardar informações sobre o número de coordenadas (*num_coords*), o número de buracos (*num_holes*) e o mínimo retângulo envolvente (*lower_x*, *lower_y*, *upper_x* e *upper_y*) de cada polígono. No Oracle Spatial o polígono é armazenado em um campo do tipo SDO_GEOMETRY

(*spatial_data*), que é um objeto espacial fornecido pela extensão, e as outras informações referentes aos polígonos não precisam ser armazenadas explicitamente.

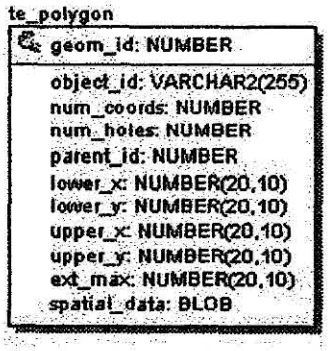



Figura 3.3.5 - Tabela de polígonos em SGBDs relacionais

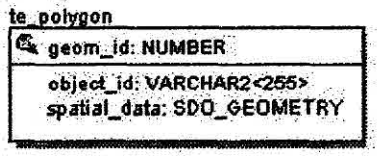



Figura 3.3.6 - Tabela de polígonos no Oracle Spatial

## 3.3 Comparação com ArcSDE

Como trabalho relacionado com a API desenvolvida, pode ser citado o ArcSDE, um aplicativo que fornece uma interface entre os ArcGIS (ArcInfo, ArcEditor, ArcView GIS e ArcIMS) e diferentes SGBDs, como Oracle, Informix, IBM DB2 e Microsoft SQL Server (ArcSDE, 2003). Na Tabela 3.1 é apresentado uma comparação entre a TerraLib e o ArcSDE, incluindo as APIs para operações espaciais.

TABELA 3.1 – TERRALIB X ARCSDE

| | **TerraLib** | **ArcSDE** |
|---|---|---|
| **Arquitetura de SIG** | Suporte para a construção de SIGs de arquitetura integrada | Suporte para os ArcGIS utilizarem uma arquitetura integrada |
| **SGDBs** | Suporte a diferentes tipos de SGBDs com ou sem extensão espacial | Suporte a diferentes tipos de SGBDs com ou sem extensão espacial |
| **Modelo de Dados** | Cria um modelo de dados lógico próprio para representar | Cria um modelo de dados lógico próprio para representar dados |

| | dados geográficos em SGBDs | geográficos em SGBDs |
|---|---|---|
| **Arquitetura de software** | Servidor de Banco de Dados (SGBD) + cliente (SIG + TerraLib) | Servidor de Banco de Dados (SGBD) + servidor de aplicação (ArcSDE) + cliente (ArcSIG) |
| **API para operações espaciais** | Operações a um alto nível de abstração | Operações a um baixo nível de abstração |

O ArcSDE possui dois tipos de APIs para operações espaciais: (1) API que fornece funções para executar operações espaciais em memória; e (2) SQL API que apenas designa a execução de uma operação em SQL, que utiliza operadores e funções espaciais, para o Informix Spatial Datablade ou para o IBM DB2 Spatial Extender.

A API desenvolvida neste trabalho disponibiliza as operações espaciais em um nível maior de abstração do que as APIs do ArcSDE. A API da TerraLib encapsula as operações espaciais de forma que o usuário não precise ter conhecimento de como elas são computadas, se é pelo SGBD com extensão espacial ou em memória. Já as APIs disponíveis no ArcSDE exigem que o usuário saiba como as operações devem ser computadas e escolher qual API utilizar.

### 3.4 Operações sobre dados vetoriais

As operações espaciais sobre dados vetoriais da API foram implementadas em dois níveis:

(1) Nível genérico: as operações foram implementadas como métodos da classe base `TeDatabase`, sendo utilizadas por todos os SGBDs que não possuem extensão espacial, como Access, MySQL e PostgreSQL.

(2) Nível específico: as operações foram reimplementadas no *driver* do SGBD Oracle Spatial, `TeOracleSpatial`, utilizando os recursos de sua extensão espacial.

As operações genéricas da classe `TeDatabase` utilizam funções da TerraLib sobre os dados espaciais em memória, recuperados do SGBD. Essas operações seguem duas etapas:

(1) Recuperação das geometrias do SGBD através de uma consulta em SQL: o SGBD retorna as geometrias como conjuntos de valores binários (BLOB), o `TeDatabase` faz a leitura de cada BLOB e transforma para uma das estruturas de dados geográficos da TerraLib (`TePolygon`, `TeLine2D`, `TePoint`, etc).

(2) Aplicação de uma função da TerraLib sobre as estruturas de dados geradas.

A Figura 3.3.7 mostra um exemplo da operação "Calcule a área do estado de Mias Gerais", seguindo as duas etapas descritas acima. Neste exemplo, é utilizada uma função chamada `AREA` da TerraLib, que recebe um `TePolygon` como argumento e retorna sua área.

As operações específicas, ou seja, do *driver* `TeOracleSpatial`, consistem em montar uma consulta em SQL utilizando os operadores e funções da extensão espacial, designando sua computação para o SGBD.

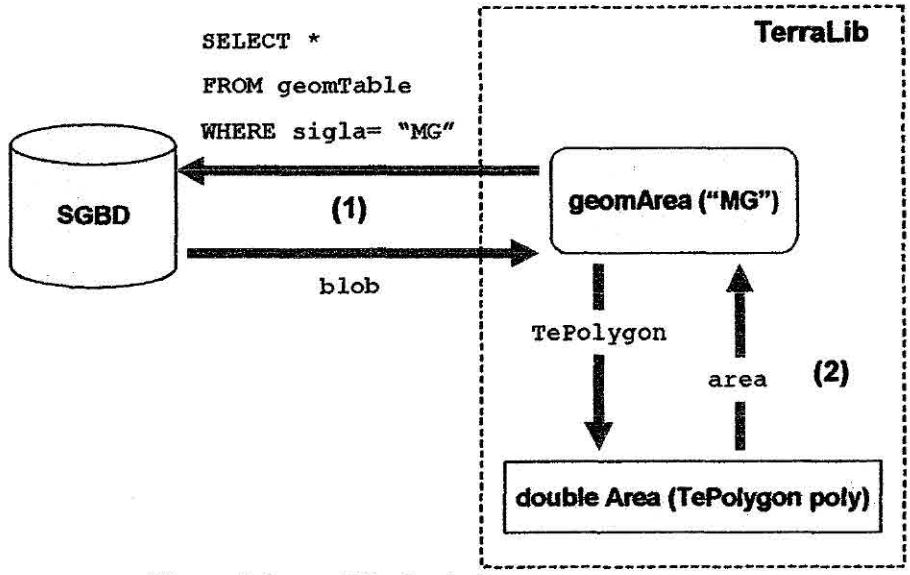



Figura 3.3.7 - Cálculo de área no `TeDatabase`

No `TeOracleSpatial`, a operação para calcular a área monta uma consulta em SQL utilizando a função SDO_AREA do Oracle Spatial. Essa consulta é passada para o SGBD que computa e retorna a área das geometrias especificadas na cláusula WHERE. A Figura 3.3.8 ilustra a operação "Calcule a área do estado de Minas Gerais", utilizando a extensão Oracle Spatial.

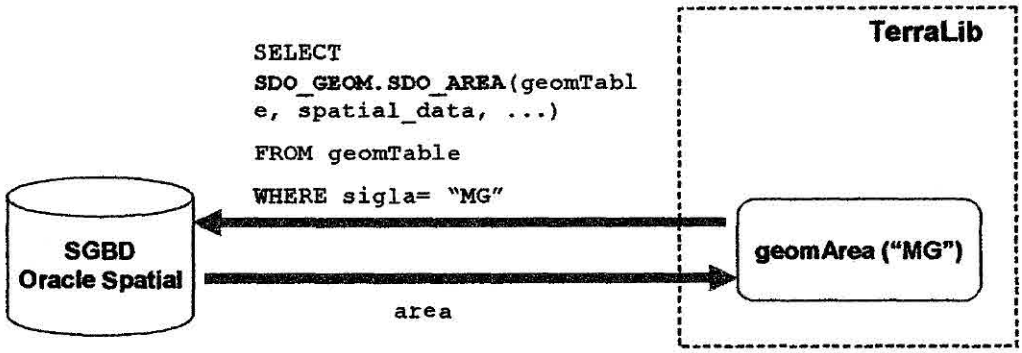



Figura 3.3.8 - Cálculo de área no TeOracleSpatial

Dentre as classes definidas por Rigaux et al (2002), a API fornece as operações espaciais sobre dados vetoriais mostradas na Tabela 3.3.

TABELA 3.3 – OPERAÇÕES ESPACIAIS DA API SOBRE DADOS VETORIAIS

| Número de argumentos da operação | Valor de retorno da operação: Booleano | Escalar | Espacial |
|---|---|---|---|
| Unário | | `calculateLength`<br>`calculateArea` | `ConvexHull`<br>`Buffer` |
| Binário | `SpatialRelation`<br>`NearestNeighbors` | `calculateDistance` | `Intersection`<br>`Union`<br>`Difference`<br>`XOr` |

As operações unárias da API são executadas sobre um determinado objeto geográfico de uma tabela espacial. As operações binárias são executadas entre pares de objetos geográficos, que podem estar armazenados em uma mesma tabela espacial ou em tabelas distintas.

Na TerraLib, cada representação geométrica de um plano de informação (PI), ou *layer*, é armazenada em uma tabela espacial. Como exemplo, serão considerados dois PIs onde um é formado pelos estados do Brasil, que é representado geometricamente por polígonos, e outro formado pelos rios do Brasil, representado geometricamente por linhas. No modelo da TerraLib, cada PI será armazenado no SGBD em uma tabela espacial distinta. Na Figura 3.3.9 esse exemplo é ilustrado.

Portanto, as operações binárias podem ser executadas entre geometrias de um mesmo PI ou entre geometrias de dois PIs distintos. Por exemplo, a consulta "selecione todos os rios que cruzam a fronteira do estado de Minas Gerais" será executada pela operação binária `SpatialRelation` entre as geometrias das duas tabelas espaciais Estados_polígonos e Rios_linhas. Já a consulta "qual é a distância entre os estados Acre e Minas Gerais" será executada pela operação binária `calculateDistance` entre as geometrias da tabela espacial Estados_polígonos.

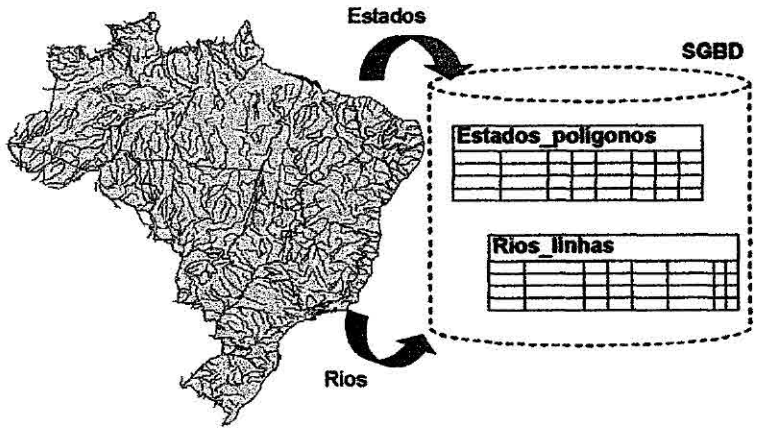



Figura 3.3.9 - Armazenamento de dados geográficos no SGBD

As funções unárias da API recebem, no mínimo, os parâmetros de entrada: nome da tabela espacial (`actGeomTable`); nome da coluna espacial (`actGeomColl`); tipo da representação geométrica armazenada na tabela (`actRep`); os identificadores dos objetos geográficos sobre os quais a operação será aplicada (`actIdsIn`); e, a variável ou a estrutura de dados que conterá o resultado, que pode ser do tipo escalar ou espacial (`TePolygonSet`). A seguir são mostradas, como exemplo, as assinaturas das operações `calculateArea` e `Buffer`:

```
bool calculateArea(const string& actGeomTable, const string&
actGeomColl, TeGeomRep actRep, Keys& actIdsIn, const double&
area);

bool Buffer(const string& actGeomTable, const string&
actGeomColl, TeGeomRep actRep, Keys& actIdsIn, TePolygonSet&
bufferSet, const double& distance);
```

Quando é passado mais de um objeto geográfico como argumento, a função **calculateArea** retorna a soma de suas áreas e a função **Buffer** retorna o conjunto das geometrias geradas a partir de todos os objetos. Como cada geometria gerada é representada por um polígono, a estrutura resultante é um conjunto de polígonos, que é representado por uma estrutura chamada `TePolygonSet`.

As funções binárias entre objetos de um mesmo PI, ou seja, de uma mesma tabela espacial, recebem os mesmos parâmetros de entrada de uma operação unária descritos acima. Como essas funções são realizadas entre pares de objetos, o identificador de cada objeto é passado como um parâmetro distinto (*objId1* e *objId2*). Para ilustrar, são mostradas, a seguir, as assinaturas das funções **calculateDistance** e **geomIntersection**:

```
bool calculateDistance(const string& actGeomTable, const string&
actGeomColl, TeGeomRep actRep, const string& objId1, const
string& objId2, const double& distance);

bool geomIntersection(const string& actGeomTable, const string&
actGeomColl, TeGeomRep actRep, const string& objId1, const
string& objId2, TeGeometryVect& geomVect);
```

A interseção entre dois objetos geográficos pode resultar em diferentes geometrias, como um polígono, uma linha ou um ponto, ou ainda em um conjunto dessas geometrias. Por isso, a estrutura de dados para armazenar esse resultado, `TeGeometryVect`, consiste em um conjunto de ponteiros para o tipo abstrato `TeGeometry`, do qual todas as outras estruturas geométricas da TerraLib, como

TePolygon, TeLine e TePoint, são derivadas. O mesmo é válido para as outras funções de conjunto como, **geomDifference**, **geomUnion** e **geomXOr**. A definição da estrutura TeGeometryVect é mostrada a seguir:

```
typedef vector<TeGeometry*> TeGeometryVect
```

As funções binárias entre objetos de dois *layers* distintos, ou seja, de duas tabelas espaciais, recebem como parâmetros de entrada: nomes das duas tabelas espaciais (*actGeomTable* e *visGeomTable*); nomes das colunas espaciais (*actGeomColl* e *visGeomColl*); tipo da representação geométrica armazenada em cada tabela (*actRep* e *visRep*); os identificadores dos objetos geográficos sobre os quais a operação será aplicada (*objId1* e *objId2*); e, a variável ou a estrutura de dados que conterá o resultado. Como exemplo, as assinaturas das funções **calculateDistance** e **geomIntersection** entre geometrias de diferentes tabelas são mostradas a seguir:

```
bool calculateDistance(const string& actGeomTable, const string&
actGeomColl, TeGeomRep actRep, const string& objId1, const
string& visGeomTable, const string& visGeomColl, TeGeomRep
visRep, const string& objId2, const double& distance);
```

```
bool geomIntersection(const string& actGeomTable, const string&
actGeomColl, TeGeomRep actRep, const string& objId1, const
string& visGeomTable, const string& visGeomColl, TeGeomRep
visRep, const string& objId2, TeGeometryVect& geomVect);
```

### 3.4.1 Relações topológicas

A função **SpatialRelation** da API retorna os objetos geográficos que possuem uma determinada relação topológica com um objeto específico. Assim como as outras operações binárias, esses objetos podem estar armazenados em uma mesma tabela espacial ou em tabelas distintas. Além disso, pode-se consultar relações topológicas entre os objetos de uma tabela espacial e um objeto geográfico específico em memória, ou seja, armazenado nas estruturas geográficas da TerraLib. A seguir são mostradas as assinaturas dessa função:

```
bool spatialRelation(const string& actGeomTable, const string&
actGeomColl, const string& actCollTable, TeGeomRep actRep, Keys&
actIdsIn, TeDatabasePortal *portal, int relate, const double&
tol);

bool spatialRelation(const string& actGeomTable, const string&
actGeomColl, TeGeomRep actRep, Keys& actIdsIn, const string&
visGeomTable, const string& visGeomColl, const string&
visCollTable, TeGeomRep visRep, TeDatabasePortal *portal, int
relate, const double& tol);

bool spatialRelation(const string& actGeomTable, const string&
actGeomColl, const string& actCollTable, TeGeomRep actRep,
TeGeometry* geom, TeGeomRep geomRep, TeDatabasePortal *portal,
int relate, const double& tol);
```

Todas as assinaturas acima recebem como parâmetros de entrada: a relação topológica a ser consultada (`relate`); a tolerância permitida (`tol`); e, a estrutura que conterá as geometrias resultantes (`portal`). As geometrias resultantes são todos os objetos geográficos da tabela espacial (`actGeomTable`) que possuem uma relação topológica específica (`relate`) com um determinado objeto geográfico, o qual pode estar armazenado na mesma tabela espacial (`actGeomTable`), em uma outra tabela (`visGeomTable`) ou pode estar em memória (`TeGeometry*`). As possíveis relações topológicas passadas como parâmetros de entrada são: `TeDISJOINT`, `TeTOUCHES`, `TeCROSSES`, `TeWITHIN`, `TeOVERLAPS`, `TeCONTAINS`, `TeINTERSECTS`, `TeEQUALS`, `TeCOVERS` e `TeCOVEREDBY`.

No `TeDatabase`, a operação **`SpatialRelation`** foi implementada em duas etapas, seguindo a mesma idéia do modelo de consulta do Oracle Spatial. A primeira etapa é executada sobre aproximações das geometrias, ou seja, sobre seu mínimo retângulo envolvente, e a segunda, sobre as geometrias exatas. As duas etapas podem ser resumidas em:

(1) Recuperar do SGBD, através de uma consulta em SQL, somente as geometrias cujo mínimo retângulo envolvente possui alguma interseção com o mínimo retângulo envolvente da geometria específica;

(2) Para cada geometria recuperada verificar se existe uma determinada relação topológica com a geometria específica, utilizando as funções topológicas da TerraLib.

Como exemplo, considera-se a consulta "selecione todos os estados do Brasil que fazem fronteira com o estado de Minas Gerais". A primeira etapa recupera do SGBD somente os estados cujo o mínimo retângulo envolvente tem alguma interseção com o mínimo retângulo envolvente do estado de Minas Gerais, como ilustrado na Figura 3.3.10. E a segunda etapa utiliza a função topológica `TeTouches` entre os estados recuperados na etapa anterior e o estado de Minas Gerais, retornando o resultado final da consulta.

Figura 3.3.10 - Primeira etapa da consulta espacial

No `TeOracleSpatial`, a operação **`SpatialRelation`** monta uma consulta em SQL utilizando o operador SDO_RELATE do Oracle Spatial e designa sua computação ao SGBD. A extensão espacial executa a consulta e retorna o resultado final. Esse operador recebe como parâmetro a relação topológica a ser consultada e utiliza a indexação espacial criada sobre as tabelas espaciais envolvidas na consulta.

Como exemplo, a SQL gerada para solucionar a consulta acima, "selecione todos os estados do Brasil que fazem fronteira com o estado de Minas Gerais" é

```
SELECT geomTable1.*
FROM   Estados_polígonos geomTable1,
       Estados_polígonos geomTable2
WHERE  geomTable2.object_id = 'MG'
AND    SDO_RELATE(geomTable1.spatial_data,
       geomTable2.spatial_data,
       'mask = TOUCH querytype = WINDOW') = 'TRUE'
```

### 3.4.2 Operação `Buffer` no Oracle Spatial

A operação **`Buffer`** do `TeOracleSpatial` monta uma consulta em SQL utilizando a função SDO_BUFFER da extensão espacial. Essa função retorna um objeto espacial do tipo SDO_GEOMETRY, que representa a extensão de uma geometria específica armazenada no banco. Esse objeto retornado é do tipo POLYGON, os dois últimos dígitos de seu atributo SDO_GTYPE são iguais a "03", e é formado por um conjunto de elementos, onde cada elemento pode ser do tipo:

- Polígono simples cujos vértices são conectados por linhas retas: SDO_ETYPE igual a "1003" ou "2003" e SDO_INTERPRETATON igual a "1";
- Polígono formado por segmentos de arcos circulares: SDO_ETYPE igual a "1003" ou "2003" e SDO_INTERPRETATON igual a "2";
- Polígono composto formado por alguns vértices conectados por linhas retas e outros por arcos circulares: SDO_ETYPE igual a "1005" ou "2005" e SDO_INTERPRETATON maior que "1".

Como exemplo, uma consulta em SQL para retornar uma nova geometria a partir de uma distância de 1.0 unidade ao redor de um objeto geográfico, o qual possui um identificador igual a "17" e está armazenado em uma tabela espacial chamada "teste_Linhas" em sua coluna espacial "spatial_data", pode ser escrita como:

```
SELECT      SDO_GEOM.SDO_BUFFER(g.spatial_data, m.diminfo, 1.0)
FROM        teste_Linhas g, USER_SDO_GEOM_METADATA m
WHERE       m.table_name = 'teste_Linhas'
AND         m.column_name = 'spatial_data'
AND         object_id = '17';
```

Essa consulta é executada pelo Oracle Spatial e retorna um SDO_GEOMETRY:

```
SDO_GEOMETRY          (2003, NULL, NULL,
SDO_ELEM_INFO_ARRAY   (1, 1005, 8, 1, 2, 2, 5, 2, 1, 9, 2, 2,
                      13, 2, 1, 15, 2, 2, 19, 2, 1, 23, 2, 2,
                      27, 2, 1),
SDO_ORDINATE_ARRAY    (18, 17, 19, 18, 18, 19, 16.4142136, 19,
                      14.7071068,    20.7071068,    14,    21,
                      13.2928932,    20.7071068,    11.2928932,
                      18.7071068,    11.2928932,    17.2928932,
                      12.7071068,  17.2928932,  14,  18.5857864,
                      15.2928932,    17.2928932,    15.6173166,
                      17.0761205, 16, 17, 18, 17))
```

O objeto SDO_GEOMETRY retornado é um polígono composto, que é representado pelo SDO_ETYPE igual a "1005". Portanto, a decodificação desse objeto deve gerar um polígono formado por segmentos de linhas retas e arcos circulares. A Figura 3.11 mostra esse polígono gerado a partir dos pontos retornados no SDO_GEOMETRY acima. Pode-se observar que alguns pontos são conectados por um segmento de linha reta e outros são conectados por um arco.

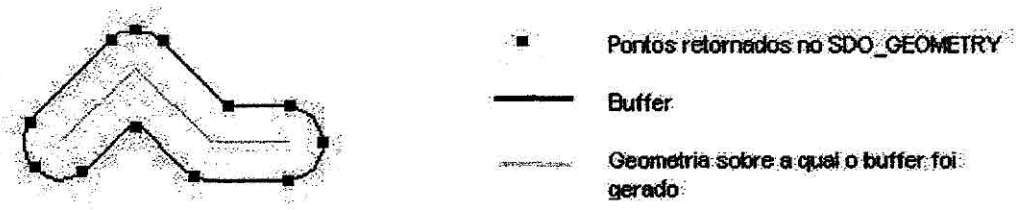



Figura 3.11 - *Buffer* formado por um polígono composto

Um arco circular no Oracle Spatial é representado apenas por três pontos e o tipo `TePolygon` da TerraLib é formado por pontos conectados por segmentos de linha reta. Portanto, para representar um polígono composto no `TePolygon`, para cada arco desse polígono é necessário gerar pontos intermediários que, ao serem ligados por segmentos

de linha reta, se aproximem de um arco. Para isso, foi necessário desenvolver uma função que gera um arco formado por *n* pontos intermediários a partir de três pontos de um arco retornados pelo Oracle Spatial.

Essa função, chamada `TeGenerateArc`, foi implementada na TerraLib e recebe como argumentos três pontos ( $pt_1$, $pt_2$ e $pt_3$) e o número *n* de pontos intermediários que devem ser gerados. Como resultado, essa função gera um arco formado por esses *n* pontos que passa pelos três pontos de entrada (Drexel University, 2003). A idéia básica dessa função é:

(1) Dado os três pontos, $pt_1$, $pt_2$ e $pt_3$, encontrar o centro da circunferência que contém esses pontos, $C(x_0, y_0)$, e o seu raio *r*;

(2) Encontrar o segmento *s* formado pelos pontos $pt_1$ e $pt_3$ e calcular o ângulo $\beta$ formado por esse segmento *s* e o centro *C* da circunferência;

(3) Calcular a variação do ângulo $\Delta\beta$ de cada ponto intermediário $pt_n$ que será gerado. Esse ângulo é calculado dividindo o ângulo $\beta$ pelo número *n* de pontos a serem gerados;

(4) Calcular o ângulo $\alpha$ formado entre o ponto $pt_1$ e o eixo horizontal da circunferência que passa pelo seu centro;

(5) Com as informações calculadas nos itens anteriores, é possível gerar os *n* pontos intermediários, utilizando as equações da circunferência:

- $Pt_n(x_n) = C(x_0) + \cos ( n*\Delta\beta + \alpha) * r$

- $Pt_n(y_n) = C(y_0) + \text{sen} ( n*\Delta\beta + \alpha) * r$

As variáveis calculadas nos itens acima são mostradas na Figura 3.12 e as funções implementadas em C++ estão disponíveis no Anexo A.

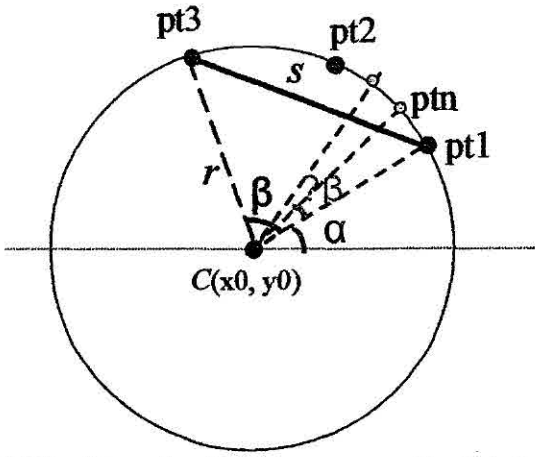



Figura 3.12 - Geração de um arco a partir de três pontos

### 3.4.3 Operações de conjunto no Oracle Spatial

As operações de conjunto da API geram as geometrias resultantes da interseção, união, diferença ou diferença simétrica entre objetos geográficos. Assim como as demais operações binárias, esses objetos geográficos podem estar armazenados em uma mesma tabela espacial ou em tabelas distintas. As funções da API são:

- `geomIntersection`: retorna a interseção topológica (operação AND) entre objetos geográficos;
- `geomDifference`: retorna a diferença topológica (operação MINUS) entre objetos geográficos;
- `geomUnion`: retorna a união topológica (operação OR) entre objetos geográficos;
- `geomXOR`: retorna a diferença simétrica topológica (operação XOR) entre objetos geográficos.

Um exemplo do resultado dessas operações está ilustrado na Figura 3.13.

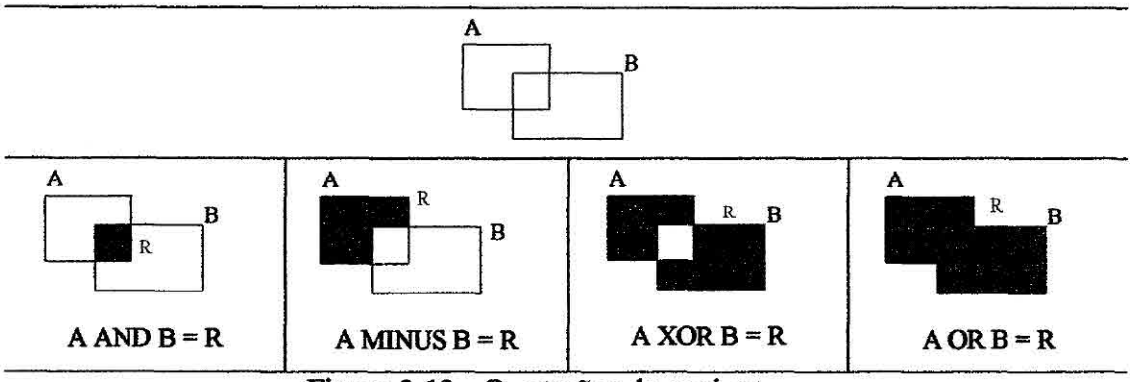



Figura 3.13 - Operações de conjunto

No `TeOracleSpatial` essas operações utilizam as funções da extensão espacial: SDO_INTERSECTION, SDO_UNION, SDO_XOR e SDO_DIFFERENCE, as quais retornam um objeto do tipo SDO_GEOMETRY, contendo a geometria resultante. Não é possível saber, a priori, o tipo do objeto que será retornado por essas funções, pois, por exemplo, a interseção entre dois polígonos pode retornar uma geometria do tipo polígono, linha ou ponto, ou até mesmo um conjunto de geometrias. Por isso, essas operações retornam uma estrutura de dados genérica, chamada `TeGeometryVect`, que pode conter o endereço de diferentes tipos de geometrias, assim como `TePolygon`, `TeLine2D` ou `TePoint`.

Como exemplo, considera-se que as geometrias A e B da Figura 3.13 estão armazenadas na coluna espacial "spatial_data" das tabelas espaciais "lotes_Poligonos" e "fronteiras_Poligonos", respectivamente. Uma consulta em SQL para calcular a diferença simétrica desses dois objetos geográficos é:

```
SELECT      SDO_GEOM.SDO_XOR(g1.spatial_data, m1.diminfo,
            g2.spatial_data, m2.diminfo)
FROM        lotes_Poligonos g1, fronteiras_Poligonos g2,
            USER_SDO_GEOM_METADATA m1, USER_SDO_GEOM_METADATA m2
WHERE       m1.table_name = 'LOTES_POLIGONOS'
AND         m1.column_name = 'SPATIAL_DATA'
AND         m2.table_name = 'FRONTEIRAS_POLIGONOS'
AND         m2.column_name = 'SPATIAL_DATA'
```

```
AND         g1.object_id = 'A'
AND         g2.object_id = 'B';
```

Essa consulta é executada pelo Oracle Spatial e retorna um SDO_GEOMETRY:

```
SDO_GEOMETRY            (2007, NULL, NULL,
SDO_ELEM_INFO_ARRAY     (1, 1003, 1, 15, 1003, 1, 29, 2003, 1),
SDO_ORDINATE_ARRAY      (24, 16, 24, 8, 32, 8, 32, 18, 28,
                         18, 28, 16, 24, 16, 10, 30, 10,
                         16, 24, 16, 24, 18, 28, 18, 28,
                         30, 10, 30, 20, 20, 20, 26, 26,
                         26, 26, 20, 20, 20))
```

O objeto SDO_GEOMETRY retornado é do tipo MULTIPOLYGON, os dois últimos dígitos de seu atributo SDO_GTYPE são iguais a "07", que é formado por um conjunto de polígonos. Portanto, a decodificação desse objeto deve gerar um `TePolygonSet`, estrutura de dados geográficos da TerraLib para representar conjuntos de polígonos.

### 3.5 Operações sobre dados matriciais

A API desenvolvida nesse trabalho fornece dois tipos de operações sobre dados matriciais: Zonal e Mask (recorte a partir de uma máscara). Essas operações foram implementadas somente no nível mais genérico, ou seja, como métodos da classe base `TeDatabase`, pois nenhuma extensão espacial oferece recursos para tratar dados matriciais. Portanto, esses métodos são utilizados por todos os SGBDs que possuem uma interface com a TerraLib, como Access, MySQL, PostgreSQL e Oracle Spatial.

A classe da TerraLib específica para tratar dados matriciais é chamada `TeRaster`. Essa classe, juntamente com outras estruturas da biblioteca, é capaz de manipular um dado matricial armazenado tanto em arquivos de diferentes formatos, como JPEG e geoTIFF, quanto em SGBDs. Nos SGBDs, cada geo-campo associado a um plano de informação, ou *layer*, é armazenado em uma tabela espacial do tipo *raster*.

Para que as operações Zonal e Mask pudessem ser aplicadas a dados matriciais armazenados tanto em arquivos em memória quanto em SGBDs, foram implementadas

as funções TeRasterStatisticsInPoly (para calcular as estatísticas de um *raster* a partir de um polígono) e TeMask (para recortar um *raster* a partir de um polígono) sobre a classe TeRaster. Assim, essas funções recebem como parâmetro de entrada um geo-campo do tipo TeRaster, o qual pode estar associado a um arquivo de formato proprietário, como JPEG ou geoTIFF, ou a uma tabela espacial do tipo *raster* do SGBD. As assinaturas dessas funções são mostradas abaixo:

```
bool TeRasterStatisticsInPoly (TePolygon& poly, TeRaster*
raster, int band, TeStatisticValMap& stat)

bool TeRasterStatisticsInPoly (TePolygon& poly, TeRaster*
raster, TeStatisticsDimensionVect& stat)

TeRaster* TeMask (TeRaster* rasterIn, TePolygon& poly,
TeIteratorStrategic st)
```

A função TeRasterStatisticsInPoly calcula as estatísticas de um geo-campo (*raster*) na região delimitada por um polígono (*poly*). No caso de geo-campos que são formados por *n* dimensões ou bandas, como as imagens se sensoriamento remoto, é possível calcular as estatísticas de uma dimensão específica passada como parâmetro (*band*). Neste caso, a função retorna uma estrutura do tipo TeStatisticValMap, que faz o mapeamento de cada estatística gerada e seu respectivo valor. Se nenhuma dimensão específica for passada como parâmetro, como na segunda assinatura, são calculadas as estatísticas de todas as bandas do *raster*. Assim, a função retorna uma estrutura do tipo TeStatisticsDimensionVect, que faz o mapeamento entre cada dimensão e suas respectivas estatísticas.

As estatísticas calculadas são: contagem, valor mínimo e máximo, soma, média, desvio padrão, variância, assimetria, curtose, amplitude, mediana, coeficiente de variação e moda. Todas as estruturas de dados e funções, implementadas em C++, utilizadas para o cálculo dessas estatísticas da operação Zonal são mostradas no Anexo B.

A função TeMask recorta um *raster* de entrada (*rasterIn*) a partir de um polígono (*poly*), segundo alguma estratégia (TeIteratorStrategic), gerando um *raster* de

saída. A estratégia define como o polígono é considerado no recorte do *raster* de entrada, se é considerado sua região interna ou externa.

Para implementar as funções acima, foi necessário desenvolver um mecanismo para percorrer o dado *raster* somente na região interna ou externa de um polígono específico. Para isso, foi implementado o conceito de iterador, sobre a classe `TeRaster`, capaz de percorrer sua estrutura segundo alguma estratégia de percurso. Essa estratégia define se o iterador percorrerá a região interna ou externa de um polígono específico.

### 3.5.1 Iteradores sobre Dados Matriciais

Iteradores são uma generalização de ponteiros; são objetos que apontam para outros objetos (Austern, 1998). Segundo Stroustrup (1999), um iterador é uma abstração da noção de um ponteiro para uma seqüência. Basicamente, seus principais conceitos são:

- "o elemento atualmente apontado" (dereferência, representada pelos operadores * e ->);
- "apontar para o próximo elemento" (incremento, representado pelo operador + +);
- "verificar se é igual a outro ponteiro, ou seja, se apontam para o mesmo lugar" (igualdade, representada pelo operador = =).

A classe `TeRaster` possui um iterador que percorre, célula a célula, todo o dado *raster*. Um dado *raster* é representado por uma estrutura matricial que pode ser de duas (linha e coluna) ou três dimensões (linha, coluna e banda). Como o tipo `TeRaster` considera um geo-campo sempre como uma estrutura matricial de três dimensões, o iterador percorre essa estrutura passando por cada linha e coluna, retornando um vetor de valores, onde cada valor está associado a uma determinada banda. Portanto, o elemento apontado pelo iterador é um vetor de valores e o seu operador de dereferência (*) retorna esse vetor. O operador de incremento (++) faz o iterador apontar para a próxima linha e coluna e é possível verificar se dois iteradores são iguais, ou seja, se eles apontam para a mesma linha e coluna do *raster*, através do operador (= =).

Iteradores são muito importantes para a programação genérica, pois fornecem uma interface entre as estruturas de dados e os algoritmos, permitindo assim um desacoplamento entre os dois (Köthe, 2000). Esse desacoplamento é essencial para a implementação de algoritmos genéricos e reusáveis (Vinhas et al, 2002).

O iterador desenvolvido nesse trabalho, chamado `iteratorPoly`, foi implementado como uma classe derivada da classe `iterator` já existente (Figura 3.14). O iterador `iterator` implementa os operadores de incremento (++) pré e pós-fixado, o de dereferência (*) e o de igualdade e diferença (= = e !=). Já o iterador `iteratorPoly` só reimplementa o operador de incremento, pois o único critério que o diferencia do iterador `iterator` é o critério de percurso. O iterador `iterator` percorre todo o *raster*, e o `iteratorPoly` só percorre o *raster* na região que está fora ou dentro dos limites de um polígono. Portanto, ele deve guardar informações sobre o polígono limitante e saber calcular, para cada linha do *raster*, as colunas que estão fora ou dentro do polígono (método `getNewSegment`). O Anexo C mostra a implementação da classe `iteratorPoly` na TerraLib em C++.

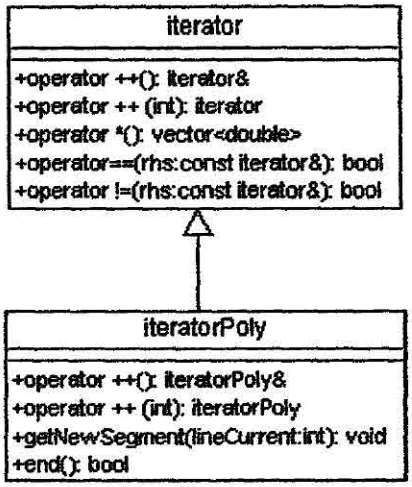



Figura 3.14 - Classe `IteratorPoly`

A seguir é mostrado, como exemplo, um procedimento para somar os valores de todos os *pixels*, da primeira banda de uma imagem, internos a um polígono, utilizando o iterador implementado. Neste exemplo, o iterador é gerado sobre uma imagem do tipo

TeRaster (Image) e recebe como parâmetros o polígono limitante do tipo TePolygon (poly) e a estratégia de percurso (TeBoxPixelIn).

```
TeRaster::iteratorPoly it = Image->begin(poly, TeBoxPixelIn);
double sum=0.0;
while(it != Image->end(poly, TeBoxPixelIn))
{
      sum += (*it)[0];  //dereferência
      ++it;             //incremento
}
```

Foram implementadas quatro estratégias de percurso para o iterador:

- TeBoxPixelIn: percorre todas as células do *raster* cujo centro é interno ao polígono;
- TeBBoxPixelInters: percorre todas as células do *raster* cujo segmento horizontal que passa pelo seu centro possui alguma interseção com o polígono;
- TeBoxPixelOut: percorre todas as células do *raster* cujo centro é externo ao polígono;
- TeBBoxPixelNotInters: percorre todas as células do *raster* cujo segmento horizontal que passa pelo seu centro não possui nenhuma interseção com o polígono.

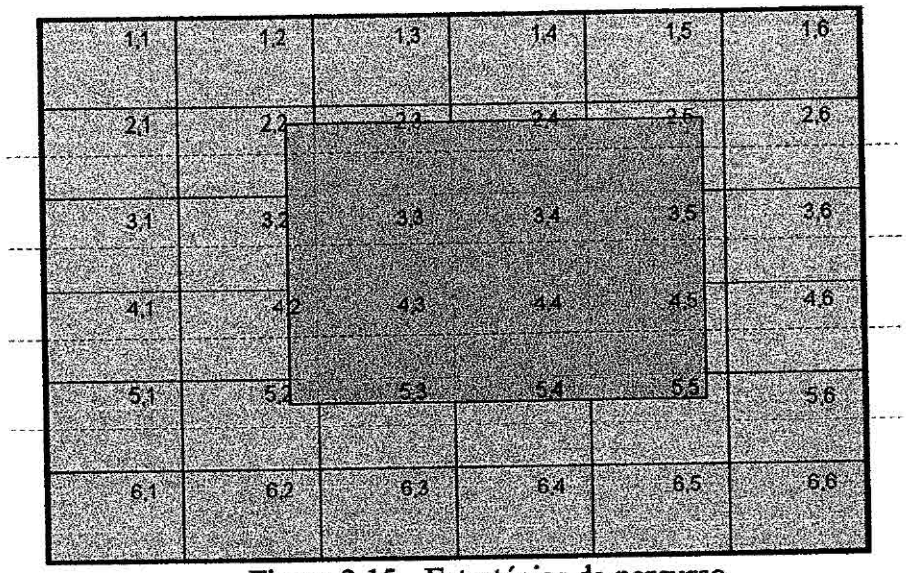


Figura 3.15 - Estratégias de percurso

Para ilustrar cada estratégia, considera-se a Figura 3.15, onde a matriz em cinza é um dado *raster*, o retângulo em azul é o polígono limitante e as retas pontilhadas vermelhas são os segmentos, para cada linha do *raster*, que passam pelo centro de todas as células dessa linha. As células percorridas segundo os critérios são:

- `TeBoxPixelIn`: (2,3), (2,4), (2,5), (3,3), (3,4), (3,5), (4,3), (4,4) e (4,5);
- `TeBBoxPixelInters`: (2,2), (2,3), (2,4), (2,5), (3,2), (3,3), (3,4), (3,5), (4,2), (4,3), (4,4) e (4,5);
- `TeBoxPixelOut`: (1,1), (1,2), (1,3), (1,4), (1,5), (1,6), (2,1), (2,2), (2,6), (3,1), (3,2), (3,6), (4,1), (4,2), (4,6), (5,1), (5,2), (5,3), (5,4), (5,5), (5,6), (6,1), (6,2), (6,3), (6,4), (6,5) e (6,6).
- `TeBBoxPixelNotInters`: (1,1), (1,2), (1,3), (1,4), (1,5), (1,6), (2,1), (2,6), (3,1), (3,6), (4,1), (4,6), (5,1), (5,2), (5,3), (5,4), (5,5), (5,6), (6,1), (6,2), (6,3), (6,4), (6,5) e (6,6).

Para calcular o segmento de interseção entre uma determinada linha do geo-campo e o polígono, é utilizada uma função chamada `TeGetIntersections`. Essa função é utilizada pelo método `getNewSegment` do **`iteratorPoly`** e retorna a seqüência

dos segmentos resultantes em uma estrutura chamada `TeCoordPairVect`. Assim, cada estratégia foi implementada como um objeto função, ou *functor* (Austern, 1998), o qual é aplicado na estrutura `TeCoordPairVect` resultante da função `TeGetIntersections`. O Anexo C mostra a implementação em C++ dessas estratégias como objeto função e sua utilização pela função `applyStrategic`.

A operação Zonal trabalha somente com as regiões do *raster* internas às zonas, que são representadas por polígonos. Portanto, a função `TeRasterStatisticsInPoly` utiliza sempre o **`iteratorPoly`** com a estratégia de percurso `TeBoxPixelIn`, como mostrado no Anexo B. Já a operação Mask, função `TeMask`, recebe a estratégia como parâmetro, podendo recortar o *raster* a partir da região interna ou externa ao polígono.

### 3.5.2 Função de estatística genérica

A função para calcular as estatísticas da operação Zonal foi implementada de forma genérica para ser utilizada por qualquer estrutura de dados que possua um iterador capaz de percorrê-la, e não somente pelo tipo `TeRaster`. As assinaturas dessa função são mostradas abaixo:

```
template<typename It>  bool TeCalculateStatistics(It& itBegin,
It& itEnd, TeStatisticValMap& stat, int dim)

template<typename It>  bool TeCalculateStatistics(It& itBegin,
It& itEnd, TeStatisticsDimensionVect& stat)

template<typename It>  bool TeCalculateStatistics(It& itBegin,
It& itEnd, TeStatisticValMap& stat)
```

Todas as assinaturas da função `TeCalculateStatistics` recebem como parâmetros de entrada um iterador apontado para o início (*itBegin*) e outro para o final (*itEnd*) da estrutura de dados. As duas primeiras assinaturas são utilizadas para calcular as estatísticas de uma estrutura de dados que possui *n* dimensões. E a última,

para calcular as estatísticas de estruturas de dados que possuem apenas duas dimensões. A implementação em C++ dessa função é mostrada no Anexo B.

Para exemplificar, é mostrada abaixo a utilização da função `TeCalculateStatistics` pela função `TeRasterStatisticsInPoly`:

```
bool TeRasterStatisticsInPoly ( TePolygon& poly, TeRaster* raster,
                                int band, TeStatisticValMap& stat)
{
  TeRaster::iteratorPoly itBegin = raster->begin(poly, TeBoxPiIn);
  TeRaster::iteratorPoly itEnd = raster->end(poly, TeBoxPixelIn);
  return(TeCalculateStatistics(itBegin, itEnd, stat, band));
}
```

Sendo genérica, essa função poderá ser utilizada para várias estruturas de dados da TerraLib que possuem um iterador, como por exemplo, a estrutura `TeSelectedObjectMap`, que armazena os atributos referentes aos objetos de um *layer* (TerraLib, 2003). Assim, para calcular as estatísticas de um determinado atributo, deve-se apenas criar um iterador para o início e outro para o final dessa estrutura e passá-los como parâmetros para a função `TeCalculateStatistics`.

### 3.5.3 Operações no `TeDatabase`

As operações sobre dados matriciais implementadas na classe `TeDatabase` utilizam as funções `TeRasterStatisticsInPoly` e `TeMask`, já mostradas anteriormente. Suas assinaturas são:

`bool` **`Zonal`** `(const string& rasterTable, const string& actGeomTable, Keys& Ids, TeObjectStatistics & result);`

`bool` **`Zonal`** `(const string& rasterTable, const string& actGeomTable, const string& actCollTable, TeObjectStatistics& result);`

`bool` **`Zonal`** `(const string& rasterTable, TePolygon& poly, TeRasterStatistics& result);`

```
bool Mask (const string& rasterTable, const string&
actGeomTable, const string& objId, const string& nameLayerOut,
TeStrategicIterator st);

bool Mask (const string& rasterTable, TePolygon& poly, const
string& nameLayerOut, TeStrategicIterator st);
```

A operação **Zonal** recebe como parâmetros de entrada o nome da tabela espacial do tipo *raster* (*rasterTable*), onde está armazenado o dado matricial, e as geometrias que definem as zonas. Essas geometrias podem estar armazenadas em tabelas espaciais (*actGeomTable*) ou estar em memória (*poly*). Quando as geometrias estão em uma tabela espacial, a função retorna uma estrutura que faz o mapeamento de cada objeto geográfico dessa tabela para suas respectivas estatísticas (`TeObjectStatistics`).

A operação **Mask**, ou de Recorte, recebe como parâmetros de entrada o nome da tabela do tipo *raster* (*rasterTable*) onde está o *raster* que será recortado, a estratégia de percurso para definir a região que será recortada (*st*) e a geometria que irá definir a máscara do recorte. Essa geometria pode estar armazenada em uma tabela espacial (*actGeomTable*) ou estar em memória (*poly*). O resultado dessa operação é uma classe `TeLayer` que contém o *raster* recortado, cujo nome é passado como parâmetro (*nameLayerOut*).

A partir do nome da tabela do tipo *raster*, as funções recuperam o dado matricial do SGBD e criam um objeto do tipo `TeRaster`. As geometrias também são recuperadas do SGBD através do nome da tabela espacial e são mapeadas para a classe `TePolygon`. E, finalmente, essas operações utilizam as funções `TeRasterStatisticsInPoly` e `TeMask` sobre os objetos `TeRaster` e `TePolygon` gerados.

# CAPÍTULO 4

# UTILIZAÇÃO E RESULTADOS

## 4.1 Introdução

As operações implementadas na API foram testadas através do TerraView, um aplicativo geográfico construído sobre a TerraLib (Ferreira et al, 2002). Na Figura 4.4.1 é mostrada a tela de interface com o usuário desenvolvida para operações sobre dados vetoriais.

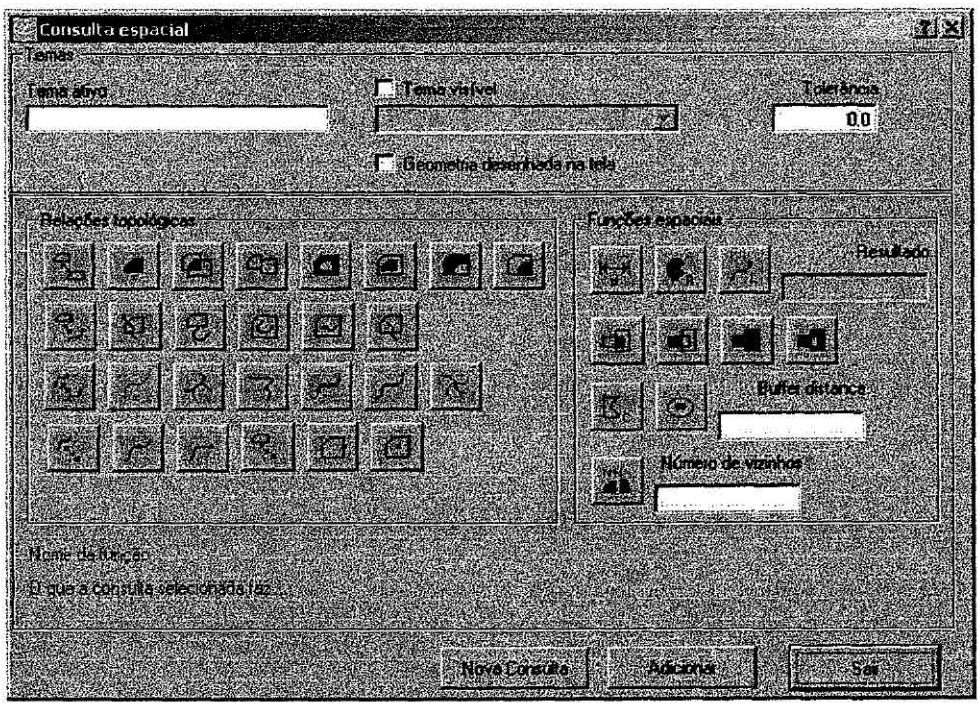



Figura 4.4.1 - Tela de operações espaciais sobre dados vetoriais

Na tela mostrada acima, o usuário escolhe os temas envolvidos e a operação que será executada. Um tema é um conceito da TerraLib que consiste em um subconjunto dos objetos geográficos de um determinado *layer*. As operações podem ser executadas entre os objetos de um mesmo tema ou de dois temas distintos.

As operações na tela são divididas em consultas sobre relações topológicas e funções espaciais. Os ícones das relações topológicas são habilitados conforme as representações geométricas dos temas selecionados. As funções espaciais disponíveis na tela são: área, comprimento ou perímetro, distância, geração de uma nova geometria a partir de uma distância em torno de uma geometria específica (*buffer*), operações de conjunto (interseção, união, diferença e diferença simétrica) e vizinhos mais próximos. Para cada operação selecionada é mostrada uma descrição de sua funcionalidade.

Para testar as operações sobre dados matriciais da API, foi desenvolvida uma interface gráfica no TerraView, mostrada na Figura 4.4.2. Nesta interface o usuário pode recortar ou gerar as estatísticas de um geo-campo, ou seja, de um tema de representação matricial, a partir de um polígono. Esse polígono pode estar em outro tema ou ter sido desenhado na tela. Para executar a operação de recorte, o usuário fornece outras informações como o nome do *layer* que será gerado e a estratégia de recorte, ou seja, se o dado *raster* será recortado a partir da região externa ou interna ao polígono.

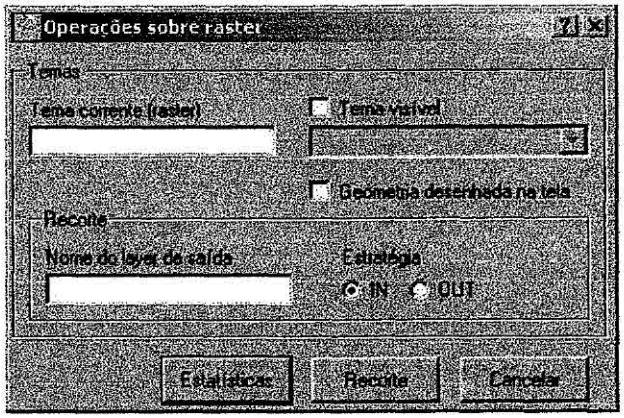



Figura 4.4.2 - Tela de operações espaciais sobre dados matriciais

## 4.2 Operações sobre dados vetoriais

A seguir, serão mostrados alguns exemplos de utilização das operações sobre dados vetoriais da API, apresentando seus resultados.

### 4.2.1 Consultas espaciais

Nesta seção, serão apresentados dois exemplos de consultas sobre relações topológicas, considerando dois temas, onde um é formado pelos estados (estados_Brasil) e o outro pelos rios (rios_Brasil) do Brasil. Esses dois temas foram construídos a partir dos *layers* mostrados na Figura 3.3.9.

Consulta 1) Encontre todos os estados do Brasil que fazem fronteira com o estado de Minas Gerais

```
//TeTheme* estados_Brasil
//TeDatabase* db (TeAdo, TeOracleSpatial, TeMySQL ou TePostgreSQL)
//typedef vector<string>  Keys

//informações sobre o tema estados_Brasil
TeGeomRep actRep = estados_Brasil->visibleRep();
string actTable = estados_Brasil->layer()->tableName(actRep);
string actColTable = estados_Brasil->collectionTable();

Keys IdsIn, IdsOut;
string obj_id = "MG";
IdsIn.push_back(obj_id);
string collName = "spatial_data";

int relate = TeTOUCHES;
int tol = 0.001;

db->spatialRelation(actTable, collName, actColTable, actRep, IdsIn,
IdsOut, relate, tol);
```

No código da consulta 1, as informações são extraídas do tema estados_Brasil: sua representação geométrica (`actRep`), o nome da tabela espacial que contém os objetos (`actTable`) e o nome da tabela de coleção onde estão armazenadas informações sobre os objetos do *layer* que fazem parte do tema (`actColTable`). O nome da coluna espacial é fixo nas tabelas geométricas do modelo da TerraLib (`collName` = "`spatial_data`"). O objeto do tipo `TeDatabase` (`db`) é instanciado conforme o

SGBD utilizado, podendo ser `TeAdo`, `TeOracleSpatial`, `TePostgreSQL` ou `TeMySQL` e, a partir do seu método **`spatialRelation`**, a consulta é executada.

Além dos parâmetros descritos acima, são passados para o método: a relação topológica a ser executada (*`relate`* = TeTOUCHES), o conjunto dos identificadores dos objetos sobre os quais a consulta será realizada (*`IdsIn`* = "MG"), a tolerância permitida (*`tol`* = 0.001) e a estrutura que conterá os identificadores dos objetos resultantes (*`IdsOut`*). Uma outra opção seria retornar todos os objetos geográficos resultantes da consulta, e não somente seus identificadores. Para isso, pode ser passado como parâmetro um objeto do tipo `TeDatabasePortal` (*`portal`*), o qual conterá as geometrias resultantes, ao invés de passar uma estrutura do tipo `Keys` (*`IdsOut`*), como mostrado abaixo:

```
TeDatabasePortal* portal = db->getPortal();
db->spatialRelation(actTable, collName, actColTable, actRep, IdsIn,
portal, relate, tol);
```

O resultado dessa consulta é mostrado nas figuras a seguir. A Figura 4.4.3 mostra o objeto selecionado na cor marrom, sobre o qual a consulta será realizada (estado de Minas Gerais). A Figura 4.4.4 mostra os objetos resultantes da consulta pintados de mostarda.

Figura 4.4.3 - Objeto selecionado

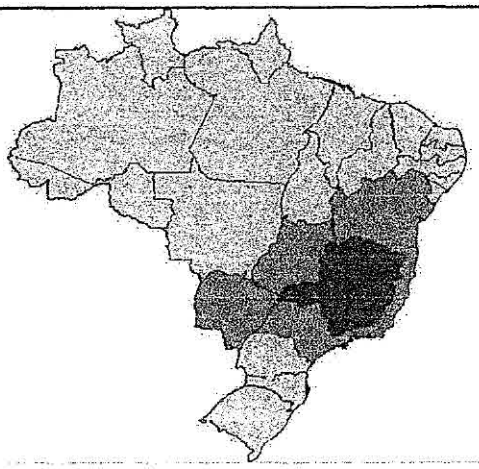


Figura 4.4.4 - Resultado da consulta

Consulta 2) Encontre todos os rios do Brasil que cruzam a fronteira do estado de Minas Gerais ou do Paraná.

```
//TeTheme* estados_Brasil, rios_Brasil
//TeDatabase* db (TeAdo, TeOracleSpatial, TeMySQL ou TePostgreSQL)
//typedef vector<string>  Keys

//informações sobre o tema estados_Brasil
TeGeomRep actRep = estados_Brasil->visibleRep();
string actTable = estados_Brasil->layer()->tableName(actRep);

//informações sobre o tema rios_Brasil
TeGeomRep visRep = rios_Brasil->visibleRep();
string visTable = rios_Brasil->layer()->tableName(visRep);
string visColTable = rios_Brasil->collectionTable();

string collName = "spatial_data";

Keys IdsIn, IdsOut;
string obj_id1 = "PR";
IdsIn.push_back(obj_id1);
string obj_id2 = "MG";
IdsIn.push_back(obj_id2);

int relate = TeCROSSES;
int tol = 0.001;

db->spatialRelation(actTable, collName, actRep, IdsIn, visTable,
collName, visColTable, visRep, IdsOut, relate, tol);
```

No código da consulta 2, o método **spatialRelation** recebe as informações sobre os dois temas envolvidos, estados_Brasil e rios_Brasil, como parâmetros de entrada, que são: suas representações geométricas (*actRep* e *visRep*), os nomes das tabelas e das colunas espaciais que contêm os objetos (*actTable*, *visTable* e *collName*) e o nome da tabela de coleção onde estão armazenadas informações sobre os objetos do *layer* que fazem parte do tema rios_Brasil (*visColTable*). O nome das

colunas espaciais de todas tabelas geométricas do modelo da TerraLib é fixo (*collName* = "*spatial_data*"). Além disso, são passados como parâmetros: a relação topológica a ser executada (*relate* = TeCROSSES), o conjunto dos identificadores dos objetos sobre os quais a consulta será realizada (*IdsIn* = "MG" e "PR"), a tolerância permitida (*tol* = 0.001) e a estrutura que conterá os identificadores dos objetos resultantes (*IdsOut*). Há também a opção de retornar todos os objetos geográficos resultantes, e não somente seus identificadores, passando um objeto do tipo `TeDatabasePortal` (*portal*), como mostrado abaixo:

```
TeDatabasePortal* portal = db->getPortal();
db->spatialRelation(actTable, collName, actRep, IdsIn, visTable,
collName, visColTable, visRep, portal, relate, tol);
```

O resultado dessa consulta é mostrado nas figuras a seguir. A Figura 4.4.5 mostra os objetos selecionados na cor amarela, sobre os quais a consulta será realizada (estados Minas Gerais e Paraná). A Figura 4.4.6 mostra os objetos, no caso os rios, resultantes da consulta, pintados de vermelho.

Figura 4.4.5 - Objetos selecionados

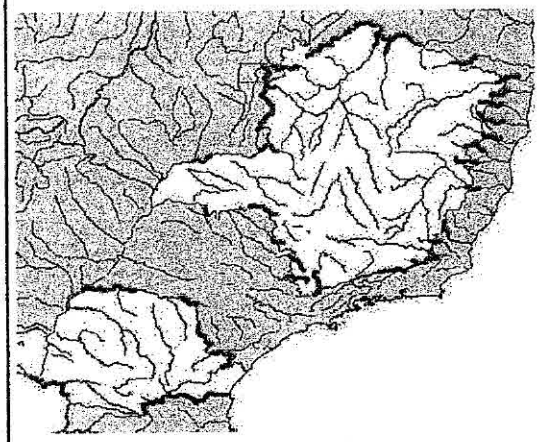


Figura 4.4.6 - Resultado da consulta

### 4.2.2 Funções espaciais

Nesta seção serão apresentados dois exemplos da utilização das operações **Buffer** e **nearestNeighbors** da API.

Exemplo 1) Gerar uma nova geometria a partir de uma distância de 0.5 ao redor dos rios Itaí e Paragua, cujos identificadores são "1140" e "1255", respectivamente.

```
//TeTheme* rios_Brasil
//TeDatabase* db = new TeOracleSpatial()
//typedef vector<string>  Keys

//informações sobre o tema rios_Brasil
TeGeomRep actRep = rios_Brasil->visibleRep();
string actTable = rios_Brasil->layer()->tableName(actRep);

string collName = "spatial_data";
TePolygonSet bufferSet;
double dist = 0.5;

Keys IdsIn;
string obj_id1 = "1140";
IdsIn.push_back(obj_id1);
string obj_id2 = "1255";
IdsIn.push_back(obj_id2);

db->Buffer(actTable, collName, actRep, IdsIn, bufferSet, dist);
```

O método **Buffer** do TeDatabase é utilizado para essa operação e recebe como parâmetros de entrada: o nome da tabela e da coluna espacial (*actTable* e *collName*) onde estão armazenados os objetos do tema; sua representação geométrica (*actRep*); o conjunto dos identificadores dos objetos sobre os quais as novas geometrias serão geradas (*IdsIn* = "1140" e "1255"); a distância considerada (*dist* = 0.5); e, a estrutura que conterá as geometrias geradas (*bufferSet*), que deve ser do tipo TePolygonSet. Como a função para gerar uma nova geometria a partir de uma

distância em torno de uma geometria específica (*buffer*) ainda não está implementada na TerraLib, o `TeDatabase` tem que ser do tipo `TeOracleSpatial` para realizar essa operação. Os resultados são mostrados na Figura 4.7.

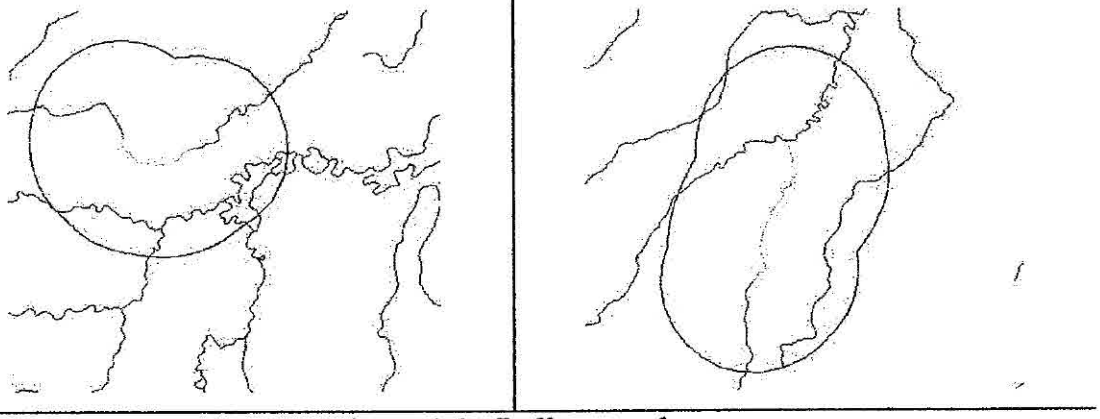


Figura 4.7 - *Buffers* gerados

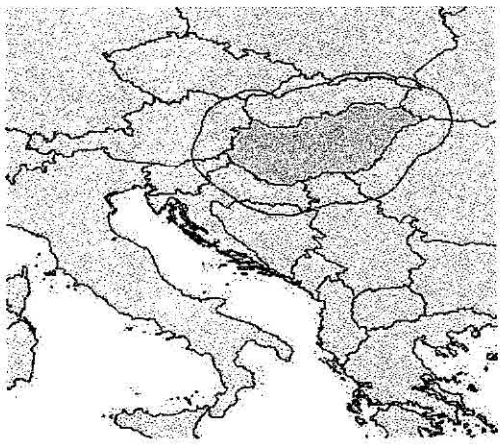


Figura 4.8 - *Buffer* do país Hungria

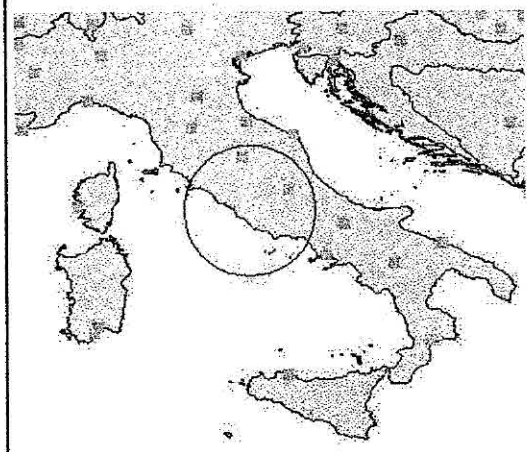


Figura 4.9 - *Buffer* da cidade de Roma

Para ilustrar a geração de novas geometrias ao redor de polígonos e pontos, são mostradas na Figura 4.8 uma geometria gerada ao redor do país Hungria e na Figura 4.9 gerada ao redor da cidade de Roma.

Exemplo 2) Encontre os 10 rios mais próximos ao rio Tapaua, cujo identificador é "1562".

```
//TeTheme* rios_Brasil
//TeDatabase* db = new TeOracleSpatial()
```

```
//informações sobre o tema rios_Brasil
TeGeomRep actRep = rios_Brasil->visibleRep();
string actTable = rios_Brasil->layer()->tableName(actRep);
string actColTable = rios_Brasil->collectionTable();

string collName = "spatial_data";

string objId = "1562";
Keys IdsOut;
int num_result = 10;

db->nearestNeighbors(actTable, collName, actColTable, actRep,
objId, IdsOut, num_result);
```

Para a operação de vizinho mais próximo é utilizado o método **nearestNeighbors** do TeDatabase. Esse método recebe como parâmetros de entrada: o nome da tabela e da coluna espacial (*actTable* e *collName*) onde estão armazenados os objetos do *layer*; o nome da tabela de coleção onde estão as informações sobre o subconjunto desses objetos que fazem parte do tema (*actColTable*); sua representação geométrica (*actRep*); o identificador do objeto sobre o qual a operação será realizada (*objId* = "1562"); o número de vizinhos esperados (*num_result*); e, a estrutura que conterá os identificadores dos objetos resultantes (*IdsOut*). Há também a opção de retornar todos os objetos geográficos resultantes, e não somente seus identificadores, passando um objeto do tipo TeDatabasePortal (*portal*), como mostrado abaixo:

```
TeDatabasePortal* portal = db->getPortal();
db->nearestNeighbors(actTable, collName, actColTable, actRep,
objId, portal, num_result);
```

A função para encontrar os vizinhos mais próximos de um objeto geográfico ainda não foi implementada na TerraLib, portanto para realizar essa operação o TeDatabase tem que ser do tipo TeOracleSpatial. A Figura 4.10 mostra o resultado dessa

operação, onde o rio Tapaua está selecionado de amarelo e seus 10 vizinhos mais próximos estão pintados de vermelho.

Ainda para ilustrar a operação de vizinho mais próximo, a Figura 4.11 mostra essa operação sendo aplicada a dois temas distintos, de rios e cidades da Europa. Essa figura ilustra o resultado da operação "encontre os cinco rios mais próximos da cidade de Roma". A cidade de Roma está selecionada de verde e os rios resultantes foram pintados de vermelho.

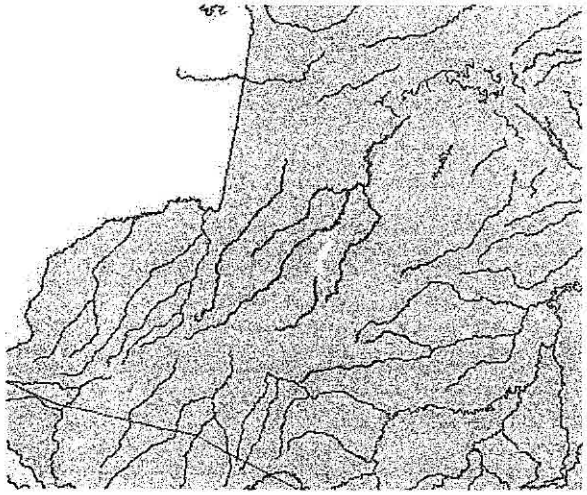


Figura 4.10 - Rios mais próximos do rio Tapaua

Figura 4.11 - Rios mais próximos da cidade de Roma

## 4.3 Operações sobre dados matriciais

Nesta seção, serão mostrados alguns exemplos de utilização das operações sobre dados matriciais da API, apresentando seus resultados.

### 4.3.1 Operação `Mask`

A operação `Mask` da API consiste em recortar um *raster* a partir de uma máscara definida por um polígono. O recorte pode ser feito a partir da região interna ou externa ao polígono, dependendo da estratégia escolhida. Esta operação gera um novo *layer* que contém o *raster* recortado.

Como exemplo, serão mostradas as operações para recortar uma imagem de Brasília, que está representada em um tema chamado Imagem_Brasilia, a partir de um polígono de um tema chamado Poligonos_Mask. A Figura 4.12 mostra esses dois temas sobrepostos, onde o polígono pintado de verde define a máscara do recorte.

Operação 1) Recortar a imagem de Brasília a partir da região interna ao polígono cujo identificador é "polígono1":

```
//TeTheme* Imagem_Brasilia
//TeTheme* Poligonos_Mask
//TeDatabase* db (TeAdo,TeOracleSpatial,TeMySQL ou TePostgreSQL)

//informações sobre o tema Imagem_Brasilia
string rasterTable  =   Imagem_Brasilia->
                        layer()->tableName(TeRASTER);

 //nome do layer de saída
 string nameLayerOut = "Imagem_Brasilia_Mask_In";

 //estratégia
 TeStrategicIterator strat = TeBoxPixelIn;

//identificador do polígono máscara
string objId = "polígono1";

// informações sobre o tema Poligonos_Mask
string geomTable    =   Poligonos_Mask->
                        layer()->tableName(TePOLYGONS);

db->Mask(rasterTable, geomTable, objId, nameLayerOut, strat);
```

No código da operação 1 mostrado acima, os nomes das tabelas onde estão armazenados o dado *raster* (`rasterTable`) e o polígono que define a máscara (`geomTable`) são extraídos dos temas Imagem_Brasilia e Poligonos_Mask, respectivamente. A operação **Mask** é um método do objeto `TeDatabase` (*db*) que é instanciado conforme o SGBD utilizado, podendo ser um `TeAdo`, `TeOracleSpatial`, `TePostgreSQL` ou

TeMySQL. Além dos nomes das tabelas, esse método recebe como parâmetros de entrada: o identificador do polígono que define a máscara (*objId* = "polígono1"), o nome do *layer* que será gerado como resultado (*nameLayerOut* = "Imagem_Brasilia_Mask_In") e a estratégia utilizada no recorte, que nesta operação é a partir da região interna à máscara (*strat* = TeBoxPixelIn). O *layer* resultante dessa operação é mostrado na Figura 4.13.

Operação 2) Recortar a imagem de Brasília a partir da região externa ao polígono cujo identificador é "polígono1":

```
//TeTheme* Imagem_Brasilia
//TeTheme* Poligonos_Mask
//TeDatabase* db (TeAdo,TeOracleSpatial,TePostgreSQL ou TeMySQL)

//informações sobre o tema Imagem_Brasilia
string rasterTable  =   Imagem_Brasilia->
                        layer()->tableName(TeRASTER);

 //nome do layer de saída
 string nameLayerOut = "Imagem_Brasilia_Mask_Out";

 //estratégia
 TeStrategicIterator strat = TeBoxPixelOut;

//identificador do polígono máscara
string objId = "polígono1";

// informações sobre o tema Poligonos_Mask
string geomTable    =   Poligonos_Mask->
                        layer()->tableName(TePOLYGONS);

db->Mask(rasterTable, geomTable, objId, nameLayerOut, strat);
```

O código mostrado acima é semelhante ao código da operação 1, só mudando o nome do *layer* gerado como resultado (*strat* = "Imagem_Brasilia_Mask_Out") e a

estratégia utilizada no recorte, que nesta operação é a partir da região externa à máscara (`strat = TeBoxPixelOut`). O *layer* resultante dessa operação é mostrado na Figura 4.14.

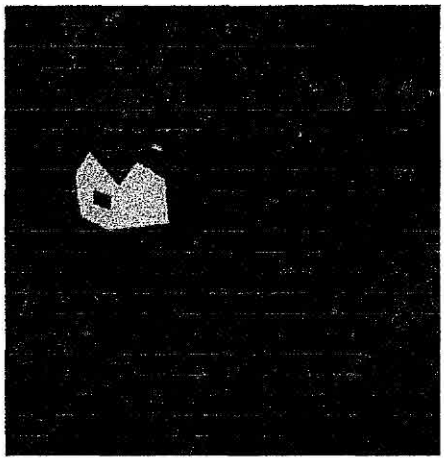


Figura 4.12 - Imagem de Brasília

Figura 4.13 - *layer* Imagem_Brasilia_Mask_In

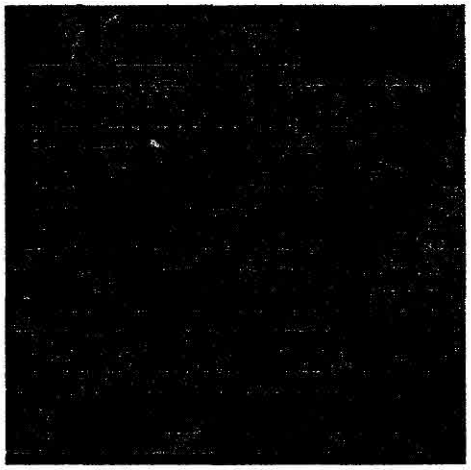


Figura 4.14 - *layer* Imagem_Brasilia_Mask_Out

### 4.3.2 Operação `Zonal`

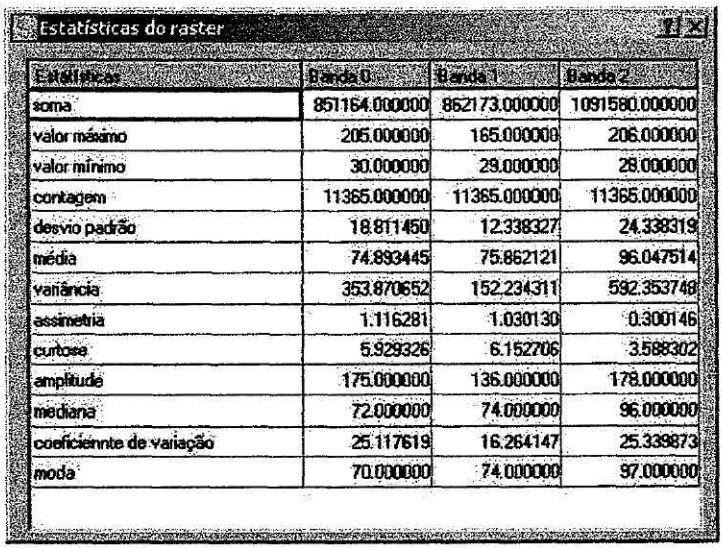
A operação `Zonal` consiste em calcular as estatísticas de uma determinada região de um geo-campo delimitada por um polígono. Como exemplo, será mostrado o código

para a operação "Calcule as estatísticas da imagem de Brasília da zona delimitada pelo polígono cujo identificador é 'poligono1'". Essa operação envolve os temas Imagem_Brasilia e Poligonos_Mask que contêm, respectivamente, a imagem de Brasília e o polígono que define a zona de interesse. A sobreposição desses temas é mostrada na Figura 4.12 e o código é mostrado a seguir:

```
//TeTheme* Imagem_Brasilia
//TeTheme* Poligonos_Mask
//TeDatabase* db (TeAdo,TeOracleSpatial,TePostgreSQL ou TeMySQL)
//typedef vector<string> Keys
//typedef map<string, TeStatisticsDimensionVect>
                                              TeObjectStatistics
TeObjectStatistics result;
Keys Ids;

//informações sobre o tema Imagem_Brasilia
string rasterTable  =   Imagem_Brasilia->
                        layer()->tableName(TeRASTER);

//identificador do polígono máscara
string Id = "poligono1";
Ids.push_back(Id);

// informações sobre o tema Poligonos_Mask
string geomTable    =   Poligonos_Mask->
                        layer()->tableName(TePOLYGONS);

db->Zonal(rasterTable, geomTable, Ids, result);
```

A operação `Zonal` é executada através de um método do objeto `TeDatabase` (*db*), que é instanciado conforme o SGBD utilizado, podendo ser um `TeAdo`, `TeOracleSpatial`, `TePostgreSQL` ou `TeMySQL`. Esse método recebe como parâmetros de entrada: os nomes das tabelas onde estão armazenados o dado *raster* (*rasterTable*) e os polígonos que definem as zonas (*geomTable*). Como mostrado

no código acima, essas informações são extraídas dos temas correspondentes. Além desses parâmetros, o método recebe: os identificadores dos polígonos que definem as zonas (*Ids*) e a estrutura de dados que conterá os resultados (`result`), que é do tipo `TeObjectStatistics`, a qual faz o mapeamento de cada zona e suas estatísticas correspondentes. Na Figura 4.15, é mostrada uma interface gráfica desenvolvida no TerraView para mostrar as estatísticas geradas nessas operações. Essa interface apresenta o conjunto de estatísticas para cada banda da imagem.




| Estatísticas | Banda 0 | Banda 1 | Banda 2 |
|---|---|---|---|
| soma | 851164.000000 | 862173.000000 | 1091580.000000 |
| valor máximo | 205.000000 | 165.000000 | 206.000000 |
| valor mínimo | 30.000000 | 29.000000 | 28.000000 |
| contagem | 11365.000000 | 11365.000000 | 11365.000000 |
| desvio padrão | 18.811450 | 12.338327 | 24.338319 |
| média | 74.893445 | 75.862121 | 96.047514 |
| variância | 353.870652 | 152.234311 | 592.353748 |
| assimetria | 1.116281 | 1.030130 | 0.300146 |
| curtose | 5.929326 | 6.152706 | 3.588302 |
| amplitude | 175.000000 | 136.000000 | 178.000000 |
| mediana | 72.000000 | 74.000000 | 96.000000 |
| coeficiennte de variação | 25.117619 | 16.264147 | 25.339873 |
| moda | 70.000000 | 74.000000 | 97.000000 |

Figura 4.15 - Tela de estatística

# CAPÍTULO 5

# CONCLUSÃO E TRABALHOS FUTUROS

Este trabalho apresentou o desenvolvimento de uma API (*Application Programming Interface*) para operações espaciais em banco de dados geográficos, implementada na TerraLib, uma biblioteca base para a construção de aplicativos geográficos de arquitetura integrada. Essa nova arquitetura consiste em utilizar SGBDs para armazenar e gerenciar dados geográficos, explorando seus recursos como controle de integridade e concorrência, gerência de transações, restrições de acessos e mecanismos para recuperação de falhas.

Com relação aos objetivos propostos, a API desenvolvida permite encapsular diferentes implementações de tipos de dados espaciais realizadas pelos diferentes fabricantes de SGBD e ainda estende as facilidades disponíveis nestes sistemas, como no caso do suporte a dados matriciais no Oracle Spatial. Como a TerraLib é uma biblioteca base para a construção desses aplicativos, a API desenvolvida fornece tais funcionalidades a um nível maior de abstração para os desenvolvedores de sistemas geográficos. Assim, esse desenvolvedor pode utilizar as operações espaciais disponíveis na biblioteca sem precisar ter conhecimento de como são computadas, podendo então, se dedicar à implementação de outras funcionalidades, como por exemplo, visualização dos dados, interface com o usuário e ferramentas para análise espacial.

Abaixo são citadas algumas restrições da API desenvolvida, que serão exploradas em trabalhos futuros:

(1) Algumas operações, quando aplicadas a dados geográficos armazenados em SGBDs que não possuem extensão espacial, não são realizadas, pois ainda não foram implementadas na TerraLib para dados em memória. Dentre essas operações estão: cálculo dos vizinhos mais próximos de um objeto; geração de

uma nova geometria a partir de uma distância em torno de uma geometria específica (*buffer*) e operações de conjunto.

(2) Há outras operações sobre dados matriciais que não estão disponíveis na API como: operações matemáticas entre dois geo-campos, reclassificação, fatiamento, ponderação e operações focais ou de vizinhança.

Ainda como trabalhos futuros, podemos citar alterações nas operações sobre dados matriciais da API quando as extensões espaciais dos SGBDORs começarem a oferecer recursos para tratar esses dados. A proposta de tratar dados matriciais já foi feita por algumas extensões, como por exemplo, Oracle Spatial.

# CAPÍTULO 6

## REFERÊNCIAS BIBLIOGRÁFICAS


Austern, M. H. **Generic Programming and the STL: Using and Extending the C++ Standard Template Library**. Massachusetts: Addison-Weslwy, 1998. 548 p.

Booch, G. **Object-Oriented Analysis and Design - with applications**. California: Benjamin/Cummings Publishing Company, 1994. 500 p.

Burrough, P. A.; McDonnell, R. A. **Principles of Geographical Information Systems**. New York: Oxford University Press, 1998. 332 p.

Câmara, G. **Modelos, Linguagens e Arquiteturas para Banco de Dados Geográficos**. São José dos Campos, SP. Tese (Doutorado em Computação Aplicada ) - INPE, 1995.

Câmara, G.; Casanova, M. A.; Hemerly, A. S.; Magalhães, G. C.; Medeiros, C. B. **Anatomia de Sistemas de Informação Geográfica**. Campinas: 10ª Escola de Computação, 1996. 193 p.

Câmara, G.; Souza, R. C. M.; Pedrosa, B.; Vinhas, L.; Monteiro A. M.; Paiva, J. A. C. P.; Gattas, M. **TerraLib: Technology in Support of GIS Innovation**. 2000. II Workshop Brasileiro de Geoinformática.

Câmara, G.; Vinhas, L.; Souza, R. C. M.; Paiva, J. A. C.; Monteiro, A. M. V.; Carvalho, M. T.; Raoult, B. **Design Patterns in GIS Development: The Terralib Experience**. 2001. III Workshop Brasileiro de Geoinformática.

Cordeiro, J.; Amaral, S.; Freitas, U.; Câmara, G. **Álgebra de Geo-Campos e suas Aplicações**. 1996. VIII Simpósio Brasileiro de Sensoriamento Remoto.

Drexel University. The Math Forum [online], 2003. <http://mathforum.org/dr.math/>. Visitado em fevereiro de 2003.

Egenhofer, M. **Spatial Information Appliances: A Next Generation of Geographic Information Systems**. 1999. First Brazilian Workshop on GeoInformatics .

Egenhofer, M. A Model for Detailed Binary Topological Relationships. **Geomatica**, v. 47, n. 3 & 4, p. 261-273, 1993.

Egenhofer, M. J. Spatial SQL: A Query and Presentation Language. **IEEE Transactions on Knowledge and Data Engineering** , v. 6, n. 1, p. 86-95, 1994.

Egenhofer, M. J.; Herring, J. R. **Categorizing Binary Topological Relations Between Regions, Lines, and Points in Geographic Databases**. Maine, USA: University of Maine, 1991.

Egenhofer, M.; Clementini, E.; Di Felice, P. Topological relations between regions with holes. **International Journal of Geographical Information Systems** , v. 8, n. 2, p. 129-144, 1994.

Egenhofer, M.; Franzosa, R. On the Equivalence of Topological Relations. **International Journal of Geographical Information Systems** , v. 9, n. 2, p. 133-152, 1995.

ESRI **ArcSDE**[online]. <http://arcsdeonline.esri.com/index.htm>. Mar. 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**. p. 57-67, 2002. XVII Simpósio Brasileiro de Banco de Dados.

Frank, A. U. Requirements for Database Systems Suitable to Manage Large Spatial Databases. **Photogrammetric Engineering & Remote Sensing** , v. 11, n. 54, p. 1557-1564, 1988.

Gamma, E.; Helm, R.; Johnson R.; Vlissides, J. **Design patterns - elements of reusable object-oriented software**. USA: Addison-Wesley, 1995.

Güting, R. An Introduction to Spatial Database Systems. **VLDB Journal**, v. 3, 1994.

Goodchild, M. F. Geographical Data Modeling. **Computers and Geoscience** , v. 18, n. 4, p. 401-408, 1992.

Guttman, A. **R-TREES. A Dynamic Index Structure for Spatial Searching**. In: Proc. SIGMOD Conf. Boston: ACM, 1984. p. 47-57.

IBM Corporation. **DB2 Spatial Extender User's Guide and Reference**[online]. <http://www-3.ibm.com> 2001.

IBM Corporation. **Working with the Geodetic and Spatial DataBlade Modules**[online]. <http://www-3.ibm.com/software/data/informix/pubs/manuals/geo_spatial.pdf> 2002.

Korth, F. H.; Silberschatz, A. **Sistemas de Bancos de Dados**. São Paulo: McGraw-Hill, 1994. 693 p.

Köthe, U. STL-Style Generic Programming with Images. **C++ Report Magazine**, v. 12, n. 1. Jan. 2000.

Lassen, A. R.; Olsson, J.; Osterbye, K. **Object Relational Modeling**. Centre for Object Technology (COT), 1998. 32 p.

Locke, P.; Belden, E.; Melnick J. **Oracle Call Interface - Programmer's Guide**. : Oracle Corporation , Mar. 1999. (No. A76975-01)

Mattos, N. **An Overview of the SQL3 Standard** . IBM Santa Teresa Lab, San Jose, CA: Database Technology Institute , 1996.

Mitrovic, A.; Djordjevic, S. **Object-Oriented paradigm meets GIS: a new era in spatial data management**. p. 141-148, 1996. YUGIS, Beograd.

Murray, C. **Oracle® Spatial - User's Guide and Reference**. Redwood City, CA: Oracle Corporation, 2001. (Part N° A88805-01)

NIST **Announcing the Standard for Database Language SQL**: National Institute of Standards and Technology, 1993. (Federal Information Processing Standards Publication 127-2)

OGC **OpenGIS Simple Features Specification For SQL** : OpenGIS Consortium, Inc, 1999. (OpenGIS Project Document 99-049)

OGC **The OpenGIS™ Guide - Introduction to Interoperable Geoprocessing**. Massachusetts, USA: Open GIS Consortium , 1996.

Paiva, J. A. C. **Topological Equivalence and Similarity in Multi-Representation Geographic Database**. Maine, USA - University of Maine, 1998.

Ramsey, P. **PostGis Manual**[online]. <<http://postgis.refractions.net>> 2002.

Ravada, S.; Sharma, J. Oracle8i Spatial: Experiences with Extensible Databases. **SSD'99. R. H. Guting, D. Papadias and F. Lochovsky**, p. 355-359, 1999.

Rigaux, P.; Scholl, M.; Voisard, A. **Spatial Databases with application to GIS**. San Francisco: Morgan Kaufmann , 2002. 408 p.

Samet, H. The Quadtree and Related Hierarchical Data Structures. **ACM Computing Surveys**, v. 16, n. 2, p. 187-260, 1984.

Shekhar, S.; Ravada, S.; Liu, X. Spatial Databases - Accomplishments and Research Needs. **IEEE Transations on Knowledge and Data Engineering**, v. 11, n. 1, Fev. 1999.

Stroustrup, B. **The C++ Programming Language**. USA: Addison-Wesley, 1999.

**TerraLib**[online]. <www.TerraLib.org> 2003.

Tomlin, C. D. **Geographic Information Systems and Cartographical Modeling**. New York: Prentice-Hall , 1990.

Vinhas, L.; Queiroz, G. R.; Ferreira, K. R.; Câmara, G.; Paiva, J. A. C. Programação Genérica Aplicada a Algoritmos Geográficos. In: IV Simpósio Brasileiro de GeoInformática. **Anais**. Caxambu, MG, 2002. v.1, p.117-122.

Voisard, A.; Schweppe, H. **Abstraction and Decomposition in Open GIS** . Berkeley, California: International Computer Science Institute, 1997. (TR-97-006)

# ANEXO A

## FUNÇÕES PARA GERAÇÃO DE ARCO

```
//! Swaps to points
inline void TeSwap(TePoint& p1, TePoint& p2)
{
	TePoint temp;

	temp = p1;
	p1 = p2;
	p2 = temp;

	return;
}


//! Given three points of a circumference, returns the center point
bool TeGetCenter(TePoint p1, TePoint p2, TePoint p3, TePoint& center)
{
	double x, y;
	double ma, mb;
	bool result = true;

	if	(	(p1.location().x()==p2.location().x()) ||
			(p1.location().y()==p2.location().y())	)
		TeSwap(p2,p3);

	if	(p2.location().x()==p3.location().x())
		TeSwap(p1,p2);

	if	(p1.location().x()!=p2.location().x())
		ma =	(p2.location().y()-p1.location().y())/
			(p2.location().x()-p1.location().x());
	else
		result=false;

	if	(p2.location().x()!=p3.location().x())
		mb =	(p3.location().y()-p2.location().y())/
			(p3.location().x()-p2.location().x());
	else
		result=false;

	if ((ma==0) && (mb==0))
		result=false;
```

```
      if (result)
      {
            x = ( ma*mb*(p1.location().y()-p3.location().y())+
                  mb*(p1.location().x()+p2.location().x())-
                  ma*(p2.location().x()+p3.location().x()))/
                  (2*(mb-ma));
            y = - (x-(p1.location().x()+p2.location().x())/2)/ma +
                  (p1.location().y()+p2.location().y())/2;

            double w= TeRound(x);
            double z= TeRound(y);
            TeCoord2D coord(w,z);
            center.add(coord);
      }
  return result;
}


//! Given three points of a circumference, returns the radius
double TeGetRadius(TePoint& p1, TePoint& p2, TePoint& p3)
{
      double s,a,b,c, result;
      a =   sqrt(pow(p1.location().x()-p2.location().x(),2)+
            pow(p1.location().y()-p2.location().y(),2));

      b =   sqrt(pow(p2.location().x()-p3.location().x(),2)+
            pow(p2.location().y()-p3.location().y(),2));

      c =   sqrt(pow(p3.location().x()-p1.location().x(),2)+
            pow(p3.location().y()-p1.location().y(),2));

      s =   (a+b+c)/2;

      result = (a*b*c)/(4*sqrt(s*(s-a)*(s-b)*(s-c)));
      return result;
}


//! Given three points of a circumference, returns an arc passing for
//! this points that is formed by a given number of points.
bool TeGenerateArc (    TePoint& p1, TePoint& p2, TePoint& p3,
                        TeLine2D& arcOut, const short& Npoints )
{
      TePoint center;
      double radius;

      if(!TeGetCenter(p1, p2, p3, center))
            return false;

      radius = TeGetRadius(p1, p2, p3);
```

```
// calculate the distance between the points p1 and p3
double length = TeDistance(p1.location(),p3.location());

// calculate the angle (in radians) between the segments (p1-p3)
// in the circle center

double thetaR = 2 * asin(length/(2*radius));

//calculate the variation of the angle in radians for each point
double deltaR = thetaR/(NPoints+1);

//verify if is counterclockwise or clockwise
TeLine2D line;
line.add(p1.location());
line.add(p2.location());
line.add(p3.location());
line.add(p1.location());
TeLinearRing ring(line);

short orient = TeOrientation(ring);

// calculate the coseno and seno of the angle (beta) between the
// circumference center (horizontal segment) and the first point

double cosBeta =  (p1.location().x()-center.location().x())/
                  radius;

double sinBeta =  (p1.location().y()-center.location().y())/
                  radius;

arcOut.add(p1.location());
double angle = deltaR;
for (int i=0; i<(NPoints+1); i++)
{
      double x,y;
      if(orient==TeCOUNTERCLOCKWISE)
      {
            //Counterclockwise
            //x = cos(beta-angle)
            x =   center.location().x() + radius *
                  ((cosBeta*cos(angle))+(sinBeta*sin(angle)));

            //y = sin(beta-angle)
            y =   center.location().y() + radius *
                  ((sinBeta*cos(angle))-(cosBeta*sin(angle)));
      }
```

```
        else if (orient==TeCLOCKWISE)
        {
              //x = cos(beta+angle)
              x =   center.location().x() + radius *
                    ((cosBeta*cos(angle))-(sinBeta*sin(angle)));

              //y = sin(beta+angle)
              y =   center.location().y() + radius *
                    ((sinBeta*cos(angle))+(cosBeta*sin(angle)));
        }

        TeCoord2D coord(x,y);
        arcOut.add(coord);
        angle += deltaR;
    }
    arcOut.add(p3.location());
    return true;
}
```

# ANEXO B

## FUNÇÕES DE ESTATÍSTICAS PARA OPERAÇÃO ZONAL

```
//! Statistics types
enum TeStatisticType
{
      TeSUM, TeMAXVALUE, TeMINVALUE, TeCOUNT, TeSTANDARDDEVIATION,
      TeKERNEL, TeMEAN, TeVARIANCE, TeSKEWNESS, TeKURTOSIS,
      TeAMPLITUDE, TeMEDIAN, TeVARCOEFF, TeMODE
};


//! A mapping of a statistic to its value
typedef   map<TeStatisticType, double>   TeStatisticValMap;


//! A mapping of an object key to its statistics
typedef   map<string, TeStatisticValMap>   TeObjStatisticsMap;


//! Associate a statistics set for each dimension
struct TeStatisticsDimension
{
      int                     dimension_;
      TeStatisticValMap       stat_;

      TeStatisticsDimension(int d, TeStatisticValMap& st):
      dimension_(d),
      stat_(st)
      {}
};


//! vector of the statistics associated with the dimensions
typedef   vector<TeStatisticsDimension>   TeStatisticsDimensionVect;


//! mapping each value to its count: histogram
typedef  map<double, int>  TeHistogram;
```

```
//! Calculate the histogram of a set of values
template <typename T> bool
TeCalculateHistogram (T itBegin, T itEnd, TeHistogram& histOut)
{
      T it = itBegin;

      //initialization
      while(it!=itEnd)
      {
            double val = (*it);
            histOut[val] = 0;
            it++;
      }

      it= itBegin;

      //calcule histogram
      while(it!=itEnd)
      {
            double val = (*it);
            histOut[val]++;
            it++;
      }

      return true;
}


//!  Calculate the statistics of a specific dimension of a data
//!  structure
template<typename It>  bool
TeCalculateStatistics   (It& itBegin, It& itEnd,
                         TeStatisticValMap& stat, int dim)
{
      double      sum, mean, minValue, maxValue, variance, assim,
                  curtose, stDev, coefVar, amplitude, moda, median;
      sum=mean=variance=assim=curtose=stDev=coefVar=amplitude=0.0;

      minValue = TeMAXFLOAT;
      maxValue = TeMINFLOAT;

      vector<double> values;

      int count=0;
      It itt = itBegin;
      while(itt != itEnd)
      {
            double val = (*itt) [dim];

            values.push_back (val);
            sum += val;
```

```
        if(minValue>val)
                minValue = val;
        if(maxValue<val)
                maxValue = val;

        ++itt;
        ++count;

        mean = sum/count;
}

for(int i=0; i<count; i++)
{
        double v= values[i];
        variance += pow((v-mean),2);
        assim += pow((v-mean),3);
        curtose += pow((v-mean),4);
}

if(!count)
        return false;

variance /= count;
stDev = pow(variance,0.5);
assim /= count;
assim /= pow(stDev,3);
curtose /= count;
curtose /= pow(stDev,4);

coefVar = (100*stDev)/mean;
amplitude = maxValue-minValue;

sort(values.begin(), values.end());

//calculate the median
if((count%2)==0)
        median = (values[(count/2)]+values[(count/2-1)])/2;
else
        median = values[(count-1)/2];

//calculate the mode
TeHistogram histog;
TeCalculateHistogram(values.begin(),values.end(), histog);

TeHistogram::iterator itHist = histog.begin();
int nCount=0;
while(itHist!=histog.end())
{
        int hCount = itHist->second;
        if (hCount > nCount)
        {
                nCount = hCount;
```

```
                moda = itHist->first;
            }
            itHist++;
        }

        //filling out the results
        stat[TeCOUNT] = count;
        stat[TeMINVALUE] = minValue;
        stat[TeMAXVALUE] = maxValue;
        stat[TeSUM] = sum;
        stat[TeMEAN] = mean;
        stat[TeSTANDARDDEVIATION] = stDev;
        stat[TeVARIANCE] = variance;
        stat[TeSKEWNESS] = assim;
        stat[TeKURTOSIS] = curtose;
        stat[TeAMPLITUDE] = amplitude;
        stat[TeMEDIAN] = median;
        stat[TeVARCOEFF] = coefVar;
        stat[TeMODE] = moda;

        return true;
}


//!   Calculate the statistics of all dimensions of a data structure
template <typename It> bool
TeCalculateStatistics ( It& itBegin, It& itEnd,
                        TeStatisticsDimensionVect& stat)
{

        vector<double>    sum, mean, minValue, maxValue, variance, assim,
                          curtose, stDev, coefVar, amplitude, moda,
                          median;

        int nb = itBegin.nBands();

        //initialization of the vetors
        for (int i=0; i<nb; i++)
        {
            sum.push_back(0.0);
            mean.push_back(0.0);
            variance.push_back(0.0);
            assim.push_back(0.0);
            curtose.push_back(0.0);
            stDev.push_back(0.0);
            coefVar.push_back(0.0);
            amplitude.push_back(0.0);
            median.push_back (0.0);
            moda.push_back (0.0);
            minValue.push_back(TeMAXFLOAT);
            maxValue.push_back(TeMINFLOAT);
        }
```

```
map<int,stats> bandValues;

int count=0;
It itt = itBegin;

while(itt != itEnd)
{
      for (int j=0; j<nb; j++)
      {
            double val = (*itt)[j];
            bandValues[j].push_back(val);
            sum[j] += val;

            if(minValue[j]>val)
                  minValue[j] = val;
            if(maxValue[j]<val)
                  maxValue[j] = val;

            int size = bandValues[j].size();
            mean[j] = sum[j]/size;
      }

      ++itt;
      ++count;
}

if(!count)
      return false;

for (int jj=0; jj<nb; jj++)
{
      for(int i=0; i<count; i++)
      {
            double v = bandValues[jj][i];
            variance[jj] += pow((v-mean[jj]),2);
            assim[jj] += pow((v-mean[jj]),3);
            curtose[jj] += pow((v-mean[jj]),4);
      }

      variance[jj] /= count;
      stDev[jj] = pow(variance[jj],0.5);
      assim[jj] /= count;
      assim[jj] /= pow(stDev[jj],3);
      curtose[jj] /= count;
      curtose[jj] /= pow(stDev[jj],4);

      coefVar[jj] = (100*stDev[jj])/mean[jj];
      amplitude[jj] = maxValue[jj]-minValue[jj];

      sort(bandValues[jj].begin(), bandValues[jj].end());
```

```
        //calculate the median
        if((count%2)==0)
              median[jj] = ((bandValues[jj][(count/2)])+
                           (bandValues[jj][(count/2-1)]))/2;
        else
              median[jj] = bandValues[jj][(count-1)/2];

        //calculate the mode
        TeHistogram histog;
        TeCalculateHistogram(   bandValues[jj].begin(),
                                bandValues[jj].end(), histog);

        TeHistogram::iterator itHist = histog.begin();
        int nCount=0;
        while(itHist!=histog.end())
        {
              int hCount = itHist->second;
              if (hCount > nCount)
              {
                    nCount = hCount;
                    moda[jj] = itHist->first;
              }
              itHist++;
        }

        //filling out the results
        TeStatisticValMap statVal;

        statVal[TeCOUNT] = count;
        statVal[TeMINVALUE] = minValue[jj];
        statVal[TeMAXVALUE] = maxValue[jj];
        statVal[TeSUM] = sum[jj];
        statVal[TeMEAN] = mean[jj];
        statVal[TeSTANDARDDEVIATION] = stDev[jj];
        statVal[TeVARIANCE] = variance[jj];
        statVal[TeSKEWNESS] = assim[jj];
        statVal[TeKURTOSIS] = curtose[jj];
        statVal[TeAMPLITUDE] = amplitude[jj];
        statVal[TeMEDIAN] = median[jj];
        statVal[TeVARCOEFF] = coefVar[jj];
        statVal[TeMODE] = moda[jj];

        TeStatisticsDimension statBand(jj, statVal);
        stat.push_back(statBand);
  }

  return true;
}
```

```
//! Calculate the statistics of a region (limited by a polygon) of a
//! raster for a specific band
bool
TeRasterStatisticsInPoly      (TePolygon& poly, TeRaster* raster,
                               int band, TeStatisticValMap& stat)
{

  TeRaster::iteratorPoly itBegin = raster->begin(poly, TeBoxPixelIn);
  TeRaster::iteratorPoly itEnd = raster->end(poly, TeBoxPixelIn);

  return (TeCalculateStatistics(itBegin, itEnd, stat, band));
}


//! Calculate the statistics of a region (limited by a polygon) of a
//! raster for each band
bool
TeRasterStatisticsInPoly      (TePolygon& poly, TeRaster* raster,
                               TeStatisticsDimensionVect& stat)
{

  TeRaster::iteratorPoly itBegin = raster->begin(poly, TeBoxPixelIn);
  TeRaster::iteratorPoly itEnd = raster->end(poly, TeBoxPixelIn);

  return (TeCalculateStatistics (itBegin, itEnd, stat));
}
```

# ANEXO C

## ITERADOR E ESTRATÉGIAS

```
//!   virtual class that define the strategics of the iteratorPoly,
//!   each strategic is a functor
class TeStrategic
{
      protected:
      TeRaster*         raster_;
      double                  y_;
      TeCoordPairVect   SegOut_;

      public:
      TeStrategic(TeRaster* r=0, double y=0): raster_(r), y_(y) {}

      void Init(TeRaster* r, double y)
      {
            raster_ = r;
            y_ = y;
      }

      virtual void strateg(double xMin, double xMax, double y) = 0;

      void operator() (TeCoordPair& pair)
      {
            //xmin and xmax of the segment (line and column index)
            double xMinSegCM = pair.pt1.x();
            double xMaxSegCM = pair.pt2.x();

            //line and column index of the segment
            TeCoord2D minSegCM(xMinSegCM, y_);
            TeCoord2D maxSegCM(xMaxSegCM, y_);

            TeCoord2D minSegLC = raster_->coord2Index (minSegCM);
            TeCoord2D maxSegLC = raster_->coord2Index (maxSegCM);

            double xMinLCd = minSegLC.x();
            double xMaxLCd = maxSegLC.x();
            double yLC = maxSegLC.y();

            //verify if it is negative
            if(xMinLCd<0)
            {
                  if(xMaxLCd<0)
                        return;
                  else
                        xMinLCd=0.5;
            }
```

```
            strateg(xMinLCd, xMaxLCd, yLC);
      }

      TeCoordPairVect result() const {return SegOut_;}
};


//!   functor TePixelBoxInPoly
class TePixelBoxInPoly: public TeStrategic
{

      public:
      TePixelBoxInPoly(TeRaster* r=0, double y=0): TeStrategic(r,y) {}

      void strateg(double xMin, double xMax, double y)
      {
            //verify if it contains the element center
            int xMinLCi = (int) xMin;
            int xMaxLCi = (int) xMax;

            if(xMin <= (xMinLCi+0.5))
                  xMin = xMinLCi+0.5;
            else
                  xMin = xMinLCi+1.5;

            if(xMax >= (xMaxLCi+0.5))
                  xMax = xMaxLCi+0.5;
            else
                  xMax = xMaxLCi-0.5;

            //new segment
            TeCoord2D minLC (xMin, y);
            TeCoord2D maxLC (xMax, y);

            TeCoordPair res;
            res.pt1 = minLC;
            res.pt2 = maxLC;

            SegOut_.push_back (res);
      }

};


//!   functor TePixelBBInterPoly
class TePixelBBInterPoly: public TeStrategic
{

      public:
      TePixelBBInterPoly(TeRaster* r=0, double y=0): TeStrategic(r,y)
                                                      {}
```

```
        void strateg(double xMin, double xMax, double y)
        {
                //element center
                int xMinLCi = (int) xMin;
                int xMaxLCi = (int) xMax;

                xMin = xMinLCi+0.5;
                xMax = xMaxLCi+0.5;

                //new segment
                TeCoord2D minLC (xMin, y);
                TeCoord2D maxLC (xMax, y);

                TeCoordPair res;
                res.pt1 = minLC;
                res.pt2 = maxLC;

                SegOut_.push_back (res);
        }
};

//! Apply the specific strategic
TeCoordPairVect
applyStrategic(   double& y, double ymin, double xmin,
                  TeStrategicIterator st, TeRaster* raster,
                  TePolygon& poly)
{
        TeCoordPairVect Segments = TeGetIntersections(poly, y);
        double resy = raster->params().resy_;

        //Inside polygon
        if((st==TeBoxPixelIn) || (st==TeBBoxPixelInters))
        {
                bool empty=false;
                if(Segments.empty())
                {
                        empty = true;
                        y -= resy;
                        while((y>=ymin) && (empty))
                        {
                                Segments = TeGetIntersections(poly, y);
                                if(!Segments.empty())
                                        empty = false;
                                else
                                        y -=resy;
                        }
                }
                if(!empty)
                {
                        if (st==TeBoxPixelIn)
                        {
```

```
                TePixelBoxInPoly strat(raster,y);
                strat = for_each( Segments.begin (),
                                  Segments.end(), strat);
                return strat.result();
            }
            else
            {
                TePixelBBInterPoly strat(raster,y);
                strat = for_each( Segments.begin (),
                                  Segments.end(), strat);
                return strat.result();
            }
        }
    }
    //Outside polygon
    else if((st==TeBoxPixelOut) || (st==TeBBoxPixelNotInters))
    {
        int nCols = raster->params().ncols_;

        if(Segments.empty())
        {
            // pass to line and column index
            TeCoord2D coordCM (xmin, y);
            double linLC = (raster->coord2Index(coordCM)).y();

            int lin = (int)linLC;

            TeCoord2D index1(0, lin);
            TeCoord2D index2(nCols-1,lin);

            TeCoordPair pair;
            pair.pt1=index1;
            pair.pt2=index2;

            Segments.push_back(pair);
            return Segments;
        }
        else
        {
            TeCoordPairVect segsIn, segResult;

            if(st==TeBoxPixelOut)
            {
                TePixelBoxInPoly strat(raster,y);
                //return the segments inside the polygon
                strat = for_each( Segments.begin (),
                                  Segments.end(), strat);
                segsIn = strat.result();
            }
            else
            {
                TePixelBBInterPoly strat(raster,y);
```

```
                    //return the segments inside the polygon
                    strat = for_each( Segments.begin (),
                                      Segments.end(), strat);
                    segsIn = strat.result();
                }

                TeCoordPairVect::iterator it = segsIn.begin();

                double colMin = 0;
                double colMax;
                double lin;

                while(it!=segsIn.end())
                {
                    TeCoord2D coord1 = (*it).pt1;
                    TeCoord2D coord2 = (*it).pt2;
                    lin = coord1.y();

                    colMax = coord1.x()-1;

                    //build the segment
                    TeCoord2D index1(colMin, lin);
                    TeCoord2D index2(colMax, lin);

                    TeCoordPair pair;
                    pair.pt1=index1;
                    pair.pt2=index2;

                    segResult.push_back(pair);

                    colMin = coord2.x()+1;
                    ++it;
                }

                //build the last segment
                TeCoord2D index1(colMin, lin);
                TeCoord2D index2(nCols-1,lin);

                TeCoordPair pair;
                pair.pt1=index1;
                pair.pt2=index2;

                segResult.push_back(pair);
                return segResult;
            }
        }

        return Segments;
}
```

```
class iteratorPoly : public iterator
{

      iteratorPoly(int colCurr, int linCurr, int nc, int nl, int nb,
                   TeRaster* pt, TePolygon& poly,
                   TeStrategicIterator str, double linMin=0.0,
                   double linMax=0.0, double colMin=0.0,
                   double colMax=0.0,
                   TeCoordPairVect& seg=TeCoordPairVect(),
                   int posSeg=0.0, int nlInPoly=0.0,
                   int nColsInPoly=0.0, bool e=true,
                   double minLinCM=0.0):
                        iterator(colCurr,linCurr,nc,nl,nb,pt),
                        poly_(poly),
                        strategy_(str),
                        linMin_(linMin),
                        linMax_(linMax),
                        colMin_(colMin),
                        colMax_(colMax),
                        segments_(seg),
                        posSegments_(posSeg),
                        nLinesInPoly_(nlInPoly),
                        nColsInPoly_(nColsInPoly),
                        end_(e),
                        linMinCM_(minLinCM)
                        {}

      iteratorPoly():
                        iterator(0,0,0,0,0,0),
                        linMin_(0.0),
                        linMax_(0.0),
                        colMin_(0.0),
                        colMax_(0.0),
                        posSegments_(0),
                        nLinesInPoly_(0),
                        nColsInPoly_(0),
                        end_(true)
                        {}

      // calculate the segment of the current line that intersect the
      // polygon
      void getNewSegment(int linCurr)
      {
                  //change to world coordinates
                  TeCoord2D coord(colMin_,linCurr+0.5);
                  TeCoord2D colLinCM = raster_->index2Coord(coord);

                  double linCM = colLinCM.y();
                  double colMinCM = colLinCM.x();
```

```
        //applyStrategic: return the segments
        segments_ = applyStrategic(linCM, linMinCM_,
                    colMinCM, strategy_, raster_, poly_);

        colMin_ = segments_[0].pt1.x();
        colMax_ = segments_[0].pt2.x();

        colCurrent_=(int)colMin_;
        posSegments_ = 0;
        end_ = false;
}

bool end(void)
{ return end_; }

int nLinesInPoly()
{ return nLinesInPoly_;}

int nColsInPoly()
{ return nColsInPoly_;}

iteratorPoly& operator++()
{
        if (++colCurrent_>colMax_)
        {
                if(++posSegments_>(segments_.size()-1))
                {
                        if(++linCurrent_>linMax_)
                        {
                                end_ = true;
                                *this = raster_
                                        ->end(poly_,strategy_);
                        }
                        else
                                getNewSegment(linCurrent_);
                }
                else
                {
                        colMin_= segments_[posSegments_].pt1.x();
                        colMax_= segments_[posSegments_].pt2.x();
                        colCurrent_=(int)colMin_;
                }
        }
        return *this;
}


iteratorPoly operator++(int)
{
        iteratorPoly temp = *this;
        ++(*this);
        return temp;
```

```
        }

    private:
        double                  linMin_, linMax_;
        double                  colMin_, colMax_;
        TeCoordPairVect         segments_;
        int                     posSegments_;
        TePolygon               poly_;
        bool                    end_;
        TeStrategicIterator     strategy_;
        int                     nLinesInPoly_, nColsInPoly_;
        double                  linMinCM_;
};
```