Designing a New Raster Sub-System for GRASS-7

Martin Hruby
IT4Innovations Centre of Excellence
Brno University of Technology
Božetěchova 2, BRNO, 61266, Czech republic
hrubym fit.vutbr.cz

Keywords: GRASS, GIS, raster sub-system, geographical data, 3D rasters

Abstract

The paper deals with a design of a new raster sub-system intended for modern GIS systems open for client and server operation, database connection and strong application interface (API). Motivation for such a design comes from the current state of API working in GRASS 6. If found attractive, the here presented design and its implementation (referred as RG7) may be integrated to the future new generation of the GRASS Geographical Information System version 7-8. The paper describes in details the concept of raster tiling, computer storage of rasters and basic raster access procedures. Finally, the paper gives a simple benchmarking experiment of random read access to raster files imported from the Spearfish dataset. The experiment compares the early implementation of RG7 with the current implementation of rasters in GRASS 6. As the result, the experiment shows the RG7 to be significantly faster than GRASS in random read access to large raster files.

Introduction and motivation

GRASS [1] is a well-known open-source GIS system with a long history, wide range of included analytical tools and rather large community of users. Most of the users cooperate on its development, they comment its quality, do report its bugs and some of them implement their own system or analytical tools in form of GRASS independent modules. The main GRASS'es strength is traditionally supposed to be in the raster analysis and so, rasters should be something very well working in GRASS. Unfortunately, the current raster Application Interface (API) of GRASS-6 does not offer functions which would comfortably support a development of complicated raster analytical algorithms. Moreover, the current raster core probably will not process large raster files efficiently. This paper is giving an alternative which, if found reasonable, might replace the current raster core with this new one – referred as RG7 (Rasters for GRASS-7) in this paper.

There are several open source initiatives and traditional software packages oriented to processing of raster files. Let us mention at least the famous GDAL [5] (eventually GDAL/OGR), Postgis Raster [6] and Raster3D [7]. The GDAL library is mostly a collection of storage formats and a very large software complex. Postgis gives mostly a storage functionality with a certain support for raster data manipulation and analysis – however, via SQL statements, which is an opposite approach to RG7. The Raster3D library is the only one software intended strictly to GRASS and, currently, is still in development. Comparing to them, the RG7 library has got a different motivation which comes mainly from the user's perspective. The main idea is to provide a very abstract view to the raster data, moreover, to provide it

in highly efficient way. And, at second, to keep the library code as simple as possible.

The paper is going to discuss the Application Interface giving the user a set of functions operating the user data (raster map layers), the format of storing raster data and the whole architecture of the raster sub-system.

Let us begin from the current state of the GRASS, it means, let us summarize the current abilities of raster API in GRASS-6 [2] (and consequently the architecture of the current raster core):

- Set of functions for opening and closing an existing raster or creating a new raster file.
- A function (`G_get_map_row(...)`) for reading a specified row from a raster layer.
- A function (`G_put_map_row(...)`) for writing a specified row to a raster layer.
- Other support routines implementing the metadata including an eventual reclass table.

Clearly, any API must contain functions for opening, creating and closing the raster data. As a new feature, we may discuss a multi-user access to the data, various rights to access the data, remote access to the data (e.g. in form of a GRASS file server), transactions an so on.

System of reading and writing a row of raster is the main point of criticism of the current state of API. The problem has two levels of discussion: at first, user's approach to the data and at second, system implementation of such a call. In the current state, an user obtains a particular row of the raster no matter what part of that he needs. This kind of a sequential approach is sufficient for rather primitive tools (r.mapcalc, r.in/out, etc.), but let us imagine an analytical algorithm requiring a random access (or better say – an index-sequential access) to the whole raster layer. *A developer implementing such a tool has to create his own library over the GRASS raster API caching the lines and serving as a necessary abstraction over the API.* As I suppose, we would rather like let the developer concentrate himself on the core algorithm of his application.

On the other hand, the current GRASS raster sub-system together with the set of raster tools is nowadays optimized for such an approach. It must me told in the very beginning: *the here presented approach is not generally better than the current one, but it has a good chance to be a good advantage for future GRASS'es developmental grow.* This paper presents some sort of an opposition to the current row access approach, even if some developers claim that this row-approach is natural for a large part of raster analytic algorithms. Some comparisons and contemplations are put in the section "Experimental results".

Let us now give the features of the RG7 raster sub-system:

- Support for extremely large raster files with a guarantee of constant time access to any part of the layer, i.e. 1) any part of the raster is accessible in the same time as another one, 2) size of the raster has no influence to the access time (or just marginal).
- Native support for multiple data formats and storing the rasters in a SQL database (for example in PostgreSQL [3] or SQlite3 [4]).
- Real multi-user access and GRASS ready to work as a file server.
- Integrated 3D-raster concept.

- Comfortable API ready to give any sort of view to the data. Development of user raster application on a very high abstraction level.
- Raw raster data and other information packages (mainly the set of various metadata) operated in a unique manner and stored in an unique place, where SQL databases are preferred[1].

Processing the extremely large rasters (size of gigabytes) together with any user-specified sliding window over the raster needs a special design of a store format and quick access to sub-region within the file. *This is why the RG7 is based on raster tiles.* Every raster file available through RG7 will be physically decomposed to a grid of so called *physical tiles.* This is the main difference to the current GRASS raster concept – not the row, but a tile is the raster atomic element of data. *A tile is a block of fixed size no matter what size has got the raster file*, i.e. raster files with long rows do not influence the complexity of their processing. The advantages and disadvantages are discussed in the section "Conclusion of the experiments".

It will be shown, that with the RG7 design, the technical approach of storing the data is absolutely independent to its logical tile representation. Moreover, many storing principles can be used, e.g. storing in a classical file managed by an operating system (referred as a OS-file), in a SQL database or virtualized via network.

The main interest of the RG7 design is focused into the set of functions making an interface between an user and the GRASS kernel. Simplifying the style of raster programming might encourage new users to bring their algorithms to the GRASS code.

Technical and geographical definition of the tiles

Definition of a tile

A tile is a geographically defined sub-region within a raster layer. A raster file itself is defined by its boundary coordinates and number of cells in north-south and east-west directions. Similarly, a tile is defined by its boundary coordinates and number of cells in both directions.

By decomposing the raster to a set of tiles, we get an organized system of small data elements (tiles) which are easy to operate. A tile becomes as an almost regular grid decomposition of a raster file, as it is shown in Figure 1. This system of recursive decomposition making a spatial index is called the Quad-Tree (let us assume this term to be a common knowledge in the GIS community).

As we may see bellow, there are several ways of defining a tile regarding its size. There are generally two approaches to raster file decomposition:

- Regular - all tiles are of the same size and the size is equal in both directions (the tile has a square shape). Constructing a spatial index in form of a Quad-Tree is not necessary to navigate through the raster. An optional implementation of raster pyramids is logical and easy.

[1] Motto: no files, no troubles.

  - Advantages - mapping a geographical coordinate to a particular tile has a constant time access, it is easy to implement and operate.
  - Disadvantages - not very practical as the input raster must have a square shape and its number of raster elements in both directions (cells) must be 2^n, where $n \in \mathbb{N}$.

- Irregular - the file is decomposed using a Quad-Tree procedure with a minimum tile attribute (see bellow). It may be applied to all existing raster files as the limitation of regularity is not required here. An optional implementation of raster pyramids is still possible.
  - Advantages - no limits for rasters in the meaning of their size.
  - Disadvantages (time issue) - each tile is of different size. Mapping a geographical coordinate to a particular tile must be done through the Quad-Tree spatial index, i. e. in logarithmic time complexity. This computational overhead can be successfully minimized in the internal algorithms of RG7, for example by using an explicit topological links among the neighbor tiles.
  - Disadvantages (space issue) - number of tree nodes (i.e. virtual tiles) grows exponentially with height of the tree. The height of tree depends on size of the raster and size of the minimum tile.

Clearly, the requirement of having all files of size in 2^n is not very practical. Therefor, the RG7 concept **strongly assumes the files of any size and implements the irregular tiling**.

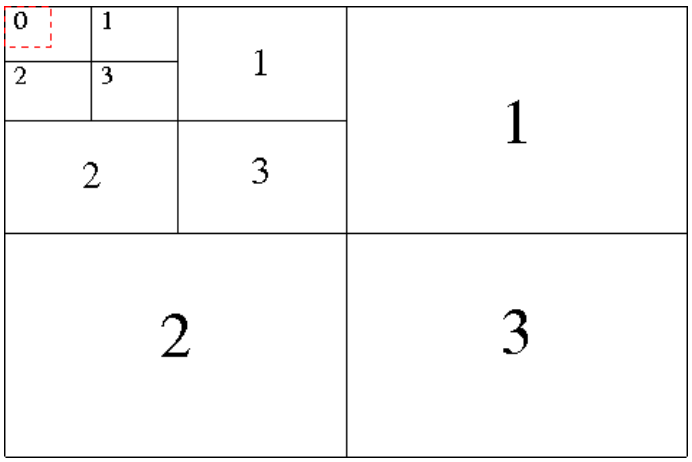

Figure 1: A demonstration of the irregular Quad-Tree

Decomposition of a raster to a tree of virtual and physical tiles

In RG7, *size of a tile is not fixed*, i.e. is not equal for all tiles, and it is influenced by so called *minimum region*. Four tiles is the minimum number of tile decomposition, i.e. having a raster (or generally any region) with $countX \times countY$ cells and minimum region of size $mregX \times mregY$ cells, the raster is split into a quadruple of virtual sub-tiles only if (1)

holds. If the condition (1) is false, the region $countX \times countY$ is not split and remains as a physical tile (see Figure 1 where the red rectangle defines the minimum region for further decomposition). The regions which allow their decomposition are then called the *virtual tiles*.

$$countX \geq 2 \cdot mregX \wedge countY \geq 2 \cdot mregY \tag{1}$$

The raster spatial index is based on Quad-Trees, where the tree nodes are are referred as *virtual tiles* and leafs (non divisible regions) are the *physical tiles*. See the Algorithm 1. The condition (1) also causes every physical tile to be generally larger then the specified minimum region, thus the minimum region does not denote a wanted size of a tile, it just define the stop condition of the decomposition. Each tile (virtual or physical) is addressed by a prefix code as described in Figures 2. In Figure 2, the top-left tile is labelled 0 and when this one tile is decomposed further (see Figure 2 on right), a prefix 0− is added.

Algorithm 1 Decomposition of a region to a tree of sub-regions

```
class Ras_vtile {
 Ras_vtile(Region reg) : region(reg) { ... }
...
 void decompose(ICoord mreg) {
   phtile = 0;

   if (region.countX() >= 2*mreg.x && region.countY() >= 2*mreg.y) {
     int mx = region.countX()/2;
     int my = region.countY()/2;

     subtiles[0] = new Ras_vtile(region.sub_region(0, my, mx, region.countY()));
     subtiles[1] = new Ras_vtile(region.sub_region(mx, my,
         region.countX(), region.countY()));
     subtiles[2] = new Ras_vtile(region.sub_region(0, 0, mx, my));
     subtiles[3] = new Ras_vtile(region.sub_region(mx, 0, region.countX(), my));

     for (int a=0; a<4; a++)
       subtiles[a]->decompose(mreg);
   } else
      phtile = new Ras_phtile(region);
 }

 Region region;
 Ras_vtile *subtiles[4];
 Ras_phtile *phtile;
}
```

In such a convention, each physical tile has got its label and using that label, one can find the tile in a quad-tree hierarchy. The label is also an index key to identify the tile in a database storage.

0-	1-
2-	3-

0-0	0-1	1-0	1-1
0-2	0-3	1-2	1-3
2-0	2-1	3-0	3-1
2-2	2-3	3-2	3-3

Figure 2: Addressing tiles in a decomposition of a raster (depth 1 and 2)

Architecture of the raster sub-system

Having defined the concept of Quad-Tree for RG7, we may proceed to description of the RG7 itself. The RG7 sub-systems consists of the following modules, each responsible for certain functionality:

- **Physical module** (PH) – PH implements a direct low level access to raster files. It creates a spatial index tree, reads and writes particular tiles. PH also implements a memory cache over tiles. This module has the major influence to practical run-time efficiency of RG7.

- **Connection module** (CN) – CN establishes a connection between a particular raster application and a physical module. CN implements a kernel which controls the user access to the files. CN also manage a possible multi-user (or multi-application) access to the files.

- **Raster window module** (RW) – RW implements an user specified view to a raster file. RW contains algorithms of raster resampling and presentation. RW specifies the user API in various formats, i.e. at least the full RG7 C++ API and GRASS standard-like C API.

- **Raster application modules** (AM) – AM is a set of user implemented raster analytical modules using RG7 API at a CN or RW level.

The RG7 API shall connect users through the CN and RW modules. Direct access to PH module is not assumed for a standard user. The direct access is allowed only for debugging and benchmarking purposes.

The whole design is prepared for object-oriented (OO) implementation in C++. Object oriented C++ is preferred for its high level of abstraction and support of semi-standard C++ libraries (Boost, STL). To keep backward compatibility with GRASS-6 raster API, a certain part of RG7 API will contain headers in classical C language. There is absolutely no technical problem in combining a C and C++ code in one software package. Moreover, in my personal opinion, the dogma of keeping GRASS in pure C might be a serious limitation in GRASS'es future development.

The Physical module of RG7

Definitions of the basic geometrical objects

Let `XCoord` denotes an abstract class implementing a 2D point in a plane[2]. Let X and Y are the two components of the coordinate – X representing the eastings and Y representing the northings.

Then, let `Coord` is defined in floating point numbers and `ICoord` in integers. `Coord` expresses a geographical coordinate (in any coordinate system) and `ICoord` just a relative coordinate, mostly an index to the raster grid.

`Region` will denote a geographical rectangular space given by its left-bottom corner (`LB:Coord`) and right-top corner (`RT:Coord`). Surely, speaking about LB and RT is equal to giving four numbers expressing north, south, east and west boundaries of the region. Rasterization of the region is given by number of cells in east-west dimension – `countX()` and south-north dimension – `countY()`. The `Region` implements a method `inside(Coord i)` returning *true*, resp. $false$, if a point i is geographically inside the region, resp. if it does not.

Basic definitions - A physical tile

A *physical tile* (`Ras_phtile` in the code) is geographically defined by its `region`. The raster data contents is stored in a 2D array, in a `matrix` object, of dimension `region.countX()` × `region.countY()` cells. Other internal attributes are not interesting at the moment. Let us see the main I/O methods of the physical tile.

The functionality of `Ras_phtile` is mainly in these methods:

- `wphys(ICoord i, dtype v)` – writes a value v to the matrix at i position.
- `rphys(ICoord i)` – returns a value at i position.
- `write_to_compressed_ba` – the function outputs a serialised stream of `matrix` contents to a compressed byte array (compressed using run-length method). The compression methods are subjects for further work and thus kept simple in the current design.
- `read_from_compressed_ba` – loads and decompresses the serialized stream to the `matrix` object.
- `allocate()` and `deallocate()` the `matrix` object. The existence of data in physical tiles is only virtual and the tile's contents is loaded just in the moment of its demand. By allocating the `matrix`(`Ras_phtile`), we mean allocating a computer memory for `matrix` object. By deallocating we mean giving the memory back on heap.

Virtuality of the raster data contents

The attribute `matrix` of `Ras_phtile` consumes a non-trivial part of computer memory. When a raster file is opened, RG7 automatically creates its tree representation made of virtual tiles and physical tiles. But no data is loaded from the database storage yet and so, no physical tile allocates a memory for the `matrix` buffers.

[2]It might be extended with the 3rd dimension very soon.

Ras_phtile object representing a particular tile stores the raster contents only virtually until its allocate() method is invoked. The method allocates a memory for matrix and the tile's contents may be loaded (read_from_compressed_ba). When the tiles is not needed, its deallocate() method frees the matrix memory. Clearly, before deallocating the buffer with some write changes, the tile's contents must be stored in the database. This memory management approach is going to be described bellow in further details.

The Physical module in C++ classes

Let us now introduce the main C++ classes making the Physical module of RG7 (See Figure 3):

- Ras_phys_file – manages a spatial index of a file and physical tiles. This class is abstract in the meaning that it just manages the tiles without any direct link to their physical storage (this is done through the following Ras_interface class). The Ras_phys_file class manages an amount of memory used by tiles via calls allocate() and deallocate() (see section "Memory management").
- Ras_interface – holds metadata (an instance of Ras_metadata) for a file and implements its particular data format. As it will be described bellow, the RG7 functionality may be extendible right through these interfaces. RG7 may implement interfaces for storing the data in files, SQL databases or even in network services.
- Ras_metadata – implements a storage of metadata for a raster file. The metadata contains a lot of raster's attributes, but at least two: region specifying the raster's geometry and minimum_region determining the raster's decomposition to tiles.

The Ras_phys_file instance represents a single opened raster file. When opening a raster file, the RG7 system instances a Ras_interface relevant for the raster file (depending on its particular data format). An instance of Ras_interface (simply the interface or iface) is responsible for the input and output of raster's metadata (at least region and minimum_region) and for all I/O operations over tiles. Then, an instance of Ras_phys_file is created (and given the interface). Having region and minimum_region, the Ras_phys_file can establish the quad-tree representing the spatial-index (Algorithm 1).

By opening a raster, the RG7 only constructs relevant data structures in computer memory (objects Ras_phys_file, Ras_interface, Ras_metadataand the spatial index). Loading the raster contents (the map itself) depends then on user's requirements. In multi-user (multi-application) access, the Ras_phys_file is instanced only once in the RG7 kernel.

Memory management over the physical tiles

As it has been already mentioned, a given raster is split to a grid of physical tiles which are supposed to contain the relevant data. Certainly, the whole raster will probably not fit the computer memory, therefor some sort of memory management must be designed. This memory management is going to be very similar to a well-known system of virtual memory managed by todays operating systems (virtual and physical memory pages).

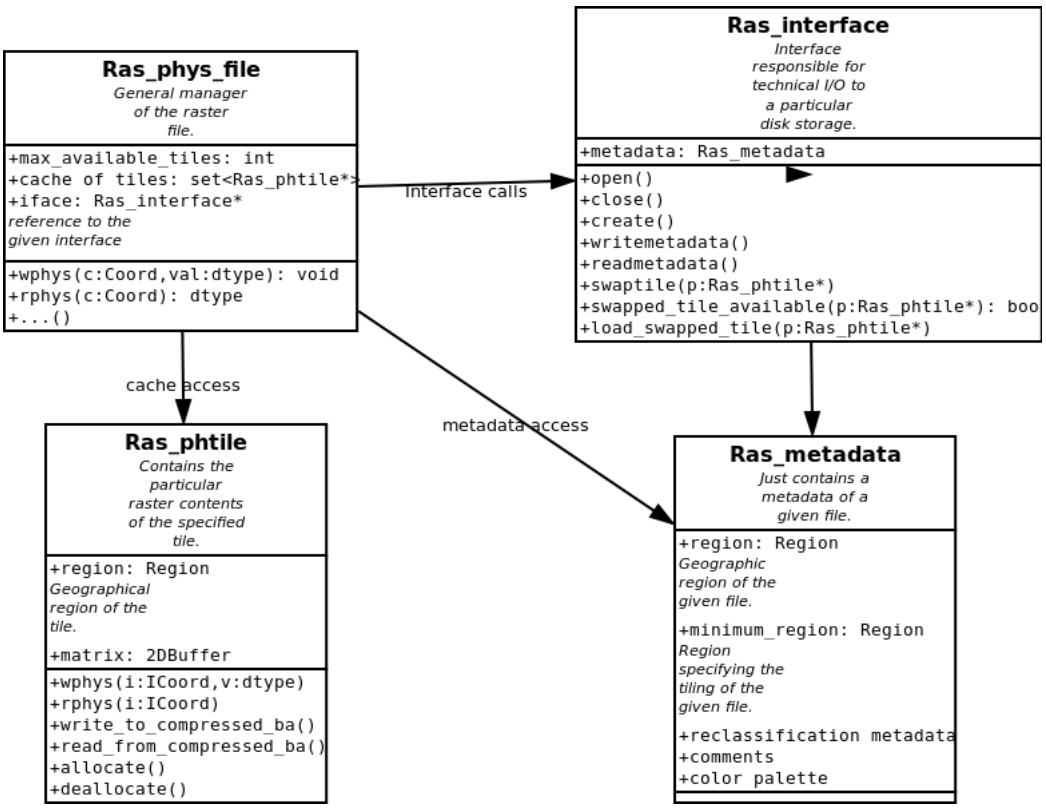

Figure 3: UML overview on the Physical module classes

A physical tile may be in one of two possible states (relevant for the tile's memory consumption/reservation) – allocated or deallocated, i.e. consuming computer memory or not. When the user requests a raster attribute at some geographical coordinate, this request is translated to its logical coordinates which identify a particular tile p and local index lc. Then:

1. If p is deallocated at the moment, `Ras_phys_file` allocates a memory for p, and asks its interface (`iface`) to load the tile from its disk storage. Then, p is allocated.

2. If p is allocated, $p.rphys(lc)$ is returned.

Certainly, not all tiles can be allocated at the same moment due some operation memory limitations. For this purpose, `Ras_phys_file` can be set to keep as maximum `max_available_tiles` physical tiles in allocated state (as default, `max_available_tiles` is set -1 and then there are no limits). As it has been mentioned, the memory management is inspired by the virtual memory management – all physical tiles contain data only virtually and the `Ras_phys_file` object assigns the memory resources on demand, it means, in a situation when there are already `max_available_tiles` physical tiles allocated and another tile is required to load its data, *one of the current allocated tiles must be deallocated.* The algorithm doing such a decision is another problem to discuss. At the current state of RG7, the `Ras_phys_file` object keeps certain access statistics on tiles and selects the latest accessed tile to be deallocated.

Virtual storage interfaces

One of the most valuable features of the RG7 design is in allowing developers to implement various formats of storing the rasters. `Ras_phys_file` operates a raster file in an abstract manner invoking its interface for certain basic operations. Every class derived from `Ras_interface` implementing the following methods may define its own data format:

- `open()` – opens an existing raster file and loads its metadata.
- `close()` – closes all files needed by the raster layer.
- `create()` – creates a new raster layer using the given metadata.
- `writemetadata()` – stores the raster file's metadata.
- `loadmetadata()` – loads the raster file's metadata.
- `swaptile(p)`– if the contents of $p.matrix$ was modified, $p.matrix$ is serialized and flushed to disk storage (or any persistent device). Invoked usually when p is selected to become deallocated or when closing the whole file.
- `swapped_tile_available(p)`– returns a boolean saying whether a raster contents of p is present on storage or not. If p is filled with NULL values, there is no need to store that fact on disk.
- `load_swapped_tile(p)`– a tile p loads its data contents from disk storage (p must be allocated before).

Having such an interface, one may implement any data format or a way of storing the raster data. For example, these interface definitions are going to be included in RG7 early design:

- `Ras_interface_sqlite` – the main assumed interface for RG7 via SQLite3 [4]. Tile's `matrix` contents is stored in an SQL database in *BLOB* records.
- `Ras_interface_postgres` – similar to `Ras_interface_sqlite`, but implemented for PostgreSQL. PostgreSQL works as an independent OS process, thus this storage processor might be faster for heavier traffic loads than SQLite3 (especially with multiple-core CPUs). This is an issue for further testing and experience.
- `Ras_interface_grass6` – an interface providing a compatibility to current GRASS-6 raster storing format. The interface has to read multiple rows to complete a 2D tile, thus it can be efficient only with classical raster row-oriented applications.
- `Ras_interface_WMS` – an interface providing an abstraction over WMS services. WMS raster files are selected as read-only then.
- `Ras_interface_sfile` – all physical tiles are stored in an unique disk file and the interface keeps an offset table in a separate disk file (similar to ESRI Shapefile .shx). Fast and rather easy to implement.

Efficient implementation of this physical storage level is essential for the RG7's read/write performance. For this reason, the `Ras_interface_sqlite` interface defines an unique database index ($RasterID \times TileID$) for fast searching in the DB storage.

Raster Metadata in RG7

Metadata are operated through the interface object using its `writemetadata()` and `loadmetadata()` methods. The loaded contents is then kept in `Ras_metadata` objects having the following parts:

- Geographical description of the raster – boundary region of the raster (referred as basic region) and its size (number of cells in both directions).
- User comments – various text fields inspired mostly in GRASS raster metadata.
- Tiling – minimum region (see 2).
- Values description – particular null value and the cell's data type (char, two-byte integer and four-byte float).
- Reclassification – reclassification table and reference to an original raster file (may be generally in different format).
- Colour palette – reference to a standard implemented palette or an user-specified palette (just for graphical presentation).

RG7 does not assume *an extra raster storage to keep explicitly the null data*, thus there must be one value reserved to specify the NULL contents of a cell. By default, the null value is set to numerical zero. Let us remind that a tile containing only null values in all cells is not stored, i.e. `swapped_tile_available(p)` interface method should return false. If such a tile is modificated by writing some non-null data, the tile is then swapped when deallocated.

The connection module of RG7

The Connection module provides the interconnection between the user application (analytical module) and the internal raster kernel managing the set of active `Ras_phys_file`. The Connection module consists of two classes:

- `Ras_kernel` – the class is instanced as a singleton `RG7`. The `RG7` object serves the users to open, create and close the requested raster files. For a required file, `Ras_kernel` returns a particular `Raster` object making a handle to a particular open raster.
- `Raster` – objects of this class provides a handle to open rasters, i.e. realize required read and write operations to the rasters. The object also gives the complete metadata information.

The `Ras_kernel` and `Raster` objects are supposed to be an only channel to a particular source of raster data from the user's perspective (to a local GIS kernel, to a remote GIS file server, etc.). There are some details in multi-user or multi-process accesses in the design to finish. At the moment, let us assume just one application processing the data through just one `Ras_kernel` channel.

The `Ras_kernel` object registers all open/created raster files and assigns a `Ras_phys_file` object for each one. Multiple open request to a unique file always leads to a single `Ras_phys_file` object.

As the `Raster` objects provide the complete information access in both read/write directions (together with the Window module), it in fact makes the basic necessary interface between an user and the raster GIS kernel. *The whole raster implementation is thus encapsulated inside this abstraction and so, the RG7 concept may serve as a new abstract API for applications even without the new tiling approach, just as an abstract layer over the classical GRASS raster engine.*

The window layer

The `RWindow` class provides a final element of RG7 C++ API. An application may specify a window which can slide within the region of `Raster` object. The important fact is that an user does not invoke explicitly the read/write operations – he just points the window as a text cursor and shifts that as he needs.

When a window is placed (or shifted/moved), its `RWindow` object requests the `Raster` object to obtain the relevant data. All data edits are made through the window as well, so if `RWindow` objects is asked to move at some other coordinate (or is being destroyed), `RWindow` object then performs the write operation automatically.

`RWindow` object offers a basic resampling of a raster based on its internal attribute `region`. If the region is left default, i.e. taken from the file's metadata (its basic region), then the window reads the raster in its original resolution and in fact, it accesses the physical raster data. Furthermore, an user may specify his own region and then the region translates the relative coordinates of the window to the geographical coordinates and identifies the source cells. By default, the resample method is set to the nearest neighbor. Specifying an own region is frequent in analytical tools respecting the standard GRASS monitor. The GRASS monitor settings (a global region accessible via `g.region`) is available by invoking `Ras_kernel::monitor()`.

The `RWindow` operations are these:

- `read(ICoord c)` – returns a value of a cell at c local position within the window.
- `write(ICoord c, dtype val)` – similarly, writes a value *val*.
- `topleft()/topright()/bottomleft()/bottomright()` – move the window on such a position.
- `shift_right()/...` – shifts the window one cell on right/left/down/up.
- `moveAt(ICoord c)/moveAt(Coord c)` – points the window at a given position.

There are various sorts of raster windows in the RG7 design:

- RWindow – basic single-layer window with user defined size and region of resampling.
- RWindowPicture – a variant of RWindow, exportable to a bitmap picture.
- RWindowRow – a variant of RWindow where the size is automatically set to $(1, columns)$ where *columns* follows the geometry of the given raster file.
- RWindowMulti – a multi-layer window. It allows opening multiple raster files with an unique window, each file either for write or read access.

- RWindow3D – a window suitable for 3D raster requests. In fact, that is a multi-level RWindow defined on a single file.

Let us see the following demonstrative example of various access windows.

```
RWindow r1(handle, ICoord(3,3));
RWindow r2(handle, ICoord(3,3), RG7.monitor());
RWindow r3(handle, ICoord(1, RG7.monitor().countX()), RG7.monitor());
RWindowRow r4(handle, RG7.monitor());
RWindowMulti r5(ICoord(3,3), RG7.monitor());
r5.add(handle);
r5.add(RG7.open("elevation.dem"), READONLY);
RWindowPicture r6(handle, RG7.monitor());
```

The `r1` window of size 3×3 is open for previously open raster using its natural (physical) resolution and for its natural region. Similarly, `r2` is created with the same window size, but defined on the global GRASS region specified by the `g.region`, i.e. including its resolution. The windows `r3` a `r4` are equivalent. The `r5` has got two layers with rasters "geology" (previously open) and "elevation.dem". Sliding `r5` will load/save raster cells of both layers simultaneously. The window `r6` is going to have its size specified by the current monitor settings, i.e. by its boundaries and resolution. In fact, it gives an whole picture of the raster as it might be seen in the `d.mon` GUI window and then printed by `r6.plot("out.bmp")` as a raster picture.

Application interface to RG7

RG7 is implemented in GNU C++, it means that its code is made to be easily understable for developers, ready for enhancements and using various STL libraries wherever it is possible. Over its core, various APIs might be specified:

- API compatible with the already existing C API standard such that absolutely no modifications to the existing raster tools will be needed.
- C++ API giving the users all the new enhancements of RG7.
- API connecting other programming languages.

It is absolutely sure that implementing RG7 may not hurt the overall functionality of GRASS, i.e. of GRASS analytical and system modules. It is a matter of time and future experience if some modules will be re-implemented to gain a higher performance coming with a new raster storing and processing.

An 3D raster support

The RG7 concept is open for 3D-raster extension with only minor modifications to its basic algorithms, I/O interface and program code.

The suggestion is the following:

- The 3rd dimension is discretized in the regular way with a constant step. It makes a set of 2D unique raster files, where identification of a particular tile is done in two steps:

identification in 2D and determination of the layer in 3rd dimension.

- Labeling the tile is extended with a number of a required layer (I/O operations provided by the interfaces).
- Window may slide on an single or on all layers (3D window).

Experimental results

This section is going to present few benchmark experiments demonstrating access times to rasters in different manners of their use. We assume the following datasets (Table 1) imported from Spearfish60 demonstration dataset [8]. The rasters were generated with appropriate `g.region res=` resolution and then exported by `r.out.ascii`. The data were then imported to RG7 software prototype. All benchmarks have been done based on the SQLite3 interface (see the section "Virtual storage interfaces"), i.e. with rasters stored in SQLite3 database. The minimum region was set 32×32 cells for almost all layers (the "geolarge" file with 64x64 tiling). The benchmark was performed on a 4xCPU Xeon 3GHz PC with 8 GB RAM running on Linux OS. The benchmark measures the whole time of the application's time run using the `time` unix utility, i.e. the time measured also includes some application overhead with initialization etc. *It should be mentioned at the very beginning of this case study, that the experiment (called the benchmark here) does not have a quality of a proper laboratory measurement and its main purpose was not estimate the exact algorithmic complexity of RG7 access times on rasters.* The purpose was rather to show the general difference between access times on GRASS and RG7 *which will be evident* and so, not very precise measurement of run times *is generally acceptable.*

Spearfish original name	Test-name	Original size (rows, columns)	Test size	Num. of tiles (imported to RG7)
soils	soils	750 x 950	750 x 950	256
geology	geol	140 x 190	140 x 190	16
geology	geolbig	140 x 190	14,000 x 19,000	65,536
geology	geol05	140 x 190	28,000 x 38,000	262,144
geology	geolarge	140 x 190	140,000 x 190,000	4,194,304 (tile $\geq$ 64x64)

Table 1: Testing raster files imported from Spearfish60 dataset.

The Table 1 consists of raster files taken from the Spearfish dataset (Spearfish original name) with their original stored resolution (Original size in rows and columns). The files has been resampled and exported in GRASS and then given a case-study identifier (Test-name) and case-study resolution (Test size). When imported to RG7, each file has been automatically decomposed to the raster tiles (Number of tiles). The "geolarge" file has been decomposed with 64x64 minimum region tiling, the others with the default 32x32 minimum region tiling.

Random physical read access

In this experiment, a given raster is open and sets its "max available tiles" attribute denoting a maximum number of tiles in the cache. Let us remind the term "maximum available tiles" (see 2) denoting a cache size of tiles in memory. The experiment proceeds a given number of iterations, where in each iteration:

1. a random coordinate c withing the raster's region is generated,
2. a cell value at the position c is requested. If the required physical tile is not present in the cache, the tile must be loaded from its disk storage (respecting the limitation of "max available tiles").

As the access is fully random, only two experiments have a sense – an experiment with a single tile cache ("max available tiles" is one) and an experiment with a non-limited cache ("max available tiles" is "full", i.e. unlimited). We should keep in mind that, with this random access, the randomly chosen cell very surely points on a different tile than in the previous iteration.

Test-name	max. available tiles	n. iterations	time [s]	avg. access time **per tile** [ms]
geol	1	10^5	4.4	0.044
soils	1	10^5	5	0.05
geolbig	1	10^5	8	0.08
geol05	1	10^5	10.3	0.10
geolbig	1	10^6	96	0.096
geol	full	10^6	0.162	–
soils	full	10^6	0.253	–
geolbig	full	10^6	8.6	0.09
geol05	full	10^6	30.4	0.03
geolarge	full	10^6	6	0.09

Table 2: Benchmark "random access" results measured on RG7

When having only one "max available tiles", almost at every iteration, the requested tile must be loaded through the given interface, i.e. from the SQLite3 database engine. The results show that obtaining any physical tile takes in average about 0.1 ms *no matter what the whole size the raster file has* (or, at least, the access times are in the same digit place). In other words, at the current implementation of RG7, this prototype can proceed approximately $10,000$ tile loads per second with absolutely random order of tiles. Let us note, that this number will be double or triple when using burst readings (database transactions, SQL selects of multiple tiles, etc.).

If the cache supports an unlimited number of physical tiles (referred as "full"), almost all tiles of the input raster must be loaded during the iterations, but just once. The complete time for "geol" and "soils" is trivial, just "geolbig" in compare to "soils" shows about 6 seconds more to load all 65536 tiles (it makes aprox. 0.09 ms/tile), which is not too bad for this early implementation of RG7. See Table 2 to get the experiment's results.

The experiment has been extended with a more detail sampling of the runtime of the benchmarking program for the "geol", "geolbig" and "geol05" raster files. The Figure 4 shows the results when using a single-tile cache and the unlimite cache. The left graph is not very surprising – the access to a tile takes some constant time, thus the resulting function for all raster files is linear. On the right side we may see, that (see the "geolbig" file) after certain number of iterations, the cache gets filled with all tiles and further iterations does not touch the disk and the request are completed within the cache itself.

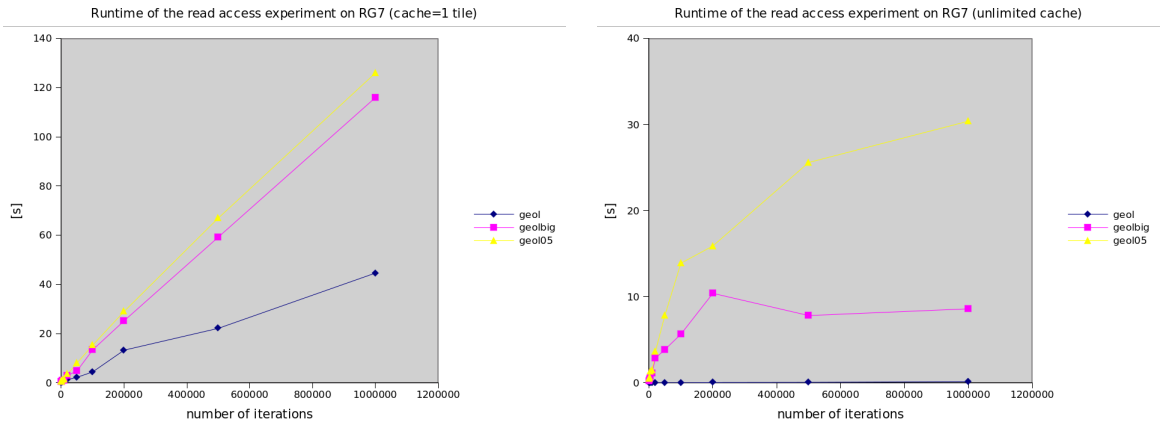

Figure 4: Benchmark "random access" results measured on RG7 with geol, geolbig and geol05 datasets. Measured runtime with single tile cache (on left) and unlimited cache (on right).

The Figure 5 shows the average access time to a tile. Let us mention mainly the effect of removing the application overhead (loading the program, establishing the quad-tree, etc.) causing the convergence of the computed average time to a certain true value.

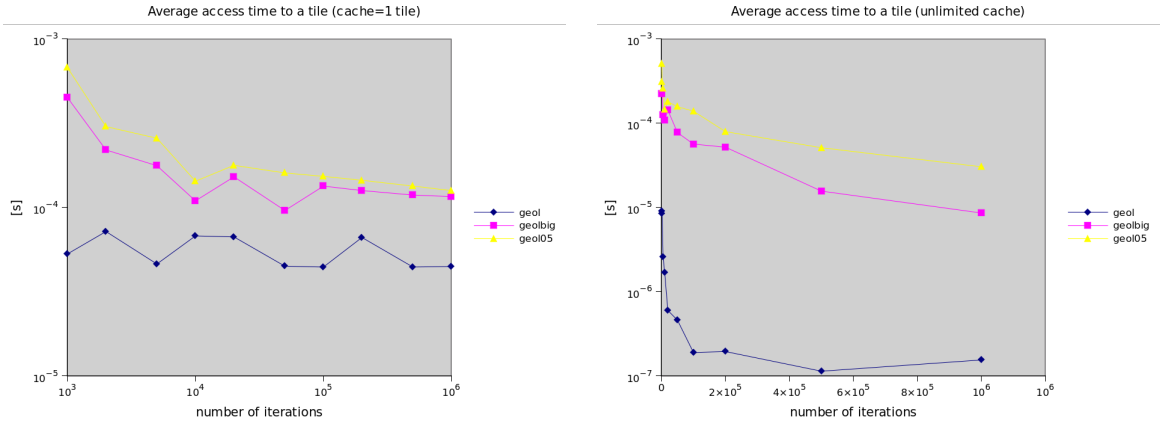

Figure 5: Benchmark "random access" results measured on RG7 with geol, geolbig and geol05 datasets. Measured average access time to a tile with single tile cache (on left) and unlimited cache (on right).

Let us proceed a similar experiment done in GRASS. The experiment was implemented using a demonstration tool called `r.example` (loop with random `Rast_get_row(infd, inrast, random()%nrows, data_type)`). The region had to be manually set regarding the current experiment, e.g. `g.region rast=geology`.

The results at the Tables 2 and 3 clearly show the following important observations:

- GRASS is significantly faster than RG7 with 1-tile cache in small files. That's probably because the GRASS kernel loads the whole file in one shot at the first read access and the further readings are then done using its internal cache. The file "geol" is also so small (around 8KB) that it takes only 2-3 pages of virtual memory and thus kept whole

Test-name	n. iterations	time [s]	avg. access time **per row** [ms]
geol	10^5	0.5	0.005
soils	10^5	1.7	0.017
geolbig	10^5	34	0.34
geol05	10^5	112	1.12
geolarge	10^5	915	9.15
geol	10^6	4.5	0.0045
soils	10^6	17	0.017
geolbig	10^6	314	0.314
geol05	10^6	963	0.963
geolarge	10^6	4,176	4.176

Table 3: Benchmark "random access" results measured on GRASS

in OS disk buffers.

- GRASS is significantly slower than RG7 in small files when RG7 has got an unlimited cache (soils, 0.253s versus 17s).
- GRASS is significantly slower in large files (see the "geolbig" results) compared mainly in case of a single tile cache – 8s versus 32s (96s versus 314s).

To conclude the first experiment, it must be told that the measurement is not very precise due to the influence of OS disk buffers which are not under the tester's control, but, anyway:

- In the category of small files, we compare GRASS results with RG7 unlimited-cache results, and, RG7 wins. We assume that the raster is surely cached in GRASS, thus this comparison is correct.
- In the category of large file, we compare GRASS results with RG7 single-tile results, and, RG7 wins. We assume the GRASS not keeping such a large file in buffers, thus, again – this comparison is correct.

Correlation between number of tiles and the access time

One might think about the time complexity of index-sequential accessing to single raster tiles. Is there any correlation between their size and expected time necessary to fetch a required tile? Certainly, there is some connection, however very small as it is shown in this sub-experiment.

I generated nine square raster files similar to the previous ones. These rasters are not generated from any existing raster file (like before), they are fully artificial but having all the same contents. See the Table 4 for the raster file definitions and for the measured results. The experiments are basically identical to the previous bundle of runs. The benchmarking program was executed with a given raster file and set to use just single tile cache. Number of iterations was set equally 1 million of iterations. The table displays the overall time of each program's execution. As we may see, through all rasters files, the million of iterations (and loading a tile) took about the same time no matter what the size of raster actually is.

File ID	Number of rows (columns)	Number of stored tiles	Runtime of the experiment [s]
g1000	1,000	256	66
g2000	2,000	1,024	67
g5000	5,000	16,384	42
g10000	10,000	65,536	46
g20000	20,000	262,144	54
g50000	50,000	1,048,576	74
g70000	70,000	4,194,304	72
g200000	200,000	16,777,216	54
g300000	300,000	67,108,864	81

Table 4: Generated raster files and runtime measurements for the experiment in Chapter 2

Random window read access

This experiment consists of random read accesses with a square window (3x3). The experiment is done with either 4 tiles cache (the window may intersect up to 4 tiles) and unlimited cache. Surely, the practical raster analyzes do not "jump" from a random point to another, the experiment just shows a possible access time when sliding a 3x3 raster window. The results (see Table 5) are not compared to GRASS as the measured times would be just a triple of the previously measured ones.

Test name	max. ava-tiles	readings	time [s]	avg. access time **per window** [μs]
geol	4	10^5	6	60
soils	4	10^5	6	60
geolbig	4	10^5	8.5	85
geol	full	10^5	0.15	1.5
soils	full	10^5	0.2	2
geolbig	full	10^5	4.7	47
geolbig	full	10^6	9	90

Table 5: Benchmark "window access" results

Conclusion of the experiments

The RG7 implementation seems to be more efficient in the here presented benchmarks than the classical GRASS raster sub-system. This paper is not a comparative study of GRASS and RG7 in raster processing performance. That's another topic for another paper which might be worked out when RG7 gets more advanced and tuned. Anyway, there is a big performance reserve and hope in processing various `RWindow` requests as the burst selects on SQLite3 database are done in shorter time than a sum of tile's individual selects. That's the point where RG7 might compete GRASS in classical row sequential analyzes as well.

There is only only one point of performance difference between these two approaches where GRASS seems to be still faster: d.mon. The d.mon utility displays a rather small number of rows and so, it reads a small number of rows from disk. Comparing to that, RG7 has to read a sequence of tiles which then complete the requested single row. The same experience

is probable in processing rasters in overview mode, i.e. not in their original resolution. Doing such a fast overview on the data is possible via raster pyramids.

Conclusions and future work

The RG7 design has been presented in this paper. The design is certainly in a very early developmental stage with, at the moment, no proper connection to its target infrastructure, and as it has been mentioned, GRASS-7 (or 8) might be the target. Moreover, the RG7 implementation is currently very theoretical and needs certain optimizations to ensure high performance of the resulting raster kernel.

Using the virtual interfaces (see the section "Virtual storage interfaces"), the RG7 library makes an abstraction over unlimited number of data storage formats and methods. The GIS system based on RG7 can operate raster data sources like GRASS raster format, RG7 Sqlite3 interface, WMS, GeoTIFF, etc. – all in the same manner.

The benchmark experiments demonstrated that a random access to a raster via RG7 is faster than via classical GRASS raster sub-system. This is pretty sure at the moment. One may say, that this success is perhaps just marginal as a large number of practical raster analyses reads the rasters sequentially and that's perfectly working in the current GRASS. However, as it was mentioned in 2, the optimized `RWindowRow` interface will probably defeat this argument.

There might be some criticism regarding the searching time overhead in Quad-tree spatial index. In fact, there are two sorts of searching trees: searching through Quad-tree spatial index and searching in database index file. Both are variants of tree data structures. *Let us mention that the practical Quad-tree height is rather small, e.g. 12 levels for raster* $200,000 \times 200,000$ *cells.* Time spent in searching within such a tree is really marginal. Similarly, in the case of the SQL database index file.

I wanted to show that tiling the raster, i.e. splitting the raster to a grid of small elements can guarantee some sort of independence on raster's size. Saying *constant access time* would be too strong as there are several aspects in computers influencing the result, however the experiments presented here give some support to the concept of RG7. The concept of RG7 has just one weakness at the moment, and that is the time spent in *generating the Quad-tree* when opening the raster. This is going to be fixed in the next RG7 development step.

Moreover, the computing performance is not the biggest issue. RG7 is attempting to show a different manner of accessing the raster data, and, I would say, a better manner than the current API provides. The idea of "windows" (or cursors?) of variable size is certainly not new. In the Computer science terminology, the rasters storage and raster API make together so called Abstract Data Type (ADT). The computer scientists know for many years that a good ADT ensures clever, efficient and flexible algorithms. On the other hand, a bad ADT kills surely the applications. Let us give a good raster ADT to the future GRASS program generation.

The RG7 design certainly counts on the standard GRASS-6 raster API ensuring a full backward compatibility with the existing raster analytical tools.

Acknowledgement: This work has been supported by the Grant Agency of Brno University of Technology No. FIT-S-11-1 *Advanced secured, reliable and adaptive IT* and the Czech Ministry of Education under the Research Plan No. MSM0021630528 *"Security-Oriented Research in Information Technology"*.

This work was also supported by the European Regional Development Fund in the IT4Innovations Centre of Excellence project(CZ.1.05/1.1.00/02.0070).

Author would like to thank to three anonymous reviewers for their useful comments and remarks.

References

1. GRASS Homepage: http://grass.fbk.eu/
2. GRASS-6 Raster API Manual: http://grass.osgeo.org/programming6/gisrasterlib.html
3. PostgreSQL Homepage: http://www.postgresql.org/
4. SQLite3 Homepage: http://www.sqlite.org/
5. GDAL Homepage: http://www.gdal.org/
6. PostGIS Homepage: http://postgis.refractions.net/
7. Raster3D Manual Page: http://grass.osgeo.org/manuals/html70_user/raster3D.html
8. Spearfish Data set: http://grass.fbk.eu/download/data6.php