Tutorial de Programação

TerraLib 3.1.3

Time TerraLib/ TerraView

Conteúdo

1.Introdução

Este tutorial descreve a biblioteca TerraLib em seus aspectos mais relevantes, incluindo o modelo conceitual do banco de dados, o modelo de armazenamento de geometrias e dados descritivos, e os mecanismos de manipulação do banco em diferentes níveis de abstração.

A TerraLib é um projeto de software livre que permite o trabalho colaborativo entre a comunidade de desenvolvimento de aplicações geográficas, servindo desde à prototipação rápida de novas técnicas até o desenvolvimento de aplicações colaborativas. TerraLib é uma biblioteca de classes escritas em C++, com código fonte aberto e distribuída como software livre (www.terralib.org).

A TerraLib fornece funções para a decodificação de dados geográficos, estruturas de dados espaço-temporais, algoritmos de análise espacial além de propor um modelo para um banco de dados geográficos (Câmara et al. 2000). A arquitetura da biblioteca é mostrada na Figura 1.1. Existe um módulo central chamado *kernel*, composto de estruturas de dados espaço-temporais, suporte a projeções cartográficas, operadores espaciais e uma interface para o armazenamento e recuperação de dados espaço-temporais em bancos de dados objeto-relacionais, além de mecanismos de controle de vizualização. A interface de recuperação e armazenamento é implementada em um módulo composto de *drivers*. Esse módulo também contém rotinas de decodificação de dados geográficos em formatos abertos e proprietários. As funções de análise espacial são implementadas utilizando as estruturas do *kernel*. Finalmente, sobre esses módulos podem ser construídas diferentes interfaces aos componentes da TerraLib em diferentes ambientes de programação (Java, COM, C++) inclusive para a implementação de serviços OpenGIS como o WMS – Web Map Server (OGIS, 2005).

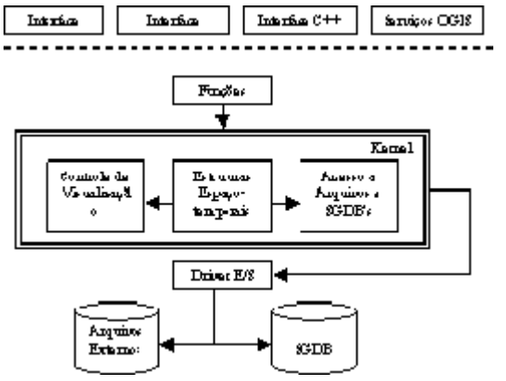


Figura 1.1 – Arquitetura da TerraLib.

Uma das características mais importantes da TerraLib é a sua capacidade de integração a sistemas de gerenciamento de bancos de dados objeto-relacionais (SGBD-OR) para armazenar os dados geográficos: tanto sua componente descritiva quanto sua componente espacial. Essa integração é o que permite o compartilhamento de grandes bases de dados em ambientes coorporativos por aplicações customizadas para diferentes tipos de usuários. A TerraLib trabalha em um modelo de arquitetura em camadas (Davis e Câmara, 2001), funcionando como a camada de acesso entre o banco e a aplicação final.

Como exemplo de um aplicativo geográfico construído usando a TerraLib podemos citar o TerraView (INPE/DPI, 2005). Na Figura 1.2 ilustramos como o TerraView utiliza a TerraLib como camada de acesso a um banco de dados sob o controle de um SGDB-OR, no exemplo o MySQL (MYSQL, 2005).

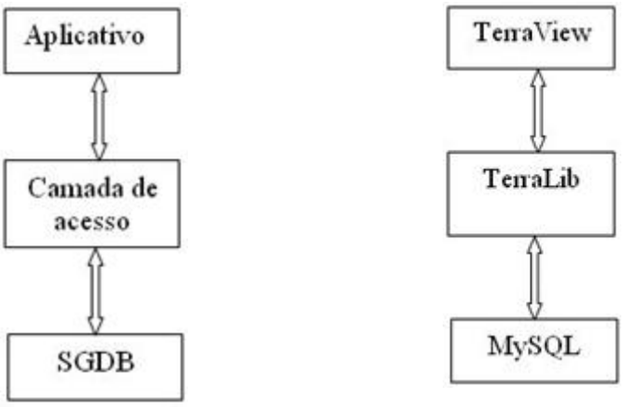


Figura 1.2 – Exemplo de uso da TerraLib como camada de acesso ao banco de dados.

A biblioteca TerraLib pode ser compilada nos sitemas operacionais Windows e Linux, e em diferentes compiladores C++. O código da TerraLib usa extensivamente os mecanismos mais atuais da linguagem C++, como a STL – Standard Template Libray, classes parametrizadas e programação multi-paradigma (Stroustroup, 1997).

Os conceitos que serão apresentados, a seguir, serão ilustrados na forma de extratos de código em C++, que devem ser incluídos em programas executáveis em linha de comando. Para compreensão dos conceitos deve se observar o banco de dados sendo manipulado através de algum front end ao gerenciador e também através de um programa completo com canvas de visualização como o TerraView.

Voltar para o índice.

2. Modelo conceitual

A TerraLib propõe não somente um modelo de armazenamento de dados geográficos em um SGBD-OR, mas também um modelo conceitual de banco de dados geográfico, sobre o qual são escritos seus algoritmos de processamento. As entidades que formam o modelo conceitual são:

- **Banco de Dados:** representa um repositório de informações contendo tanto os dados geográficos quanto o seu modelo de organização. Um banco de dados pode ser materializado em diferentes SGDB's – Sistemas Gerenciadores de Bancos de Dados, comerciais ou de domínio público. O único requisito da TerraLib é que o SGDB possua a capacidade de armazenar campos binários longos, ou uma extensão própria capaz de criar tipos abstratos espaciais, e que possa ser acessado por alguma camada de software.
- **Layer:** um layer representa uma estrutura de agregação de um conjunto de informações espaciais que são localizadas sobre uma região geográfica e compartilham um conjunto de atributos, ou seja, um layer agrega coisas semelhantes. Como exemplos de layers podem ser citados os mapas temáticos (mapa de solos, mapa de vegetação), os mapas cadastrais de objetos geográficos (mapa de municípios do Distrito Federal) ou ainda dados matriciais como cenas de imagens de satélites.
  Independentemente da representação computacional adotada para tratar o dado geográfico, matricial ou vetorial, um layer conhece qual a projeção cartográfica da sua componente espacial.
  Layers são inseridos no banco de dados através da importação de arquivos de dados geográficos em formatos de intercâmbio como shapefiles, ASCII-SPRING, MID/MIF, GeoTiff, JPEG ou dbf. A biblioteca fornece as rotinas de importação desses arquivos. Layers também podem ser gerados a partir de processamentos executados sobre outros layers já armazenados no banco.
- **Representação:** trata do modelo de representação da componente espacial[1] dos dados de um layer e pode ser do tipo vetorial ou matricial. Na representação vetorial, a TerraLib distingue entre representações formadas por pontos, linhas ou áreas e também outras representações mais complexas formadas a partir dessas como células e redes. Para representações matriciais, a TerraLib suporta a representação de grades regulares multi-dimensionais.
  A TerraLib permite que um mesmo geo-objeto de um layer possua diferentes representações vetoriais (por exemplo, um município pode ser representado pelo polígono que define os seus limites e/ou pelo ponto onde está localizado em sua sede). A entidade Representação da TerraLib guarda algumas informações sobre o dado, como o seu menor retângulo envolvente ou a resolução horizontal e vertical de uma representação matricial.
- **Projeção Cartográfica:** serve para representar a referência geográfica da componente espacial dos dados geográficos. As projeções cartográficas permitem projetar a superfície terrestre em uma superfície plana. Diferentes projeções são usadas para minimizar as diferentes deformações inerentes ao processo de projeção de um elipsóide em um plano. Cada projeção é definida a partir de certo número de parâmetros como o Datum planimétrico de referência, paralelos padrão e deslocamentos (Snyder,1987).
- **Tema:** serve principalmente para definir uma seleção sobre os dados de um layer. Essa seleção pode ser baseada em critérios a serem atendidos pelos atributos descritivos do dado e/ou sobre a sua componente espacial.Um tema também define o visual, ou a forma de apresentação gráfica da componente espacial dos objetos do tema. Para o caso de dados com uma representação vetorial a componente espacial é composta de elementos geométricos como pontos, linhas ou polígonos. Para os dados com uma representação matricial, sua componente espacial está implícita na estrutura de grade que a define, regular e com um espaçamento nas direções X e Y do plano cartesiano. Os temas podem definir também formas de agrupamento dos dados de um layer, gerando grupos, os quais possuem legendas que os caracterizam.
- **Vista:** serve para definir uma visão particular de um usuário sobre o banco de dados. Uma vista define quais temas serão processados ou visualizados conjuntamente. Além disso, como cada tema é construído sobre um layer com sua própria projeção geográfica, a vista define qual será a projeção comum para visualizar ou processar os temas que agrega.
- **Visual:** um visual representa um conjunto de características de apresentação de primitivas geométricas. Por exemplo, cores

de preenchimento e contorno de polígonos, espessuras de contornos e linhas, cores de pontos, símbolos de pontos, tipos e transparência de preenchimento de polígonos, estilos de linhas, estilos de pontos, etc.

- **Legenda:** uma legenda caracteriza um grupo de dados, dentro de um tema, apresentados com o mesmo visual, quando os dados do tema são agrupados de alguma forma.

Voltar para o índice.

3. Bancos de Dados Geográficos com TerraLib

Uma das características importantes da TerraLib é sua capacidade de integração com diferentes Sistemas Gerenciadores de Bancos de Dados (SGBDs) objeto-relacionais para armazenar dados geográficos, tanto sua componente descritiva quanto sua componente espacial. Essa integração é o que permite o compartilhamento de grandes bases de dados, em um ambiente coorporativo, por aplicações customizadas para diferentes tipos de usuários.

Para que uma aplicação possa acessar diferentes gerenciadores de bancos de dados de maneira transparente, a TerraLib fornece, através da classe abstrata `TeDatabase`[2],o conceito de um banco de dados TerraLib.

3.1. TeDatabase e Drivers

Na TerraLib existe a possibilidade de integração com diferentes SGBD's, comerciais ou de domínio público. O conceito de um Banco de Dados TerraLib independe de SGBD e é fornecido pela biblioteca através da classe abstrata TeDatabase. Esta classe abstrata contém os métodos necessários para criar, popular e consultar um banco de dados. Ela é derivada em classes concretas, chamadas drivers, que resolvem para diferentes SGDB's comerciais e de domínio público, as particularidades de cada um de forma que as aplicações possam criar bancos de dados em diferentes gerenciadores. A TerraLib fornece alguns drivers em sua distribuição padrão, como mostrado na Figura 3.1. Drivers para outros gerenciadores podem ser criados através da implementação dos métodos virtuais definidos na classe base.

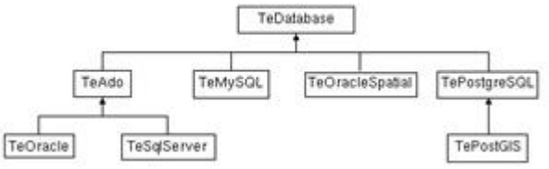


Figura 3.1 – Drivers para bancos de dados fornecidos pela TerraLib.

Tipicamente, as aplicações que usam a TerraLib processam ponteiros para a classe TeDatabase, inicializados com instancias concretas de algum driver como mostrado no exemplo a seguir.

Exemplo 3.1 - Conexão a um servidor de bancos de dados com a TerraLib.

```
void main()
{
        TeDatabase* db;
        int op;
        cout << "Selecione qual gerenciador deseja usar:
        1) ACCESS 2) MySQL \n";
        cin >>op;
        if (op == 1)
                db = new TeAdo();     // Usa ACCESS através da biblioteca ADO
        else
                db = new TeMySQL();   // Usa MySQL
// ... usa somente db
```

3.2. Modelo de um Banco TerraLib

Fisicamente, um banco de dados TerraLib é formado por um conjunto de tabelas em um SGBD-OR, onde são armazenados tanto os dados geográficos (suas geometrias e seus atributos) quanto um conjunto de informações sobre a organização desses dados no banco, ou seja, o modelo conceitual da biblioteca. Essas tabelas podem ser divididas em dois tipos:

- **Tabelas de Metadados**: são usadas para guardar os conceitos TerraLib e possuem formato pré-definido, de forma a serem decodificadas pelas classes. O conjunto das tabelas de metadados é chamado de Modelo de Dados Conceitual.
- **Tabelas de Dados**: são usadas para guardar os dados geográficos em si, tanto sua componente espacial quanto descritiva. O esquema das tabelas para armazenar a componente descritiva do dado não é pré-definido. Por exemplo, a tabela que contém os atributos dos bairros de uma cidade pode ser:

BairrosSP

ID	Nome	Populacao	Area
3550089	Tatuapé	200000	25000
3550070	Moema	145000	60000

Já as tabelas que contém a componente geográfica, ou geométrica do dado, têm um formato pré-definido para que as classes TerraLib possam decodificar o dado lá armazenado, criando instâncias em memória dessas geometrias. Por exemplo, considerando que cada bairro possua uma representação poligonal (seus limites), esses são armazenados em uma tabela de polígonos:

Polygons1

geom_id	object_id	Num_coords	num_holes	spatial_data	...
1	3550070	...	...	...	...
2	3550089	...	...	...	...

Para cada tipo de geometria (polígonos, linhas, pontos, células, arcos e nós) há um formato pré-definido de tabela. Por exemplo, as tabelas que armazenam polígonos seguem o formato mostrado acima.

Essas duas tabelas (de atributos e geometrias) armazenam o conceito de objeto geográfico. A ligação entre a componente geométrica e descritiva de um objeto geográfico é feita pelo relacionamento entre o campo **object_id** de uma tabela geométrica e um campo qualquer da tabela de atributos descritivos. Assim no exemplo acima temos que o objeto 3550070 (Moema) é representado pelo polígono 1 e o objeto 3550089 (Tatuapé) pelo polígono 2.

As tabelas de metadados são automaticamente criadas quando se cria um novo banco TerraLib. Para acessar bancos de dados já existentes, as aplicações abrem conexões a eles, sendo que uma aplicação pode manter conexões a mais de um banco de dados ao mesmo tempo. Ao criar ou abrir uma conexão a um banco de dados as aplicações devem informar os parâmetros exigidos pelo SGDBOR onde está armazenado o banco, como mostra o Exemplo 3.2. Ao final da execução da aplicação todas as conexões devem ser fechadas.

Exemplo 3.2 - Criar um banco TerraLib chamado "TerraTeste", usando um gerenciador de dados MySQL rodando na máquina local e supondo que existe um usuário chamado “root” sem senha de acesso.

```
#include <TeMySQL.h>
int main()
{
        // Parâmetros do servidor de bancos de dados
        string host = "localhost";
        string dbname = "TerraTeste";
        string user = "root";
        string password = "";

        // Cria um novo banco
        TeDatabase* db = new TeMySQL();
        if (!db->newDatabase(dbname,user, password, host))
        {
                cout << "Erro: " << db->errorMessage() << endl;
                return 1;
        }

        cout << "O banco de dados \"" << dbname;
        cout <<"\" foi criado com sucesso no servidor MySQL localizado em\""
                 << host;
        cout << "\" para o usuario \"" << user << "\"!"  << endl;
        // Fecha o banco de dados
        db->close();
        delete db;
        return 0;
}
```

Exemplo 3.3 - Abrir o banco criado no exemplo anterior através de um front end e observar as tabelas criadas.

As tabelas de metadados (Figura 3.2) servem para armazenar os conceitos descritos na Seção 2. Cada layer criado no banco gera um registro na tabela chamada **te_layer**, o campo **layer_id** contém a identificação única de cada layer no banco de dados. Os outros campos dessa tabela armazenam o nome e o mínimo retângulo envolvente do layer, ou seja, o mínimo retângulo envolvente de todas as geometrias associadas ao layer.

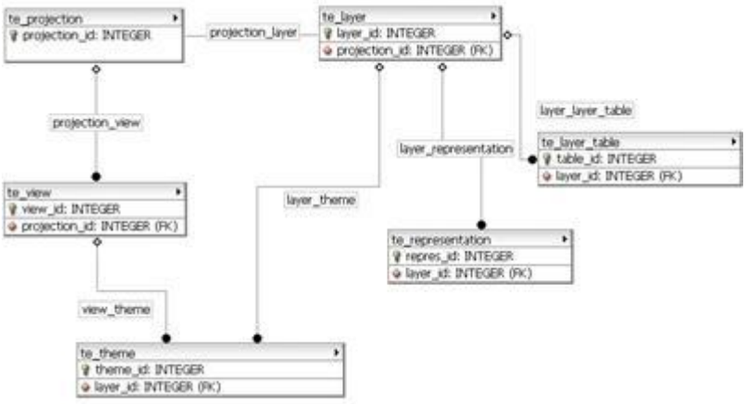


Figura 3.2 – Principais tabelas do modelo conceitual da TerraLib.

Cada representação geométrica associada a um layer gera um registro na tabela **te_representation**. Cada tabela de atributos associada a um layer gera um registro na tabela **te_layer_table**.

Cada vista criada no banco de dados gera um registro em uma tabela chamada de **te_view**. Cada instância de projeção cartográfica é armazenada no banco na tabela **te_projection**. Layers e vistas possuem referência a um registro da tabela de projeções. Cada tema criado gera um registro na tabela **te_theme**. Cada tema possui uma referência para vista no qual está definida.

Para compreender melhor as tabelas do modelo de metadados e os relacionamentos entre elas é interessante observar o conteúdo dessas tabelas após a execução de uma seqüência típica de operações:

- Criar um banco de dados;
- Importar um dado geográfico de um arquivo para um layer do banco de dados;
- Criar uma vista;
- Criar um tema usando o layer criado e inseri-lo na vista.

As tabelas de dados serão descritas mais adiante após falarmos sobre o modelo de geometrias e de atributos de TerraLib.

Exemplo 3.4 - Para utilizar um banco de dados já criado anteriormente é necessário estabelecer uma conexão ao mesmo. Ao final da aplicação a conexão ao banco de dados deve ser sempre fechada.

```
#include<TeMySQL.h>
int main()
{
        // Cria uma conexão a um servidor de banco de dados
        TeDatabase* db = new TeMySQL();
        // Parametros do servidor de bancos de dados
        string host ="localhost";
        string bname = "TerraTeste";
        string user = "root";
        string password = "";
        if (!db->connect(host,user, password,dbname,0))
        {
                cout<< "Erro: " << db->errorMessage() << endl;
                return 0;
        }
        cout << “Aberta conexão ao banco: “ << dbname;
        db->close();
        delete db;
}
```

A seguir serão mostrados os principais conceitos, sua representação em classes e em tabelas do modelo conceitual.

Voltar para o índice.

4. Layer

No universo da TerraLib o conceito de um ***layer*** refere-se a uma estrutura de agregação de informações espaciais sobre elementos que estão localizados em uma mesma região geográfica e compartilham o mesmo conjunto de atributos. Ou seja, um layer agrega elementos semelhantes.
Como exemplos de layers podem ser citados mapas temáticos (mapa de solos), mapas cadastrais de objetos geográficos (mapa de municípios do Distrito Federal) ou ainda dados matriciais como imagens de satélites. Um layer existe em memória como uma

instância da classe `TeLayer`[3] e está registrado em um banco TerraLib como um registro da tabela do modelo conceitual **te_layer**.

Os layers são criados através da importação de dados geográficos em formatos conhecidos, proprietários de Sistemas de Informação Geográfica específicos. Como exemplos, podem ser citados o formato Shapefile, usado pelos produtos da Environmental Systems Research Institute, Inc. (ESRI), ou o MapInfo Interchange File (MID/MIF) dos produtos MapInfo. Para dados matriciais existem formatos mais gerais como GeoTIFF ou JPEG. Algumas funcionalidades de TerraLib também geram novos layers a partir de alguns já existentes.

Exemplo 4.1 - Criar um layer através da importação de um arquivo MID/MIF.

```
// Conectar-se a um banco...
// Importa arquivo
string filename = ""../data/Distritos.mif";
TeLayer* newLayer = TeImportMIF(filename,db_);
if (newLayer)
        cout << "Arquivo MID/MIF importado para o banco\n";
else
{
        cout <<"Erro na importacao do dado\n";
        db_->close();
}
```

Observar que a tabela **te_layer** do modelo conceitual possui um registro onde estão guardadas as informações sobre o layer recém criado. Além disso, duas novas tabelas de dados foram criadas, uma para armazenar os polígonos que representam a componente espacial do dado e outra para guardar seus atributos descritivos.

Voltar para o índice.

5. Projeção Cartográfica

Uma projeção cartográfica estabelece uma relação entre pontos da superfície terrestre e seus correspondentes no plano de projeção do mapa. Apesar de haver um grande número de projeções, elas se reduzem à dois grandes grupos: o das projeções conformes, em que os ângulos são conservados, e o das projeções equivalentes, em que as áreas são conservadas.
Cada projeção depende de certo número de parâmetros, como paralelo-padrão, meridiano de origem e datum planimétrico. O datum planimétrico é definido pelo posicionamento do elipsóide de referência em relação à superfície da Terra. Uma projeção existe em memória como uma instância da classe `TeProjection`[4] e registrado em um banco TerraLib como um registro da tabela do modelo conceitual **te_projection**.

Exemplo 5.1 - Criar duas projeções diferentes e fazer conversão de projeções.

```
TeDatum dSAD69 = TeDatumFactory::make("SAD69");               // SAD69 Spheroid
TeDatum dWGS84 = TeDatumFactory::make("WGS84");                        // WGS84 Spheroid

// UTM: Latitude de Origem -45.0
// TeCDR: "Converter para Radianos a partir de Grau Decimal"
TeUtm* pUTM = new TeUtm(dSAD69,-45.0*TeCDR);
// Polyconic: Latititude de origem -45.0
// TeCDR: “Converter paraRadianos a partir de Grau Decimal"
TePolyconic* pPolyconic = new TePolyconic(dWGS84,-45.0*TeCDR);
// Coordenada original em UTM

TeCoord2D pt1(340033.47,7391306.21);
// Conversao de UTM para Policonica
pUTM->setDestinationProjection(pPolyconic);
// Converte para Lat Long
TeCoord2D ll = pUTM->PC2LL(pt1);
// Converte a projecao de saida
TeCoord2D pt2 = pPolyconic->LL2PC(ll);
printf("UTM ->Policonica \n");
printf("(%.4f, %.4f)-> ",pt1.x(), pt1.y());
printf("(%.4f, %.4f)\n",pt2.x(), pt2.y());

// Conversao de Policonica para UTM
pPolyconic->setDestinationProjection(pUTM);
ll = pPolyconic->PC2LL(pt2);
pt1 = pUTM->LL2PC(ll);
```

```
printf("\nPolyconic-> UTM \n");
printf("(%.4f,%.4f) -> ",pt2.x(), pt2.y());
printf("(%.4f, %.4f)\n",pt1.x(), pt1.y());
```

Alguns formatos de intercâmbio de dados geográficos podem trazer a informação sobre em qual projeção cartográfica esse dado se refere (ex. MID/MIF, GeoTIFF). Quando se desconhece a projeção cartográfica, TerraLib oferece um sistema de coordenadas *default*, não geográfico, chamado `TeNoProjection`. **Exemplo 5.2** - Importar um dado sem projeção cartográfica e atribuir a projeção correta posteriormente.

```
TeLayer* layer = new TeLayer(layerName, db);
string filename = "../data/EstadosBrasil.shp";
string tablename = "BrasilIBGE";
string linkcolumn = "SPRROTULO";

if (TeImportShape(layer, filename, tablename, linkcolumn))
        cout >> "The shapefile was imported successfully into the
             TerraLib database!\n" << endl;
else
        cout >> "Erro: Falha ao importar o shapefile!\n" << endl;

TeDatum sad69 = TeDatumFactory::make("SAD69");
TePolyconic* proj = new TePolyconic(sad69, -54.0*TeCDR);
layer->setProjection(proj);  // Ajusta a projecao correta
```

Verificar na tabela **te_projection** a distinção entre a projeção do layer importado no Exemplo 5.2 e o layer importado no Exemplo 4.1.

No Exemplo 5.2 deve-se observar que o usuário especificou explicitamente qual deveria ser o nome da tabela de atributos - “BrasilIBGE”.
Um dos componentes do formato shapefile é um arquivo DBF, que contém os atributos descritivos. Sabendo que nesse arquivo existe uma coluna chamada “SPRROTULO” com valores únicos, o usuário a utilizou para identificar unicamente cada estado e para ligar as geometrias e atributos dos objetos ou elementos desse *layer*.

Voltar para o índice.

6. Representação Geométrica

Os objetos geográficos podem ter diferentes representações geométricas, ou geometrias, para representar sua localização no espaço. Por exemplo, dependendo do tipo de análise, um município pode ser representado por um ponto ou por um polígono, ou um rio pode ser representado por um polígono ou por uma linha.

6.1. Modelo de Geometrias

Conforme descrito na Seção 2, na TerraLib os dados geográficos são agregados em *layers*. *Layers* são formados por conjuntos de objetos, onde cada objeto possui uma identificação única, um conjunto de atributos descritivos e uma representação geométrica. Essa seção descreve o modelo de classes de geometria da TerraLib.

A classe base da qual derivam todas as geometrias de TerraLib é chamada de `TeGeometry`[5].
Cada geometria possui uma identificação única, a referência ao seu menor retângulo envolvente e a identificação do objeto geográfico que representa. As geometrias vetoriais de TerraLib são construídas a partir de coordenadas bi-dimensionais representadas na classe chamada de `TeCoord2D`[6].

Essas geometrias são:

- ***Pontos*:** representados na classe `TePoint, TeCoord2D`;
- ***Linhas*:** composta de um ou mais segmentos são representadas na classe `TeLine2D`, implementada como um vetor de duas ou mais `TeCoord2D`;
- ***Anéis*:** representados pela classe `TeLinearRing`, são linhas fechadas, ou seja, a última coordenada é igual a primeira. A classe `TeLinearRing` é implementada como uma instância única de uma `TeLine2D` que satisfaz a restrição de que a primeira coordenada seja igual a última;
- ***Polígonos*:** representados pela classe `TePolygon`, são delimitações de áreas que podem conter nenhum, um ou mais buracos, ou filhos. São implementados como um vetor de `TeLinearRing`. O primeiro anel do vetor é sempre o anel externo enquanto que os outros anéis, se existirem, são buracos ou filhos do anel externo.

A fim de otimizar a manutenção das geometrias em memória as classes de geometrias de TerraLib são implementadas segundo o padrão de projeto *handle/body* onde a implementação é separada da interface (Gamma et al., 1995). Além disso, as implementações são referências contadas, ou seja, cada instância de uma classe de geometria guarda o número de referências feitas a ela, inicializado com zero quando a instância é criada. Cada vez que uma cópia dessa instância é solicitada apenas o seu número de referências é incrementado e, cada vez que uma instância é destruída, o número de referências a ela é decrementada. A instância é

efetivamente destruída apenas quando esse número chega ao valor zero.

Outro aspecto importante das classes de geometria da TerraLib é que elas são derivadas ou da classe `TeGeomSingle` ou da classe `TeGeomComposite`, representando, respectivamente, que são geometrias com um único elemento menor (como um `TePoint`) ou que podem ser compostas de outros elementos menores (como é o caso de `TePolygon, TeLine2D e TeLinearRing`).
Esse padrão de compostos de elementos menores (Gamma et al. 1995) é aplicado também em classes que formam conjuntos de pontos, linhas, polígonos e são representados nas classes `TePointSet, TeLineSet` e `TePolygonSet`.
Os padrões de projeto são implementados com o recurso de classes parametrizadas como mostrado na Figura 6.1, o que torna o código mais reutilizável.

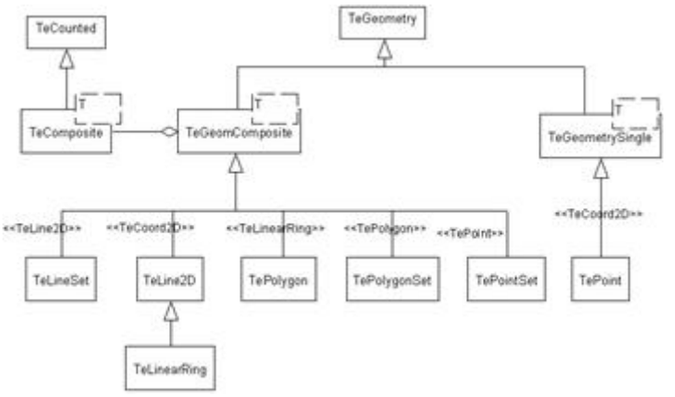


Figura 6.1 – Diagrama das principais classes de geometria da TerraLib (adaptado de Queiroz, 2003)

As geometrias matriciais são representadas na classe TeRaster, que será descrita por completo na Seção 10. Um exemplo de criação de geometrias em memória é mostrado no Exemplo 6.1.

Exemplo 6.1 - Criação de geometrias vetoriais em memória.

```
// Cria um conjunto de linhas
TeLine2D reta;
reta.add(TeCoord2D(500,500));
reta.add(TeCoord2D(600,500));
reta.add(TeCoord2D(700,500));
reta.objectId("reta");

TeLine2D ele;
ele.add(TeCoord2D(700,700));
ele.add(TeCoord2D(800,600));
ele.add(TeCoord2D(900,600));
ele.objectId("ele");

TeLineSet ls;
ls.add(reta);
ls.add(ele);

// Cria um conjunto de polígonos
// Um polígono simples

TeLine2D line;
line.add(TeCoord2D(900,900));
line.add(TeCoord2D(900,1000));
line.add(TeCoord2D(1000,1000));
line.add(TeCoord2D(1000,900));
line.add(TeCoord2D(900,900));

TeLinearRing r1(line);
TePolygon poly1;
poly1.add(r1);
poly1.objectId("spoli");

// Um polígono com um filho
TeLine2D line2;
line2.add(TeCoord2D(200,200));
```

```
line2.add(TeCoord2D(200,400));
line2.add(TeCoord2D(400,400));
line2.add(TeCoord2D(400,200));
line2.add(TeCoord2D(200,200));
TeLinearRing r2(line2);

TeLine2D line3;
line3.add(TeCoord2D(250,250));
line3.add(TeCoord2D(250,300));
line3.add(TeCoord2D(300,300));
line3.add(TeCoord2D(300,250));
line3.add(TeCoord2D(250,250));
TeLinearRing r3(line3);
TePolygon poly2;
poly2.add(r2);
poly2.add(r3);
poly2.objectId("cpoli");
TePolygonSet ps;
ps.add(poly1);
ps.add(poly2);

// Cria um conjunto de pontos.
TePoint p1(40,40);
p1.objectId("ponto1");
TePointSet pos;
pos.add(p1);
```

A TerraLib implementa uma estrutura de dados de espaços celulares, que juntamente com o suporte para predicados temporais atende às necessidades de implementação de modelos dinâmicos baseados em espaços celulares. Espaços celulares podem ser vistos ou como uma estrutura matricial generalizada onde cada célula armazena mais que um valor de atributo ou como um conjunto de polígonos que não se interceptam. Essa estrutura traz como uma vantagem a possibilidade de armazenar conjuntamente, numa única estrutura, todo o conjunto de informações necessárias para descrever um fenômeno espacial complexo, o que beneficia aspectos de visualização e interface. Todas as informações podem ser apresentadas da mesma forma que objetos geográficos com representação vetorial. Para atender a essa necessidade, a TerraLib propõe mais uma geometria chamada `TeCell`, que representa uma célula em um espaço celular materializado na classe `TeCellSet`. O Exemplo 6.2 mostra como criar um espaço celular a partir de um *layer* armazenado em um banco.

Exemplo 6.2 - Criação de um espaço celular.

```
// Recupera o layer de estados
TeLayer* estados = new TeLayer("Brasil");
db->loadLayer(estados);
// Cria um espaço celular sobre a extensão do layer
// de estados, onde cada célula tem 1/100 metros da extensão horizontal
// do layer de estados
TeLayer* espaco_cel = TeCreateCells("CellsEstados", estados,
                          estados->box().width() / 100,
                          estados->box().width() / 100,
                          estados->box(), true);
```

Cada célula possui uma identificação única e uma referência a sua posição dentro do espaço celular, a qual pode ser associada diferentes atributos conforme o modelo dinâmico sendo construído.

6.2. Modelo de armazenamento de geometrias

A proposta da TerraLib, conforme mostramos na Figura 1.2, é trabalhar em bancos de dados geográficos que podem armazenar tanto atributos descritivos como atributos geométricos (como pontos, linhas, polígonos e dados matriciais). Esses bancos de dados podem ser construídos em SGDB's que possuem extensões espaciais, ou seja, possuem a capacidade de criar tipos espaciais e manipulá-los como tipos básicos e fornecem mecanismos eficientes de indexação e consulta (Shekkar e Chawla, 2002). Podem também ser construídos em SGDB's que oferecerem somente a capacidade de criar tabelas com campos do tipo binário longo. Na TerraLib esses dois tipos de SGDB's são usados de maneira transparente através da classe abstrata `TeDatabase`.

O modelo de armazenamento de geometrias em um banco leva em conta questões relativas à eficiência no seu armazenamento e na sua recuperação, e também a existência ou não de um tipo espacial no SGDB. Todas as tabelas que armazenam as geometrias

possuem os campos:

- `geom_id`: do tipo inteiro, que armazena a identificação única da geometria;
- `object_id`: do tipo texto, que armazena a identificação única do objeto geográfico ao qual a geometria está relacionada;
- `spatial_data`: armazena o dado geométrico. O tipo desse campo depende do SGDB onde está armazenado o banco. Para SGDB's com extensão espacial é o tipo fornecido pela extensão. Para SGDB's sem a extensão espacial é um binário longo.

Para os SGDB´s sem extensão espacial as tabelas de geometrias do tipo linhas e polígonos possuem outros campos para armazenar o mínimo retângulo envolvente da geometria (`lower_x`, `lower_y`, `upper_x`, `upper_y` todos do tipo real). Esses campos são indexados pelos mecanismos fornecidos pelo SGBD e serão usados pelas rotinas de recuperação como os indexadores espaciais do dado. Para os SGDB's com extensão, a coluna com o dado espacial é indexada espacialmente pelo mecanismo oferecido pela extensão.

A Figura 6.2 mostra a diferença entre uma tabela de geometria do tipo polígono criada em um banco sem extensão espacial e em um banco com extensão espacial. No segundo caso o tipo "GEOMETRY" representa o tipo espacial fornecido pela extensão. No caso das geometrias do tipo polígono, o modelo de armazenamento em campos longos, prevê que cada anel do polígono é armazenado em um registro da tabela. O anel externo contém a informação sobre o número de filhos que o polígono possui e os anéis internos guardam a identificação de seu pai, ou seja, do anel externo no qual estão contidos. Essa forma de armazenamento permite a recuperação parcial de polígonos com um grande número de filhos (por exemplo, uma operação de *zoom* em um mapa em uma área grande como a Amazônia legal, em uma escala pequena). Como são armazenados os retângulos envolventes de cada filho é possível recuperar somente o pai e os filhos que estão dentro do retângulo envolvente definido pelo *zoom*. Isso representa uma otimização no processamento desse dado.

Tabela de polígonos em SGDB´s sem extensão espacial

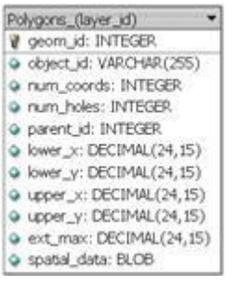


Tabela de polígonos no PostgreSQL com a extensão PostGIS

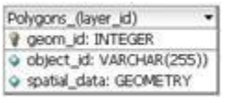


Figura 6.2 – Modelo de armazenamento de geometrias do tipo polígonos (adaptado de Ferreira, 2003).

A Figura 6.3 mostra como são as tabelas que armazenam geometrias do tipo linhas e a Figura 6.4 as tabelas para geometrias do tipo pontos.

Tabela de linhas em SGDB´s sem extensão espacial

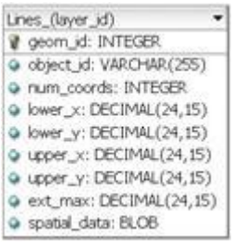


Tabela de linhas no PostgreSQL com a extensão PostGIS

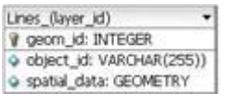


Figura 6.3 – Modelo de armazenamento de geometrias do tipo linhas.

Tabela de pontos em SGDB´s sem extensão espacial

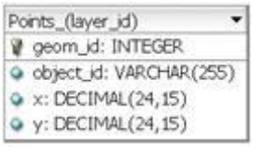


Tabela de pontos no PostgreSQL com a extensão PostGIS

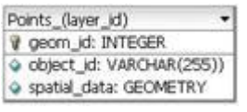


Figura 6.4 – Modelo de armazenamento de geometrias do tipo pontos.

Em um banco TerraLib, para cada layer, cada tipo de representação geométrica é armazenado em uma tabela diferente, cujo formato é específico para o tipo de representação. Uma representação geométrica de um layer existe em memória como uma instância da estrutura `TeRepresentation` e registrado em um banco TerraLib como um registro da tabela do modelo conceitual **te_representation**. Observar na tabela **te_representation** as representações geométricas dos layers importados nos exemplos anteriores. No próximo exemplo será usada uma funcionalidade de TerraLib para criar uma segunda representação geométrica para

objetos com representação poligonal.

Exemplo 6.3 - Criar uma representação geométrica de pontos associada a cada distrito. O ponto será o centróide do polígono que delimita o distrito.

```
TeLayer* distritos = new TeLayer("Distritos");
if (!db->loadLayer(distritos))
{
        cout << "Falha ao tentar carregar o layer \"Distritos\": "
          << db->errorMessage() << endl << endl;
        db->close();
        return 1;
}

// Checa se a tabela de pontos a ser criada já existe no banco
string pointsTableName = distritos->tableName(TePOINTS);
if (pointsTableName.empty() == false)
{
        if (db->tableExist(pointsTableName))
        {
                cout << "A tabela de pontos \"" << pointsTableName
               << "\" já existe no banco!\n\n";
                db->close();
                return 1;
        }
}
TeRepresentation* repPol = distritos->getRepresentation(TePOLYGONS);
TePointSet centroids;     //gera centróides
if (db->Centroid(repPol->tableName_, TePOLYGONS, centroids) == false)
{
        cout << "Falha ao criar centroides: " << db->errorMessage();
        db->close();
        return 1;
}
//  Adiciona uma nova representação ao layer
distritos->addPoints(centroids);
cout << "Centroides criados!\n\n";
cout <<"\nNumero de poligonos: "<< distritos->nGeometries(TePOLYGONS);
cout <<"\nNumero de linhas: "   << distritos->nGeometries(TeLINES);
cout <<"\nNumero of pontos: " << distritos ->nGeometries(TePOINTS);
```

Observar na tabela **te_representation** quantas representações geométricas existem para o layer “Distritos”, de quais tipos são e em quais tabelas de geometrias estão armazenadas.

Voltar para o índice.

7. Atributos Descritivos

Os atributos descritivos dos objetos de um *layer* são representados em tabelas relacionais onde cada campo representa um atributo do objeto. Um dos campos deve conter a identificação do objeto e seus valores são repetidos nas tabelas de geometria permitindo assim a ligação entre os atributos descritivos e a geometria do objeto.

Cada *layer* pode ter uma ou mais tabelas de atributos e ao serem inseridos no banco de dados cada tabela de atributo é registrada na tabela de metadados chamada **te_layer_table**. Nessa tabela é registrada também o nome do campo que é a chave primária e identificador do objeto. Esse campo serve de ligação entre atributos e geometrias. Ao serem criados, os temas selecionam quais tabelas do *layer* irão usar.

Quando a tabela de atributos não possui nenhuma informação temporal e cada registro representa os atributos de um objeto diferente, a tabela é chamada de **tabela estática**. Essas classificação semântica das tabelas de atributos é uma característica da TerraLib e tem por objetivo definir funções e processamentos dependentes desses tipos. Isso ficará mais claro adiante quando falarmos do processamento de dados espaço-temporais.

Algumas tabelas de atributos, chamadas **tabelas externas**, não representam nenhum objeto definido em um *layer*, mas podem possuir algum campo com valores coincidentes com os valores de um campo de uma tabela estática de um *layer*. Através de uma operação de junção por esses campos coincidentes uma tabela externa pode acrescentar informações aos objetos de um *layer*.
Por isso, tabelas externas podem ser incorporadas ao banco e registradas como tal, ficando disponíveis para serem usadas em todos os temas do banco.

Uma tabela de atributos existe em memória como uma instância da classe `TeTable`[7] e está registrada em um banco TerraLib como um registro da tabela do modelo conceitual **te_layer_table**.

Na TerraLib, as tabelas de atributos são classificadas de acordo com as características dos dados armazenados. Por exemplo, tabelas de atributos que armazenam informações dinâmicas, ou seja, que mudam ao longo do tempo (`TeAttrEvent`, `TeFixedGeomDynAttr`, `TeDynGeomDynAttr`, `TeGeomAttrLinkTime`), informações de mídia (`TeAttrMedia`) ou informações estáticas (`TeAttrStatic`).
Dependendo do tipo da tabela alguns campos da tabela **te_layer_table** devem ser preenchidos.

As tabelas de atributos mais comuns são as que armazenam informações estáticas, ou seja, que não variam ao longo do tempo. Para registrar esse tipo de tabela de atributos é necessário informar a qual layer ela pertence e qual de seus campos faz a ligação com as tabelas de geometria, ou seja, qual é o atributo identificador do objeto.

Exemplo 7.1 - Criação de uma tabela de atributos em memória.

```
// Cria a lista de atributos da tabela
TeAttributeList attList;
TeAttribute at;
at.rep_.type_ = TeSTRING;
at.rep_.numChar_ = 16;
at.rep_.name_ = "IdObjeto";
at.rep_.isPrimaryKey_ = true;
attList.push_back(at);

at.rep_.type_ = TeSTRING;           // um atributo do tipo string
at.rep_.numChar_ = 255;
at.rep_.name_ = "nome";
at.rep_.isPrimaryKey_ = false;
attList.push_back(at);

at.rep_.type_ = TeREAL;             // um atributo do tipo real
at.rep_.name_ = "val1";
at.rep_.isPrimaryKey_ = false;
attList.push_back(at);

// Cria uma instância da tabela em memória
TeTable attTable("teste2Attr",attList,"IdObjeto","IdObjeto");
TeTableRow row;
row.push_back("reta");
row.push_back("L reta");
row.push_back("11.1");
attTable.add(row);
row.clear();
row.push_back("ele");
row.push_back("L ele");
row.push_back("22.2");
attTable.add(row);
row.clear();
row.push_back("spoli");
row.push_back("Pol simples");
row.push_back("33.3");
attTable.add(row);
row.clear();
row.push_back("ponto1");
row.push_back("Um ponto");
row.push_back("55.5");
attTable.add(row);
row.clear();
row.push_back("cpoli");
row.push_back("Pol buraco");
row.push_back("44.4");
attTable.add(row);
row.clear();
```

Observar no banco criado nos exercícios anteriores as tabelas de dados e as entradas nas tabelas de metadados.

7.1 Tabelas Estáticas

São tabelas que contêm atributos que não variam no tempo.
Além disso, devem possuir um campo com valores únicos, o qual serve de ligação com uma ou mais tabelas de geometrias. Um determinado layer pode possuir uma ou mais tabelas de atributos. O conjunto de propriedades ou atributos dos objetos de um layer é formado pela união de todas as suas tabelas de atributos.

Os formatos de dados MID/MIF, Shapefile e SPRING Geo/Tab são compostos por dois arquivos, um com a geometria e outro com os atributos de um conjunto de objetos. No caso do formato MID/MIF e Shapefile a ligação entre as geometrias e os atributos se dá pela ordem, ou seja, a primeira geometria do arquivo de geometrias juntamente com o primeiro conjunto de atributos do arquivo de atributos formam o primeiro objeto, a segunda geometria do arquivo de geometrias juntamente com o segundo conjunto de atributos do arquivo de atributos formam o segundo objeto, e assim por diante. No caso do formato GEO/Tab, a geometria e o conjunto de atributos contêm explicitamente a identificação do objeto ao qual pertencem (Tabela 7.1).

Tabela 7.1 Formatos de dados geográficos

Formato	Geometrias	Atributos	Ligação Geometria/Atributos
MID/MIF	*.mif	*.mid	Ordem
Shapefile	*.shp	*.dbf	Ordem
Spring Geo/Tab	*.geo	*.tab	Explícita

Os importadores de Shapefile e MID/MIF, por default, criam uma coluna de atributos extra, com valores relativos à ordem de entrada no arquivo, que é usada para ligar as geometrias e atributos. Porém, quando o usuário conhece os seus dados e sabe que já existe uma determinada coluna que contém valores únicos e pode ser usada como identificador do objeto, ele pode usar essa coluna para fazer a ligação entre a geometria e atributos. O mesmo vale para as tabelas de atributos em formato CSV ou DBF. Como será mostrado no exemplo a seguir.

Exemplo 7.2 - Importar um arquivo MID/MIF formando um layer de bairros de Recife. Importar uma segunda tabela de atributos para esse layer a partir de um arquivo em formato CSV.

```
string filename = "../data/BairrosRecife.MIF";
if (db->layerExist("BairrosRecife"))
{
        cout << "Erro: o layer \"BairrosRecife\" ja existe
               no banco de dados!\n" << endl;
        db->close();
        return 1;
}
// Importa o arquivo MID/MIF
TeLayer* newLayer = new TeLayer("BairrosRecife", db, 0);
// Seleciona o campo ID_ para ser o identificador de cada bairro
if (TeImportMIF(newLayer, filename,"BairrosRecife", "ID_"))
        cout << "O arquivo MID/MIF foiimported para o banco TerraLib
                 com sucesso!\n" << endl;
else
{
        cout << "Falha ao importar o arquivo MID/MIF!\n";
        db->close();
        return 1;
}
// Importa o arquivo csv
filename = "../data/BairrosRecife2.csv";
if (db->tableExist("BairrosRecife2"))
{
cout << "Ja existe uma tabela com o nome \"BairrosRecife2\"
       no banco!\n\n";
                return 1;
        }
cout << "Importando a segunda tabela de atributos a partir de um arquivo CSV...\n";
// Cria a definicao das colunas contidas no arquivo CSV
TeAttributeList attList;
TeAttribute column1;
column1.rep_.name_  = "BAIRRO_ID";
```

```
column1.rep_.type_ = TeSTRING;
column1.rep_.isPrimaryKey_ = true;
column1.rep_.numChar_ = 32;
attList.push_back(column1);

TeAttribute column2;
column2.rep_.name_  = "ORDEM";
column2.rep_.type_ = TeINT;
attList.push_back(column2);

TeAttribute column3;
column3.rep_.name_  = "NOME";
column3.rep_.type_ = TeSTRING;
column3.rep_.numChar_ = 32;
attList.push_back(column3);

TeAttribute column4;
column4.rep_.name_  = "NOME_PADRAO";
column4.rep_.type_ = TeSTRING;
column4.rep_.numChar_ = 32;
attList.push_back(column4);

TeTable attTable2("BairrosRecife2",attList, "BAIRRO_ID",
                      "BAIRRO_ID", TeAttrStatic);
if (!newLayer->createAttributeTable(attTable2))
{
        cout << "Erro ao tentar criar a tabela \"BairrosRecife2\" no
                 banco de dados!\n\n";
        return 1;
}
TeAsciiFile csvFile(filename);
while (csvFile.isNotAtEOF())
{
        TeTableRow trow;
        csvFile.readNStringCSV (trow, 4, ';');
        if (trow.empty())
                break;
        csvFile.findNewLine();
        attTable2.add(trow);
}
if (attTable2.size() > 0)
{
        if (!newLayer->saveAttributeTable(attTable2))
        {
                cout << "Erro ao salvar a tabela \"BairrosRecife2\" no
                       banco!\n\n";
                return 1;
        }
        attTable2.clear();
}
cout << "A tabela foi importada com sucesso...\n";
db->close();
```

Verificar no banco as tabelas criadas e as entradas na tabela **te_layer_table**.

Voltar para o índice.

7.2 Tabelas Externas

Tabelas externas são tabelas de atributos que podem acrescentar informações aos objetos de um ou mais layers, mas que não possuem um campo de ligação direta com as tabelas de geometria. Essas tabelas podem ser consideradas como não espaciais.

As tabelas externas podem fazer parte de um tema através da ligação de um de seus campos com qualquer campo de uma tabela de

atributos estática. As tabelas externas são dinâmicas e não pertencem a um layer específico, podendo ser ligadas a diferentes temas construídos a partir de diferentes layers.

Exemplo 7.3 - Importar uma tabela de atributos externa.

```
// Importa o arquivo dbf
string filename = "../data/SOCEC.dbf";
if (db->tableExist("SOCEC"))
{
         cout << "Ja existe uma tabela nomeada \"SOCEC\" no banco!\n\n";
         return 1;
}
if (TeImportDBFTable(filename,db))
         cout << "O arquivo dbf \"SOCEC.dbf\" foi importado com sucesso, como uma
            tabela exetrna no banco de dados!\n\n";
else
{
         db->close();
         cout << "Falha ao importar o arquivo dbf!\n";
         return 1;
}
TeAttrTableVector tableVec;
if (db->getAttrTables(tableVec, TeAttrExternal))
{        cout << "Tabelas exetrnas no banco de dados: "
            << tableVec.size() << endl;
         TeAttrTableVector::iterator it = tableVec.begin();
         while (it != tableVec.end())
         {  cout << (*it).name() << " Coluna com a chave primaria: "
               << (*it).uniqueName() << endl << endl;
                  ++it;
         }
}
db->close();
```

Voltar para o índice.

8. Acessando um banco de dados TerraLib

8.1 Acesso através do Layer

As rotinas de importação que inserem um arquivo de dados geográfico são escritas usando os métodos da classe `TeLayer`. Essa classe fornece métodos para a inserção de geometrias e atributos. Através desses métodos os registros nas tabelas do modelo conceitual são preenchidos corretamente, fazendo as referências ao identificador do *layer* quando necessário.

O Exemplo 8.1 deve ser lido como uma seqüência do Exemplo 6.2 e do Exemplo 6.3 e mostra como criar um *layer* e como usar seus métodos para inserir no banco de dados as geometrias e atributos criados em memória.

Exemplo 8.1 - Inserção de geometrias e atributos no banco através da classe

```
// Cria uma projeção
TeDatum mDatum = TeDatumFactory::make("SAD69");
TeProjection* pUTM = new TeUtm(mDatum,0.0);

// Cria um novo layer chamado "TesteLayer"
string layerName = "TesteLayer";
if (db->layerExist(layerName))
{
         cout << "Ja existe um layer chamado \"TesteLayer\" no banco!\n\n";
         db->close();
         return 1;
}

TeLayer* layer = new TeLayer(layerName, db, pUTM);
if (layer->id() <= 0)      // O layer não foi criado corretamente
```

```
{
        cout << "Erro: " << db->errorMessage() << endl;
        db->close();
        return 1;
}

// Adiciona a representacao geometrica de linhas e armazena na tabela chamada
//  "TesteLayerLines"
if (!layer->addGeometry(TeLINES, "TesteLayerLines"))
{
        cout << "Erro: " << db->errorMessage() << endl;
        db->close();
        return 1;
}

// Cria um conjunto de linhas
TeLine2D reta;
reta.add(TeCoord2D(500,500));
reta.add(TeCoord2D(600,500));
reta.add(TeCoord2D(700,500));
reta.objectId("reta");

TeLine2D ele;
ele.add(TeCoord2D(500,600));
ele.add(TeCoord2D(600,600));
ele.add(TeCoord2D(700,700));
ele.add(TeCoord2D(800,600));
ele.add(TeCoord2D(900,600));
ele.objectId("ele");

TeLineSet ls;
ls.add(reta);
ls.add(ele);

// Adiciona o conjunto de linhas ao layer
if (!layer->addLines(ls))
{
        cout << "Erro: " << db->errorMessage() << endl;
        db->close();
        return 1;
}
else
cout << "Linhas inseridas no novo layer!\n";

// Cria um conjunto de poligonos
// Um poligono simples
TeLine2D line;
line.add(TeCoord2D(900,900));
line.add(TeCoord2D(900,1000));
line.add(TeCoord2D(1000,1000));
line.add(TeCoord2D(1000,900));
line.add(TeCoord2D(900,900));
TeLinearRing r1(line);
TePolygon poly1;
poly1.add(r1);
poly1.objectId("spoli");

// Um poligono com um buraco
TeLine2D line2;
line2.add(TeCoord2D(200,200));
line2.add(TeCoord2D(200,400));
```

```
line2.add(TeCoord2D(400,400));
line2.add(TeCoord2D(400,200));
line2.add(TeCoord2D(200,200));

TeLinearRing r2(line2);

TeLine2D line3;
line3.add(TeCoord2D(250,250));
line3.add(TeCoord2D(250,300));
line3.add(TeCoord2D(300,300));
line3.add(TeCoord2D(300,250));
line3.add(TeCoord2D(250,250));

TeLinearRing r3(line3);

TePolygon poly2;
poly2.add(r2);
poly2.add(r3);
poly2.objectId("cpoli");

TePolygonSet ps;
ps.add(poly1);
ps.add(poly2);

// Adiciona o conjunto de poligonos ao layer
// Como o método addGeometry não foi chamado antes, a tabela
// de poligonos terá um nome padrão
if (!layer->addPolygons(ps))
{
        cout << "Error: " << db->errorMessage() << endl;
        db->close();
        return 1;
}
else
        cout << "Poligonos inseridos no novo layer!\n";

// Cria um conjunto de pontos
TePoint p1(40,40);
p1.objectId("ponto1");
TePoint p2(65,65);
p2.objectId("ponto2");

TePointSet pos;
pos.add(p1);
pos.add(p2);
// Adiciona um conjunto de pontos ao layer
// Como o método addGeometry não foi chamado antes, a tabela
// de pontos terá um nome padrão.
if (!layer->addPoints(pos))
{
        cout << "Erro: " << db->errorMessage() << endl;
        db->close();
        return 1;
}
else
        cout << "Pontos inseridos no novo layer!\n";

// Cria uma tabela de atributos
// Define a lista de atributos
TeAttributeList attList;
TeAttribute at;
```

```
at.rep_.type_ = TeSTRING;
at.rep_.numChar_ = 16;
at.rep_.name_ = "object_id";
at.rep_.isPrimaryKey_ = true;
attList.push_back(at);

at.rep_.type_ = TeSTRING;       // atributo do tipo string
at.rep_.numChar_ = 255;
at.rep_.name_ = "nome";
at.rep_.isPrimaryKey_ = false;
attList.push_back(at);

at.rep_.type_ = TeREAL; // atributo do tipo float
at.rep_.name_ = "val1";
at.rep_.isPrimaryKey_ = false;
attList.push_back(at);

at.rep_.type_ = TeINT;          // atributo do tipo integer
at.rep_.name_ = "val2";
at.rep_.isPrimaryKey_ = false;
attList.push_back(at);

// Cria uma tabela de atributos associada ao novo layer,
// chamada "TesteLayerAttr"
TeTable attTable("TesteLayerAttr", attList, "object_id", "object_id");          // cria a tabela em memoria
if (!layer->createAttributeTable(attTable))
{
         cout << "Erro: " << db->errorMessage() << endl << endl;
         db->close();
         return 1;
}


// Cada linha está relacionada com uma das geometries através do seu
// objetc_id
TeTableRow row;
row.push_back("reta");
row.push_back("An straight line");
row.push_back("11.1");
row.push_back("11");
attTable.add(row);
row.clear();

row.push_back("ele");
row.push_back("A mountain shaped line");
row.push_back("22.2");
row.push_back("22");
attTable.add(row);
row.clear();

row.push_back("spoli");
row.push_back("A simple polygon");
row.push_back("33.3");
row.push_back("33");
attTable.add(row);

row.clear();
row.push_back("ponto1");
row.push_back("A point");
row.push_back("55.5");
row.push_back("55");
```

```
attTable.add(row);
row.clear();

row.push_back("ponto2");
row.push_back("Another point");
row.push_back("66.6");
row.push_back("66");
attTable.add(row);
row.clear();

row.push_back("cpoli");
row.push_back("A polygon with hole");
row.push_back("44.4");
row.push_back("44");
attTable.add(row);
row.clear();

// Salva a tabela no banco
if (!layer->saveAttributeTable( attTable ))
{
        cout << "Erro: " << db->errorMessage() << endl << endl;
        db->close();
        return 1;
}
else
cout << "Objetos inseridos no novo layer!\n\n";
```

O Exemplo 8.1 mostra como controlar a inserção de cada tipo de geometria (pontos, linhas ou polígonos) podendo inclusive especificar o nome para as tabelas de geometrias (como no caso da inserção de linhas). É interessante verificar o conteúdo do banco de dados após a execução deste programa.
Devem ser notados o conteúdo das tabelas de metadados e as referências feitas aos nomes das tabelas de geometrias, aos nomes das tabelas de atributos, ao identificador do *layer*, e o registro dos nomes de colunas usados para relacionar geometrias e atributos. Na Figura 8.1 são representados os conteúdos dessas tabelas e seus relacionamentos. Para efeito de clareza nem todos os campos das tabelas são representados. As linhas contínuas representam relacionamentos físicos entre os campos das tabelas e as linhas pontilhadas representam relacionamentos em termos de conteúdo. Por exemplo, o nome da tabela de atributos e o nome do campo que faz o seu relacionamento com as tabelas de geometrias são registrados na tabela **te_layer_table**.

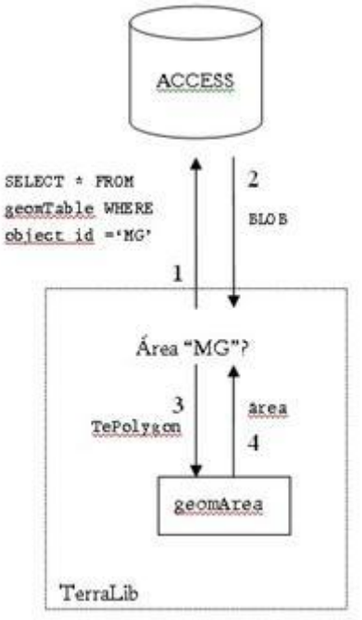


Figura 8.1 – Conteúdo do banco após a execução do Exemplo 8.1.

A recuperação dos dados armazenados no banco, também pode ser feita diretamente através da classe `TeLayer`. Essa classe fornece alguns métodos para recuperar as geometrias de um determinado tipo como mostrado no Exemplo 8.2.

Exemplo 8.2 - Recuperação de geometrias através do *layer*.

```
TePolygonSet ps2;
```

```
layer1->getPolygons(ps2);
cout << "Numero de objetos com geometria do tipo poligono: " << ps2.size();
ps2.clear();
// recupera somente os polígonos associados ao objeto com o
// identificador "spoli"
layer1->loadGeometrySet("spoli", ps2);
```

Os métodos para recuperação de geometrias e atributos através da classe `TeLayer` são poucos, mas a classe `TeDatabase` é uma interface completa de acesso ao banco de dados, essa interface será descrita a seguir.

8.2 Acesso através da interface TeDatabase

A classe `TeDatabase` é uma interface completa de manipulação do banco de dados através da qual é possível inserir, alterar e recuperar as entidades do modelo conceitual e os dados geográficos. Ela contém todos os métodos para criar as tabelas de metadados, as tabelas de geometrias bem como uma tabela de atributos como ilustrado no Exemplo 8.3. Esse exemplo mostra também como recuperar um *layer* do banco e posteriormente como removê-lo. A remoção do *layer* implica que todos os seus dados e metadados são fisicamente removidos do banco bem como todas as outras entidades do modelo conceitual que fazem referência a ele. Por exemplo, ao remover um *layer* do banco todos os temas criados sobre esse *layer* devem ser removidos também.

Exemplo 8.3 - Métodos de criação e modificação de tabelas da classe `TeDatabase`.

```
db->createConceptualModel(); // cria todo o modelo conceitual
// cria uma tabela que armazena geometrias do tipo polígono
db->createPolygonGeometry("TabelaPoligonos");

// cria uma tabela de atributos descritivos
TeAttributeList meusAttributos;
TeAttribute coluna;
coluna.rep_.type_ = TeSTRING;
coluna.rep_.name_ = "coluna1";
coluna.rep_.numChar_ = 10;
meusAttributos.push_back(coluna);

db->createTable("TabelaAtributos", meusAttributos);

// cria uma nova coluna em uma tabela já existente
TeAttributeRep novaColuna;
novaColuna.type_ = TeSTRING;
novaColuna.name_ = "novaCol";
novaColuna.numChar_ = 10;
db->addColumn("TabelaAtributos", novaColuna);
// recupera um layer do banco
TeLayer *layer = new TeLayer("Distritos");
// carrega todas as informações sobre o layer chamado Distritos
db->loadLayer(layer);
// remove o layer
db->deleteLayer(layer->id());
```

A classe `TeDatabase` também é capaz de submeter ao banco comandos escritos em SQL – Structured Query Language como mostrado no Exemplo 8.5. Apesar da SQL ser considerada padrão para SGDB's relacionais, podem existir algumas variações em termos da capacidade de execução de um comando SQL. O método `TeDatabase::execute` retorna verdadeiro ou falso caso o comando foi executado com sucesso ou não, respectivamente. A classe `TeDatabase` armazena o erro retornado pelo SGDB resultante do último comando executado sobre o banco. A classe TeDatabase também fornece métodos para a inserção de geometrias e atributos nas tabelas do banco de dados como mostra o Exemplo 8.4.

Exemplo 8.4 - Inserções no banco através da classe TeDatabase.

```
TePolygonSet polyset;
db->insertPolygonSet("tabPoligono", polyset);

TeLineSet lineset;
db->insertLineSet ("tabLine", lineset);

TePointSet pointset;
db->insertPointSet("tabPoint", pointset);
```

```
TeTable tabAttributes;
insertTable(tabAttributes);
```

Exemplo 8.5 - Execução de um comando SQL.

```
string sql = "UPDATE TabelaAtributos SET novaCol = 'xxx' ";
if (db->execute(sql))
        cout << "Comando executado com sucesso!";
else
        cout << "Erro: " << db->errorMessage();
```

Note que o método `TeDatabase::execute` não deve ser usado para comandos que representam consultas ao banco, ou seja, que retornam valores ou registros, mas somente para comandos que alteram as tabelas e registros do banco.

8.2.1. A classe TeDatabasePortal

A classe `TeDatabase` fornece um mecanismo mais flexível de consulta ao banco de dados do que apenas através dos métodos da classe `TeDatabase`.
Esse mecanismo é implementado pela classe `TeDatabasePortal`. Os portais de consultas são criados a partir de qualquer instância da classe `TeDatabase` que possua uma conexão aberta. Os portais podem submeter consultas SQL ao banco e disponibilizar os registros resultantes da consulta para a aplicação processar. Um banco pode criar quantos portais forem necessários, porém após serem consumidos os resultados todo portal deve ser destruído. O Exemplo 8.6 mostra o uso típico de um portal.

Exemplo 8.6 - Uso do `TeDatabasePortal`.

```
// Abre um portal no banco de dados
TeDatabasePortal* portal = db->getPortal();
if (!portal)
         return -1;

// Submete uma consulta sql a uma tabela
string sql = "SELECT * FROM TabelaAtributos";
if (!portal->query(sql))
{
         cout << "Não foi possível executar..." << db->errorMessage();
   delete portal;
   return -1;
}
// Consome todos os registros resultantes
while (portal->fetchRow())
{
         cout << "Atributo 1: " << portal->getData(0) << endl;
   cout << "Atributo 2: " << portal->getData("nome") << endl;
   cout << "Atributo 3: " << portal->getDouble(2) << endl;
}
// Libera os portal para fazer uma nova consulta
portal->freeResult();

// Submete uma nova consulta
sql = "SELECT SUM(val1) FROM TabelaAtributos WHERE val1 > 33.5";
if (portal->query(sql) && portal->fetchRow())
         cout << "Soma: " << portal->getDouble(0);
delete portal;
```

Analisando o exemplo vemos que após a chamada do método `TeDatabasePortal::query` os registros ficam disponíveis para serem consumidos em seqüência. Um ponteiro interno é posicionado em uma posição *anterior ao primeiro registro*.
A cada chamada ao método `TeDatabasePortal::fetchRow` o ponteiro interno é incrementado e o valor verdadeiro é retornado quando o ponteiro está em um registro válido ou falso caso contrário. Dessa forma é possível fazer o *loop* em todos os registros retornados.

Os campos de um registro do portal podem ser acessados de duas formas: pela ordem do campo na seleção expressa na SQL (iniciando em zero), ou diretamente pelo nome do campo. O valor do campo pode ser obtido como uma *string* de caracteres através do método `TeDatabasePortal::getData`[8] independentemente do tipo do campo ou através dos métodos que especificam o tipo

de retorno (`getDouble, getInt, getDate` ou `getBlob`).
Na segunda forma é necessário que a aplicação conheça o tipo do campo sendo acessado.

A classe `TeDatabasePortal` também pode executar consultas sobre uma tabela de geometrias e acessar os campos resultantes como os tipos geométricos da TerraLib, como mostrado no Exemplo 8.7.

Exemplo 8.7 - Uso de um portal para recuperar geometrias

```
// Submete uma consulta sobre uma tabela de geometrias
string q =" SELECT * FROM Polygons11 WHERE object_id='cpoli'";
       q += " ORDER BY parent_id, num_holes DESC, ext_max ASC";

if (!portal->query(q) || !portal->fetchRow())
{
         delete portal;
         return false;
}
bool flag = true;
do
{
         TePolygon poly;
         flag = portal->fetchGeometry(poly);
         // usa o polígono retornado
} while (flag);
delete portal;
```

Voltar para o índice.

9. Tema

Um tema representa um subconjunto dos objetos de um layer.
Esse subconjunto pode conter todos os objetos do layer ou somente aqueles que satisfazem algumas restrições definidas no tema, como por exemplo, uma restrição sobre algum de seus atributos. O tema contém informações sobre o visual da apresentação gráfica da componente geométrica do dado. Quando os objetos do tema são agrupados, ele guarda informações sobre o tipo de agrupamento usado, o número de grupos e as legendas geradas.

Um tema existe em memória como uma instância da classe `TeTheme` e está registrado em um banco TerraLib como um registro da tabela do modelo conceitual **te_theme**.

Após serem definidas os critérios que geram um tema, ou seja, quais as restrições devem ser aplicadas, os objetos que atendem a esses critérios são registrados em uma tabela de coleção, materializando assim as definições do tema. As tabelas de coleção pertencem ao modelo conceitual. Para cada tema gerado é criada também uma tabela de coleção, que tem por finalidade otimizar a manipulação dos objetos desse tema.

9.1 Vista

Uma vista representa uma visão sobre os dados de um banco TerraLib. Ou seja, define quais os temas são apresentados simultaneamente. Como cada tema pode vir de um layer com projeção diferente, a vista também determina qual a projeção comum para apresentar os temas que agrega. Assim, os temas agregados que possuem uma projeção diferente da vista são remapeados para a projeção da vista. É importante notar que o sistema de coordenadas não cartográfico (“NoProjection”) não pode ser remapeado para projeção nenhuma.

Uma vista existe em memória como uma instância da classe `TeView` e está registrado em um banco TerraLib como um registro da tabela do modelo conceitual **te_view**.

9.2 Visual

Um visual representa um conjunto de características de apresentação de primitivas geométricas. Por exemplo, cores de preenchimento e contorno de polígonos, espessuras de contornos e linhas, cores de pontos, símbolos de pontos, tipos e transparência de preenchimento de polígonos, estilos de linhas, estilos de pontos, etc.

Um visual existe em memória como uma instância da classe `TeVisual` e está registrado em um banco TerraLib como um registro da tabela do modelo conceitual **te_visual**.

Exemplo 9.1 - Criar uma vista e dois temas a partir de um layer. O primeiro sem nenhuma restrição e o segundo com uma restrição de atributos.

```
// Abre uma conexão com o servidor MySQL
// Carrega o layer "Distritos" que contém dados da cidade de São Paulo
TeLayer* dist = new TeLayer("Distritos");
```

```
db->loadLayer(dist);

TeProjection* proj = dist->projection();
// Cria uma vista com na mesma projeção do layer
string viewName = "SaoPaulo";
// Checa se já existe uma vista com este nome no banco
if (db->viewExist(viewName))
{
        db->close();
        return 1;
}
TeView* view = new TeView(viewName, user);
view->projection(proj);
if (!db->insertView(view))         // salva a vista no banco
{
        db->close();
        return 1;
}
// Cria um tema que conterá todos os objetos do layer
//(nenhuma restrição é aplicada)
TeTheme* theme = new TeTheme("DistritosSaoPaulo", dist);
view->add(theme);

// Seta o visual default das geometrias dos objetos do layer
// Para polígonos será setada a cor azul
TeColor color;
TeVisual polygonVisual(TePOLYGONS);
color.init(0,0,255);
polygonVisual.color(color);
theme->setVisualDefault(polygonVisual, TePOLYGONS);
// Para os pontos será setada a cor vermelha
TeVisual pointVisual(TePOINTS);
color.init(255,0,0);
pointVisual.color(color);
pointVisual.style(TePtTypeX);
theme->setVisualDefault(pointVisual, TePOINTS);

// Marca todas as representações geométricas visíveis
int allRep = dist->geomRep();
theme->visibleRep(allRep);

// Mostra todas as tabelas de atributos do layer
theme->setAttTables(dist->attrTables());
// Salva o tema no banco
if (!theme->save())
{
        db->close();
        return 1;
}

// Constrói a coleção de objetos associada ao tema
if (!theme->buildCollection())
{
        db->close();
        return 1;
}
cout << "O tema \"DistritosSaoPaulo\" foi criado sem restricoes!\n";
// Cria um tema com a seguinte restrição de atributo:
// Distritos com população maiores que 100,000 pessoas

TeTheme* themeRest = new TeTheme("Pop91GT100000", dist);
```

```
themeRest->setAttTables (dist->attrTables());

// Ajusta a restrição de atributo
string restAttr = " Pop91 > 100000 ";
themeRest->attributeRest(restAttr);
// Marca todas as representações geométricas visíveis
themeRest->visibleRep(allRep);

// Ajusta o visual
themeRest->setVisualDefault(polygonVisual, TePOLYGONS);
themeRest->setVisualDefault(pointVisual, TePOINTS);
// Insere o tema na vista
view->add(themeRest);
// Salva o tema no banco
if (!themeRest->save())
{
        db->close();
        return 1;
}

// Constrói a coleção de objetos associada ao tema
if (!themeRest->buildCollection())
{
        db->close();
        return 1;
}
cout << "O tema  \"Pop91GT100000\" foi criado com restrição de atributo!\n\n";
db->close();
```

Observar no banco as tabelas de coleção e as entradas nas tabelas **te_theme** e **te_view**. Usar o aplicativo TerraView como ferramenta de visualização do banco TerraLib.

Um tema pode decidir quais as tabelas de atributos de um layer ele irá utilizar, como mostrado no Exemplo 9.2. Após executar este exemplo, verificar no banco os registros criados na tabela **te_theme_table**.

Exemplo 9.2 - Criar um tema selecionando apenas a segunda tabela de attributos de um layer.

```
TeLayer* recife = new TeLayer("BairrosRecife",db);
TeTable attTable2;
recife->getAttrTablesByName("BairrosRecife2",attTable2);
TeView* viewrec = new TeView("Recife",user);
viewrec->projection(recife->projection());
if (!db->insertView(viewrec))      // salva a vista no banco
{
        cout << "Falha ao inserir a vista \"Recife\" no banco: " <<
            db->errorMessage() << endl;
        db->close();
        return 1;
}
TeTheme* themerec = new TeTheme("Recife", recife);
viewrec->add(themerec);
themerec ->addThemeTable(attTable2);
themerec->visibleRep(recife->geomRep());
if (!themerec->save() || !themerec->buildCollection())
{
        cout << "Erro ao tentar salvar o tema \' Recife \': "
          << db->errorMessage() << endl;
        db->close();
        return 1;
}
cout << "Tema criado ..\n";
db->close();
```

9.3 Agrupamento de objetos de um Tema

O tema engloba também o conceito de agrupamento sobre seus objetos, baseado em seus atributos. Para isso existem várias funções de agrupamento como por valor único, por quantil ou passo igual. A cada grupo gerado é associada uma legenda, que finalmente aponta para um visual característico daquele grupo. Além disso, a legenda contém outras informações referentes ao grupo como limite inferior e superior, o número de objetos pertencentes a esse grupo ou o rótulo associado a ele.

O agrupamento presente em um tema existe em memória como uma instância da classe `TeGrouping` e está registrado em um banco TerraLib como um registro da tabela do modelo conceitual **te_grouping**. Cada um dos grupos encontrados existe em memória como uma instância da classe `TeLegendEntry`, que contém as informações sobre o grupo e seu visual de apresentação. Em um banco TerraLib os grupos são persistidos em registros da tabela **te_legend**, e os visuais na tabela **te_visual**.

Exemplo 9.3 - Criar um agrupamento sobre um tema já existente.

```
// Carrega um tema previamente criado chamado DistritosSaoPaulo.
// Esse tema foi criado sobre um layer dos distritos metropolitanos
// da cidade de Sao Paulo
TeTheme* trmsp = new TeTheme("DistritosSaoPaulo");
if (!db->loadTheme(trmsp))
{
         db->close();
         return 1;
}
trmsp->resetGrouping();    // Limpa qualquer agrupamento previamente realizado

TeAttributeRep rep;
// O agrupamento sera baseado no atributo chamado "area_km2"
// que contém a área de cada distrito meyropolitano
rep.name_ = " area_km2 ";
rep.type_ = TeREAL;

TeGrouping equalstep3;     // define um modo específico de agrupar os objetos
equalstep3.groupAttribute_ = rep;
equalstep3.groupMode_ = TeEqualSteps;       // Passos iguais em três classes (grupos)
equalstep3.groupNumSlices_ = 3;

// Gera os grupos baseados na definição do agrupamento
if (!trmsp->buildGrouping(&equalstep3))
{
         db->close();
         return 1;
}

// Associa um visual diferente para cada grupo
TeVisual visual(TePOLYGONS);                // grupo 1: verde escuro
TeColor color(0,200,0);
visual.color(color);
trmsp->setGroupingVisual(1,visual,TePOLYGONS);

color.init(0,150,0);
visual.color(color);               // grupo 2: verde
trmsp->setGroupingVisual(2,visual,TePOLYGONS);

color.init(0,100,0);
visual.color(color);               // grupo 3: verde claro
trmsp->setGroupingVisual(3,visual,TePOLYGONS);

if (!trmsp->saveGrouping())                 // salva a definição do agrupamento
{
         db->close();
         return 1;
}
db->close();
```

Observar as modificações nas tabelas de coleção, de agrupamento, visual e legenda. Visualizar o resultado no TerraView.

Voltar para o índice.

10. Dados matriciais

Dados matriciais, ou raster, na TerraLib são considerados como um tipo de geometria associada a um determinado objeto e são armazenados em tabelas de geometria raster segundo um determinado padrão. A interface de manipulação de dados raster na Terralib é feita através das classes:

- `TeRaster`[9]: classe genérica de manipulação de um dado raster independente de formato, tamanho de cada elemento do raster ou dispositivo de armazenamento. Para isso fornece os métodos `setElement` e `getElement`, com os quais podem ser escritas as aplicações.
- `TeRasterParams`: estrutura que armazena todos os parâmetros específicos de um determinado raster. Esses parâmetros incluem número de linhas, colunas e bandas, tipo do elemento, número de bits por elemento, projeção, retângulo envolvente, etc. Cada `TeRaster` possui um `TeRasterParams`.
- `TeDecoder`: classe responsável por permitir o acesso a um dado raster específico. O decoder associado a um dado raster pode ser especificado explicitamente ou inferido de algum parâmetro, por exemplo, extensões de arquivos de dado raster.

Usando a classe `TeRaster` e seus parâmetros pode-se escrever uma rotina que percorra alguns elementos de um dado raster:

```
void mostra (TeRaster* rst)
{
    TeRasterParams params = rst->params();
    cout << "Nro Bandas: " << params.nBands() << endl;
    cout << "Nro Linhas: " << params.nlines_ << endl;
    cout << "Nro Colunas: " << params.ncols_ << endl;
    cout << "Projeção: " << params.projection()->name() << endl;
    double val;
    if (rst->getElement(45,2,val,0))
       cout << "valor do elemento linha 2 e coluna 45: " << val << endl;
    if (rst->getElement(67,2,val,0))
       cout << "valor do elemento linha 2 e coluna 67: " << val << endl;

    if (rst->getElement(300,2,val,0))
       cout << "valor do elemento linha 2 e coluna 300: " << val <<endl;
         else
       cout << "nao existe elemento na linha 2 e coluna 300!" << endl;
}
```

Exemplo 10.1 - Criar dados matriciais em diferentes formatos e dispositivos, utilizando a rotina mostrada acima para visualizar algumas informações desse dado.

Parte A - Manipular um raster em memória

```
void main() {
// inicializa o sistema de decodificacao de dados raster
TeInitRasterDecoders();
// define uma projecao
TeProjection* noproj = new TeNoProjection();

// define parametros de um dado raster em memória
TeRasterParams  params;
params.projection(noproj);                  // projecao
params.setPhotometric(TeRasterParams::TeMultiBand);// valor photometrico
params.nBands(1);                                   // numero de bandas
params.setDataType(TeFLOAT);                // tamanho de cada elem
params.boundingBoxLinesColumns(0,0,99,99,100,100);
params.decoderIdentifier_ = "MEM";  // decodificador a ser usado
params.mode_ = 'c';        // escrita

cout << "Box: [" << params.box().x1_ << ",";
cout << params.box().y1_ << ",";
cout << params.box().x2_ << ",";
```

```
cout << params.box().y2_ << "]\n";
cout << "Bounding Box: [" << params.boundingBox().x1_ << ",";
cout << params.boundingBox().y1_ << ",";
cout << params.boundingBox().x2_ << ",";
cout << params.boundingBox().y2_  <<  "]\n\n";

TeRaster* rastermem = new TeRaster(params);
rastermem->init();
if (!rastermem->status())
{
         cout << "Falha ao criar raster em memoria." << endl;
         return 1;
}
// preenche com alguns valores
for (int l=0; lsetElement(c,l,c+1);
}
mostra(rastermem);
}
```

Parte B - Acessa um raster em um arquivo binário sem formato ou header.

```
// define parametros de um dado raster em memória
TeRasterParams  params;
params.projection(noproj);
params.setPhotometric(TeRASTERMULTIBAND);
params.nBands(1);
params.setDataType(TeUNSIGNEDCHAR);
// parametros do retangulo envolvente
params.topLeftResolutionSize(7404193,321045,
                             30,30,589,703,true);
params.decoderIdentifier_ = "MEMMAP";
params.fileName_ = "../data/sp_589x703.raw";
params.mode_ = 'r';
cout << "Box: [" << params.box().x1_ << ",";
cout << params.box().y1_ << ",";
cout << params.box().x2_ << ",";
cout << params.box().y2_  <<  "]\n";
cout << "Bounding Box: [" << params.boundingBox().x1_ << ",";
cout  << params.boundingBox().y1_<< ","  ;
cout << params.boundingBox().x2_ << "," ;
cout << params.boundingBox().y2_  <<  "]\n\n";
TeRaster* rasterraw = new TeRaster(params);
rasterraw->init();
if (!rasterraw->status()){
         cout << "Falha ao incializar raster do arquivo." << endl;
         return;
}
 mostra(rasterraw);
}
```

Parte C - Criar um dado raster a partir de um dado em formato GeoTiff. Todos os parâmetros são conhecidos nesse formato.

```
void main() {
// inicializa o sistema de decodificacao de dados raster
TeInitRasterDecoders();
TeRaster* rastertif = new TeRaster("../data/nat1.tif");
rastertif->init();
if (!rastertif->status())  {
         cout << "Falha ao incializar raster do arquivo." << endl;
         return;
}
```

```
TeRasterParams params = rastertif->params();

cout << " \n --- Raster TIFF --- \n";
mostra(rastertif);
TeRaster* rasterjpg = new TeRaster("../data/ sampa.jpg");
rasterjpg ->init();
if (!rasterjpg->status())  {
        cout << "Falha ao incializar raster do arquivo." << endl;
        return;
}
cout << " \n --- Raster JPEG ---- \n";
mostra(rasterjpg);
}
```

10.1. Armazenamento em um banco de dados TerraLib

Assim como no caso das geometrias vetoriais, existe um padrão proprietário de TerraLib para o armazenamento de geometrias matriciais em tabelas relacionais que leva em conta questões como otimização de armazenamento e eficiência na recuperação. Em um banco de dados TerraLib, as geometrias são armazenadas em campos binários longos de tabelas relacionais ou em tipos abstratos de dados fornecidos por SGBD com extensão espacial, os quais são codificados e decodificadas pelas classes da TerraLib. Os dados matriciais ainda não são tratados pelas extensões espaciais conhecidas.

TerraLib propõe que, para o armazenamento de um dado matricial em um banco de dados, esse seja inicialmente divido em blocos de tamanhos fixos. Cada bloco é espacialmente indexado, unicamente identificado e armazenado em um registro de uma tabela de geometria matricial.

Esse armazenamento é feito através da rotina de importação `TeImportRaster`.Essa rotina importa um dado matricial como uma geometria associada a um objeto de um layer.

Exemplo 10.2 - Importar um dado raster para um banco de dados TerraLib

```
#include <TeDatabase.h>
#include <TeMySQL.h>
#include <TeInitRasterDecoders.h>
#include <TeImportRaster.h>
int main()
{
        TeInitRasterDecoders(); // Inicializa o decodificador do raster
        TeRaster image("../data/sampa.jpg"); // Acessa a imagem de entrada
        if (!image.init())
        {
                cout  << "Não é possível acessar a imagem de entrada!" << endl
               << endl;
                return 1;
        }
        // Parametros do servidor
        string host = "localhost";
        string dbname = "TerraTeste";
        string user = "root";
        string password = "";
        TeDatabase* db = new TeMySQL();
        if (!db->connect(host, user, password, dbname))
        {
                image.clear();
                cout << "Erro: " << db->errorMessage() << endl << endl;
                return 1;
        }
        string layerName = "SampaJPEG";
        if (db->layerExist(layerName))
        {
                cout << "O banco já contem um layer com o nome \"";
                cout << layerName << "\"!" << endl << endl;
                db->close();
                return 1;
```

```
        }
        // Cria um layer para receber a geometria matricial
        TeLayer* layer = new TeLayer(layerName, db, image.projection());
        if (layer->id() <= 0)
        {
                image.clear();
                db->close();
                cout << "O layer de destino nao pode ser criado!\n"
               << db->errorMessage() << endl << endl;
                return 1;
        }
        // Importa o raster para o layer
        if (!TeImportRaster(layer, &image, 128, 128))
        {
                image.clear();
                db->close();
                cout << "Falha ao importar imagem!\n" << endl;
                return 1;
        }
        db->close();
        cout << "A imagem JPEG foi importada com sucesso!\n\n";
        return 0;
}
```

A rotina de importação também permite que um conjunto de dados raster sejam agregados em uma mesma representação, formando um mosaico. Para isso o segundo (e subseqüentes dados) devem ser importados para uma geometria já existente em um layer.

Exemplo 10.3 - Construir uma geometria raster através do mosaico de 2 arquivos de imagens diferentes (nat1.tiff e nat2.tiff).

```
#include <TeDatabase.h>
#include <TeMySQL.h>
#include <TeInitRasterDecoders.h>
#include <TeImportRaster.h>
int main()
{
        TeInitRasterDecoders();    // Inicializa o decodificador do raster
        // Acessa a imagem de entrada
        TeRaster img1("../data/nat1.tif");
        if (!img1.init())
        {
                cout << "Não é possível acessar a primeira imagem de entrada!"
                << endl << endl;
                return 1;
        }
        TeRaster img2("../data/nat2.tif");
        if (!img2.init())
        {
                cout << " Não é possível acessar a segunda imagem de entrada!"
                << endl << endl;
                return 1;
        }
    string host = "localhost";
    string dbname = "TerraTeste";
    string user = "root";
    string password = "";

    TeDatabase* db = new TeMySQL();
    if (!db->connect(host, user, password, dbname))
    {
                cout << "Error: " << db->errorMessage() << endl << endl;
    }
```

```
        // Cria um layer para receber a geometria matricial (mesma
        // projecao do raster)
        string layerName = "NatividadeMosaic";
        // Checa se ja existe um layer com esse nome no banco
        if (db->layerExist(layerName))
        {
                db->close();
                cout << "O banco já contem um layer com o nome \"";
                cout << layerName << "\"!" << endl << endl;
                return 1;
        }
        TeLayer* layer = new TeLayer(layerName, db, img1.projection());
        if (layer->id() <= 0)
        {
                db->close();
                cout << "O layer de destino não pode ser criado!\n"
                << db->errorMessage() << endl;
                return 1;
        }

        // Importa a primeira imagem para o layer
        if (!TeImportRaster(layer,&img1,256,256,
                        TeRasterParams::TeNoCompression,"",255,true,
                        TeRasterParams::TeExpansible))
        {
                db->close();
                cout << "Falha ao importar a primeira imagem\n\n!";
                return 1;
        }
        else
                cout << "A primeira imagem foi importada com sucesso!\n\n";

        delete layer;
        layer = new TeLayer(layerName, db);
        // Mosaica a Segunda imagem nomeso layer
        if (!TeImportRaster(layer, &img2, 256, 256,
                        TeRasterParams::TeNoCompression, "", 255,
                        true, TeRasterParams::TeExpansible))
        {
                db->close();
                cout << "Falha ao importar a segunda imagem\n\n!";
                return 1;
        }
        else
                cout << "A segunda imagem foi adicionada com sucesso!"
                << endl << endl;
        db->close();
        return 0;
}
```

A recuperação de um dado raster armazenado em um banco TerraLib é feita diretamente pelo layer.

Exemplo 10.4 - Recuperar a geometria raster associada a dois objetos diferentes de um layer chamado “Brasilia”.

```
TeLayer* brasLayer = new TeLayer("Brasilia", db); // recupera layer
TeRaster* bras = brasLayer->raster();       // recupera primeiro raster

TeRaster* sul = brasLayer->raster("sul");   // geometria do objeto "sul"
TeRaster* norte = brasLayer->raster("norte"); // geometria do objeto "norte"
```

Voltar para o índice.

11. Operações espaciais

A TerraLib oferece um conjunto de operações espaciais sobre geometrias vetoriais e matriciais (uma revisão da literatura e descrição mais detalhada sobre operações espaciais pode ser encontrada em Ferreira, 2003). As operações espaciais implementadas na TerraLib podem ser divididas em:

- Determinação de relacionamentos topológicos entre geometrias vetoriais: são baseados na matriz de 9-intersecções dimensionalmente estendidas onde são formalizadas relações como toca, intercepta, cobre, disjunto ou é-coberto-por;
- **Funções métricas**: cálculo de área, comprimento ou perímetro e distância entre geometrias;
- **Operações que geram novas geometrias**: *buffer* (gera uma nova geometria a partir de uma distância de um objeto específico), centróides e geração de uma geometria convexa;
- **Operações que combinam geometrias**: como diferença, união e intersecção e diferença simétrica;
- **Operações zonais e focais sobre geometrias matriciais**: como a obtenção de medidas estatísticas sobre uma região de um dado matricial e operações de recorte de uma região do dado matricial.

As funções que implementam essas operações espaciais estão definidas nos arquivos `TeGeometryAlgorithms`, `TeBufferRegion` e `TeOverlay` do kernel da biblioteca e recebem como parâmetros os tipos geométricos da TerraLib. Os exemplos a seguir mostram como essas funções são utilizadas.

Exemplo 11.1 - Dado um polígono, calcular sua área e perímetro, gerar seu centróide e verificar quais pontos estão dentro dele. Neste exemplo, as geometrias estão em memória como instâncias das classes geométricas da TerraLib.

```
//Seta a precisão que sera considerada nas operações espaciais
TePrecision::instance().setPrecision(0.001);

// criar um polígono
TeLine2D line;
line.add(TeCoord2D(480275.38, 6667799.34));
line.add(TeCoord2D(483337.32, 6666464.65));
line.add(TeCoord2D(486006.70, 6668898.50));
line.add(TeCoord2D(484593.50, 6671410.86));
line.add(TeCoord2D(481688.58, 6672431.51));
line.add(TeCoord2D(480275.38, 6667799.34));
TeLinearRing ring(line);
TePolygon poly;
poly.add(ring);

//calcula area
double area = TeGeometryArea(poly);
//calcula o cumprimento
double len = TeLength(ring);
//gera o centroide
TeCoord2D centroid = TeFindCentroid(poly);

//criar pontos
TePointSet points;
TePoint point1(TeCoord2D(480000.38, 6667799.34));
point1.objectId ("point_1");
points.add (point1);

TePoint point2(TeCoord2D(483180.30, 6668191.90));
point2.objectId ("point_2");
points.add (point2);

TePoint point3(TeCoord2D(483180.30, 6660000.90));
point3.objectId("point_3");
points.add (point3);

TePointSet::iterator it = points.begin();
while(it != points.end())
{
        if (TeWithin((*it), poly))
```

```
   cout << "O ponto " << (*it).objectId() << " esta
               dentro do poligono." << endl ;
        ++it;
}
```

O exemplo acima utilizou as funções `TeGeometryArea`, `TeLength`, `TeFindCentroid` e `TeWithin` e uma precisão que é definida através da classe `TePrecision`.

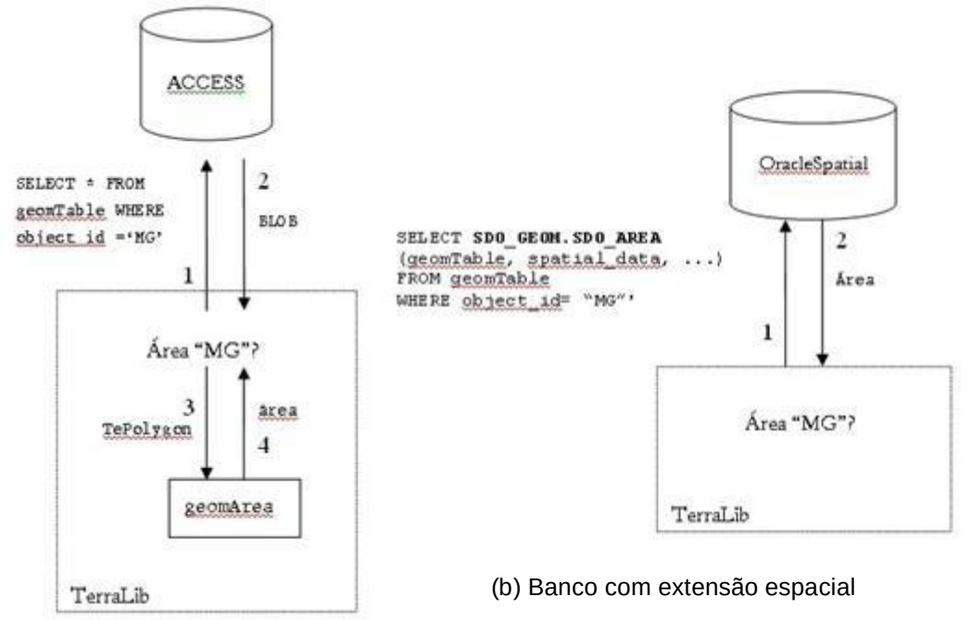
Exemplo 11.2 - Calcular a união e a interseção entre dois polígonos.

```
// criar um polígono 1
TeLine2D line1;
line1.add(TeCoord2D(480275.38, 6667799.34));
line1.add(TeCoord2D(483337.32, 6666464.65));
line1.add(TeCoord2D(486006.70, 6668898.50));
line1.add(TeCoord2D(484593.50, 6671410.86));
line1.add(TeCoord2D(481688.58, 6672431.51));
line1.add(TeCoord2D(480275.38, 6667799.34));
TeLinearRing ring(line1);
TePolygon poly1;
poly1.add(ring);

// criar um polígono 2
TeLine2D line2;
line2.add(TeCoord2D(484000.00, 6668000.00));
line2.add(TeCoord2D(484000.00, 6670000.00));
line2.add(TeCoord2D(485000.00, 6670000.00));
line2.add(TeCoord2D(485000.00, 6668000.00));
line2.add(TeCoord2D(484000.00, 6668000.00));
TeLinearRing ring2(line2);
TePolygon poly2;
poly2.add(ring2);

TePolygonSet polSet1;
polSet1.add(poly1);
TePolygonSet polSet2;
polSet2.add(poly2);
TePolygonSet polSetResult;

//Calcula a união entre os polígonos
bool res = TeOVERLAY::TeUnion(polSet1, polSet2, polSetResult);
polSetResult.clear();
//Calcula a interseção entre os polígonos
res = TeOVERLAY::TeIntersection(polSet1, polSet2, polSetResult);
```

11.1. Interface Genérica de Operações Espaciais

As operações espaciais também estão implementados em um conjuntos de métodos da classe `TeDatabase` que funcionam como uma Interface Genérica de Operações Espaciais ou API para Operações Espaciais. Essa interface é genérica porque, através da classe `TeDatabase`, esses métodos podem ser chamados da mesma maneira em *drivers* para SGDB's com extensão espacial ou sem extensão espacial. *Drivers* para SGDB's com extensão espacial implementam a interface de operações especiais usando as funções da extensão espacial. *Drivers* que não possuem a extensão implementam a interface através de funções espaciais básicas escritas sobre as classes de geometria, após a sua recuperação do banco de dados. Na Figura 11.1 mostramos a diferença de execução de uma operação espacial, cálculo de área de um objeto, executada em um *driver* sem extensão espacial (p. ex. ACCESS) e um *driver* com extensão espacial (p. ex. Oracle Spatial).

As operações sobre dados matriciais foram implementadas somente usando as classes da TerraLib, uma vez que as extensões espaciais ainda não fornecem as operação sobre dados matriciais. Uma descrição completa da interface genérica de operações espaciais em um banco de dados geográfico pode ser encontrada em Ferreira (2003).



(a) Banco sem extensão espacial

(b) Banco com extensão espacial

Figura 11.1 – Representação da operação de cálculo de área (adaptado de Ferreira, 2003).

A API para Operações Espaciais pode ser utilizada quando os dados geográficos estão armazenados em um banco TerraLib. Os exemplos a seguir mostram operações sobre layers em um banco TerraLib.

Exemplo 11.3 - Recuperar todos os estados adjacentes (`TeTOUCHES`), todos os rios que cruzam (`TeCROSSES`) e todas as cidades que estão dentro (`TeWITHIN`) do estado de identificador 23.
Este exemplo utiliza a Interface Genérica de Operações Espaciais da classe `TeDatabase`.

```
// Recupera do banco de dados um layer de polígonos, um de linha e
// outro de pontos
TeLayer* states = new TeLayer("states");    // states: polígonos
db->loadLayer(states);
TeLayer* rivers = new TeLayer("rivers");    // rivers: linhas
db->loadLayer(rivers);
TeLayer* cities = new TeLayer("cities");    // cities: pontos
db->loadLayer(cities);

// Armazena a identificação dos objetos resultantes
vector<string> objsOut;
vector<string> objsIn;  // objetos a serem consultados
objsIn.push_back("23");

// Seta a precisão que sera considerada nas operações espaciais
TePrecision::instance().setPrecision(
TeGetPrecision(states->layer()->projection()));

// Recupera os estados que tocam o estado 23
bool res = db->spatialRelation(states->tableName(TePOLYGONS),
TePOLYGONS, objsIn, objsOut, TeTOUCHES);
if (res)
{
        cout << "Estados que tocam o estado \"23\": \n";
        unsigned int i;
        for (i=0; i<objsOut.size(); i++)
                cout << "Estado: " << objsOut[i] << endl;
}

// Recupera os rios que cruzam o estado 23
res = db->spatialRelation(states->tableName(TePOLYGONS),
TePOLYGONS, objsIn,
```

```
rivers->tableName(TeLINES), TeLINES,
objsOut, TeCROSSES);
if(res)
{
        cout << "\nRios que cruzam o estado \"23\": \n";
        unsigned int i;
        for (i=0; i<objsOut.size(); i++)
                cout << "Rios: " << objsOut[i] << endl;
}

// Recupera as cidades que estão dentro do estado 23
res = db->spatialRelation(states->tableName(TePOLYGONS), TePOLYGONS,
objsIn, cities->tableName(TePOINTS),
TePOINTS, objsOut, TeWITHIN);
if (res)
{
        cout << "\nCidades que estão dentro do estado \"23\": \n";
        unsigned int i;
        for (i=0; i<objsOut.size(); i++)
                cout << "Cidade: " << objsOut[i] << endl;
}

db->close();
return 0;
```

Pode-se observar que o tipo da relação topológica é passado como parâmetro da função. As relações possíveis estão definidas no arquivo `TeDataTypes` e são: `TeDISJOINT`, `TeTOUCHES`, `TeCROSSES`, `TeWITHIN`, `TeOVERLAPS`, `TeCONTAINS`, `TeINTERSECTS`, `TeEQUALS`, `TeCOVERS`, `TeCOVEREDBY`.

Exemplo 11.4 - Recuperar todos os eventos de crime que ocorreram a uma distância máxima de 500 metros da fronteira do bairro São Geraldo (identificador 60).

```
// Recupera o layer "BairrosPoA"
TeLayer* bairros = new TeLayer("BairrosPoA");
db->loadLayer(bairros);
// Recupera o layer "OcorrenciasPoA"
TeLayer* events = new TeLayer("OcorrenciasPoA");
db->loadLayer(events);

// Guarda o identificador dos objetos resultantes
vector<string> objsOut;
vector<string> objsIn;  // objects to be queried
objsIn.push_back("60");

// Seta a precisão que sera considerada nas operações espaciais
TePrecision::instance().setPrecision(TeGetPrecision(bairros->projection()));

// Gera um buffer de 500 metros de distancia ao redor do bairro
TePolygonSet bufferPol;
if (!db->buffer(bairros->tableName(TePOLYGONS), TePOLYGONS,
                                objsIn, bufferPol, 500))
{
        cout << "Falha ao generar o buffer!" << endl;
        return 1;
}
// Recupera os eventos que estão dentro do buffer
if(!db->spatialRelation(events->tableName(TePOINTS), TePOINTS,
                                        &bufferPol, objsOut, TeWITHIN))
{
        cout << "Falha ao recuperar os eventos!" << endl;
        return 1;
}
```

```
// Mostra o resultado
cout << " Eventos encontradas:" << endl << endl;

for(unsigned int i=0; i<objsOut.size(); i++)
        cout << "      " << objsOut[i] << endl;

db->close();
return 0;
```

O exemplo anterior gerou um *buffer* de distância de 500 metros a partir das fronteiras do bairro São Geraldo e utilizou a geometria gerada na operação espacial. O resultado da operação de *buffer* é um conjunto com dois pólígonos onde o primeiro é o *buffer* externo e o segundo é o *buffer* interno.

Exemplo 11.5 - Recortar um dado matricial a partir de uma máscara definida por um polígono.

```
//Conecta com o banco TerraTeste
//Inicializa os decoders do raster
TeInitRasterDecoders();

//Carrega o layer Brasilia
TeLayer* layerBrasilia =  new TeLayer("Brasilia_RGB");
db->loadLayer(layerBrasilia);
string rasterTable =  layerBrasilia->tableName(TeRASTER);

//Carrega o layer
TeLayer* layerPolBrasilia = new TeLayer("BrasiliaPol");
db->loadLayer(layerPolBrasilia);

//Carrega o polígono que será usado como máscara no recorte
TePolygonSet ps;
layerPolBrasilia->loadGeometrySet("0",ps);
TePolygon poly = ps[0];

//Recorta o dado matricial considerando a parte interna da máscara ou
// do polígono. Essa função gera um novo layer raster.
string newLayer = "BrasiliaRecortadoIn";
db->mask (rasterTable, poly, newLayer, TeBoxPixelIn);

//Recorta o dado matricial considerando a parte externa da máscara ou
// do polígono. Essa função gera um novo layer raster.
newLayer = "BrasiliaRecortadoOut";
db->mask (rasterTable, poly, newLayer, TeBoxPixelOut);

//operação zonal
TeStatisticsDimensionVect result;
db->zonal(rasterTable, poly, result);
```

O Exemplo 11.5 mostra as operações Zonal e Recorte de um dado matricial. Na operação zonal, um conjunto de estatísticas (por exemplo, soma, valor máximo, valor mínimo, contagem, média, variância, etc.) é calculado sobre a região do dado matricial contida dentro de um polígono que representa a zona de interesse. O resultado é fornecido em uma estrutura própria que armazena as estatísticas calculadas em cada dimensão do dado matricial.

Na operação de recorte o dado matricial é recortado pela zona de interesse, o resultado é um novo *layer* no banco de dados. A implementação dessa função foi feita usando o conceito de *iterador* sobre uma representação matricial, ou seja, um ponteiro que percorre todos os elementos do dado matricial. No caso da operação de recorte, foi usada uma especialização do conceito de *iterador* sobre dados matriciais, de forma que esse percorra somente os elementos que possuam uma certa relação com a região de interesse. As relações possíveis são que a área do elemento esteja toda dentro da região, que a área do elemento esteja toda fora da região, que o centro do elemento esteja dentro da região ou que o centro do elemento esteja fora da área de interesse.

Voltar para o índice.

12. Manipulação de dados espaço-temporais

Os atributos e as geometrias de um objeto ou elemento geográfico podem mudar ao longo do tempo, onde cada mudança gera uma nova instância no tempo desse objeto. A TerraLib tem uma proposta para tratar dados espaço-temporais, ou seja, para considerar a componente temporal dos dados geográficos, quando houver. Como exemplos de dados espaço-temporais podemos identificar:

- ***Eventos***: ocorrências independentes no espaço e no tempo, que possuem um rótulo temporal de quando o evento ocorreu. Esse é o caso de crimes que ocorrem em uma determinada localização de uma cidade em uma determinada data e hora. Cada objeto espacial associado a um associado a um evento terá uma única identificação no banco de dados;
- ***Objetos Mutáveis***: os quais possuem uma geometria fixa ao longo do tempo, porém seus atributos descritivos podem variar. Um exemplo desse tipo de objeto são os dados manipulados por modelos dinâmicos baseados em espaços celulares, ou estações fixas de coletas de dados;
- ***Objetos em Movimento*** os quais possuem limites e localizações, ou seja, geometrias que podem variar no tempo assim como seus atributos descritivos. Esse é o caso quando se acompanha a evolução de dos lotes de uma cidade, ou dos polígonos de desmatamento de uma região.

No modelo de banco de dados geográfico proposto pela TerraLib os dados são registrados semanticamente em função da sua natureza temporal com o objetivo de possam ser construídas funcionalidades específicas relativas a essa natureza temporal. Entre essas funcionalidades estão incluídos os algoritmos de agrupamento temporal, visualização de animações ou criação de temas com restrições temporais e ferramentas de modelagem dinâmica. Assim, quando uma tabela de atributos, associada a um *layer*, é inserida em um banco de dados, uma das informações registradas na tabela de metadados **`te_layer_table`** é qual o seu tipo, e dependendo desse tipo outras informações são necessárias. Além das tabelas estáticas e externas descritas anteriormente os outros tipos de tabelas são:

- ***Eventos:*** uma tabela de atributos de eventos. É preciso registrar qual de seus campos possui a identificação única do evento (e que faz a ligação com sua geometria) e quais de seus campos possuem a informação temporal da ocorrência do evento. A informação temporal normalmente é um intervalo (tempo inicial e tempo final), mas para rótulos temporais pontuais o tempo inicial é igual ao tempo final;
- ***Atributos Variáveis:*** uma tabela de atributos relacionada a dados do tipo objetos mutáveis. É preciso identificar quais o campo possui a identificação única do objeto no banco (e que faz a ligação com sua geometria), qual campo identifica de maneira única cada versão, ou instância de atributos associada ao objeto, e quais campos possuem a informação temporal que determina o intervalo de validade dessa instância.

O modelo de armazenamento dos objetos em movimento ainda está sendo desenvolvido na TerraLib, uma vez que esse é o caso mais complexo. Além de se considerar a variação no tempo associado aos atributos dos objetos é preciso considerar a variação associada às suas geometrias. Cada geometria tem que armazenar um intervalo (ou instante) de validade e é necessário propor um mecanismo de associação, em um determinado instante ou intervalo, entre a geometria e os atributos do objeto.

Exemplo 12.1 - Utilizar o TerraView para importar um dado temporal de ocorrências de crimes de um arquivo e observar as entradas geradas na tabela **`te_layer_table`**.

12.1. Estruturas da TerraLib para representar dados espaço-temporais

As classes da TerraLib para representar dados espaço-temporais são: `TeSTInstance`, `TeSTElement`, `TeSTElementSet`. `TeSTInstance` representa uma instância, ou seja, uma versão no tempo de um elemento ou objeto geográfico. Uma instância contém o identificador do elemento ou objeto ao qual pertence, um intervalo de tempo de validade (classe `TeTimeInterval`), suas geometrias (classe `TeMultiGeometry`) e seus atributos (classe `TePropertyVector`). Todas as instâncias de um mesmo elemento ou objeto geográfico são agrupadas pela classe `TeSTElement`.

A classe `TeSTElementSet` é um container em memória de instâncias (`TeSTInstance`) de um layer ou tema. Portanto, um `TeSTElementSet` pode estar associado a um layer ou a um tema. Se estiver associado a um layer, conterá todas as instâncias de todos os objetos ou elementos pertencentes a esse layer. Se estiver associado a um tema, conterá somente as instâncias dos elementos de um layer que satisfazem as restrições sobre atributos, espaciais e temporais definidas nesse tema. Esse container oferece mecanismos para percorrê-lo (iteradores) e recuperar atributos e geometrias de uma instância no tempo de um objeto.

Um `TeSTElementSet` pode ser preenchido com os objetos de um layer ou tema armazenados em um banco TerraLib através de funções disponíveis na biblioteca, como mostrado nos exemplos a seguir.

Exemplo 12.2 - Recuperar todos os objetos de um layer armazenado em um banco TerraLib. Neste exemplo, o `TeSTElementSet` é preenchido somente com alguns atributos do layer.

```
// Carrega o layer chamado EstadosBrasil
TeLayer* estados =  new TeLayer("EstadosBrasil");
db_->loadLayer(estados);

// Inicia as estratégias de consultas
TeInitQuerierStrategies();

// Cria um elementSet a partir do layer "estados"
TeSTElementSet steSet (estados);

// Define quais atributos serão armazenados no elementSet.
// O nome do atributo deve estar no formato:
// "nome_tabela.nome_atributo"
vector<string> attrs;
attrs.push_back("EstadosBrasil.NOME_UF");
```

```
attrs.push_back("EstadosBrasil.CAPITAL");

// Preenche o elementSet sem as geometries, apenas com
// os atributos NOME_UF e CAPITAL
bool loadGeometries = false;
bool loadAllAttributes = false;
if(!TeSTOSetBuildDB(&steSet, loadGeometries, loadAllAttributes, attrs))
{
        cout << "Erro! " << endl;
        return 1;
}

// Mostra o número de elementos do elementSet
cout << "Numero de elementos: " << steSet.numElements() << endl;

// Passa por todas as instancias do conjunto usando um iterador
TeSTElementSet::iterator it = steSet.begin();
while ( it != steSet.end())
{
        TeSTInstance st = (*it);
        // retorna os atributos
        TePropertyVector vectp = st.getPropertyVector();

        cout << "Id: " << st.objectId() << " ---------------- " << endl;
        for (unsigned int i=0; i<vectp.size(); i++)
        {
                cout << vectp[i].attr_.rep_.name_ << "  =  ";
                cout << vectp[i].value_ << endl;
        }
        cout << endl;
        ++it;
}

db->close();
return 0;
```

Exemplo 12.3 - Recuperar todos os objetos de um tema criado com duas restrições, uma espacial e outra por atributo. Neste exemplo, o tema é criado em memória, ou seja, não é persistido no banco TerraLib. Esse tema conterá todas as ocorrências de crime do tipo “ameaça” (restrição por atributo) e que ocorreram no bairro chamado Santo Antônio (restrição espacial).

```
// Carrega o layer chamado BairrosPoA
TeLayer* bairrosPA =  new TeLayer("BairrosPoA");
db_->loadLayer(bairrosPA))

// Carrega a geometria do distrito Santo Antonio (id = 48)
TePolygonSet ps;
bairrosPA->loadGeometrySet("48", ps);

// Carrega um layer chamado OcorrenciaPoA
TeLayer* OcorrenciaPoA =  new TeLayer("OcorrenciasPoA");
db->loadLayer(OcorrenciaPoA);

// Cria um tema na memória a partir do layer Ocorrencias com restricoes
TeTheme* Ocorrencias = new TeTheme("Ocorrencias", OcorrenciaPoA);
TeAttrTableVector attrTables;
OcorrenciaPoA->getAttrTables(attrTables);
Ocorrencias->setAttTables(attrTables);

// Restricao especial: dentro do distrito Santo Antonio
Ocorrencias->setSpatialRest (&ps, TePOINTS, TeWITHIN);
// Restrição de atributo: tipo = "ameaça"
Ocorrencias->attributeRest(" EVENTO = 'Ameaça' ");
```

```
// Inicia as estratégias de consultas
TeInitQuerierStrategies();

// Cria um elementSet a partir do tema
TeSTElementSet  steSet(Ocorrencias);

// Constrói o elementSet com geometrias e todos os atributos
bool loadGeometries = true;
bool loadAllAttributes = true;
vector<string>  attrs;
if(!TeSTOSetBuildDB(&steSet, loadGeometries, loadAllAttributes, attrs))
{
        cout << "Erro! " << endl;
        return 1;
}

// Mostra quanto elementos o elementSet possui
cout << "Numero de elementos: " <<  steSet.numElements() << endl;

// Passa por todas as instancias do conjunto usando um iterador
TeSTElementSet::iterator it = steSet.begin();
while ( it != steSet.end())
{
        TeSTInstance st = (*it);
        // Obtém o valor do atributo
        string event;
        st.getPropertyValue("EVENTO", event);
        cout << " Identificador do Objeto :  " << st.objectId() << endl;
        cout << " Evento    :  " << event  << endl ;

        // Obtém a geometria
        if(st.hasPoints())
        {
                TePointSet pset;
                st.getGeometry (pset);
                cout<< "     id do ponto: "<< pset[0].objectId ()     << endl;
                for(unsigned int j=0; j<pset.size (); ++j)
                {
                        string point =  Te2String(pset[j].location().x(), 7)
                +";"+
                        Te2String(pset[j].location().y(), 7);
                        cout<< "          ponto: " << " ("+ point + ") " <<
                endl    << endl;
                }
        }
        ++it;
}
db->close();
return 0;
```

12.2. Mecanismo para recuperar dados espaço-temporais: TeQuerier

A classe `TeQuerier` implementa um mecanismo flexível para recuperar e percorrer as instâncias (`TeSTInstance`) de um determinado layer ou tema armazenado em um banco TerraLib. Essa classe contém uma série de parâmetros que definem quais instâncias de quais objetos e quais atributos devem ser recuperados. Uma vez definidos esses parâmetros, o `TeQuerier` é responsável por acessar um banco de dados TerraLib, aplicar as restrições definidas pelos parâmetros e retornar as instâncias que satisfazem essas restrições, uma a uma.

A classe `TeSTElementSet`, mostrada na Seção 12.1, é um container que armazena em memória todas as instâncias dos elementos de um tema ou layer. As funções utilizadas para preencher um `TeSTElementSet` utilizam, internamente, a classe `TeQuerier`. Um usuário deve utilizar o `TeQuerier` quando for necessário armazenar as instâncias retornadas em uma estrutura de dados própria ou quando tiver interesse apenas em percorrer as instâncias sem armazená-las em memória.

Os parâmetros do mecanismo `TeQuerier` define, além de quais instâncias serão retornadas, o que será preenchido em cada instância. Uma instância pode conter: nenhum, alguns ou todos os atributos descritivos; todas ou nenhuma geometria; e o tempo válido dessa instância (se o tema ou layer for temporal). Os exemplos 12.4 e 12.5 mostram como recuperar instâncias de um layer utilizando `TeQuerier`.

Exemplo 12.4 - Recuperar todas as instâncias de um layer, com todos os seus atributos e sem geometrias.

```
// Conecata-se a um banco....
// Carrega o layer
TeLayer* ocorrencias = new TeLayer("OcorrenciasPoA");
db_->loadLayer(ocorrencias);

bool loadAllAttributes = true;
bool loadGeometries = false;

// Inicia as estratégias de consultas
TeInitQuerierStrategies();

// Seta os parametros do querier - carrega todos os atributos
//  e nenhuma geometria do layer "ocorrencias"
TeQuerierParams querierParams(loadGeometries, loadAllAttributes);
querierParams.setParams(ocorrencias);

TeQuerier  querier(querierParams);

// Carrega as instancia do layer baseadas nos parametros do querier
if(!querier.loadInstances())
return 1;

// Retorna a lista dos atributos carregados
TeAttributeList attrList = querier.getAttrList();

cout << " Atributos Carregados  ------------------  " << endl;
// Mostra o nome dos atributos
for(unsigned int i=0; i<attrList.size(); ++i)
         cout << attrList[i].rep_.name_ << endl;
cout << endl;

// Passa por todas as instancias
TeSTInstance sti;
while(querier.fetchInstance(sti))
{
         cout << " Objeto: " << sti.objectId() << " --- " << endl << endl;
      // Mostra cada atributo (o seu nome e valor)
         TePropertyVector vec = sti.getPropertyVector();
         for(unsigned int i=0; i<vec.size(); ++i)
         {
                  string  attrName = vec[i].attr_.rep_.name_;
                  string  attrValue = vec[i].value_;
                  cout << attrName  << " : " << attrValue << endl;
         }
}
db_->close ();
return;
```

Exemplo 12.5 - Recuperar todas as instâncias de um layer, com apenas dois atributos e com suas geometrias

```
// Conecta-se a um banco de dados...
// Carrega o layer OcorrenciasPoA
TeLayer* ocorrencias = new TeLayer("OcorrenciasPoA");
db_->loadLayer(ocorrencias);
```

```
bool loadGeometries = true;
vector<string> attributes;
attributes.push_back ("OcorrenciasPoA.ENDERECO");
attributes.push_back ("OcorrenciasPoA.BAIRRO");
// Inicia as estratégias de consultas
TeInitQuerierStrategies();
// Seta os parametros do querier - carrega apenas os atributos "ENDERECO"
// e "BAIRRO" e as geometrias do layer ocorrencias
TeQuerierParams querierParams(loadGeometries, attributes);
querierParams.setParams(ocorrencias);
TeQuerier  querier(querierParams);

// Carrega as instanciass do layer baseado nos parametros do querier
if(!querier.loadInstances())
        return 1;

// Passa por todas as instancias
TeSTInstance sti;
while(querier.fetchInstance(sti)) {
cout << " Objeto: " << sti.objectId() <<  endl << endl;

        // Mostra cada atributo, seu nome e valor
        TePropertyVector vec = sti.getPropertyVector();
        for(unsigned int i=0; i<vec.size(); ++i) {
                string  attrName = vec[i].attr_.rep_.name_;
                string  attrValue = vec[i].value_;
                cout << attrName  << " : " << attrValue << endl;
        }
        // Mostra as geometrias
        if(sti.hasPoints())       {
                TePointSet pointSet;
                sti.getGeometry(pointSet);

                for(unsigned int j=0; j<pointSet.size(); ++j)
                        cout << " Ponto : ( "  << pointSet[j].location().x(),7
                 <<     "," << pointSet[j].location().y(),7 << " )";
        }
        cout << endl << endl;
}
db_->close ();
return;
```

Os exemplos a seguir mostram como recuperar instâncias de um tema utilizando `TeQuerier`. Neste caso, o `TeQuerier` considera todas as restrições do tema (atributo, temporal e espacial).

Exemplo 12.6 - Recuperar todas as instâncias de um tema, com todos os atributos e geometrias. Nesse exemplo, os bairros têm duas representações geométricas, ponto e polígono. Portanto, as instâncias retornadas contêm as duas representações.

```
// Conecta-se ao banco de dados...
// Carega o tema "DistritosSaoPaulo"
TeTheme* bairros = new TeTheme("DistritosSaoPaulo");
db_->loadTheme(bairros);

// Inicia as estratégias de consultas
TeInitQuerierStrategies();

// Todos os atributos e geometrias
bool loadGeometries = true;
bool loadAllAttributes = true;

// Seta os parametros do querier - carrega todos os atributos e
// geometrias
```

```
TeQuerierParams querierParams(loadGeometries, loadAllAttributes);
querierParams.setParams(bairros);

TeQuerier  querier(querierParams);

// Carrega as instancias do layer baseado nos parametros do querier
if(!querier.loadInstances())
        return 1;
// Passa por todas as instancias
TeSTInstance sti;
while(querier.fetchInstance(sti)) {
        cout << " Objeto: " << sti.objectId() << " ---- " << endl << endl;

        // Mostra cada atributo(nome e valor)
        TePropertyVector vec = sti.getPropertyVector();
        for(unsigned int i=0; i<vec.size(); ++i) {
                string  attrName = vec[i].attr_.rep_.name_;
                string  attrValue = vec[i].value_;
                cout << attrName  << " : " << attrValue << endl;
        }
        // Obtém as geometrias
        if(sti.hasPolygons()){
                TePolygonSet polSet;
                sti.getGeometry(polSet);
                for(unsigned int i=0; i<polSet.size(); ++i)
                {
                        TeCoord2D centroid = TeFindCentroid(polSet[i]);
                        string p = "( "+ Te2String(centroid.x(), 7) +", "+
                  Te2String(centroid.y(), 7) +")";
                        cout << " Centroide do Poligono : " << p << endl;
                }
        }
        if(sti.hasPoints()) {
                TePointSet ponSet;
                sti.getGeometry(ponSet);
                for(unsigned int i=0; i<ponSet.size(); ++i)
                {
                        string p = "( "+ Te2String(ponSet[i].location().x())
                     +", "+  Te2String(ponSet[i].location().y()) +")";
                        cout << " Ponto : " << p << endl;
                }
        }
        cout << endl << endl;
}
delete(bairros);
return 1;
```

Os exemplos 12.7 e 12.8 mostram como utilizar a restrição espacial do `TeQuerier`. Quando uma restrição espacial é atribuída ao `TeQuerier`, apenas as instâncias que satisfazem essa restrição são recuperadas. No exemplo 12.7, a restrição espacial é definida por um retângulo ou `TeBox`. No exemplo 12.8, a restrição espacial é definida pelas geometrias de um outro tema.

Exemplo 12.7 - Dado um retângulo ou box (`TeBox`), recuperar todas as instâncias de um tema que estão dentro desse retângulo.

```
// Conecta-se a um banco de dados...
// Carrega o layer OcorrenciasBH
// Inicia as estratégias de consultas
TeInitQuerierStrategies();

// Todos os atributos e geometrias
bool loadGeometries = false;
bool loadAllAttributes = true;
```

```
// Seta os parametros do querier
TeQuerierParams querierParams(loadGeometries, loadAllAttributes);
querierParams.setParams(ocorrencias);

//Seta a restrição espacial
TeBox box(609033.62, 794723.35, 613455.34, 800417.99);
querierParams.setSpatialRest(box, TeWITHIN);

TeQuerier  querier(querierParams);

// Carrega as instancias do layer baseado nos parametros do querier
querier.loadInstances();

// Passa por todas as instancias
TeSTInstance sti;
while(querier.fetchInstance(sti))
{
        cout << " Object: " << sti.objectId() << " ----- " << endl;
// Mostra cada atributo (nome e valor)
        TePropertyVector vec = sti.getPropertyVector();
        for(unsigned int i=0; i<vec.size(); ++i)
        {
                string  attrName = vec[i].attr_.rep_.name_;
                string  attrValue = vec[i].value_;
                cout << attrName  << " : " << attrValue << endl;
        }
        cout << endl << endl;
}
db_->close ();
return 0;
```

Exemplo 12.8 - Para cada elemento de um tema X, recuperar todas as instâncias de um outro tema Y que estão dentro da geometria desse elemento. Nesse exemplo, teremos dois `TeQuerier`, um para o tema Bairros e outro para o tema Ocorrencias. E para cada bairro, serão recuperadas todas as ocorrências que estão dentro desse bairro.

```
// Conecta-se a um banco de dados...
// Carrega o tema "OcorrenciasBH"
// carrega o tema "BairrosBH"

// Inicia as estratégias de consultas
TeInitQuerierStrategies();

// ------------ querier para o tema "bairros"
bool loadGeometries = true;
vector<string> attrs;
attrs.push_back ("Bairrobh.nobai");

TeQuerierParams querParamsBair(loadGeometries, attrs);
querParamsBair.setParams(bairros);

TeQuerier  querBairros(querParamsBair);
if(!querBairros.loadInstances())
{
        cout << " Sem dados!!! " << endl;
        return 1;
}

// Carrega cada elemento e passa o querier como uma restrição espacial
TeSTInstance bairro;
while(querBairros.fetchInstance (bairro))
{
        string name;
```

```
        bairro.getPropertyValue(name, 0);
        cout << " Bairro: " << name << endl;

        TePolygonSet setP;
        if(!bairro.getGeometry (setP))  // Acessa o polygon set
                continue;

        // ------------ querier para o tema ocorrencias
        loadGeometries = false;
        vector<string>attributes;
        attributes.push_back ("ocorrenciasbh.Nrbo");
        TeQuerierParams querParamsOcr(loadGeometries, attributes);
        querParamsOcr.setParams(ocorrencias);

        // restrição espacial
        querParamsOcr.setSpatialRest(&setP);

        TeQuerier  querOcorrencias(querParamsOcr);
        if(!querOcorrencias.loadInstances())
        {
                cout << " Sem dado!!! " << endl;
                continue;
        }
        TeSTInstance sto;
        while(querOcorrencias.fetchInstance(sto))
        {
                // Mostra cada atributo (nome e valor)
                TePropertyVector vec = sto.getPropertyVector();
                for(unsigned int i=0; i<vec.size(); ++i)
                {
                        string  attrName = vec[i].attr_.rep_.name_;
                        string  attrValue = vec[i].value_;
                                cout << attrName  << " : " << attrValue << endl;
                }
                // Mostra a data
                cout << endl << " Data : " << sto.getInitialDateTime();
            cout << endl;
        }
}
db_->close ();
return 0;
```

O `TeQuerier` pode agrupar instâncias por objeto ou elemento, por restrição espacial ou por chronon (quando o tema for temporal). Para executar esse agrupamento, o `TeQuerier` deve receber, como parâmetros, as funções de agregação que serão utilizadas nos atributos descritivos. O Exemplo 12.9 agrupa as instâncias por objeto, o Exemplo 12.10 agrupa todas as instâncias que satisfazem uma restrição espacial e o Exemplo 12.11 agrupa todas as instâncias por chronon e por objeto.

Exemplo 12.9 - Agrupar as instâncias por objeto. Esse exemplo agrupa todas as coletas (instâncias) por armadilha (objeto). Ou seja, para cada armadilha o `TeQuerier` soma os número de ovos coletados nessa armadilha.

```
// Conecta-se a um banco de dados...
// Carrega o tema "coletas"

// Inicia as estratégias de consultas
TeInitQuerierStrategies();

// Qual função de agregação será utilizada em cada atributo
bool loadGeometries = false;
TeGroupingAttr attributes;

pair<TeAttributeRep,TeStatisticType>
        attr1(TeAttributeRep("Coletas.NRO_OVOS_PAL1"), TeSUM);
```

```
attributes.insert(attr1);

pair<TeAttributeRep, TeStatisticType>
        attr2(TeAttributeRep("Coletas.NRO_OVOS_PAL2"), TeSUM);
attributes.insert(attr2);

pair<TeAttributeRep, TeStatisticType>
        attr3(TeAttributeRep("Coletas.NRO_OVOS_PAL3"), TeSUM);
attributes.insert(attr3);

pair<TeAttributeRep, TeStatisticType>
        attr4(TeAttributeRep("Coletas.NRO_OVOS"), TeSUM);
attributes.insert(attr4);

// Seta os parametros do querier - carrega apenas os atributos do
// agrupamento
TeQuerierParams querierParams(loadGeometries, attributes);
querierParams.setParams(coletas);

TeQuerier  querier(querierParams);

// Carrega as instancias do layer baseado nos parametros do querier
if(!querier.loadInstances())
{
        db_->close ();
        return;
}

// Passa por todas as instancias
TeSTInstance sti;
while(querier.fetchInstance(sti))
{
        cout << " Object: " << sti.objectId()<< endl;

        // Mostra cada atributo (nome e valor)
        TePropertyVector vec = sti.getPropertyVector();
        for(unsigned int i=0; i<vec.size(); ++i)
        {
                string  attrName = vec[i].attr_.rep_.name_;
                string  attrValue = vec[i].value_;
                cout << attrName  << " : " << attrValue << endl;
        }
        cout << endl << endl;
}
db_->close ();
return;
```

Exemplo 12.10 - Agrupar todas as instâncias que satisfazem uma restrição espacial. Esse exemplo agrupa todas as coletas por bairro. Ou seja, a restrição espacial será definida por cada bairro do tema Bairros e o `TeQuerier` soma os números de ovos coletados em cada bairro.

```
// Conecta-se a um banco de dados...
// Carrega os temas
TeTheme* coletas = new TeTheme("Coletas");
db_->loadTheme(coletas);
TeTheme* bairros = new TeTheme("ibge_bairros");
db_->loadTheme(bairros);

// Inicia as estratégias de consulta
TeInitQuerierStrategies();

// estatisticas das coletas
```

```
TeGroupingAttr attributes;

pair<TeAttributeRep, TeStatisticType>
        attr1(TeAttributeRep("Coletas.NRO_OVOS_PAL1"), TeSUM);
attributes.insert(attr1);

pair<TeAttributeRep, TeStatisticType>
 attr2(TeAttributeRep("Coletas.NRO_OVOS_PAL2"), TeSUM);
attributes.insert(attr2);

pair<TeAttributeRep, TeStatisticType>
        attr3(TeAttributeRep("Coletas.NRO_OVOS_PAL3"), TeSUM);
attributes.insert(attr3);

pair<TeAttributeRep, TeStatisticType>
        attr4(TeAttributeRep("Coletas.NRO_OVOS"), TeSUM);
attributes.insert(attr4);

// ------------ querier para o tema "bairros"
bool loadGeometries = true;
vector<string> attrs;
attrs.push_back ("BAIRRO2000_REC.NOME");

TeQuerierParams querParamsBair(loadGeometries, attrs);
querParamsBair.setParams(bairros);

TeQuerier  querBairros(querParamsBair);
if(!querBairros.loadInstances())
{
        cout << " Sem dados!!! " << endl;
        return;
}

// Carrega cada elemento, passando como restrição espacial
TeSTInstance bairro;
while(querBairros.fetchInstance (bairro))
{
        string name;
        bairro.getPropertyValue(name, 0);
        cout << " Bairro: " << name << endl;

        TePolygonSet setP;
        if(!bairro.getGeometry (setP))  // Acessa o polygon set
                continue;

        // ------------ querier para o tema coletas
        loadGeometries = false;

        TeQuerierParams querParamsCol(loadGeometries, attributes);
        querParamsCol.setParams(coletas);

        // restrição espacial
        querParamsCol.setSpatialRest(&setP);
        TeQuerier  querColetas(querParamsCol);

        if(!querColetas.loadInstances())
        {
                cout << " Sem dados!!! " << endl;
                continue;
        }
        TeSTInstance coleta;
```

```
        while(querColetas.fetchInstance(coleta))
        {
                // Mostra cada atributo (nome e valor)
                TePropertyVector vec = coleta.getPropertyVector();
                for(unsigned int i=0; i<vec.size(); ++i)
                {
                        string  attrName = vec[i].attr_.rep_.name_;
                        string  attrValue = vec[i].value_;
                        cout << attrName  << " : " << attrValue << endl;
                }
        }
}
db_->close ();
return;
```

Exemplo 12.11 - Agrupar instâncias por chronon e por objeto. Esse exemplo agrupa todas as coletas por mês e por armadilha.

```
// Conecta-se a um banco de dados...
// Carrega o tema "coletas"
// Inicia as estratégias de consulta>
TeInitQuerierStrategies();

// Atributo que define o agrupamento
bool loadGeometries = false;
TeGroupingAttr attributes;

pair<TeAttributeRep, TeStatisticType>
attr1(TeAttributeRep("Coletas.NRO_OVOS_PAL1"), TeSUM);
attributes.insert(attr1);

pair<TeAttributeRep, TeStatisticType>
attr2(TeAttributeRep("Coletas.NRO_OVOS_PAL2"), TeSUM);
attributes.insert(attr2);

pair<TeAttributeRep, TeStatisticType>
attr3(TeAttributeRep("Coletas.NRO_OVOS_PAL3"), TeSUM);
attributes.insert(attr3);

pair<TeAttributeRep, TeStatisticType>
attr4(TeAttributeRep("Coletas.NRO_OVOS"), TeSUM);
attributes.insert(attr4);

// Seta os parametros do querier - carrega apenas os atributos do
// agrupamento
TeQuerierParams querierParams(loadGeometries, attributes);
querierParams.setParams(coletas, TeMONTH);

TeQuerier  querier(querierParams);
// número de frames gerados pelo chronon TeMONTH
int numTimeFrames = querier.getNumTimeFrames();

// Carregar as instancias do tema para cada frame ou mês
for(int frame=0; frame < numTimeFrames; ++frame)
{
TeTSEntry ts;
querier.getTSEntry(ts, frame);

cout << " Time frame: " << Te2String(frame) <<  endl << endl;
string initialDate = ts.time_.getInitialDateTime("DDsMMsYYYY");
string finalDate = ts.time_.getFinalDateTime("DDsMMsYYYY");
```

```
cout << " Intervalo de Tempo: " << initialDate << " to " << finalDate;
if(!querier.loadInstances(frame))
         continue;

// Passa por todos os elementos
TeSTInstance sti;
while(querier.fetchInstance(sti))
{
        cout << " Objeto: " << sti.objectId() << endl;
        // Mostra cada atributo (nome e valor)
        TePropertyVector vec = sti.getPropertyVector();
        for(unsigned int i=0; i<vec.size(); ++i)
        {
                string  attrName = vec[i].attr_.rep_.name_;
                string  attrValue = vec[i].value_;
                cout << attrName  << " : " << attrValue << endl;
        }
}
cout << endl << endl;
}
getchar();
db_->close ();
return;
```

Voltar para o índice.

13. Análise e Estatísticas Espaciais

13.1. Matriz de proximidade

Matriz de proximidade é um conceito importante para análise e estatísticas espaciais. A TerraLib oferece recursos para gerar matriz de proximidade seguindo diferentes critérios, como por exemplo adjacência, distância mínima e vizinho mais próximo, e para ponderar e fatiar a matriz segundo algum atributo. Uma matriz de proximidade é gerada a partir de um `TeSTElementSet`.
Nos exemplos a seguir, será mostrada a construção de uma matriz de proximidade a partir de um `TeSTElementSet` segundo diferentes critérios.

Exemplo 13.1 - Criar uma matriz de proximidade segundo o critério de distância mínima e percorrer os vizinhos de cada objeto. No critério de distância mínima dois objetos são considerados vizinhos se a distância entre eles for menor do que uma distância definida. Neste exemplo a distância é de 12000 metros.

```
// Conecta-se a um banco de dados...
// Carrega o DistritosSP
TeLayer* DistritosSP = new TeLayer("DistritosSP");
db->loadLayer(DistritosSP);

// Inicia as estratégias de consulta
TeInitQuerierStrategies();

// Cria um STElementSet a partir do layer DistritosSP
TeSTElementSet steSet(DistritosSP);

// Constrói o STElementSet com as geometrias
vector<string> attrs;
if(!TeSTOSetBuildDB(&steSet, true, false, attrs))
{
        cout << "Erro! " << endl;
        return 1;
}

// Mostra quatos elementos existem no elementSet
cout << "Numero de elementos: " <<  steSet.numElements() << endl;

// Cria uma estratégia de matriz de proximidade
TeProxMatrixLocalDistanceStrategy<TeSTElementSet> sc_dist(&steSet,TePOLYGONS, 12000.00);
```

```
TeGeneralizedProxMatrix<TeSTElementSet> proxMat(&sc_dist);
proxMat.setCurrentConstructionStrategy(&sc_dist);

// Constrói a matriz de proximidadde
if(!proxMat.constructMatrix())
{
        cout << "Erro na cnstrução da matriz de proximidade! " << endl;
        db->close ();
        return 1;
}

// Mostra os vizinhos de cada objeto da matriz
TeSTElementSet::iterator it = steSet.begin();
while ( it != steSet.end()){
        cout<< " Os vizinhos do elemento  "<< (*it).objectId() << "
                sao: " << endl;

        // Obtem os vizinhos de um elemento
        TeNeighboursMap neighbors = proxMat.getMapNeighbours((*it).objectId());
        TeNeighboursMap::iterator itN = neighbors.begin();
        while (itN != neighbors.end())     {
                cout<< "         "<<(*itN).first << endl;
                ++itN;
        }
        cout << endl;
        ++it;
}
db->close();
return 0;
```

13.2. Estatísticas Espaciais

Exemplo 13.2 - Calcular algumas estatísticas espaciais usando uma matriz de proximidade gerada através do critério de adjacência e gravar as estatísticas resultantes em um banco TerraLib. No critério de adjacência dois objetos são considerados vizinhos se suas fronteiras se tocam.

```
// Conecta-se a um banco de dados...
// Carrega o layer DistritosSP
TeLayer* DistritosSP =  new TeLayer("DistritosSP");
db->loadLayer(DistritosSP);

// Inicia as estratégia de consulta
TeInitQuerierStrategies();

// Cria um STElementSet para o laer DistritosSP
TeSTElementSet steSet(DistritosSP);

// Preenche o STElementSet apenas com o atributo "Pop91" para
// calcular estatisticas e não carrega as geometrias
vector<string> attrs;

attrs.push_back ("Distritos.Pop91");
if(!TeSTOSetBuildDB(&steSet, false, false, attrs)){
        cout << "Erro! " << endl;
        return 1;
}

// Mostra quanto elementos existem no elementSet
cout << "Numero de elementos: " <<  steSet.numElements() << endl;

// Constrói a matriz de proximidade
TeProxMatrixLocalAdjacencyStrategy        sc_adj (&steSet,  TePOLYGONS);
```

```
TeGeneralizedProxMatrix<TeSTElementSet>        proxMat(&sc_adj);

if(!proxMat.constructMatrix()){
        cout << "Erro na construcao da matriz de de proximidade! " << endl;
        return 1;
}

// Calcula a média global
double mean = TeFirstMoment (steSet.begin(), steSet.end(), 0);
cout << "Média Global " << mean << endl << endl;

// Calcula os desvios (Z) para cada objeto and armazena no STElementSet
if(!TeDeviation (steSet.begin(), steSet.end(), mean))
{
        cout << "Erro ao calcular Z! " << endl;
        return 1;
}

// índice do atributo desvios (Z) no STElementSet
int indexZ = 1;

// Calcula a media local (WZ) para cada objeto e armazenar
// no STElementSet
if(!TeLocalMean (&steSet, &proxMat, indexZ))
{
        cout << "Erro ao calculae Media Local! " << endl;
        return 1;
}

// índice do atributo média local (WZ) no STElementSet
int indexWZ = 2;

// Mostra os indices Z e WZ gerados para cada objeto
TeSTElementSet::iterator it = steSet.begin();
while ( it != steSet.end()){
        TeSTInstance obj = (*it);
        //Gets attribute value
        string valZ, valWZ;
        obj.getPropertyValue(valZ, indexZ);
        obj.getPropertyValue(valWZ, indexWZ);
        cout << " Identificador do Objeto :  " << obj.objectId() << endl;
        cout << " Z               :  " << valZ   << endl;
        cout << " WZ      :  " << valWZ  << endl << endl;
        ++it;
}

// Insere os atributos gerados (Z e WZ) na tabela DistritosSP
if(!TeUpdateDBFromSet (&steSet, "DistritosSP"))
{
        cout << "Erro ao atualizar o banco de dados! " << endl;
        return 1;
}
db->close();
return 0;
```

Voltar para o índice.

14. Referências Biliográficas

BAILEY, T.; GATRELL, A. **Interactive Spatial Data Analysis.** London: Longman Scientific and Technical,1995.

CÂMARA, G.; SOUZA, R. C. M.; PEDROSA, B.; VINHAS, L.; MONTEIRO, A. M. V.; PAIVA, J. A.; CARVALHO, M. T.; GATTASS, M. TerraLib: Technology in Support of GIS Innovation. In: II **Simpósio Brasileiro em Geoinformática**, GeoInfo2000, 2000, São Paulo.

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

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**.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLÍSSIDES, J. **Design Patterns - Elemens of Reusable Object-Oriented Software**. Reading, MA: Addison-Wesley,1995. 395 p.

INPE-DPI. O aplicativo TerraView. Disponível em: <http://www.dpi.inpe.br/terraview>. Acesso em: maio 2005.

MySQL AB. **MySQL reference manual**. Disponível em: <http://www.mysql.org>.Acesso em: abril 2005.

Open GIS Consortium. Web Map Service Version 1.13. Disponível em: <http://www.opengeospatial.org>. Acesso em: maio 2005.

QUEIROZ, G. R. **Algoritmos Geométricos para Banco de Dados geográficos: da teoria à prática na TerraLib.**São José dos Campos, SP: INPE - Instituto Nacional de Pesquisas Espaciais, 2003. Dissertação de Mestrado, Computação Aplicada, 2003.

QUEIROZ, G. R.; FERREIRA, K. R., 2005. SGBD com extensões espaciais. In: CASANOVA, M. A.; CÂMARA, G.; DAVIS, C.; VINHAS, L.; QUEIROZ, G. R., eds., **Bancos de Dados Geográficos**, v. 1: São José dos Campos, SP, Editora MundoGeo, p. 506.

SHEKHAR, S.; CHAWLA, S. **Spatial Databases: A Tour.** Prentice Hall, 2002.

SNYDER, J. P. **Map Projections – A Working Manual**. Washington, DC, USA: United States Government Printing Office, 1987.

SOUZA, R. C.; CÂMARA, G.; AGUIAR, A. P. D. **Modeling Spatial Relations by Generalized Proximity Matrices**.In: V Simpósio Brasileiro de Geoinformática, 2003, Campos do Jordão. Anais do GeoInfo 2003. São José dos Campos : SBC, 2003.

STROUSTROUP, B. **C++ Programming Language** - 3rd Edition. Reading, MA: Addison-Wesley, 1997. 911 p.

VINHAS, L.; FERREIRA, K. R.; , 2005. Descrição da TerraLib. In: CASANOVA, M. A.; CÂMARA, G.; DAVIS, C.; VINHAS, L.; QUEIROZ, G. R., eds., **Bancos de Dados Geográficos**, v. 1: São José dos Campos, SP, Editora MundoGeo, p. 506.

---

[1] O termo representação espacial, no contexto da TerraLib, é muitas vezes usado de maneira análoga ao termo geometria.

[2] A declaração desta classe encontra-se no arquivo TeDatabase.h, na pasta kernel.

[3] A declaração desta classe encontra-se no arquivo TeLayer.h, nas pasta kernel.

[4] A declaração desta classe encontra-se no arquivo TeProjection.h, na pasta kernel.

[5] A declaração desta classe encontra-se no arquivo TeGeometry.h, na pasta kernel.

[6] A declaração desta classe encontra-se no arquivo TeCoord2D.h, na pasta kernel.

[7] Esta classe encontra-se declarada no arquivo TeTable.h, na pasta kernel.

[8] O ponteriro de retorno deve ser copiado, pois o próximo comando getData pode invalidar o ponteiro retornado anteriormente.

[9] A declaração desta classe encontra-se no arquivo TeRaster.h, na pasta kernel.

Voltar para o índice.