The GEO++ System: an Extensible GIS

Tom Vijlbrief
TNO Institute for Perception,
P.O. Box 23, 3769 ZG Soesterberg, The Netherlands.
Email: tom@izf.tno.nl
and
Peter van Oosterom
TNO Physics and Electronics Laboratory,
P.O. Box 96864, 2509 JG The Hague, The Netherlands.
Email: oosterom@fel.tno.nl

In this paper we present a classification of the architectures of Geographic Information Systems (GIS). Most commercial GISs are closed. This means that if certain functionality is not available, it is impossible for the users to extend or modify the system for their own purpose. We then present a solution based on the extensible database management system "Postgres", in which new data types and operators may be defined. The resulting extensible GIS is called *GEO*$^{++}$. We illustrate this powerful capability with two examples.

1 Introduction

Most commercial Geographic Information Systems (GISs) are based on a relational DataBase Management System (DBMS), such as Oracle or Ingres. One obvious drawback of the standard DBMSs is that they cannot manipulate geographic data. That is, there are no geometric attribute types (e.g., point, polylines, polygons) and operators (e.g., distance, intersection, circumference, area).

This paper is structured as follows. In Section 2 we give a classification of GIS architectures. The subsequent section describes the geometric extensions to the open DBMS *Postgres* (Stonebraker, Rowe, & Hirohama, 1990) that we implemented. Section 4 enumerates the basic capabilities of our Postgres GIS frontend *GEO*$^{++}$. The implementation of the system has been described in an earlier paper (van Oosterom & Vijlbrief, 1991). The real power of GEO^{++} is demonstrated in Section 5, in which the system is extended with user-defined types.

2 GIS Architectures

The standard DBMSs do not provide basic geographic data types so that it is impossible to store geographic data in a natural manner and to pose queries such as: "Select all towns with more than 10,000 inhabitants that are located within 3 kilometers from a lake." These systems also lack multi-dimensional access methods (or index mechanisms), which are required because geographic data sets are often very large.

Three different types of system architectures have been suggested to overcome these problems: *dual architecture*, *layered architecture*, and *integrated architecture*. The term dual architecture was first introduced in (van Oosterom & Vijlbrief, 1991). A similar classification was described in (Bennis *et al.*, 1990, 1991). They used the terms *partial DBMS architecture*, *shell architecture*, and *full DBMS architecture* for respectively dual architecture, layered architecture, and integrated architecture. In the next subsections, the pros and cons of these architectures are discussed. Note that it is not always easy to classify a specific system. For example *Smallworld GIS* (Chance, Newell, & Theriault, 1990) possesses characteristics of both the layered and the integrated architecture.

2.1 Dual Architecture

The most common type of commercial GIS architecture is the dual one. Dual architecture GISs have a separate subsystem for storing and retrieving spatial data, whilst thematic information is stored in a relational DBMS. This dual architecture is not conceptually elegant and also reduces the performance. An object that has both a thematic and a spatial component, has parts in both subsystems that are linked by a *common identifier*. In order to retrieve an object, the two subsystems have to be queried and the answer has to be composed. Figure 1 illustrates this dual GIS architecture.

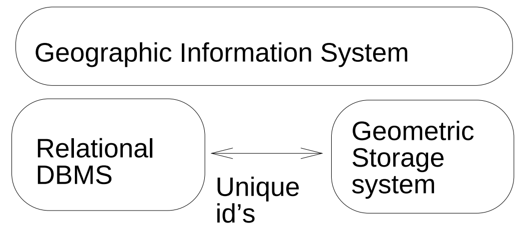

Figure 1: Dual GIS Architecture

Typical examples of GISs with dual architecture are: ARC/INFO (ESRI, 1989; Morehouse, 1989), MGE (Intergraph, 1990), SICAD (Schilcher, 1985; Siemens Data Systems Division, 1987), and ARGIS 4GE (Unisys Corporation, 1989).

The advantage of the dual architecture is that it is partly based on a standard DBMS and that the storage and retrieval of spatial data can be efficient. However, this method has some severe drawbacks. The existence of two storage subsystems implies that query optimization is not possible to the full extent. A relational DBMS offers transactions that are atomic, durable, and serializable. Storing data outside the relational DBMS can result in losing the transaction semantics, because the two storage managers each have their own locking protocol. The final drawback of the dual architecture is that integrity constraints can be violated. For example, an entity can still exist in the spatial storage subsystem while it has been deleted from the relational DBMS.

2.2 Layered Architecture

Most drawbacks of the dual architecture are caused by the fact that there are two storage managers each with their own responsibilities. It is possible to store the spatial data in the pure relational data model (van Oosterom, van Hekken, &

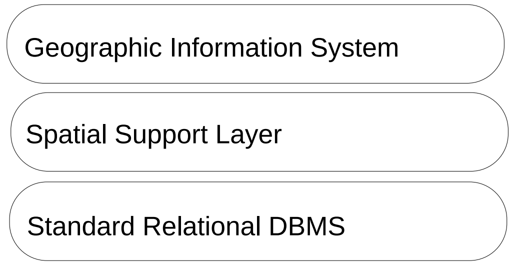

Figure 2: Layered GIS Architecture

Woestenburg, 1989; van Roessel, 1987). This implies that the support for transaction semantics and integrity constraints is restored. However, in order to fit the data into the relational model, the coherent geographic entities have to be broken into multiple parts, which are stored in separate tables. Retrieving the original geographic entities has to be done by joins of relations, making the system slower and more difficult to use.

The user may be freed from formulating difficult queries by some "intelligent" translations in the layer on top of the standard relational database. This layer translates geographic queries (in "GeoSQL"; Geographic Structured Query Language) into standard SQL-queries and it may also implement spatial indexes. These indexes are implemented by means of auxiliary relations that contain the required index data. This makes spatial access faster, but the queries become even more complicated, because they have to use the auxiliary relations. This indirect implementation of an access method is less efficient than a direct implementation in the DBMS kernel. The layered GIS architecture is depicted in figure 2. System 9 (Pedersen & Spooner, 1988) from Prime, GEOVIEW (Waugh & Healey, 1987) from the University of Edinburgh, and SIRO-DBMS (Abel, 1989) from CSIRO Australia are characteristic examples of layered architecture systems.

2.3 Integrated Architecture

The inconvenient and inefficient mapping from the user data model to the relational tables can be avoided if more attribute types and access methods are added to the system. This solution is chosen in the integrated GIS architecture. In contrast to the other two types of architectures, this architecture cannot be based on a standard relational DBMS. It requires an extensible (and often object-oriented) DBMS. This is illustrated in figure 3, where the spatial extension is completely embedded in the DBMS. Users may extend the DBMS with their own basic abstract data types (ADTs). An obvious drawback of this approach is that users have to implement their own types within the DBMS environment, which may be quite complicated. However, once this task has been performed, the advantages of this approach become clear. The implementation of the data model is easier, because the appropriate geometric types are available now. The formulation of spatial queries is directly supported in the extensible query language by means of added spatial operators such as distance, area, and intersection.

Perhaps the most important advantage is the good performance of these systems. The direct implementation of data types in the kernel of the DBMS is very efficient. Another facility supporting system performance, is that one may provide own spatial access methods. The development of integrated GIS architectures depends on the availability of open DBMSs. Characteristic examples of integrated GIS architectures are TIGRIS (Herring, 1987) from Intergraph and the research oriented system GéoTropics (Bennis *et al.*, 1990) from the

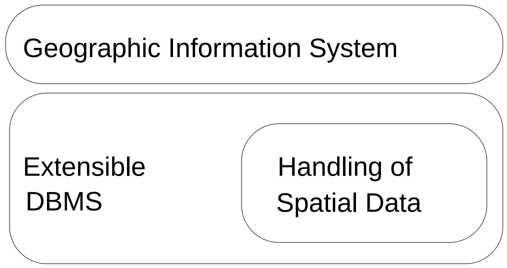

Figure 3: Integrated GIS Architecture

University of Paris VI and IGN France. Our own system, GEO++ (van Oosterom & Vijlbrief, 1991), also possesses an integrated architecture. The system-component "Handling of Spatial Data" (see figure 3) is implemented in the open Postgres environment.

3 Geometric Data Types

The first step in the implementation of a GIS with an integrated architecture is extending the DBMS with geometric data types and operators. In our case, Postgres itself has some geometric capabilities. It comes with the R-tree (Guttman, 1984) spatial index structure and with the following types: `point`, `lseg`, `path`, and `box`. The type `lseg` implements a single line segment. Polylines and polygons may be represented by `path`, which is a variable length array of `lseg`. The special case of a two dimensional axes-parallel rectangle is represented by the type `box`. Some useful functions and operators are provided (test for overlapping boxes, test if a point lies inside a box, the distance between two points), but more are required for a more complete geographic extension of the DBMS.

We have implemented both 2- and 3-dimensional geometric data types. Therefore, the type name ends with a number that indicates the dimension. The 2D data types are: `POINT2`, `POLYLINE2`, and `POLYGON2`. Instead of using the Postgres system type `path`, we want to reflect the *semantic* difference between polyline and polygon by providing separate data types for them. These types have to be supported with more geometric functions. Function names reflect the purpose, the dimension, and the operand types. For example, the function `Equal2PntPnt` checks if two 2-dimensional points are equal. The functions are classified into groups, each illustrated with a few examples (operator symbols are indicated within square brackets):

Return boolean values:
`Equal2PntPnt [==]`,
`On2PntPln [-->]`,
`Contain2PgnPgn [-->]` (Note the operator overloading), and
`Overlap2PlnPgn [&&]`.

Return atomic geometric objects:
`GravCenter2Pgn [@]`,
`MinBoundRec2Pgn [|=|]`, and
`ConvexHull2Pnts` [~] (Pnts indicate a variable length array of `POINT2`s).

Return complex geometric objects:
`Inter2PgnPgn [^]`,
`Voronoi2Pnts [<|]`, and
`Delaunay2Pnts [|>]` as described in (Preparata & Shamos, 1985).

Return scalar values:
`Distance2PntPnt [<->]`,
`MinDist2PgnPgn [<->]`,
`Length2Pln [--]`, and
`Area2Pgn [*]`.

4 GEO++ Capabilities

GEO^{++} is a general-purpose GIS front-end for Postgres. The system has a "direct-manipulation user-interface," it allows us to implement real-world GIS systems, and to experiment with the user-interface and various data structures and storage techniques. Some of the expected applications are: electronic nautical charts and various Command and Control (C2) systems. The current prototype system is written in C++ (Stroustrup, 1986) and uses the ET++ (Weinand, Gamma, & Marty, 1988, 1989) class library. The architecture is shown in figure 4. Most of the components will be described in this and the following section.

The screendump (figure 5) shows a part of the user-interface of GEO++.

The GEO++ system has the following basic capabilities:

- Viewing of Postgres relations[1].
- Addition or editing of Postgres tuples in viewed relations.
- Graphically composing the *where clause* restriction for the viewed relation in a tree format that represents the structure of this expression. The graphical editor inspects the Postgres Meta Relations (system catalogs) which contain information about the existing operators. In this way, users have access to all defined operators and types, including types they have defined themselves.
- The composer checks the types of arguments to operators and does not allow clauses that would result in syntactical errors. Composed query-trees can be saved and reloaded for different relations.
- GEO++ also has its own Meta Relations. For instance, a relation that describes the Regular Expression that each entered Postquel type constant must match. This is used by GEO++ to ensure the construction of correct *where clauses*.
- Displaying the result of queries on a geographical map. Point attributes are shown as colored icons. These icons and their colors are also specified in GEO++ Meta Relations or are taken from the displayed relation. Polylines and polygon (the Postgres `path` type) can also be shown. The width (polylines) and colors are specified in the same way as for the Point attributes.
- Labeling geographic entities on the map. The labels are formed by the attribute name (optional) and the current value.
- The users can shuffle the drawing orders of the displayed queries or temporary hide them. Also parameters of the *where clauses* can easily be modified.
- Picking of displayed objects. This shows the corresponding Postgres tuple.
- Editing and creating `point` and `path` attributes on the displayed map.
- Zooming and panning of the displayed map and of a second map display which shows the context of the main map. This is supported by a spatial access method: the R-tree.

[1] Some object-oriented terminology: a relation is an object class, a tuple is an object instance, and an attribute is an instance variable.

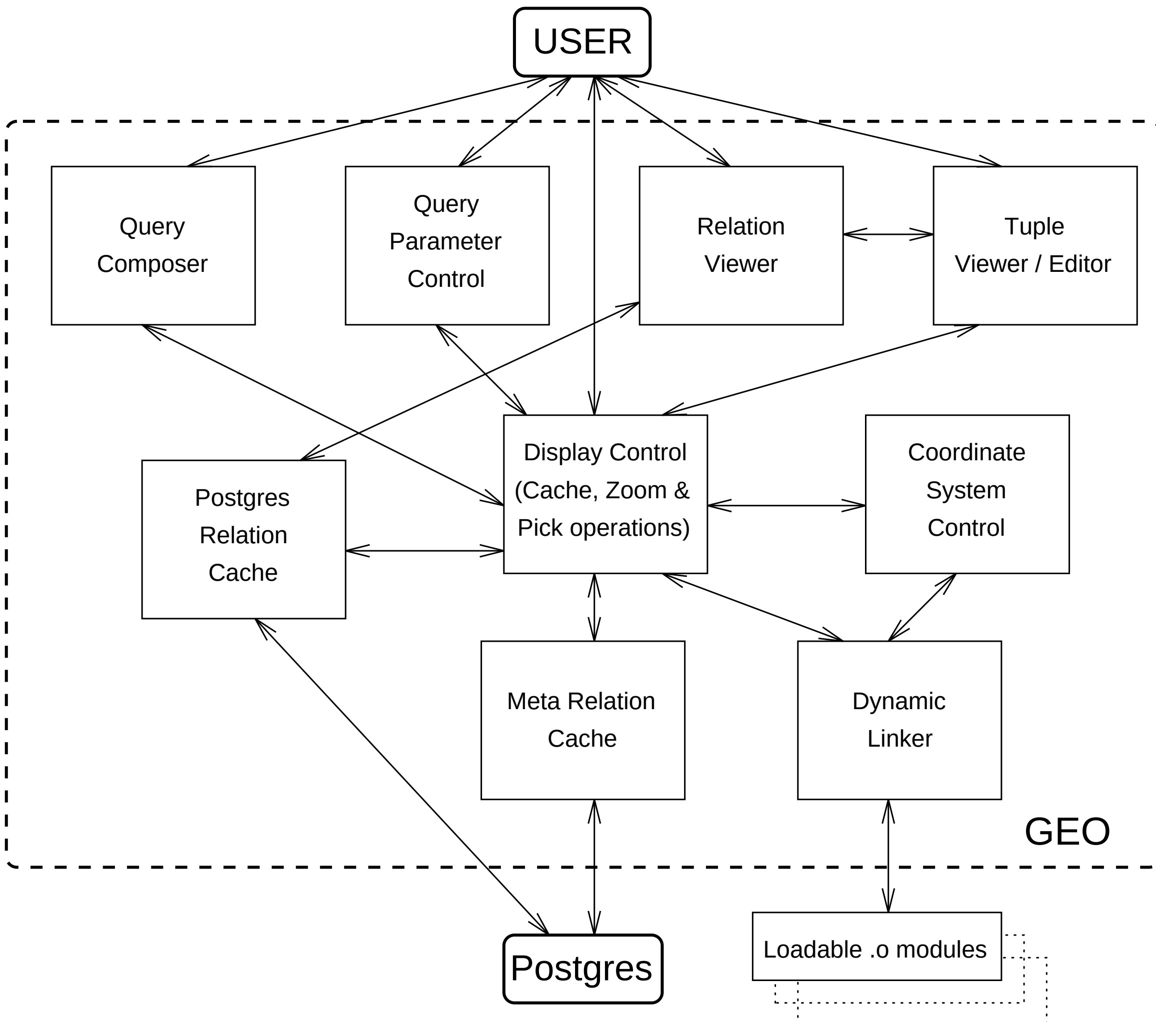

Figure 4: The GEO++ functional decomposition

- Annotating a map with text, lines, polylines, symbols, etc., like a normal drawing program. Multiple annotations (from different users) may be displayed simultaneously, thereby allowing communication about map contents between two or more remote users.

- Automatic redraw of changed relations by means of Postgres Asynchronous rules (triggers). This capability allows the creation of dynamic map displays with moving objects.

5 Extending GEO++

Just as Postgres is extensible by defining new abstract data types and coding their implementation in the programming language C, GEO++ is extensible by defining new `QueryShapes` and coding them in C++. The advantage of C++ is that one can inherit most of the functionality of the parent `QueryShape`.

The new `QueryShapes` must implement a `draw` method and a `distance` method that are used for drawing the `QueryShapes` on the map and for the `pick` operation. Typically, a GEO++

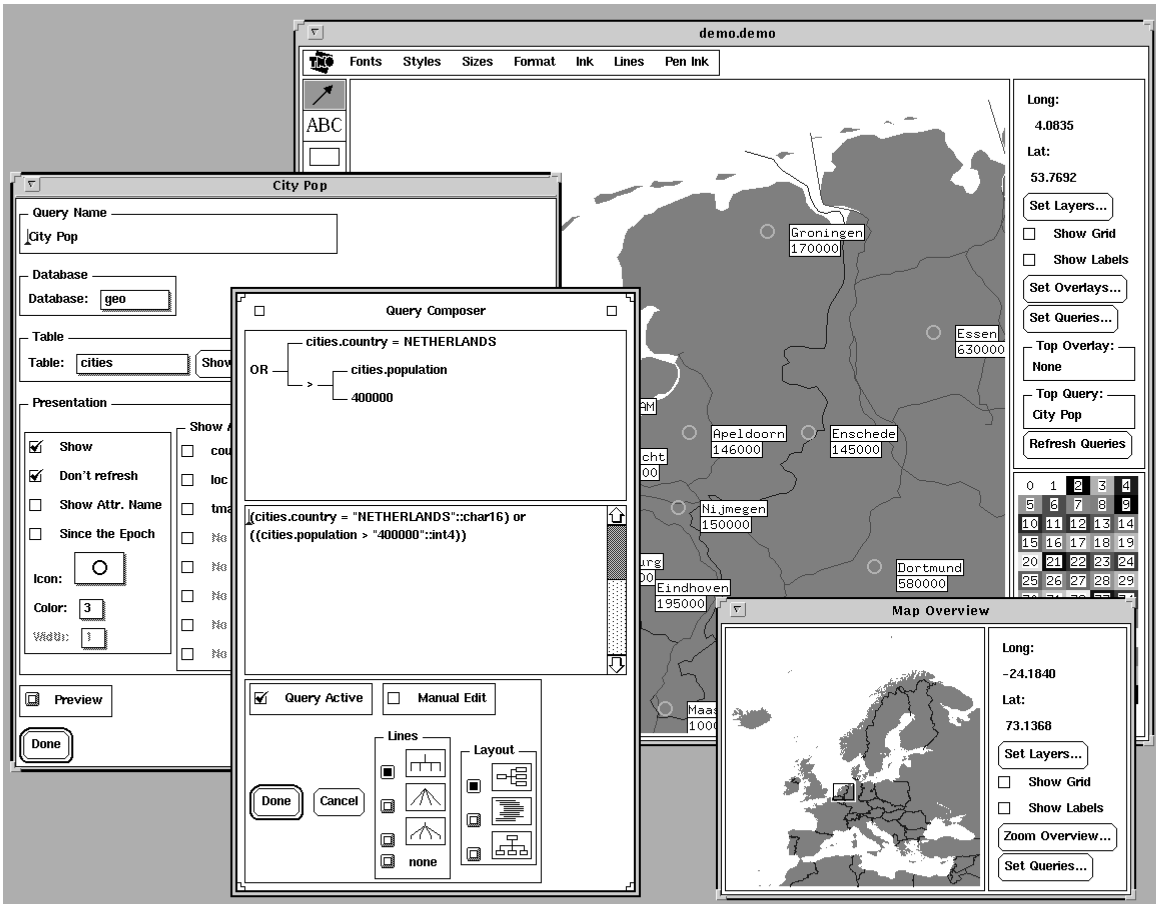

Figure 5: The GEO++ user interface

`QueryShape` matches a user-defined geometric type in Postgres. The geometric operations are implemented in Postgres and the visualization is implemented by new user-defined GEO++ `QueryShapes`.

By means of extending the GEO++ system one has almost unlimited flexibility in the way data is stored in Postgres and displayed with GEO++. Below, we show how `QueryShapes` that display data of our own set of Geometric types (Section 3) are entered into a GEO++ meta relation. This is required because GEO++ was originally based on the standard geometric types of Postgres.

We have implemented for example, a `Pln2Shape`, a `Pgn2Shape` and a `Pnt2Shape` that visualize the corresponding new Postgres geometric types. GEO++ consults the Meta Relation `geo_dyninfo` when displaying relations. If we have a relation `bordersmap` that contains instances of our `Pln2Shape` example then we would enter this data with the following Postquel command:

```
append geo_dyninfo (
  relname="bordersmap",
  dynfunc="make_Pln2_shape",
  dynfile="MapShape.o",
  relattr="plndata,color,width",
  bboxattr="bbox")
```

This specifies that

- The tuples in the relation `bordersmap` are displayed by calling the C++ function `make_Pln2_shape` defined in

MapShape.o. GEO++ will attach the .o file to its own executing image at runtime. This is implemented by means of the GNU dynamic linker package `dld` (Ho & Olsson, 1991). Details on the actual C++ implementation of QueryShapes are not within the scope of this paper. You should study the GEO++ source distribution for implementation details.

- The function is called with the attributes `plndata`, `color` and `width` as arguments.
- The objects in `bordersmap` have a bounding box that is stored in the attribute `bbox`.

5.1 Raster Capabilities

We have extended the current GEO++ system (originally an vector GIS) with the capability to display raster data. This clearly proves the "open" character of GEO++. We implemented a `QueryShape` that displays raster type data, thereby creating a hybrid system. Raster data is stored in variable size clusters (e.g., 32 x 32 or 64 x 64) that store rectangular parts of the raster type data. This method takes full advantage of the R-tree access method. The values at the edges of a cluster could be duplicated from the neighbor cluster of each edge. This allows efficient implementation of operations that need access to the neighbors of each raster cell.

A separate program converts a raster data source file (e.g. satellite imagery or a scanned paper map) to tuples for a Postgres relation. This relation should have at least the following attributes:

sizex (int2) The x-dimension of the raster data cluster (e.g. 32 or 64).

sizey (int2) The y-dimension of the raster data cluster.

area (box) The rectangle covered by this cluster on the earth surface.

data (bytea) The byte array (with e.g. 32 x 32 entries) containing the actual values (e.g. color indices) of the raster cells.

Figure 6 shows a background scanned road map (the center area of the Netherlands) on which a road network has been digitised.

The dark wide line shows the result of a fastest path analyses. This fastest path is computed by a separate program (an Unix shell script which is invoked from a customisable GEO++ user interface menu). The script is passed the start and end points of the desired path as indicated by the user and it retrieves the road fragments start/end points, their (adjusted) length and their `oid` (Postgres Object Identifier) from the Postgres relations and feeds this information to a (C++) program which computes the shortest path. The length of the roads in the network are adjusted in order to compensate for the difference in average driving speed on highways (90 km/h) and non-highways (60 km/h); see figure 7. The shortest path computing script inserts the oids of the edges composing the fastest path into a Postgres relation (PathOids). These oids can be used in a view definition which is used to display the path with GEO++:

```
define view FastPath (roads.all)
where roads.oid = PathOids.the_oid
```

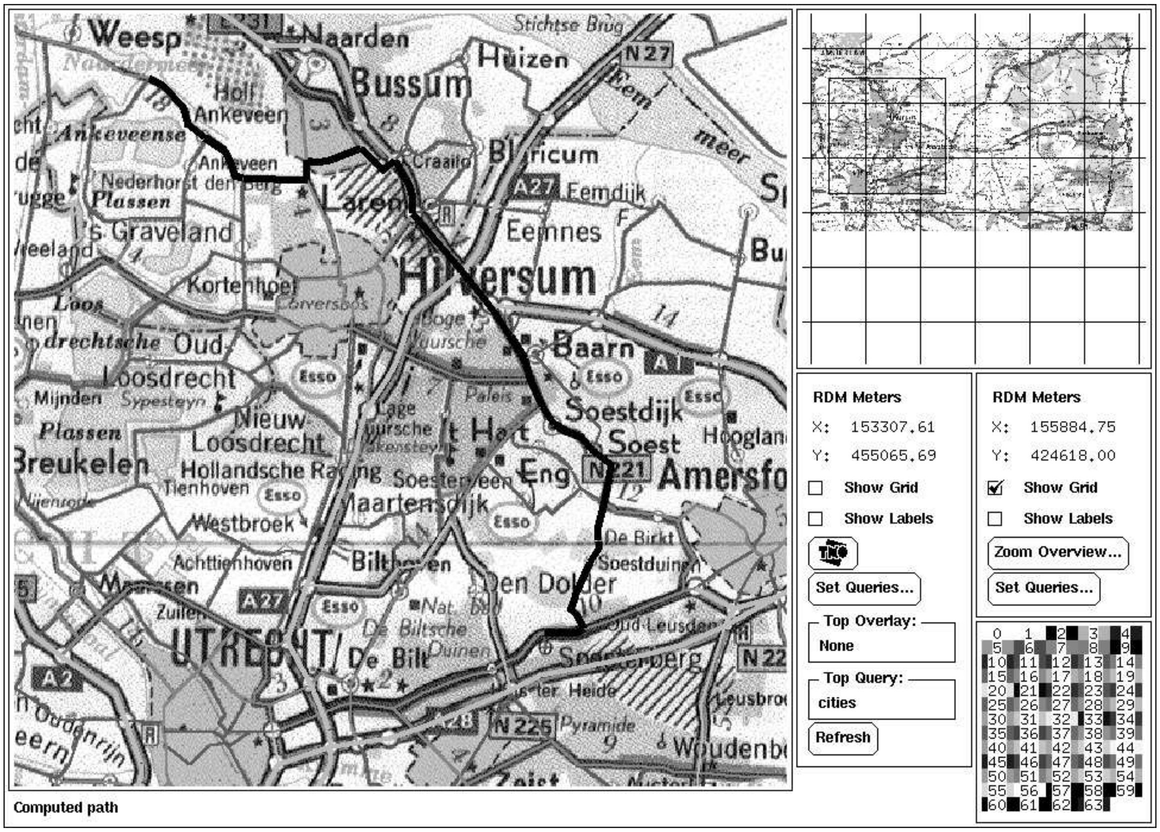

Figure 6: An hybrid map display

6 Conclusion

The current GEO^{++} system contains less functionality than some of the commercial GIS systems. However, our sound extensible DBMS/GIS architecture allows users to incorporate the functionality they require.

At this moment, we have the following on-going and planned research projects:

- 3D data types, operations, and visualization techniques.
- Raster operations (image processing).
- Enhance the cartographic interface by means of additional `QueryShapes` which:
  - Display more context information (legend, north, scale).
  - Produce thematic maps (choropleths, prism-maps, maps with pie-charts, etc.).
- Integrated multiple levels of map detail; reactive data structures (van Oosterom, 1991).
- Topologically structured map layers and operations such as network analysis, topologic editing, and map overlay (de Hoop & van Oosterom, 1991; Molenaar, 1989).

The GEO^{++} binaries (Sun/Sparc) and the source code are available. These may be obtained by anonymous ftp from `postgres.berkeley.edu` or `archive.eu.net`.

```
retrieve (the_oid= roads.oid,
          frompnt= PointSelector(roads.polyline, 1),
          topnt= PointSelector(roads.polyline, -1),
          length= Length2Pln(roads.polyline) / 1500.0
) where roads.category = 1  /* Highways (90 km/h = 1500 m/min) */

retrieve (the_oid= roads.oid,
          frompnt= PointSelector(roads.polyline, 1),
          topnt= PointSelector(roads.polyline, -1),
          length= Length2Pln(roads.polyline) / 1000.0
) where roads.category != 1  /* other roads */
```

Figure 7: Two example Postquel queries to extract shortest path analysis input data

*References

Abel, D. J. (1989). SIRO-DBMS: A Database Tool-kit for Geographical Information Systems. *International Journal of Geographical Information Systems*, *3(2)*, 103–116.

Bennis, K., David, B., Quilio, I., & Viémont, Y. (1990). GéoTropics Database Support Alternatives for Geographic Applications. in *4th International Symposium on Spatial Data Handling, Zürich*, pp. 599–610 Columbus, OH. International Geographical Union IGU.

Bennis, K., David, B., Morize-Quilio, I., Thévenin, J. M., & Viémont, Y. (1991). GéoGraph: A Topological Storage Model for Extensible GIS. in *Auto-Carto 10*, pp. 349–367.

Chance, A., Newell, R. G., & Theriault, D. G. (1990). An Object-Oriented GIS: Issues and Solutions. in *EGIS'90: First European Conference on Geographical Information Systems, Amsterdam, the Netherlands*, pp. 179–188 Utrecht. EGIS Foundation.

de Hoop, S., & van Oosterom, P. (1991). Topology in Postgres. Working paper.

ESRI (1989). ARC/INFO: The Georelational Model Revisited. *ARC News*, *11(1)*, 9.

Guttman, A. (1984). R-Trees: A Dynamic Index Structure for Spatial Searching. *ACM SIGMOD*, *13*, 47–57.

Herring, J. R. (1987). TIGRIS: Topologically Integrated Geographic Information System. in *Auto-Carto 8*, pp. 282–291.

Ho, W. W., & Olsson, R. A. (1991). An Approach to Genuine Dynamic Linking. *Software Practice & Experience*, *21(4)*, 375–390.

Intergraph (1990). MGE: The Modular GIS Environment. brochure.

Molenaar, M. (1989). Single Valued Vector Maps: A Concept in Geographic Information Systems. *Geo-Informationssysteme*, *2(1)*, 18–26.

Morehouse, S. (1989). The Architecture of Arc/Info. in *Auto-Carto 9*, pp. 266–277.

Pedersen, L.-O., & Spooner, R. (1988). Data Organization in System 9.

WILD Heerbrugg.

Preparata, F. P., & Shamos, M. I. (1985). *Computational Geometry.* Springer-Verlag, New York.

Schilcher, M. (1985). Interactive Graphic Data Processing in Cartography. *Computers & Graphics*, *9(1)*, 57–66.

Siemens Data Systems Division (1987). SICAD: The Geographical Information System for Modern Mapping. brochure.

Stonebraker, M., Rowe, L. A., & Hirohama, M. (1990). The Implementation of Postgres. *IEEE Transactions on Knowledge and Data Engineering*, *2(1)*, 125–142.

Stroustrup, B. (1986). *The C++ Programming Language.* Addison-Wesley, Reading, Mass.

Unisys Corporation (1989). Open Geographic Information Systems Forum. brochure.

van Oosterom, P., & Vijlbrief, T. (1991). Building a GIS on Top of the Open DBMS "Postgres". in *Proceedings EGIS'91: Second European Conference on Geographical Information Systems*, pp. 775–787 Utrecht. EGIS Foundation.

van Oosterom, P., van Hekken, M., & Woestenburg, M. (1989). A Geographic Extension to the Relational Data Model. in *Geo'89 Symposium, The Hague*, pp. 319–333 The Hague. Shape Technical Centre.

van Oosterom, P. (1991). The Reactive-Tree: A Storage Structure for a Seamless, Scaleless Geographic Database. in *Auto-Carto 10*, pp. 393–407.

van Roessel, J. W. (1987). Design of a Spatial Data Structure using the Relational Normal Forms. *International Journal of Geographical Information Systems*, *1(1)*, 33–50.

Waugh, T. C., & Healey, R. G. (1987). The GEOVIEW Design: A Relational Data Base Approach to Geographical Data Handling. *International Journal of Geographical Information Systems*, *1(2)*, 101–118.

Weinand, A., Gamma, E., & Marty, R. (1988). ET++: An Object Oriented Application Framework in C++. in *OOPSLA'88*, pp. 46–57 New York. ACM.

Weinand, A., Gamma, E., & Marty, R. (1989). Design and Implementation of ET++: A Seamless Object-Oriented Application Framework. *Structured Programming*, *10(2)*, 63–87.