

The new GRASS 5.1 vector architecture

Radim Blazek*, Markus Neteler*, Roberto Micarelli**

* ITC-irst, Via Sommarive, 18, Italy
tel +39 0461 314 -520 (fax -591),
E-mail: blazek@itc.it, neteler@itc.it

** E-mail: mi.ro@iol.it

Abstract

The presentation describes the new GRASS 5.1 vector library architecture. This new architecture overcomes the vector limitations of GRASS 4.x-5.0 by extending the vector support with attributes stored in external relational databases and by new 3D capabilities. Besides internal file based storage the geometry may alternatively be stored in PostGIS database. This enables users to maintain large data sets with simultaneous write access. External GIS formats such as SHAPE-files may be used directly without necessity of format conversion.

1 Background

In GIS the vector data model is used for geographic phenomena which may be represented by geometric entities (primitives) like points, lines, and areas. In GRASS 5.1 the vector data model includes the description of topology, where besides the coordinates describing the location of the points, lines, boundaries and centroids also their spatial relations are stored. In general topological GIS require a data structure, where common boundary between two adjacent areas is stored as a single line, simplifying the map maintenance.

The vector support is probably the component of GRASS which is currently undergoing the most significant changes. This paper describes the new architecture and the status of implementation.

1.1 History and motivation

The versions GRASS 4.x-5.0 provide a 2D vector library. This 2D library supports two different types of lines: area edges as well as lines which represent linear type features such as roads, or streams. Since 2001 a new vector library which is presented here is developed on top of existing 2D library. The old vector architecture shows numerous limitations. In terms of the geometry model optional storage of a third dimension is missing as well as multi-layer support. In terms of attribute management it lacks reliable attributes support. Also the option to link external vector data sources is not given. The new GRASS 5.1 vector architecture tries to overcome these limitations.

1.2 GRASS 5.1 vector architecture features

The current GRASS 5.1 vector architecture implementation covers the new 3D, multi-attribute, multi-layer vector features. Current available are:

- multi-layer – features in one vector may represent more layers and may be linked to more external tables;
- multi-attribute – attributes saved in external Relational Database Management System (RDBMS) connected through DBMI library and drivers;
- 2D and 3D vector geometry with topology;
- multi-format – external data formats supported (SHAPE-file, PostGRASS etc.);
- portability – platform independent internal format for 32bit, 64bit etc. computer architectures;
- integrated “Directed Graph Library” – to support vector network analysis (written by Roberto Micarelli, Italy);
- spatial index – based on R-tree method (under development).

These new features and a library overhaul require upgrading of all existing vector modules during migration from GRASS 5.0 to 5.1. Additionally new modules are under development (see Section 3.2).

2 GRASS 5.1 Vector architecture

The architecture comprises two major parts: the geometry model and the attribute management.

2.1 General Architecture description

The **geometry data** of vector maps are stored in an arc-node representation, consisting of curves (called arcs or lines). A line is stored as a series of x,y coordinate pairs, optionally a z coordinate is stored. The end points of a line are called nodes. Two consecutive x,y pairs define a line segment. Lines, either single, or in combination with others, may form areas. In this case they are of type *boundary* (in GRASS 5.0 also called *area edge*). Boundaries may not intersect. The vector types are shown in Figure 1.

Two different vector types are used to store vector areas: boundaries and centroids. A closed ring of boundaries defines a vector area. The end points of a vector line are called *nodes*, coordinate pairs forming the line are called *vertices*. An area in an area represents in topology an island within an area. Vector geometry can be stored in several ways internally or outside GRASS. Supported are SHAPE-files (with full access to the related dBase file), PostGRASS (based on PostgreSQL with PostGIS extension, [5], [7]) and, planned, OGR Simple Features Library ([8]). PostgreSQL is a hybrid object-oriented and relational database management system. PostGIS adds support for geographic objects to PostgreSQL. The OGR driver allows for plugging data sets in various formats into GRASS without the need of data import. These formats comprise Arc/Info-coverages, DGN, SDTS, MapInfo, GML and more. While native format and PostGRASS provide full topology, for non-topological formats such as SHAPE and data sets linked through OGR “pseudo-topology” is built.

Attribute data of vector maps may be stored in several ways. The former *dig_cats* files are no longer used, now vector attributes are stored in (external) database tables. Records in a table are linked to vector entities by field and category number. This category number (attribute ID) may be shared between vector entities as the same category can be assigned to several vector entities. To assign attribute information to area vector data, the area *centroid* is used.

All attributes are handled through a common layer between the GRASS modules (commands) and the connected RDBMS. This common layer is the DBMI library (DataBase Management Interface) which provides several drivers: PostgreSQL (planned), ODBC ([1]) and a xBase DBF driver. Other drivers may be added to the DBMI library in future. While the PostGIS extension to PostgreSQL stores vector geometry, the underlying PostgreSQL engine can also be used to manage attributes. For this purpose a PostgreSQL driver is planned. ODBC supports several RDBMS such as MySQL, Oracle, PostgreSQL and other database management systems. This concept was designed to achieve high flexibility for the system’s configuration.

Figure 2 shows the complete vector architecture. GRASS 5.1 is released under the terms of the GNU General Public License.

2.2 Geometry Model

GRASS allows users to store the geometrical component of a vector map layer in the binary vector format. It allows for access to vectors on 2 levels:

- level 1 – pure geometry and lines categories;
- level 2 – topology (information about areas, islands, connectivity etc) and level 1 functionality.

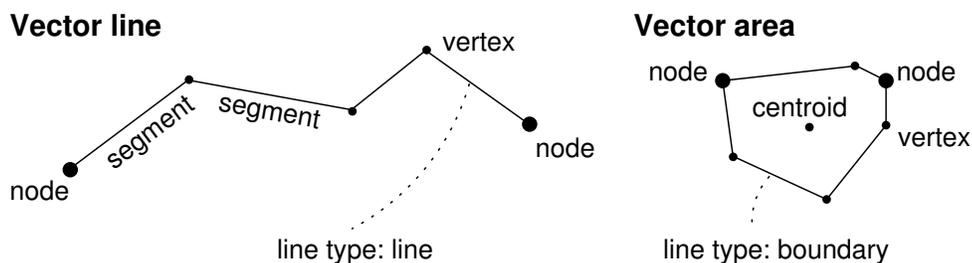


Figure 1: GRASS 5.1 vector terminology.

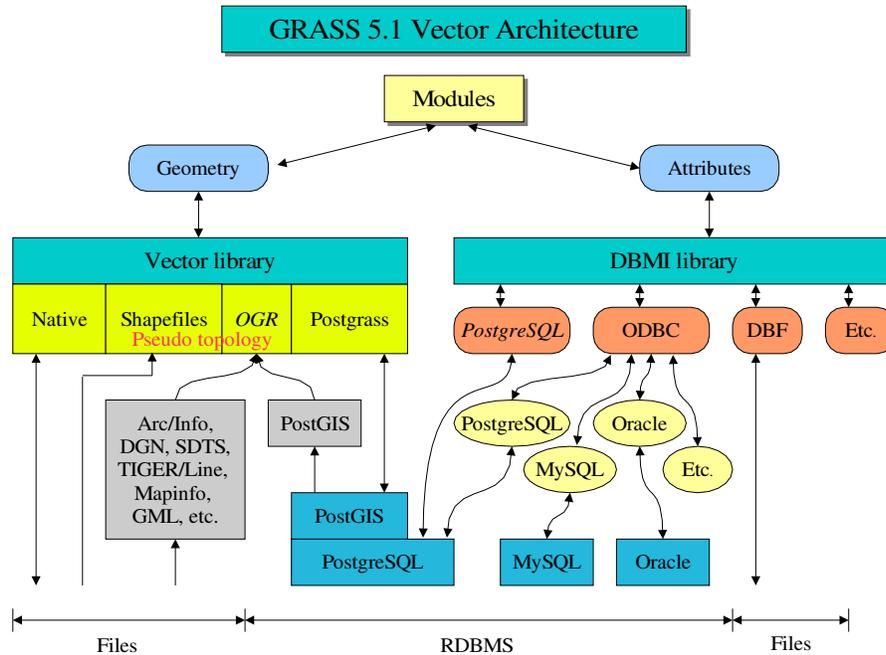


Figure 2: GRASS 5.1 vector library architecture with geometry and attribute libraries. Library parts written in *italics* are not yet implemented.

While level 1 data are stored in a *coord file* (or alternate formats), the level 2 data are stored in a *topo file*. Lines of boundary type internally carry identifiers of left and right areas, but line-type lines carry no such identifiers.

The vector library was introduced in GRASS 4.0 to hide vector files' formats and structures. However in GRASS 5.0 modules often access vector structures directly. The new implementation aims at hiding internal structures and providing access through higher-level vector functions. All modules should read data only through the `Vect_*()` functions. Planned is support for updating (writing) vectors on level 2 by vector library calls.

When dealing with vector maps, the number of lines and nodes in a map can reach a significantly high amount. To enable a GIS to manage spatial data efficiently, an index mechanism is required that allows for quickly retrieving vector entities from the database according to their spatial locations. In multidimensional spaces, special spatial indexing algorithms have to be used. The spatial index planned for GRASS 5.1 will be based on a R-tree implementation which provides efficient multidimensional spatial searches. An R-tree is a hierarchical data structure and a generalization of B-trees (binary trees) for multidimensional spaces.

Geometry data can be stored in various formats which provide different level of scalability, speed of data access and multi-user support.

2.2.1 Native format

The native format is used for vector coordinates using flat files stored in GRASS mapsets. Directory structure and file names are changed comparing to previous GRASS versions. All vector files for one vector map are stored in one directory now in the following style:

```
$MAPSET/vector/vector_name/
```

Such a directory contains these files:

- head – text file, header information (formerly part of dig/);
- coord – binary file, contains the vector data (formerly dig/ file);
- topo – binary file, topology (formerly dig_plus/).

The **head file** is an unordered list of key/value entries to describe ORGANIZATION, DIGIT DATE, DIGIT NAME, MAP NAME, MAP DATE, MAP SCALE, OTHER INFO, ZONE, and MAP THRESH. When using native vector format this *head file* is generated automatically.

The **coor file** consists of a head and a body part. The head part stores information about the format version, byte order, number of spatial dimensions. The body part contains the vector line records. These records carry information about the line type, status of a line (alive or dead, are categories attached to this line), the number of attached categories, a field number as category identifier to distinguish between more categories being appended to one line, the category value, the number of vertices and the coordinates of the vertices.

The **topo file** contains topological information of the map including bounding boxes.

2.2.2 PostGRASS

An alternate option to store vector geometry is to use the new PostGRASS approach. The use of an RDBMS allows for simultaneous editing by several users and provides further advantages as only available through RDBMS usage. Here geometry information is saved in an external PostGIS database in form of points and lines, topology is maintained by GRASS and saved as GRASS topology file. PostGRASS uses PostGIS which is an extension to PostgreSQL to allow for storage of geographic objects into PostgreSQL tables. PostGIS is a spatial database much like ESRI's *SDE* and Oracle's *OracleSpatial*. PostgreSQL and PostGIS are Free Software (BSD License and GNU LGPL resp.). PostGRASS allows for linking vector maps to one or several database stored on one or several hosts.

PostGRASS was developed by a group of students at the RCOST Institute at University of Sannio (Benevento, Italy, [6]), supervised by G. Antoniol. The functionality of PostGRASS in GRASS 5.1 is the same as for native format. That means that the user can read and write both geometry and categories. Everything is hidden in vector library and the GRASS modules do not know about any difference.

Regular (native format) vector data are saved in the directory `$MAPSET/vector/vector_name/` with *coor*, *head*, and *topo* files. For PostGRASS the directory must contain *head* and *frmt* files. These files for PostGRASS vector support must currently be written by hand before the user can start working with vector maps.

- *head* – text file, header information (formerly part of *dig/*)
- *frmt* – text file, contains informations where to find the vector data.

The **head file** format is as for the native format (see above Section 2.2.1).

The **frmt file** format is similar to head file (i.e. pairs of key and value):

```

FORMAT: postgis # required, always postgis for PostGIS
HOST:          # optional, name of computer PostgreSQL is running on,
               # if not specified then localhost is used
PORT:         # optional, port the PostgreSQL server is listening on
DATABASE:    # required, database name
USER:        # optional, user name
PASSWORD:    # optional, user password
GEOM_TABLE:  # required, name of geometry table
CAT_TABLE:   # required, name of category table
GEOM_ID:     # optional, name of unique identifier column in
               # geometry table, default: id
GEOM_TYPE:   # optional, name of type column in geometry table,
               # default: type
GEOM_GEOM:   # optional, name of geometry column in geometry table,
               # default: geom
CAT_ID:      # optional, name of id column in category table,
               # default: id
CAT_FIELD:   # optional, name of field column in category table,
               # default: field
CAT_CAT:     # optional, name of category column in category table,
               # default: cat

```

Example for PostGRASS *frmt file*:

```

FORMAT: postgis
HOST: 123.456.0.1
PORT: 5432

```

```

DATABASE: mcat
USER: eleonora
PASSWORD: password
GEOM_TABLE: geomtable
CAT_TABLE: cattable
GEOM_ID: geom_id
GEOM_TYPE: geom_type
GEOM_GEOM: geom_geom
CAT_ID: cat_id
CAT_FIELD: cat_field
CAT_CAT: cat_cat

```

Database structure and storage system

Primitive's geometry (point, line, boundary, centroids will be saved in one column (geometry PostGIS type), each primitive in one row. Types are distinguished by codes in one column (integer). Several categories (cats) may be attached to each primitive. These categories are distinguished by field number. Categories are linking to external database table containing further attributes. That means that one primitive may be linked to several tables. In PostGRASS categories are saved in a table which are separated from the table containing geometry information. This approach allows for creating views for joined geometry and attribute tables. Such view may be read by other clients which are not able to handle links to several tables.

2.2.3 SHAPE-file support

Since the ESRI SHAPE vector format is well established in the GIS world (although it shows several disadvantages), direct support was implemented in the library. It is possible to display and query SHAPE-files in GRASS 5.1 without further need of data conversion. Attributes saved in DBF file may be read through the DBMI library. For areas pseudo topology will be built, instead of the standard GRASS topology. For registration inside a GRASS LOCATION a *frmt file* is required to be generated:

- *frmt* – text file, contains informations where to find the vector data

Format of the *frmt file*:

```

FORMAT: shape          # format
SHAPE:                 # path to SHAPE files
CAT_COLUMN:           # optional, column name with integer ID which will
                      # be used as category number

```

Example for a *frmt file* (areas.shp):

```

FORMAT: shape
SHAPE: /home/radim/gdata/g51test/shp/areas
CAT_COLUMN: CAT_ID

```

2.2.4 OGR (planned)

GDAL, which is already used in GRASS 5.0, is a translator library [9] for raster geospatial data formats that is released under an Open Source license. As a library, it presents a single abstract data model to the calling application for all supported formats. The related OGR library (which lives within the GDAL source tree) provides similar capabilities for simple features vector data. OGR is intended to provide an OpenGIS Simple Features inspired terminology ([3]) and view of data sets.

The planned OGR support will extend the capabilities of direct vector format read support to Arc/Info Binary Coverage, GML (Geography Markup Language), Microstation DGN, ESRI SHAPE-file, UK NTF, SDTS, U.S. Census TIGER/Line, IHO S-57 (ENC), MapInfo File, OGDI Vectors, and PostgreSQL.

Since OGR is written in C++, first a set of C wrapper functions will have to be developed before the OGR support can be implemented. Through OGR it will be possible to display and query the various formats in GRASS 5.1 without further need of data conversion. Categories will be read from the original data sets, however, additional attributes must be linked through the DBMI library (this may be considered as strength or weakness). For areas pseudo topology is built, instead of the standard GRASS topology.

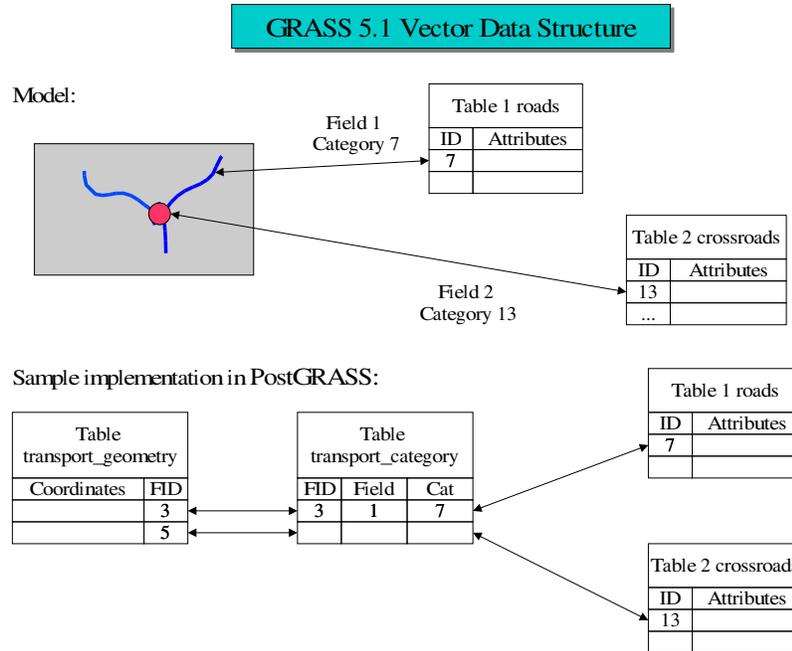


Figure 3: Attribute management implemented in GRASS 5.1 vector library architecture.

2.3 Attributes Model

The attribute management was completely changed in GRASS 5.1 to achieve multi-attribute capabilities managed within external databases. The former `dig_cats` files are not used any more and all vector attributes are stored in external database. Connection with database is done through DBMI library (DataBase Management Interface) with its integrated drivers. At time of this writing drivers for ODBC and DBF files are available. A PostgreSQL driver is planned (note that PostgreSQL can already be accessed through ODBC).

In general records in table are linked to vector entities by field number and category number (compare Figure 3). The field number identifies a database table and the category number identifies the record record. I.e. for unique combination `mapset + map + field + category` exists one unique combination `driver + database + table + row`. Each element may have none, one or more categories (cats). More cats are distinguished by field number (field). The flexibility of this approach even supports the extreme case, that one vector entity may be linked to attributes in different tables in different databases connected by different drivers at the same time.

The DBMI library contains drivers such as DBF, ODBC, and PostgreSQL (planned). The functionality of the database support varies with the capabilities of the underlying RDBMS. Main features are multi-attributes support for various data types, also multiple tables may optionally be linked to one or many vector entity/ies. SQL (Structured Query Language) is used for all drivers.

2.3.1 DBF driver

A simple DBMI driver is the xBase DBF driver. It may become the default driver and is also used for SHAPE-file support. At time this driver supports a subset of SQL statements, currently are supported:

```
SELECT FROM WHERE
INSERT INTO
UPDATE WHERE
DELETE FROM WHERE
CREATE TABLE
DROP TABLE
```

As “where” conditions are supported:

`=, <=, >=, <, >, <>, AND`

This driver may be improved in future. Modifications are needed as the current implementation requires a full file rewrite even when a single entry should be modified in a DBF table.

2.3.2 ODBC driver

The ODBC driver enables GRASS to connect to PostgreSQL, MySQL, Oracle and other external database management systems. In general the attribute tables written by GRASS 5.1 through DBMI/ODBC may also be accessed externally from other software which is connected to a RDBMS.

2.3.3 PostgreSQL driver (planned)

The planned PostgreSQL driver enables direct access to PostgreSQL databases without the necessity of ODBC. This should speed up the database connection and simplify the installation.

2.3.4 Definition of a database connection

The definition of a database connection is done in a *DB file* within a mapset:

- DB – text file, contains information which driver, database, table and key column should be used.

Each row in this DB file contains names separated by spaces in following order (names/rows in squared brackets are optional):

```
map[@mapset] field table key database driver
[map[@mapset] field table [key [database [driver]]]]
[...]
```

For each mapset + map + field must be defined table, key column, database, and driver, this definition is written then into a text file `$MAPSET/DB`. Definitions in current mapset may override definitions from DB file in other mapset if a mapset is specified along a map name. Wildcards (`*` and `?`) may be used in map and mapset names. The variables `$GISDBASE`, `$LOCATION`, `$MAPSET`, `$MAP`, `$FIELD` may be used in table, key, database and driver names. Note that `$MAPSET` is not the current mapset but the mapset of the map the rule is defined for.

Examples for DB files

The following example of a *DB file* for SHAPE file links the map “areas.shp” with field 1 to table “areas.dbf” (as the DBF driver is used) stored in directory “`$GISDBASE/$LOCATION/shp`” with key column “cat_id”:

```
areas.shp 1 areas cat_id $GISDBASE/$LOCATION/shp dbf
```

Another, more general example for a DB file using the DBF driver may look as follows:

```
* 1 mytable id $GISDBASE/$LOCATION/$MAPSET/vector/$MAP dbf
```

This definition says that all vector maps (“*”) are linked to tables “mytable.dbf” by category of field 1 to the key column “id”. The DBF table files, all with names `mytable.dbf` are saved within the related vector subdirectories of each map. The *database* entry is the full path to the directory containing the individual `.dbf` files.

A third example for a DB file contains several entries:

```
water* 1 rivers id /home/grass/dbf dbf
water* 2 lakes lakeid /home/guser/mydb
trans 1 roads key basedb odbc
trans 5 rails
```

These definitions defines more fields for one map i.e. in one map may be more features linked to more tables. Definition on first 2 rows are applied for example on maps `water1`, `water2`, ... so that more maps may share one table. Also DBF and ODBC drivers are used in parallel.

As last example a DB file which overwrites a definition from PERMANENT (as possible for any user outside the PERMANENT mapset):

```
water@PERMANENT 1 myrivers id /home/guser/mydbf dbf
```

This definition overwrites definition saved in `PERMANENT/DB` and links map from PERMANENT mapset to a user’s table.

2.4 External Directed Graph Library hosted by GRASS

The Directed Graph Library or DGLib ([2]) provides functionality for vector network analysis. This library released under GPL is hosted by the GRASS project (in the CVS server). As stand-alone library it may also be used by other software project.

A graph is a system of logical connections between a collection of objects called vertices. Graphs are usually represented by a picture, so that each vertex is shown as a point, with the connections shown as line segments. These vertices are also commonly referred to as nodes, edges referred to as arcs. A directed graph (digraph) consists of a finite set of vertices, and a finite set of edges, where an edge is an ordered pair of vertices. A directed graph has the property that edges have a direction, this is the reason for defining an edge as an *ordered* pair of vertices often referred to as the *head* and the *tail* of the edge.

The original design idea behind DGLib was to support middle sized graphs in RAM with a near-static structure that doesn't need to be dynamically modified by the user program; ability to read graphs from input streams and process them with no need to rebuild internal trees. A representation has been defined, where graph data is stored in 32bit word arrays and each element pointer is converted to a relative offset. This representation is *serializable* from/to input/output streams and allows fast load-and-use processing. Graphs need to be de-serialized in order to be edited. In further refactorings the library has evolved to support dynamic changes and state-independent algorithm (algorithms can be run on both serializable or editable graphs).

DGLib defines a serializable graph as being in FLAT state and a editable graph as being in TREE state. The implementation makes intensive use of *libavl* ([4]) AVL data structures to support TREE state.

So far DGLib defines three different graph versions, version 1 supports directed graph with a weak concept of the edge, it can support many applications where one doesn't need to know about the input edges of a node (in-degree) and where there is no requirement to directly retrieve edges by their identifier but only by head/tail combinations. Version 2 adds in-degree support and a true edge addressing, yet supporting directed graph. Version 3 uses the same internal representation of version 2 but activates code branches to support undirected graphs.

The DGLib user can control a number of static features and can attach a arbitrary amount of data to each node (node-attributes) and each edge (edge-attributes). Attributes are not considered to be part of the graph structure and can be edited also when the graph is in FLAT state.

Graph traversal in neither recursive nor hook (callback) based, but built on the use of *traversers* for nodes and edges. By default, traversal is ordered by node and edge identifiers but can optionally be ordered by other means. For example, it is often useful to visit edges on a *weight order* basis (greedy algorithm), this is possible via *prioritizers* that are activated by setting specific *graph options*.

Both preemptive (blocking) and non-preemptive (non-blocking/multiplexed) I/O is supported, although GRASS does not actually use graph storage it may be easily required by any other library user. Thread safety is so far ensured by a data separation design that keeps all application context states into stack containers, whose life cycle is controlled by the user program. Each graph is a separate container and two or more graphs never conflict. In addition algorithms (ie. shortest path) can safely share the same graph, while concurrent editing on the same graph is unsafe.

As DGLib is under development, only a bunch of polynomial time algorithms have been implemented, and the basic structure is being stressed to be a mature core to possibly time wasting computations. Current algorithms are: *shortest path*, *depth spanning*, and *minimum spanning*. Spanning algorithms silently behave as arborescences when applied to directed graphs. A clip callback function, optionally supplied by the user, comes called by the library while traversing the graph in order to alter default algorithm behavior (i.e. user can control access to specific graph segments while computing shortest path).

The Directed Graph Library library provides functionality to assign costs to lines and/or nodes. That means that costs can be accumulated while traveling along polylines. The user can assign individual costs to all lines and/or nodes of a vector map and later calculate shortest path connections based on the accumulated costs. Applications are transport analysis, connectivity and more.

3 GRASS 5.1 Vector Modules

All vector modules (commands) from GRASS 5.0 have to undergo an overhaul before integration into the new code repository. This cleanup will check and modify the functions used for vector data processing, address portability/ANSI-C compliance problems as well as standardize the module parameters and flags.

At time of this writing some modules from previous releases were upgraded and also some new modules introduced into the system. The `db.*` modules known from GRASS 5.0 are functional for the 5.1 attributes management.

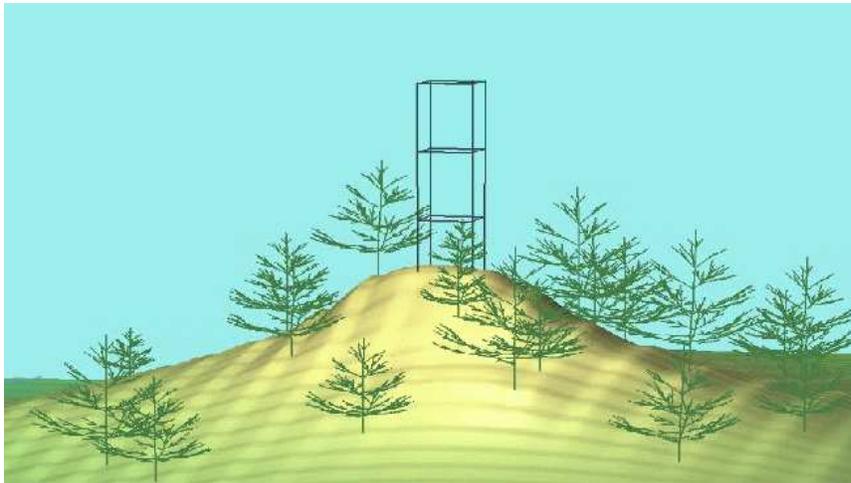


Figure 4: GRASS 3D vector maps (tower and trees) rendered with NVIZ.

3.1 Upgraded vector related modules

Upgraded modules well known from previous releases are `d.vect` and `d.what.vect` (display commands). Partly rewritten are `v.in.ascii` as well as `v.out.ascii`. Also the visualization tool NVIZ was modified to display 3D vectors (see Figure 4). Other vector modules will be migrated to GRASS 5.1 in future.

3.2 New modules

Several new modules have already been developed:

- `d.m`: GRASS 5.1 display manager for displaying raster and vector maps. It allows to save map selections along with chosen vector color, optional spatial queries, scripts into a project file.
- `d.path`: This module searches the shortest path for selected starting and ending nodes in a vector map. The module uses the Direct Graph Library (see Section 2.4) to calculate accumulated costs when traveling along the vector lines. These costs can be either the line lengths, or costs attached as attributes to both arcs and nodes (stored into a database table).
- `v.category`: This module allows for attaching, deleting or report vector categories.
- `v.convert`: This tool converts between GRASS vector versions (4.x and 5.1 formats).

4 The future

To improve the vector geometry support, write access on level 2 (topology) should be developed. To improve the speed of geometry data access a spatial index for vector geometry should be implemented. Future development of GRASS 5.1 may comprise the implementation of the OGR support for vector geometry.

It is suggested to migrate the GRASS 5.0 sites management to the vector format. This will reduce the software maintenance efforts as well as improve the sites attributes management capabilities.

Next planned development for the DGLib will focus on Salesman Problem, Postman Problem and Steiner Tree/Arborescence.

Forms support should be developed to display graphically map query results.

For the DBMI library an improvement of the DBF driver is required since this driver may become the default driver. Also the implementation of a PostgreSQL driver is desired.

The DBMI library may be extended to support storage of images, films, or audio data as attributes, using BLOBs (Binary Large Objects). Through that users are enabled to attach e.g. video sequences to a certain point in the map.

References

- [1] Harvey, P. et al., 2002. unixODBC software. <http://www.unixodbc.org/>
- [2] Micarelli, R., 2002. Directed Graph Library. <http://grass.itc.it/dglib/>
- [3] OpenGIS Consortium, 1999. OpenGIS Simple Features Specifications.
<http://www.opengis.org>
- [4] Pfaff, B., 2002. GNU libavl, binary search trees and balanced trees library.
<http://www.msu.edu/~pfaffben/avl/>
- [5] PostgreSQL Development Team, 2002. PostgreSQL Database Management System.
<http://www.postgresql.org/>
- [6] RCOST Institute at University of Sannio (Benevento, Italy), 2002. PostGRASS driver.
<http://www.rcost.unisannio.it/antoniol/students/PostGrass/>
- [7] Refrations Research Inc, 2002. PostGIS: Geographic objects for PostgreSQL.
<http://postgis.refrations.net/>
- [8] Warmerdam, F., 2002. OGR Simple Features Library Software.
<http://gdal.velocet.ca/projects/opengis/>
- [9] Warmerdam, F., 2002. GDAL Geospatial Data Abstraction Library.
<http://www.remotesensing.org/gdal/>