MINISTÉRIO DA CIÊNCIA E TECNOLOGIA
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

INPE-12838-PRE/8128

CAPÍTULO 13

TRATAMENTO DE DADOS MATRICIAIS NA TERRALIB

Lúbia Vinhas
Ricardo Cartaxo Modesto de Souza

Bancos de dados geográficos

INPE
São José dos Campos
2005

13 Tratamento de dados matriciais na TerraLib

Lúbia Vinhas
Ricardo Cartaxo Modesto de Souza

13.1 Introdução

Esse capítulo descreve a solução adotada na biblioteca TerraLib (Câmara et al., 2000) para o tratamento de dados matriciais incluindo a decodificação de diferentes formatos e o seu armazenamento e recuperação em bancos de dados objeto-relacionais.

Os bancos de dados geográficos ainda possuem um desafio: o tratamento eficiente de dados matriciais, mais especificamente de dados de imagens de satélites. As imagens de sensoriamento remoto são uma das mais procuradas fontes de informação espacial para pesquisadores interessados em fenômenos geográficos de larga escala. A variedade de resoluções espectrais e espaciais das imagens de sensoriamento remoto é grande, variando desde as imagens pancromáticas do satélite IKONOS, com resoluções espaciais de um metro, até as imagens polarimétricas de radar que em farão parte da nova geração de satélites RADARSAT e JERS. Os recentes avanços da tecnologia de sensoriamento remoto, com o desenvolvimento de novas gerações de sensores, têm trazido consideráveis avanços nas aplicações de monitoramento ambiental e de planejamento urbano (Moreira, 2004).

A construção de bancos de dados geográficos capazes de manipular imagens de satélites tem sido amplamente estudado na literatura e a abordagem principal tem sido o desenvolvimento de servidores de dados especializados, como é o caso do PARADISE (Patel et al., 1997) e do RASDAMAN (Reiner et al., 2002). A vantagem principal dessa abordagem é a capacidade de melhora de desempenho no caso de grandes bases de dados de imagens. A principal desvantagem dessa abordagem é a

necessidade de um servidor específico o que sobrecarrega o SIG – Sistemas de Informação Geográfica.

Além das imagens de satélite os SIG também devem ser capazes de tratar outros tipos de geo-campos, cuja representação computacional mais adequada é a matricial. Esses dados incluem grades numéricas que representam fenômenos qualitativos contínuos, como superfícies de fenômenos físicos (por exemplo, altimetria), ou sociais (por exemplo, exclusão social). A Figura 13.1 mostra dois exemplos de dados geográficos diferentes com representação matricial: uma foto aérea e uma superfície de altimetria.

Figura 13.1 – Exemplos de dados matriciais.

Os SIG's precisam trabalhar de maneira integrada com dados matriciais de diferentes naturezas e na mesma base de dados geográfica, que pode conter também geo-objetos com representação vetorial.

A implementação de um sistema de manipulação de dados matriciais, como grades e imagens de sensoriamento remoto, em um SGDB-OR, juntamente com funções para tratar outros tipos de dado espaciais é de grande importância para os SIG, e traz vantagens sobre sistemas especializados para construir bases de dados de imagens.

A infra-estrutura objeto-relacional pode ser usada para construir um sistema de indexação espacial, o que permite a consulta e recuperação de dados matriciais integrada à interface genérica de manipulação de dados matriciais. Através da indexação adequada, algoritmos de compressão e sistemas de *cache* um desempenho satisfatório pode ser conseguido, para

os SIG's manipulando bases de dados compostas de representações matriciais e vetoriais.

13.2 Características dos dados matriciais

Dados matriciais em TerraLib ser entendidos como qualquer dado representado em uma estrutura de matriz retangular com M linhas × N colunas. Exemplos desse tipo de dado são grades regulares com valores de uma determinada grandeza (como uma grade de altimetria) ou imagens de sensoriamento remoto.

Dados matriciais podem possuir outras dimensões além das duas representadas no plano cartesiano das linhas e colunas. Ou seja, a cada elemento da matriz podem estar associados um ou mais valores. Por exemplo, as imagens multi-dimensionais de sensoriamento remoto possuem em cada elemento, ou *pixel*, da imagem o valor relativo a uma banda espectral do instrumento imageador que obteve a imagem. Por esse ser o caso mais típico de dados matriciais multi-dimensionais, em TerraLib dizemos que um dado matricial possui M linhas × M colunas × B bandas, como representado na Figura 13.2.

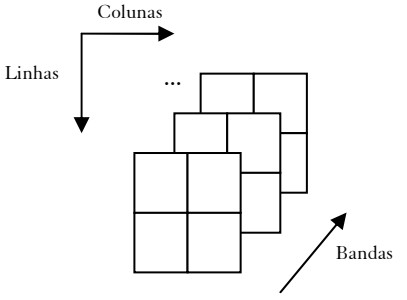

Figura 13.2 – Representação matricial multi-dimensional.

Cada posição da representação matricial é chamada de elemento (equivalente a um *pixel*, no caso de imagens) e costuma ser ter sua posição identificada por um par (linha × coluna), onde a posição (0,0) equivale à linha mais superior e a coluna mais à esquerda da matriz. Cada elemento,

em uma dimensão, pode conter um valor dentro de um determinado intervalo de valores possíveis. Esse intervalo dependente do tamanho computacional (ou digital) reservado para seu armazenamento e todos os elementos possuem o mesmo tamanho. Alguns tamanhos possíveis para cada elemento são:

- *Unsigned Char*: ocupa 8 bits por elemento. Pode armazenar valores entre 0 a 255;
- *Short*: ocupa 16 bits por elemento. Pode armazenar valores entre – 32.68 to 32.67
- *Float*: ocupa 32 bits por elemento. Pode armazenar valores entre 3.4E-38 a 3.4E+38 (7 dígitos);
- *Double*: ocupa 64 bits por elemento. Pode armazenar valores entre 1.7E- 308 a 1.7E+308 (15 dígitos).

No caso das imagens de sensoriamento remoto o tamanho do elemento relaciona-se com a resolução espectral da imagem e no caso de outros tipos de dados esse valor relaciona-se com a precisão do dado que está sendo representado.

Computacionalmente, a representação matricial não é esparsa, ou seja, todo elemento da matriz possui um valor. Porém muitas vezes é necessário ser possível a criação de uma representação esparsa, ou seja, uma representação onde em alguns pontos da matriz não existe um valor válido. Isso pode ser feito através da escolha de um valor como indicativo da ausência de dado naquele elemento da matriz. Esse valor deve pertencer ao intervalo de valores válidos para os elementos da matriz e é chamado de valor *dummy* ou "*nodata*". A Figura 13.3 (a) mostra um dado matricial, com 3 linhas × 3 colunas × 1 banda. Se o valor 0 for considerado como valor *dummy*, podemos dizer que nas posições (0,1) e (2,2) não existem valores válidos.

Outra característica dos dados matriciais e que os valores em cada elemento podem representar índices para uma tabela de combinações de valores, como no caso típico de imagens do tipo paleta. O valor de cada elemento é uma chave para uma tabela que contém uma tripla de valores R, G e B como mostrado na Figura 13.3(b).

1	**0**	**2**
1	**2**	**3**
2	**1**	**0**

(a) Valores Dummy

Chave	R	G	B
1	255	0	0
2	0	255	0
3	0	0	255

(b) Tabela de cores

Figura 13.3 – Característica da representação matricial.

Finalmente, as representações matriciais são caracterizadas pela resolução de cada elemento da grade, ou seja, a extensão nas direções horizontal e vertical de cada elemento da matriz quando mapeado para a área da superfície da Terra que representa. No caso das imagens de satélite esses parâmetros equivalem à resolução espacial da imagem.

A seção seguinte descreve os mecanismo de tratamento de dados matriciais de TerraLib.

13.3 Tratamento de dados matriciais em TerraLib

Imagens e/ou quaisquer outros tipos de dados com representação matricial são manipulados em TerraLib por três classes básicas:

- Uma classe genérica chamada `TeRaster`;
- Uma estrutura de dados que representa todos os parâmetros que caracterizam um dado matricial `TeRasterParams`;
- Uma classe genérica de decodificação de formatos e acesso a dispositivos de armazenamento de dados matriciais chamada `TeDecoder`.

13.3.1 A classe TeRaster

A classe `TeRaster` é uma interface genérica de acesso aos elementos de um dado matricial. Seus dois métodos básicos são: `setElement` e `getElement`, os quais inserem e recuperam, respectivamente, valores de cada elemento da representação matricial como um valor real de dupla precisão. O Exemplo 13.1 mostra como criar uma representação matricial em

memória através da classe `TeRaster`. A representação matricial tem 10 linhas, 10 colunas, 2 bandas e cada elemento (em cada banda) pode conter um valor do tipo *unsigned char*.

```
TeRaster rasterMem(10,20,2,TeUNSIGNEDCHAR);
```

Exemplo 13.1– Criação de uma representação matricial em memória.

Antes que o dado esteja disponível para ser manipulado, todas as inicializações necessárias, como alocações de memória, devem ser executadas e estão concentradas no médodo `TeRaster::init`. Após a chamada desse método cada instância da classe `TeRaster` possui um valor de estado que indica quais as operações (leitura, escrita) podem ser executadas sobre ele. O Exemplo 13.2 mostra a chamada e teste de estado na criação de um dado matricial através da classe `TeRaster`.

```
// 1) cria um objeto raster
TeRaster rasterMem(10,20,2,TeUNSIGNEDCHAR);
// 2) faz inicializações
rasterMem.init();
// 3) verifica resultado
if (rasterMem.status() == TeNOTREADY)
     return false;
```

Exemplo 13.2 – Inicialização de um dado matricial.

13.3.2 A classe TeRasterParams

Um dado matricial é caracterizado por uma série de parâmetros que podem ser divididos nas seguintes categorias:

- *Dimensão*: número de linhas, número de colunas e número de bandas;
- *Geográficos*: projeção cartográfica, resolução horizontal, resolução vertical e menor retângulo envolvente de todo o dado. Como cada

elemento de um dado matricial tem área, podemos identificar dois tipos de retângulo envolvente: o que passa pelo centro dos elementos das bordas (chamamos de box) e o que passa pelos extremos dos elementos das bordas da matriz (chamamos de bounding box) como representado na Figura 13.4;

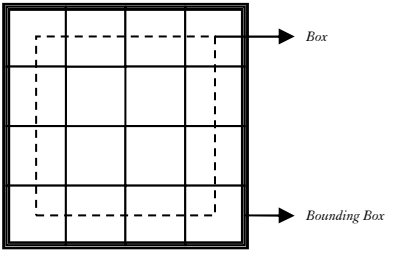

Figura 13.4 – Diferença entre box e bounding box.

- *Características dos elementos*: tamanho digital de cada valor de um elemento e interpretação do valor (paleta ou não);
- *Armazenamento*: nome de arquivo ou tabela de banco de dados onde está armazenado, tipo acesso permitido;
- *Forma de decodificação*: mecanismo usado para ler ou escrever um valor em um elemento.

A classe `TeRasterParams` contém um super conjunto de todos os possíveis parâmetros que descrevem um dado matricial, por isso cada objeto da classe `TeRaster` possui como um dos seus membros um objeto da classe `TeRasterParams`, que pode ser acessado através do método `TeRaster::params` como mostro o Exemplo 13.3.

```
void describe(TeRasterParams& par)
{
  cout << "Nome: " << par.fileName_ << endl;
  cout << "Modo de acesso: " << par.mode_ << endl;
  cout << "Nro bandas: " << par.nBands() << endl;
  cout << "Nro linhas: " << par.nlines_ << endl;
  cout << "Nro colunas: " << par.ncols_ << endl;
  cout << "Resolucao X: " << par.resx_ << endl;
```

```
  cout << "Resolucao Y: " << par.resy_ << endl;
  cout << "Identificacao do decodificador: " << 
par.decoderIdentifier_<< endl;
  cout << "Projecao: " << par.projection()->name() << endl;
  cout << "Retangulo envolvente: ";
  TeBox bb = par.boundingBox();
  cout << bb.x1_ << " " << bb.y1_ << " " << bb.x2_ << " " << 
bb.y2_ << endl;
}
TeRaster rasterMem(10,10,2,TeUNSIGNEDCHAR);
rasterMem.init();
if (rasterMem.status() == TeNOTREAD)
   return 0;
describe(rasterMem.params());
```

Exemplo 13.3 – Parâmetros do dado matricial.

Os diferentes formatos de armazenamento de dados matriciais podem trazer todos ou alguns dos parâmetros citados acima. Vejamos como exemplo:

- *Tiff*: dados matriciais nesses formatos trazem informações sobre o número de Linhas, o número de Colunas, o número de Bandas, o tipo dos elementos e a interpretação do elemento (se é do tipo paleta e sua tabela de cores);
- *GeoTiff*: é uma extensão do formato Tiff específica para o armazenamento de dados geográficos (grades numéricas ou imagens) e traz além das informações do Tiff, a projeção e retângulo envolvente da imagem, o tipo e a resolução espacial dos elementos, e a sua projeção cartográfica. O GeoTiff é o formato de intercâmbio de dados matriciais do consórcio OpenGIS (OPENGIS, 2005);
- *JPEG*: formato para armazenamento de imagens, traz informações sobre o número de Linhas, o número de Colunas e o número de

Bandas. O formato JPEG permite armazenar apenas dados com 1 ou 3 bandas e o tipo dos elementos é sempre unsigned char;

- *RAW*: servem para armazenar grades ou imagens como arquivos binários sem nenhuma formatação e não trazem nenhuma informação sobre a representação matricial.

A estrutura `TeRasterParams` armazena também o nível de acesso ao dado, que pode assumir 3 opções:

1. 'r': somente leitura. Caso o dado não exista o método `TeRaster::init` falhará;
2. 'c': criação de um novo. Caso o dado já exista, será apagado e recriado. Será garantido acesso para leitura e escrita;
3. 'w': leitura e escrita. Pressupõe que o dado já exista. Caso contrário o método `TeRaster::init` falhará.

13.3.3 A classe `TeDecoder`

A idéia principal da classe `TeRaster` é poder representar dados matriciais que podem estar em memória, em arquivos em diferentes formatos (como TIFF ou JPEG), ou ainda dentro do um banco de dados geográfico. Com o objetivo de desacoplar a classe `TeRaster` das diferentes alternativas de armazenamento, as requisições de acesso aos elementos da representação matricial são tratadas pela classe `TeDecoder`. Esse mecanismo é um exemplo de uso do padrão de projeto "*Strategy*" (Gamma, Helm et al. 1995). A classe `TeDecoder` é abstrata e TerraLib já fornece um conjunto de decodificadores de imagens e grades. Novos decodificadores podem ser criados através da especialização da classe `TeDecoder` implementando os métodos `init`, `clear`, `setElement` e `getElement`.

A Tabela 13.1 mostra as características das classes de decodificadores já implementadas em TerraLib.

Tabela 13.1 – Decodificadores de Dados Matriciais implementados em TerraLib

Decodificadores	*Identificador*	*Extensão*	*Acesso*	*Descrição*
`TeDecoderMemory`	"MEM"	---	Criação Leitura Escrita	Dados matriciais como uma matriz de valores em memória
`TeDecoderJPEG`	"JPEG"	".jpg" ".jpeg"	Criação Leitura Escrita	Dados matriciais armazenados em arquivos no formato JPEG. Carregam todo o dado para memória
`TeDecoderTIFF`	"TIFF"	".tif" ".tiff"	Criação Leitura	Dados matriciais armazenados em arquivos no formato GeoTIFF ou TIFF
`TeDecoderSPR`	"SPR"	".spr" ".txt"	Criação Leitura Escrita	Dados matriciais armazenados em arquivos no formato ASCII Spring. Carregam todo o dado para memória
`TeDecoderMemoryMap`	"MEMMAP"	".raw"	Criação Leitura Escrita	Dados matriciais armazenados em arquivos binários raw, limitados a aproximadamente 2Gb de tamanho
`TeDecoderFile`	"FILE"	".bin"	Criação Leitura Escrita	Dados matriciais armazenados em arquivos binários *raw*
`TeDecoderDatabase`	"DB"	---	Criação Leitura Escrita	Dados matriciais armazenados em um banco de dados TerraLib

A associação de um decodificador específico a um dado matricial pode ser feita explicitamente, mas também foi implementado um mecanismo mais flexível usando o padrão de projeto "*Factory*" (Gamma, Helm et al. 1995). Existe uma fábrica de decodificadores que constrói a instância de decodificador correta de acordo com um ou mais parâmetros do dado matricial. Um exemplo típico de uso desse mecanismo automático de seleção é associação de extensões de arquivos a decodificadores específicos. Quando possível, a associação de extensões de arquivos a decodificadores é o que permite que um objeto da classe `TeRaster` possa ser instanciado a partir somente de um nome de arquivo. Essa associação pode ser modificada, conforme interesse do usuário da biblioteca, editando-se a função `TeInitRasterDecoders`. O Exemplo 13.4 mostra o acesso a dois dados matricias, nesse caso arquivos de imagens em dois formatos diferentes, TIFF e JPEG, de forma transparente.

```
// Mostra os valores em um dado matricial
void showRaster(TeRaster& rst)
{
  int i,j,b;
  double val;              // lê e mostra valores
  for (b=0; b<rst.params().nBands();++b)  {
      cout << "\nValores na banda: " << b << \n;
      for (i=0; i<rst.params().nlines_; ++i) {
          for (j=0; j<rst.params().ncols_; ++j) {
            rst.getElement(j,i,val,b);
            cout << val << " ";  }
       cout << endl << endl;
      }
  }
int main()
{
  TeInitRasterDecoders();        // inicializa decodificadores
  TeRaster rasterTIF("d:/Dados/TIFF/brasM.tif");
  if (!rasterTIF.init())
```

```
        return 0;
    TeRaster rasterJPEG("d:/Dados/JPEG/brasilia.jpg");
    if (!rasterJPEG.init())
        return 0;
    cout << "Dados no arquivo TIFF: \n";
    showRaster(rasterTIF);
    cout << "\n\nDados no arquivo JPEG: \n";
    showRaster(rasterJPEG);
}
```

Exemplo 13.4 – Exemplo de uso da classe `TeRaster`.

A Figura 13.5 mostra o relacionamento entre as três classes principais da interface de manipulação de dados matriciais de TerraLib. A classe `TeRaster` é uma geometria como as geometrias vetoriais descritas no Capítulo 12. Cada objeto da classe `TeRaster` possui uma instância de um objeto da classe `TeDecoder`, e ambas possuem um objeto da classe `TeRasterParams`, ou seja, o decodificador tem uma versão dos parâmetros do dado matricial que decodifica.

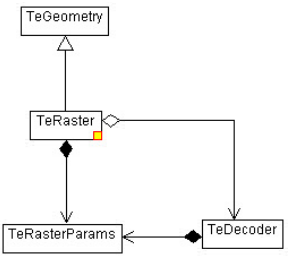

Figura 13.5 – Relacionamento entre as classes da interface de manipulação de dados matriciais.

Esse mecanismo facilita a escrita de algoritmos de manipulação de dados matriciais porque é flexível o bastante para fornecer um acesso uniforme a diferentes formatos de dados matriciais como mostra o Exemplo 13.5.

```
//! Copia os valores em um dado raster para outro
void copiaRaster(TeRaster& rstIn, TeRaster& rstOut)
{
  int i,j,b;
  double val;                              // lê valores
  for (b=0; b<rstIn.nBands();++b) {
      for (i=0; i<rstIn.nLines();++i) {
       for (j=0; j<rstIn.nCols();++j) {
          rstIn.getElement(j,i,val,b);
          rstOut.setElement(j,i,val,b);
       }
      }
  }
}

int main()
{
  TeInitRasterDecoders(); // inicializa decodificadores

  TeRaster rasterTIF("d:/Dados/TIFF/brasM.tif");
  if (!rasterTIF.init())
       return 1;

  TeRasterParams par = rasterTIF.params();    // copia todos os
parâmetros
  par.fileName_ = "d:/Dados/JPEG/bras.jpg";
  par.mode_ = 'c';
  par.decoderIdentifier_ = "JPEG";

  TeRaster rasterJPEG(par);
  if (!rasterJPEG.init())
       return 1;
```

```
  copiaRaster(rasterTIF, rasterJPEG);
  return 0;
}
```

Exemplo 13.5 – Criação de um dado matricial a partir de outro.

Pode se observar no exemplo acima que foram aproveitados todos os parâmetros da imagem de origem com exceção de três: o nome do arquivo destino, o modo de criação e a identificação do decodificador. O construtor do objeto `TeRaster` destino recebe uma instância de `TeRasterParams` e não mais um nome de arquivo. Assim a nova imagem será criada no formato JPEG no arquivo especificado. A função de cópia continua usando apenas os métodos `TeRaster::setElement` e `TeRaster::getElement`.

Quando se deseja instanciar usar a classe `TeRaster` para acessar um dado cujo formato não inclui todas as informações necessárias, existe a possibilidade de se instanciar esse objeto a partir de uma estrutura de parâmetros e previamente preenchida. O Exemplo 13.6 mostra como acessar um dado matricial guardado como uma matriz binária em um arquivo *raw*. Como esse formato não possui nenhum tipo de descritor seus parâmetros devem ser explicitamente informados.

```
TeDatum sad69 = TeDatumFactory::make("SAD69");
TeUtm* proj = new TeUtm(sad69,-45.0*TeCDR);

TeRasterParams par;
par.fileName_ = "D:/Dados/RAW/grade.raw";
par.decoderIdentifier_ = "MEMMAP";
par.nBands(1);
par.mode_ = 'r';
par.setDummy(-9999.9);
par.setDataType(TeFLOAT);
par.topLeftResolutionSize(185350.0,824800.0,1000,500,10,10,true
);
```

```
par.projection(proj);

TeRaster grade(par);
if (!grade.init())
   return 1;
   //... usa dado
grade.clear();
```

Exemplo 13.6 – Criação de um objeto `TeRaster` a partir de parâmetros.

O método `TeRaster::clear` deve ser chamado ao final da utilização do dado matricial. Esse método libera memória ou quaisquer estruturas que foram alocadas na chamada do método `TeRaster::init`. De maneira geral a ordem de passos para se utilizar um objeto da classe `TeRaster` é a seguinte:

1. Definir parâmetros
2. Instanciar um objeto `TeRaster` a partir dos parâmetros
3. Chamar método `TeRaster::init`
4. Verificar se estado do objeto é o esperado
5. Utilizar objeto
6. Chamar método `TeRaster::clear`

Uma vez executado o método `TeRaster::init` não se deve alterar os parâmetros do dado matricial, pois as inicializações feitas utilizaram os valores então definidos. Qualquer modificação subseqüente pode invalidar a capacidade de decodificação do dado requerendo uma nova chamada ao método `TeRaster::init` como mostrado no Exemplo 13.7.

```
TeRasterParams par;
```

```
par.decoderIdentifier_ = "MEM";
par.nBands(1);
par.mode_ = 'c';
par.setDummy(255);
par.setDataType(TeUNSIGNEDCHAR);

TeRaster rMem(par);
if (!rMem.init())                // aloca espaço para 100
elementos
   return 0;

rMem.params().nlines_ = 200;     // altera número de elementos
rMem.params().ncols_ = 200;
rMem.setElement(199,199,0);       // ERRO!

rMem.init()                      // CERTO! Reinicializar raster
rMem.setElement(199,199,0);      // antes de acessar novos
elementos
```

Exemplo 13.7 – Reinicializando um objeto `TeRaster`.

13.3.4 Incluindo informações geográficas

Para que o dado matricial seja geo-referenciado, ou seja, para que se possa identificar para cada linha e coluna uma localização sobre a superfície terrestre é preciso que seu os parâmetros contenham as informações sobre uma projeção cartográfica, as coordenadas de seu retângulo envolvente e a resolução espacial de cada elemento em unidades da projeção. Esses dados devem ser informados nos parâmetros ou obtidos do próprio formato de armazenamento e não devem ser alterados depois da inicialização sob pena de invalidar o geo-referenciamento. O Exemplo 13.8 mostra como incluir as informações geográficas em um dado matricial.

```
TeDatum sad69 = TeDatumFactory::make("SAD69");
TeUtm* proj = new TeUtm(sad69,-45.0*TeCDR);
```

```
TeRasterParams par;
par.decoderIdentifier_ = "MEM";
par.nBands(1);
par.mode_ = 'r';
par.setDummy(255);
par.setDataType(TeUNSIGNEDCHAR);

//define o retângulo envolvente do dado e sua resolução
par.lowerLeftResolutionSize(309695,7591957,20,20,3000,2000,true
);
// define a projeção do dado
par.projection(proj);

TeRaster grade(par);
if (!par.init())
   return 1;

for (i=0; i< grade.nLines();++i) {          //     retorna     a
coordenada geográfica
   for (j=0; j< grade.nCols();++j) {      // de cada elemento
        char mess[100];
        TeCoord2D xy = grade.index2Coord(TeCoord2D(j,i));
        sprintf(mess,"[%d,%d]:(%.2f,%.2f) ",i,j,xy.x(),xy.y());
        cout << mess;
    }
    cout << endl;
}
```

Exemplo 13.8 – Associação de parâmetros geográficos.

Os parâmetros geográficos de um dado matricial que permitem sua navegação em coordenadas de uma determinada projeção cartográfica (*bounding box*, *box*, número de linhas, colunas, resoluções horizontais e verticais) são correlacionados e devem ser coerentes. A classe `TeRasterParams` possui os métodos informar parte deles e

automaticamente atualizar os outros (`lowerLeftResolutionSize`, `boxResolution`, `upperLeftResolutionSize`, `boundingBoxResolution`, etc.).

13.3.5 Remapeamento de dados raster

Um objeto da classe `TeRaster` propriamente inicializado pode ser parâmetro de funções que usam os seus métodos `getElement` e `setElement` como mostrado anteriormente. A classe `TeRasterRemap` é um exemplo de implementação de uma função que copia um dado matricial para outro, resolvendo diferenças em termos de geometria. Essas diferenças podem ser em relação ao número de linhas, colunas, resoluções diferentes ou retângulos envolventes diferentes, ou ainda, projeções diferentes. A Figura 13.6 mostra o fluxo de operações da classe `TeRasterRemap`.

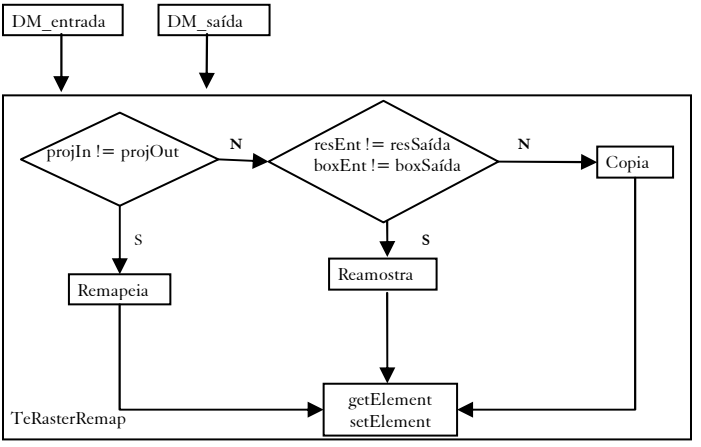

Figura 13.6 – A classe `TeRasterRemap`.

O Exemplo 13.9 mostra três casos de uso dessa função: para degradar a resolução de uma imagem, para recortar um pedaço de uma imagem e para remapear (trocar de projeção) uma imagem.

```
#include "TeRasterRemap.h"
```

```
int main()
{
  TeInitRasterDecoders();
  TeRaster rEntrada("d:/Dados/TIFF/brasM.tif");       // raster
em projeção UTM
  if (!rEntrada.init())
  {
     cout << "Erro obtendo dado de entrada!\n";
     return 1;
  }

   TeRasterParams parS = rEntrada.params();
   TeBox bb = parS.boundingBox();              // mantém box
   double resx = parS.resx_*2.0;               //    degrada    a
resolução espacial
   double resy = parS.resy_*2.0;

parS.boundingBoxResolution(bb.x1_,bb.y1_,bb.x2_,bb.y2_,resx,res
y);
   parS.fileName_ = "d:/Dados/JPEG/subBras.jpg";
   parS.decoderIdentifier_ = "JPEG";            // troca formato
de saída
   parS.mode_ = 'c';
  TeRaster rSaidal(parS);
   if (rSaidal.init())
   {
    TeRasterRemap  subamostra(&rEntrada,&rSaidal);     //  faz
reamostragem
    subamostra.apply();
   }
   parS = rEntrada.params();
   parS.fileName_ = "d:/Dados/JPEG/zoomBras.jpg";
   parS.decoderIdentifier_ = "JPEG";    // define box menor

parS.boundingBoxResolution(184000.0,8247000.0,194500.0,8248000.
0, parS.resx_,parS.resy_);
```

```
  parS.mode_ = 'c';

   TeRaster rSaida2(parS);
   if (rSaida2.init())
   {
    TeRasterRemap zoom(&rEntrada,&rSaida2);    // faz recorte
    zoom.apply();
  }

  parS = rEntrada.params();
  parS.fileName_ = "d:/Dados/JPEG/latlongBras.jpg";
  parS.decoderIdentifier_ = "JPEG";
  TeDatum sad69 = TeDatumFactory::make("SAD69");
  TeLatLong* latlong = new TeLatLong(sad69);
  parS.projection(latlong);
  parS.mode_ = 'c';

  TeRaster rSaida3(parS);
  if (rSaida3.init())
  {
      TeRasterRemap remap(&rEntrada,&rSaida3);  // remapeia
      remap.apply();
  }
}
```

Exemplo 13.9 – A classe `TeRasterRemap`.

A funcionalidade da classe `TeRasterRemap` pode ser usada em diversos contextos como a importação de dados para um banco e o apresentação desse dado em uma janela de visualização.

13.3.6 *Iteradores* sobre dados matriciais

Um grande número de algoritmos de processamento de dados geográficos não dependem de uma implementação particular de uma estrutura de dados, mas apenas de alguma propriedade semântica fundamental, como a capacidade de mover entre elementos de uma estrutura de agregação ou

comparação entre dois elementos da estrutura. Por exemplo, o algoritmo para calcular o histograma de um dado geográfico não necessita conhecer se o dado está organizado como um conjunto de pontos, um conjunto de polígonos ou um conjunto de *pixels*. Tudo que é necessário é a que a estrutura forneça a capacidade de percorrer uma lista de elementos e obter um valor pra cada um deles. Esse paradigma de desenvolvimento é chamado de programação genérica (Austern 1998). Programação genérica visa o projeto de módulos interoperáveis baseados na separação entre algoritmos e estruturas de dados.

A classe `TeRaster` fornece *iteradores*, ou seja, ponteiros generalizados que permitem o percorrimento de uma representação matricial. Em sua versão mais simples os *iteradores* de dados matriciais começam na primeira linha e na primeira coluna do dado e a cada operação de avançar o *iterador* move-se para a próxima coluna até o fim da linha quando então se move para a primeira coluna da linha seguinte. *Iteradores* são úteis, pois permitem que sejam escritos algoritmos em função dos *iteradores* e não de uma estrutura de dados em particular. O Exemplo 13.10 mostra como percorrer uma representação matricial através de *iteradores*.

```
TeRaster* img = new TeRaster("./dados/imagem.jpg");
TeRaster::iterator it = img.begin();
while (it != img.end())
{
  cout << (*it);
}
```

Exemplo 13.10 – Percorrimento de um dado matricial através de *iterador*.

Usando esse princípio, a classe `TeRaster` também fornece *iteradores* especializados para percorrimento dos elementos da representação matricial que estão delimitados por uma região de interesse (definida como um `TePolygon`) como mostra o Exemplo 13.11 .

```
TeRaster* img = new TeRaster("./dados/imagem.jpg");
TePolygon region;
```

```
//... cria a região de interesse
TeRaster::iteratorPoly itB = img.begin(region);
TeRaster::iteratorPoly itE = img.end(region);
while (it != itE)
{
  cout << (*it);
}
```

Exemplo 13.11 – Percorrimento de região de um dado matricial limitada a uma região.

As seções anteriores mostraram a interface de manipulação de dados matriciais oferecidas por TerraLib. A próxima seção trata das questões relativas ao armazenamento e recuperação de dados matriciais em um banco de dados TerraLib.

13.4 Dados matriciais em bancos de dados TerraLib

Um banco TerraLib é composto de um conjunto de planos de informação, ou *layers*, onde cada *layer* é formado por um conjunto específico de dado geográfico (como um mapa de solos, uma imagem ou um mapa cadastral). Para o armazenamento desses dados em um SGBDOR – Sistema Gerenciador de Banco de Dados Objeto-Relacional, cada *layer* está associado com um conjunto de tabelas que armazenam a componente descritiva e espacial do dado. Em um banco TerraLib vai existir uma tabela diferente para cada tipo de geometria associada a informação contida no *layer* (pontos, linhas, polígonos, células e representações matriciais). O acesso ao dado espacial armazenado no SGDB-OR é feito através de uma interface que encapsula as diferenças internas de cada SGDB-OR. Essa interface mapeia os tipos espaciais TerraLib em tipos característicos de cada SGDB-OR usando os mecanismos nativos de indexação e otimização se disponíveis. As duas classes abstratas que definem a interface de acesso aos dados espaciais são: `TeDatabase` e `TeDatabasePortal` que permitem: (a) estabelecimento de conexões com um servidor de banco de dados; (b) execução de commandos SQL; (c) definção de indíces; (d) definição de integridade referencial; (e) execução

de consultas espaciais. Para saber mais sobre a interface de acesso a um banco de dados TerraLib o leitor deve recorrer ao capítulo 12.

O consórcio OpenGIS propõe uma especificação de componentes capazes de tratar geo-campos, chamados de *Grid Coverages*, mas, diferentemente de geometrias vetoriais não propõe um modelo de armazenamento desses dados em um banco de dados (OPENGIS, 2005). Trabalhos anteriores (Patel et al., 1997) (Reiner et al., 2002) mostram que a estratégia mais apropriada para trabalhar com grandes bases de dados de imagens é uma combinação de divisão por blocos, ou *tiling*, e a criação de uma pirâmide de multi-resolução. Essa combinação tem sido usada também em alguns servidores de mapas via web (DPI/INPE, 2005) e (Barclay et al, 2000). O esquema de *tiling* é usado como indexação espacial, de forma que quando se deseja recuperar uma parte da imagem apenas os blocos relevantes para essa parte serão recuperados. A pirâmide de multi-resolução é útil na visualização de imagens grandes e evita acessos desnecessários, nos níveis da pirâmide onde a resolução é degradada, menos blocos são recuperados. Essa abordagem foi adotada em TerraLib de forma integrada a todo o mecanismo descrito nas seções anteriores de forma que um banco TerraLib possa guardar quaisquer dados com uma representação computacional matricial e não somente imagens.

A TerraLib trata uma representação matricial como mais uma tipo de geometria, sendo assim, um *layer* possui um conjunto de objetos cuja componente espacial pode ser uma representação matricial. Muitas vezes, um *layer* representa um geo-campo e nesse caso a geometria se refere ao *layer* e não a objetos discretos que agrega. A tabela de representações geométricas contém um registro para indicar a existência da representação matricial para o *layer*, esse registro, no campo **`geom_table`** informa o nome da tabela que contém as informações sobre a representação matricial de cada objeto do *layer*. Essa tabela de representações matriciais de um *layer* possui um conjunto de informações gerais sobre a representação: número de linhas e colunas, sua resolução espacial, a largura e altura (em número de elementos) dos blocos em que cada banda da representação espacial foi dividida e a indicação da possível técnica de compressão aplicada aos blocos. Finalmente, o campo **`spatial_data`**, aponta para a tabela de

geometrias matriciais onde os blocos que formam a representação estão armazenados. A Figura 13.7 mostra os relacionamentos entre as tabelas que compõe o modelo de armazenamento de dados matriciais de TerraLib.

Na tabela de geometrias matriciais, cada registro possui a identificação única do bloco, seu retângulo envolvente e, em um campo binário longo, o bloco comprimido ou não. Todos os blocos de uma tabela de geometrias matriciais possuem a mesma largura e a mesma altura. Esse modelo de armazenamento de dados matriciais pode ser implementado em qualquer SGDB-OR uma vez que não faz uso de nenhum recurso especial do banco.

Poucos SGDB's com extensão espacial propõem um modelo de armazenamento de dados matriciais, um deles é a extensão espacial do Oracle como pode ser visto em ORACLE (2005).

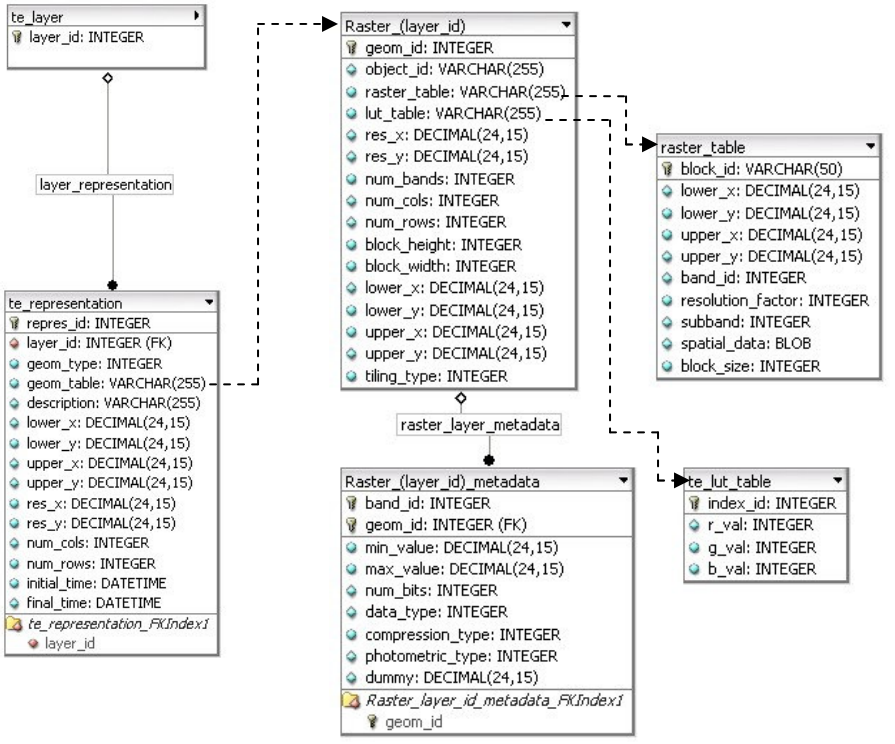

Figura 13.7 – Modelo de armazenamento de representações matriciais de TerraLib.

13.4.1 Modelo de divisão em blocos

O mecanismo de divisão do dado matricial em blocos visa a atender a dois requisitos específicos: (a) eficiência, conseguida pela capacidade de recuperação de parte do dado matricial; (b) flexibilidade, conseguida dada pela capacidade de acrescentar novos dados matriciais a uma representação já armazenada (por exemplo, ao se fazer o mosaico de duas cenas de imagens de satélite).

TerraLib disponibiliza duas formas de divisão de dados matriciais em blocos: não expansível e expansível. Essas duas formas diferentes de

divisão implicam em diferentes maneiras de se mapear um elemento da representação matricial como um todo para um elemento dentro de bloco armazenado. A seguir explicamos as duas maneiras.

a) Não expansível

Significa que a divisão dos blocos é feita a partir da linha 0 e coluna 0 da representação matricial. Cada bloco contém $L \times A$ elementos, onde L é a largura do bloco e A a sua altura. Cada bloco é identificado pela sua ordem dentro da divisão total da representação, como mostra a Figura 13.8.

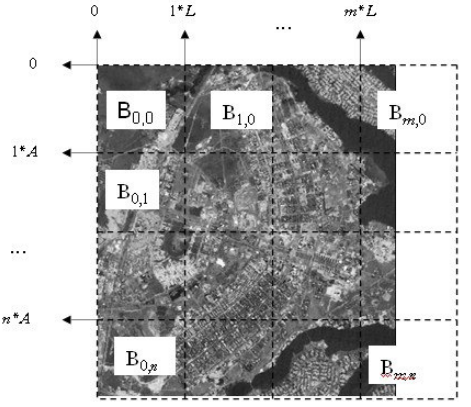

Figura 13.8 – Divisão por blocos de maneira não expansível.

Um elemento que está posição *(j, i)* da representação original está no bloco $B_{m,n}$ tal que $m = \text{INT}(i/L)$, $n = \text{INT}(j/A)$, onde INT(x) representa a parte inteira de x.

Observa-se na figura que a última linha de blocos e a última coluna de blocos ($B_{...,n}$ e $B_{m,..}$) podem não estar completos, ou seja, não existem valores na representação matricial para preencher todo o bloco. Isso acontece se o número de linha da representação não é múltiplo de A ou o número de colunas da representação não é múltiplo de L. Esses elementos do bloco são preenchidos com o valor *dummy*.

Ao se escolher a opção de divisão por blocos não expansível, não será possível acrescentar, ou *mosaicar*, posteriormente novos dados matriciais ao dado armazenado.

b) Expansível

Significa que a divisão dos blocos não é feita na referência das linhas e colunas, mas sim a partir posições arbitrárias calculadas de acordo com as coordenadas geográficas do dado matricial. Isso permite que novos dados matriciais possam ser acrescentados a uma representação matricial já armazenada, desde que seus parâmetros geográficos sejam coerentes. A identificação de cada bloco é dada por seu posicionamento dentro de uma divisão da área geográfica referente ao dado matricial em blocos de tamanho $L*ResX$ e $A*ResY$, onde $ResX$ e $ResY$ são as resoluções espaciais do dado matricial, iniciando na posição 0,0 do sistema de referência geográfica do dado. O esquema de divisão de blocos de maneira expansível é ilustrado na Figura 13.9. É importante notar que na divisão expansível os limites dos blocos estão em uma referência geográfica e não relacionada à linha e coluna de uma matriz. Isso é o que permite áreas de intersecção geográfica entre dois dados arquivos de imagens, por exemplo, sejam armazenados no mesmo bloco, como destacado na figura.

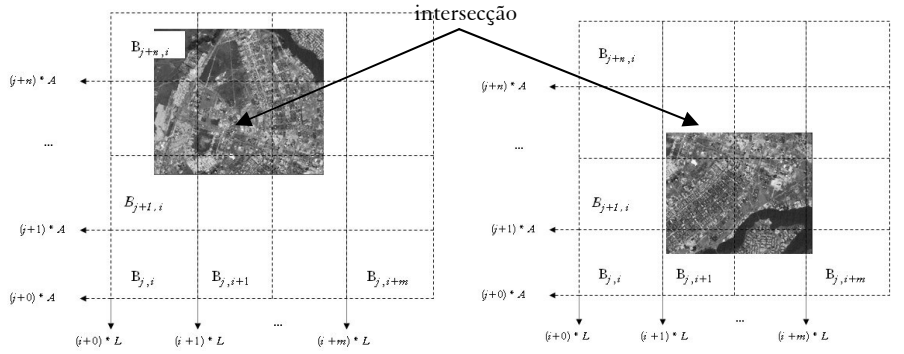

Figura 13.9 – Divisão por blocos de maneira expansível.

Um elemento que está posição (*j, i)* da representação original está no bloco $B_{m,n}$ tal que $m=\text{INT}(x/LG)$ e $n=\text{INT}(y/AG)$ onde (x,y) é a coordenada geográfica de (j,i), $LG=L*resX$ e $AG=A*resY$.

Assim como no caso da divisão não expansível alguns blocos ficam incompletos (observar a Figura 13.9) e são preenchidos com o valor *dummy*.

Através do retângulo envolvente de cada bloco, é possível saber seu retângulo envolvente em termos de linhas e colunas da representação matricial total e assim encontrar a posição do elemento procurado dentro do bloco.

O Exemplo 13.12 mostra o código escrito para executar importar um arquivo de imagens para um banco TerraLib criando um novo *layer*. Depois um segundo arquivo é importado indicando que deve ser armazenado na mesma representação e, portanto executando o mosaico. A Figura 13.10 mostra o resultado visual da operação.

```
TeRaster bras1(“./dados/Brasilia1.tif”);
TeRaster bras1(“./dados/Brasilia1.tif”);
TeLayer *msc = new TeLayer("Brasilia", db, bras1.projection());
TeImportRaster(msc,bras1,128,128,TeJPEG,"",255,TeExpansible));
TeImportRaster(msc,bras2,128,128,TeJPEG,"",255,TeExpansible))
```

Exemplo 13.12 – Mosaico de dois arquivos de imagens para uma mesma representação matricial.

Os parâmetros da rotina de importação indicam qual o tamanho dos blocos, qual o algoritmo de compressão, qual o valor a ser usado como *dummy* e qual o método de divisão por blocos.

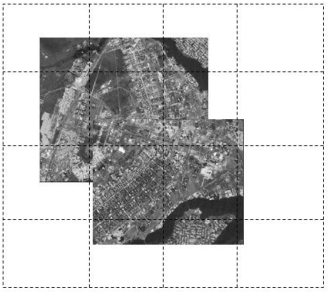

Figura 13.10 – Resultado do mosaico de dois arquivos de imagens.

Os mecanismos de divisão por blocos e de mosaico são válidos tanto para imagens quanto para grades numéricas.

13.4.2 A estrutura de multi-resolução

As operações de visualização de dados matriciais grandes são limitadas pelos dispositivos de saída. Por exemplo, considere uma imagem com 20000 × 20000 *pixels* sendo mostrada em um *display* de 1000 × 1000 pontos. O usuário controlando a aplicação de visualização pode começar obtendo uma visão geral da imagem e depois prosseguir ampliando áreas menores de seu interesse. Em cada situação, ou seja, em cada ampliação, não deveria ser necessário a recuperação simultânea de toda a imagem do banco de dados. Na situação ideal, a recuperação da imagem deveria ser

da ordem de magnitude da capacidade do dispositivo de saída. Esse requisito pode ser atendido se a imagem armazenada no banco de dados esteja organizada em uma estrutura de multi-resolução. Combinada com o esquema de divisão por blocos, a pirâmide de multi-resolução armazena, em seu nível mais baixo, os blocos com a resolução total da imagem. Nos níveis mais altos são armazenados blocos com resolução degradada. Dessa forma a aplicação pode controlar, de acordo com a capacidade do dispositivo de saída, em qual nível recuperar os blocos de interesse.

Com essa motivação, TerraLib implementa um esquema pirâmide de multi-resolução. Tradicionalmente são criadas resoluções com fatores multiplicativos em potência de dois, de forma que um nível superior contenha ¼ do número de blocos necessários para se armazenar o dado no nível inferior (ou seja, de melhor resolução) como mostrado na Figura 13.11.

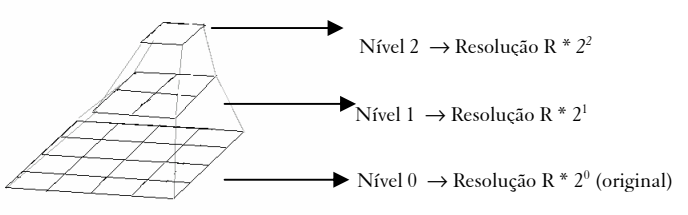

Figura 13.11 – Exemplo de pirâmide com resoluções degradas de fator 2.

TerraLib permite que a aplicação informe qual o fator de degradação, a cada nível, deseja aplicar para construir a pirâmide como mostra o Exemplo 13.13.

```
// importa resolução original res X= 30, res Y=30
 TeImportRaster(nativade,nat,512,512,TeJPEG,"obj1");
 TeBuildLowerResolution(nativade nat, 2, ,"obj1");
 // resX = resY = 60
 TeBuildLowerResolution(nativade nat, 4, ,"obj1");
// resX = resY = 120
```

```
 TeBuildLowerResolution(nativade nat, 8, ,"obj1");
// resX = resY = 240
```

Exemplo 13.13 – Importando resoluções extras.

As resoluções extras podem ser criadas nos dois modelos de divisão por blocos: expansível e não expansível. A resolução do bloco também é parte de sua identificação única, por isso os blocos com resoluções degradadas podem ser armazenados na mesma tabela que os blocos na resolução original.

13.4.3 Processamento de consultas e recuperação de dados

Assim como no caso das geometrias vetoriais (ver capítulo 12), existem vários níveis de acesso aos dados matriciais armazenados no banco de dados TerraLib. Através da classe `TeDatabase` e `TaDatabasePortal`, a aplicação pode submeter consultas SQL diretamente sobre a tabela de geometria matricial e pode consumir os registros retornados.

```
TeDatabase portal = db->getPortal();
// seleciona todos os blocos na resolução original
string sql = "SELECT * FROM RasterLayert_01_Raster" WHERE
resolution_factor = 1";
portal->query(sql);
while (portal->fetchRow())
{
  char* blob;
   long size;
   portal-> getRasterBlock(blob, size);
   // usa o blob retornado
}
```

Exemplo 13.14 – Consulta SQL sobre uma tabela de representação matricial.

O problema com essa abordagem é que a aplicação fica responsável por descompactar e decodificar o *blob*. Por isso, em um nível conceitual mais

alto, a classe `TeRaster` possui um esquema de seleção dos blocos com um dado fator de resolução e que interceptam um certo retângulo envolvente. O resultado da seleção é consumido como se fosse um portal, porém o retorno é uma instância de decodificador já inicializado, que pode ser usado para construir uma representação matricial independente em memória como mostrado no Exemplo 13.15.

```
TeBox bb(183557, 8246310, 194987, 8258940);
int resFac = 2;
TeRasterParams parBlock;
// seleciona os blocos com um certo fator de resolução e que
// interceptam um determinado retângulo envolvente
if (raster->selectBlocks(bb, resFac,parBlock))
{
   // Processa cada bloco como um dado matricial independente
em memoria
  TeRaster* block = new TeRaster;
  TeDecoderMemory* decMem = new TeDecoderMemory(parBlock);
  decMem->init();
  do
  {
       flag = raster->fetchRasterBlock(decMem);
       block->setDecoder(decMem);
  } while (flag);
   raster->clearBlockSelection();
}
```

Exemplo 13.15 – Esquema de seleção de blocos de uma determinada representação matricial.

Aumentando um pouco mais o nível de abstração, através da classe `TeLayer` é possível se obter a sua representação matricial como uma instância da classe `TeRaster` que pode então acessar seus elementos individualmente através dos métodos `setElement` e `getElement`, ou através de *iteradores* como mostrado nos exemplos das seções anteriores. O Exemplo 13.16 mostra como pode ser feito esse acesso.

```
TeLayer img("Brasilia");
db->loadLayer(img);
TeRaster* bsbImg = img->raster();
//… acessa elementos individuais da imagem
double val;
bsbImg->getElement(0,0,val,0);
```

Exemplo 13.16 – Acessando uma representação matricial de um *layer*.

13.4.4 A classe `TeDecoderDatabase`

No Exemplo 13.16, internamente é criado um decodificador para um dado matricial armazenado em uma banco de dados TerraLib (representado pela classe `TeDecoderDatabase`), escondendo todos os detalhes de tabelas e compressões associados ao dado armazenado. O decodificador de dados matriciais em um banco de dados TerraLib levou em conta algumas questões de eficiência para permitir o acesso a elementos individuais da representação. Existe um mecanismo de *cache* de blocos em memória a fim de reduzir o número de acessos ao banco de dados. O identificador único de cada bloco é usado como chave no armazenamento no banco de dados e também no sistema de *cache*. Cada vez que a aplicação requisita o valor de um elemento com as coordenadas (i,j) na banda b, em uma resolução r, se o bloco que contém o elemento não está na memória, é recuperado e colocado no sistema de *cache*. No próximo acesso a um elemento, o sistema inicialmente verifica se o bloco que o contém está no sistema de *cache*. Se estiver no o valor do elemento é recuperado diretamente da memória sem a necessidade de se acessar o banco novamente. O número máximo de blocos mantido no sistema de *cache* é controlado pela aplicação. Quando é necessário trazer um bloco para o *cache* e não existe mais espaço, o bloco menos recentemente usado é liberado do *cache*, caso tenha sido modificado seu conteúdo é atualizado no banco de dados. O sistema de cache é implementado na classe `TeDecoderVirtualMemory` da qual a classe `TeDecoderDatabase` é derivada. O sistema de troca de blocos no *cache* atual é simples, onde os blocos mais usados são mantidos por mais tempo. No entanto, esse é um ponto onde podem ser desenvolvidas outras estratégias mais adaptadas a cada caso.

A classe `TeDecoderDatabase` permite que dados matriciais armazenados em um banco de dados TerraLib possam ser usados da mesma maneira que dados matriciais armazenados em memória ou em arquivos em formatos proprietários. Juntamente com a classe `TeRasterRemap` aplicações como a importação de um dado matricial para o banco de dados ou a visualização de um dado matricial em um dispositivo de saída é facilmente implementada. A Figura 13.12 mostra o esquema usado para importar um dado matricial para um banco TerraLib.

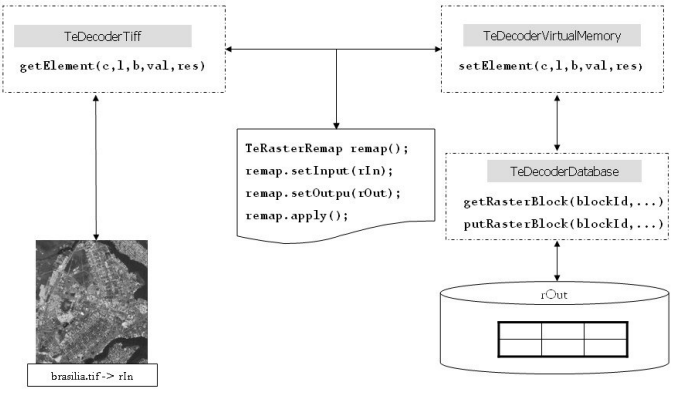

Figura 13.12 – Esquema da importação de um dado matricial para o banco.

13.5 Um exemplo de aplicação

Na TerraLib, o esquema de manipulação de dados matriciais foi funciona em todos os sistemas gerenciadores de banco de dados que implementam a interface `TeDatabase` como ORACLE, MySQL, PostgreSQL e Access. Um exemplo de aplicação é o SIG TerraView, que possui as funções para importar para um banco de dados TerraLib, visualizar e exportar dados matriciais em diversos formatos. A Figura 13.13 mostra um a tela do TerraView mostrando um *layer* com representação matricial, no caso uma imagem, sobreposta por uma dado vetorial, o limite dos estados da região norte.

A representação matricial do *layer* mostrado foi construída como um mosaico de duas imagens vindas de arquivos em formato GeoTiff. A

primeira imagem possui 7020 × 7984 *pixels* e 3 bandas, segunda imagem 7414 × 8239 *pixels* e também 3 bandas. Os dois arquivos GeoTiff juntos ocupavam 335Mb.

As imagens foram divididas em blocos de 512 × 512 *pixels* e foram construídos cinco níveis de resolução, cada nível degradando a resolução do nível anterior de um fator de dois. Os blocos foram armazenados em um banco TerraLib armazenado no SGDB ACCESS, e comprimidos pelo formato JPEG com um nível de qualidade de 75%. A tabela de geometrias matriciais ficou com um total de 1500 blocos ocupando um espaço de armazenamento de 472 Mb. Para visualizar a imagem total foram decodificados cinco blocos com um fator de resolução de 64 vezes a resolução original.

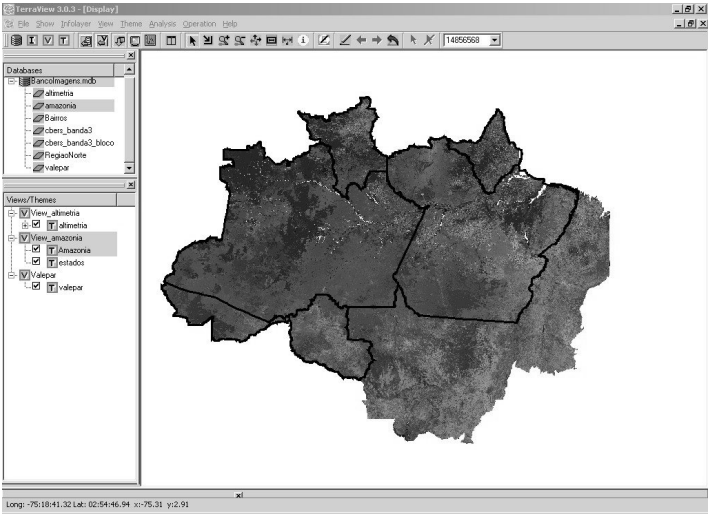

Figura 13.13 – Tela principal do TerraView.

A Figura 13.14 mostra o resultado da ampliação de aproximadamente um quarto da imagem e a aplicação precisou descomprimir 16 blocos com um fator de resolução de oito vezes a resolução original.

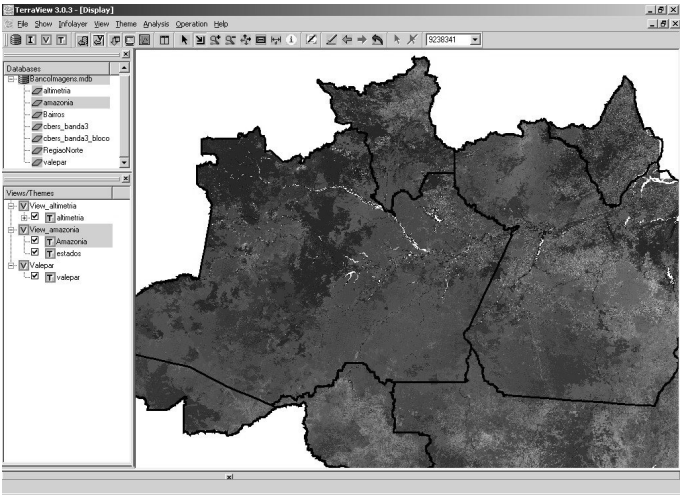

Figura 13.14 – Primeira ampliação.

Finalmente, a Figura 13.15 mostra uma ampliação detalhada, para a qual a aplicação descomprimiu seis blocos na resolução original.

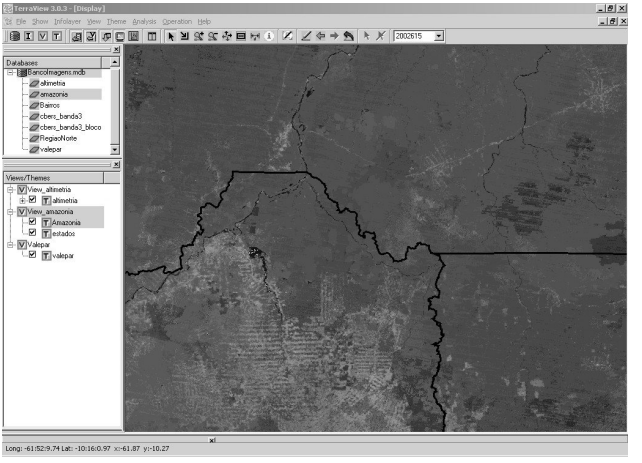

Figura 13.15 – Segunda ampliação.

A Figura 13.16 mostra a visualização de uma grade de altimetria, armazenada segundo o mesmo esquema do armazenamento da imagem. A grade é mostrada como se fosse uma imagem em tons de cinza onde o valor mais baixo da grade está mapeado para o preto e o valor mais alto está mapeado para o branco.

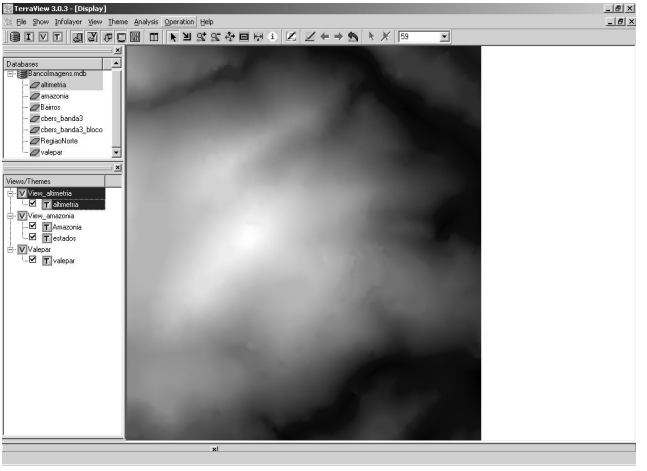

Figura 13.16 – Visualização da grade de altimetria.

Referências

AUSTERN, M. (1998). **Generic Programming and the STL : Using and Extending the C++ Standard Template Library**. Reading, MA, Addison-Wesley.

BARCLAY, T.; GRAY, J.; SLUTZ, D. Microsoft TerraServer: A Spatial Data Warehouse. In: **ACM SIGMOD International Conference on Management of Data**,2000,Dallas, Texas, USA. ACM Press, p. 307-318.

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.

DPI/INPE. **Mosaico do Brasil**. Disponível em: <http://www.dpi.inpe.br/mosaico>. Acesso em: jan. 2005.

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

MOREIRA, M. A. **Fundamentos do Sensoriamento Remoto e Metodologias de Aplicação**. UFV, 2003.

Open GIS Consortium. OpenGis Implementation Especification: Grid Coverages. Revision 1.0. Disponível em: <http://www.opengeospatial.org>. Acesso em: abril 2005.

ORACLE Corporation. Managing Geographic Raster Data Using GeoRaster. Disponível em <http://www.oracle.com>. Acesso em: abril 2005.

PATEL, J., J. YU, et al. (1997). Building a Scalable Geo-Spatial DBMS: Technology, Implementation, and Evaluation. SIGMOD Conference, Tucson, Arizona.

REINER, B., K. HAHN, et al. (2002). Hierarchical Storage Support and Management for Large-Scale Multidimensional Array Database Management Systems. **3th International Conference on Database and Expert Systems Applications (DEXA)**, Aix en Provence, France.