*In Proceedings of the $6^{th}$ International Symposium on Spatial Data Handling, Edinburgh, Scotland, September 5–9, 1994.*

# Integrating Complex Spatial Analysis Functions in an Extensible GIS

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

**In this paper we present a framework for integrating complex spatial analysis functions in an open Geographic Information System (GIS). Two different aspects of integration are described: 1. integration of the *user interface* and 2. the *data sharing* between analysis functions and the GIS. Special attention will be paid to topology, as this plays an important role in many analysis functions. We designed a system which implements this framework based on an extensible relational database management system (the eRDBMS Postgres, but another eRDBMS can be used instead) and the extensible GIS called 'GEO$^{++}$'. The main integration tool is the new 'Builder' language, which provides the required glue between GEO$^{++}$ and other modules. This allows efficient integration with respect to both the user interface and the data sharing aspects. The advantages for the user are: a. faster/easier analyses, b. seamless integration with company data, c. efficient development of task-specific user interfaces, and d. the interaces of the analysis modules and the core GIS have an uniform look-and-feel.**

## 1 Introduction

The characteristic capabilities of a Geographic Information System (GIS) are the manipulation and combination of several spatial data sets with spatial analysis functions. The spatial data itself is the common factor to many of these functions and must therefore be stored in a DataBase Management System; DBMS. In (van Oosterom & Vijlbrief, 1991; Vijlbrief & van Oosterom, 1992) it was described how we stored all geographic data, both thematic and spatial, in one single database. This solution, the *integrated GIS architecture*, is based on an extensible DBMS and has many advantages over the *dual GIS architecture* in which spatial and thematic data are stored in two separate subsystems. In GEO$^{++}$, we currently use the extensible relational DBMS Postgres (Stonebraker, Rowe, & Hirohama, 1990), which we extended with some abstract data types (ADTs) for spatial data: point, line, and region. New spatial data types can be added to the system by providing an ADT implementation and a QueryShape, which implements the draw and pick functions in GEO$^{++}$. Note that the functionality in Postgres is similar to the future SQL3 standard (Robinson & Tom, 1993),

Fig. 1: A spatial data model and some simple spatial analyses

```
a. create town (name=text, population=int4, region=POLYGON2)
b. create road (name=text, construct=date, shape=POLYLINE2)

c. retrieve (town.all) where Area2Pgn(town.region) > 10.000
d. retrieve (town.all) where Box2Pgn(town.region) && "(0,0,800,400)"::BOX2
e. retrieve (road.all) where Length2Pln(road.shape) < 5.000
                         and road.construct > "1 Jan 1990"
f. retrieve (town.all) where MinDist2PgnPln(town.region, road.shape) < 500
                         and road.name = "A12"
```

as used in for example *Montage*, the commercial successor of the Postgres research DBMS. Though we'll use Postgres examples in this paper, the method is general and can be applied in combination with any eRDBMS.

Many different types of spatial analyses can be expressed as database queries in the query language using the spatial ADTs. The results of a query can be presented in the standard tabular form or on a map display using the QueryShapes to depict the retrieved objects. Using GEO$^{++}$'s graphical query composer it is easy to specify these queries, as syntax errors are avoided and the graphical representation of the query also supports easier manipulation.

Simple spatial analyses can be performed by creating queries which use the operators of the spatial ADTs. For example, assume we have created the data model in Fig. 1a and b. The first query (Fig. 1c) finds all towns with an area greater then 10.000 square meters by applying the `Area2Pgn` function to the attribute `region`. The second query (Fig. 1d), converts the region to a bounding box and tests if it has overlap (operator `&&`) with a given rectangle. Note that in query Fig. 1e, both the spatial and thematic attributes are used in an uniform manner in the where-clause to find all roads shorter than 5.000 meters constructed after 1 Jan. 1990. Spatial index structures are required to efficiently solve these spatial queries. Postgres provides an R-tree index (Guttman, 1984), but others can be added; e.g., the Reactive-tree (van Oosterom & Schenkelaars, 1993; van Oosterom, 1993).

More complicated types of analyses can be performed by using *joins* based on the spatial attributes. The last query (Fig. 1f) retrieves every town that lies within 500 meters (buffer-zone) of one specific road, the 'A12'. The query can be made more general by dropping the second part of the where-clause: find all towns that lie within 500 meters of a road. In a sense, this is the optimal type of integration of spatial analyses in a GIS. However, certain types of analyses are more difficult to execute in this way. For example the query which finds the shortest path from A to B over the network of roads is hard to express in Postquel. An alternative is to regard the road network as one complex object in which the individual roads (sub-objects) are linked. Now, a shortest path function can be applied to the complex object road-network. Unfortunately, this solution is also not feasible in many DBMSs.

In general, it is more difficult to perform complex spatial analyses with one single database query if the operands are complex objects. A feasible way to implement this is by using separate dedicated programs. A few examples are:

- map-overlay computation, which can be executed in raster or vector mode (Dougenik, 1979; Akman, Franklin, Kankanhalli, & Narayanaswami, 1989; van Roessel, 1990; van Oosterom, 1994);
- computation of buffer-zones around point, line, or area features;
- shortest path in linear networks, network analyses in general: Dijkstra or A* search algorithms (Dijkstra, 1959; Hart, Nilsson, & Raphael, 1968);
- shortest path through weighted regions; CCM, which is very different from linear networks; see Section 4;
- Voronoi diagram, Delaunay triangulation (Preparata & Shamos, 1985);
- physical simulations: ground water model-

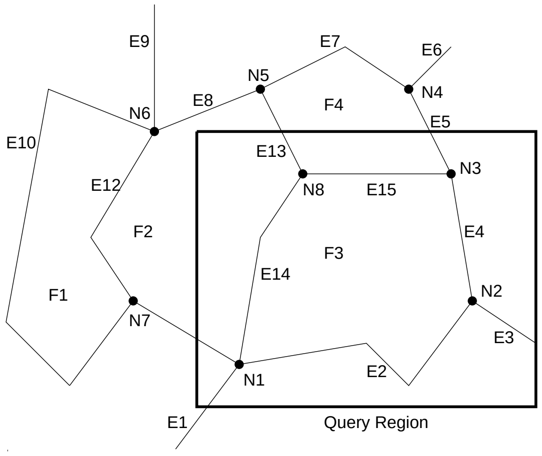


Fig. 2: Missing edges with `bbox` selection

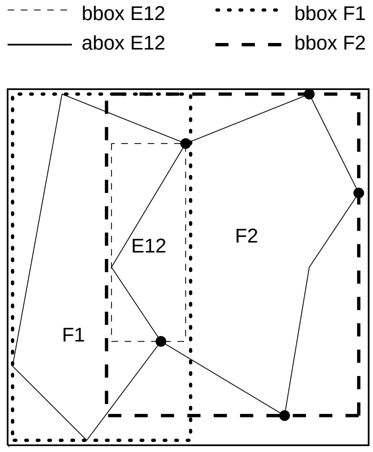


Fig. 3: The `abox` of an edge

ing, erosion, runoff, etc.;

- three dimensional analyses: slopes, visibility map, volumes, conversion between different representations (TIN, grid, contour) (Floriani & Magillo, 1993; Fisher, 1993; Watson, 1992).

When considering the integration of spatial analyses in a GIS, two important aspects have to be distinguished:

- *data sharing* between the GIS and analysis software. Several techniques are available for achieving this: ASCII files, binary files, common database, shared memory, dynamic loading of functions (Ho & Olsson, 1991) into the address space of the core GIS process. Goodchild classifies the types of integration into the following categories (1992):
  1. *Stand-alone* spatial analysis software and no data exchange;
  2. *Loose coupling* data exchange between GIS and spatial analysis software through ASCII or binary files;
  3. *Close coupling* spatial analysis software using the macro language and 'hooks' of the core GIS: one common data model is used;
  4. *Full integration* of spatial analysis software and the GIS into one program: data is in the 'same program'.

  Most existing GIS/analyses environments fall in category 2 or 3.

- *user interface* integration between analysis functions and the core GIS (setting of parameters with buttons, sliders, textfields; picking of locations from the map; output of maps, tables, records, etc.). The graphical user interface should be consistent for the core GIS and the different spatial analyses modules (even if they are written in different programming languages) so that the user does not have to change the style of interaction.

Quite a few spatial analysis functions are based on a topological data structure. As will be described in Section 2, this structure can be modelled and stored in an eRDBMS, but it poses some special problems for visualization in GEO$^{++}$. Section 3 describes the BUILDER language, our tool for integrating spatial analyses and other modules in a (GIS) system. This section also describes the GEO$^{++}$ features which are integrated it into this environment. One specific example of spatial analysis, cross-country movement planning, is given Section 4. The main objective is to design and implement an integrated GIS/analysis environment.

## 2 Topology

The advantages of using a planar topological structure in a GIS database are well-known: it avoids redundancy when storing common boundaries and it is very suitable for certain types of spatial analyses, e.g., shortest path computations in a network (de Hoop & van Oos-

Fig. 4: A polygonal area data model

```
a. create boundaries (b_id=int4, line=POLYLINE2, bbox=box, abox=box)
                     /* possibly also include  a_id_l, a_id_r */
b. create areas      (a_id=int4, b_ids=int4[], bbox=box)
c. create geo_topol  (ind_relname=text, ind_relattr=text,
     topoltype=text, /* linear, polygonal network, hierarchy,... */
     ref_count=int,  /* #objects referred 1,2,...or variable (0) */
     ref_relname=text, ref_relid=text, ref_relvis=text, ref_relbbox=text)
d. append geo_topol  (ind_relname="areas", ind_relattr="b_ids",
     topoltype="ind polygon to polyline",
     ref_count=0,    /* variable nr of references */
     ref_relname="boundaries", ref_relid="b_id",
     ref_relvis="line", ref_relbbox="abox")
e. append geo_dyninfo (relname="boundaries",relattr="line",
     bboxattr="bbox", dynfunc="bin_pln2_shape", is_bin="t",
     dynfile="$GEOHOME/dynamic/Geo2Shapes.o", the_oid="oid")
f. append geo_dyninfo (relname="areas",relattr="b_ids",
     bboxattr="bbox", dynfunc="bin_tpgn2_shape", is_bin="t",
     dynfile="$GEOHOME/dynamic/Topol2Shapes.o", the_oid="oid",
     cleanup_func="tpgn2_clean")
```

terom, 1992; de Hoop, van Oosterom, & Molenaar, 1993). Topology describes relationships (Egenhofer & Franzosa, 1991; Clementini, Felice, & van Oosterom, 1993) between objects and can be used to create hierarchies of objects consisting of sub-objects; e.g., a city may consist of several districts.

The drawback of using a topological structure is that the visualization is often indirect and therefore slower; e.g., areas in a topologically structured polygonal map. First, the references from areas to edges have to be retrieved, then the actual edges have to be retrieved (based on the references), and finally, the polygons have to be reconstructed from the edges. Also, special attention has to be paid to islands (or holes) in polygons.

A difficulty when using topology is that it is data model dependent unlike the normal type POLYGON2, which is implemented as an ADT within the eRDBMS environment. So somehow during the visualization, the dependencies on the data model have to be made explicit to GEO$^{++}$. Therefore, we introduced a new GEO$^{++}$ meta class `geo_topol` which contains the data model dependent topological information. Existing GEO$^{++}$ meta classes are: `geo_edit`, `geo_dyninfo`, etc. The use of `geo_dyninfo` is explained in (Vijlbrief & van Oosterom, 1992). In the `geo_edit` table user defined actions can be specified, which are executed when an object is picked on the map display. Note that this action can be the start of some kind of spatial analysis in which the picked object plays a special role.

### 2.1 An area partitioning

Another problem that occurs when using topology for an area partitioning is that when the user selects a part of the database (indicated by a box) for display on the screen, it is not clear which edges should be retrieved from the database. Using the overlap test with the bounding boxes of the edges is not sufficient; e.g. see Fig. 2: edge 'E12' is not retrieved, so polygon 'F2' cannot be drawn. It is also quite inefficient to first select all areas based on their bounding box, and then select the edges based on references in the retrieved areas. However, the advantages of topology are too important to ignore, so we designed the following solutions:

1. The `geo_topol` meta class indicates the class (and its attribute) that stores the edges of the polygons and the class (and its attributes) that contains the referenced edge_id and `POLYLINE2` pair. Care must be taken to assure that every edge has its own unique id. This is achieved by using the Postgres rule system.
2. Every edge contains its own bounding box `bbox` and the bounding box of the union

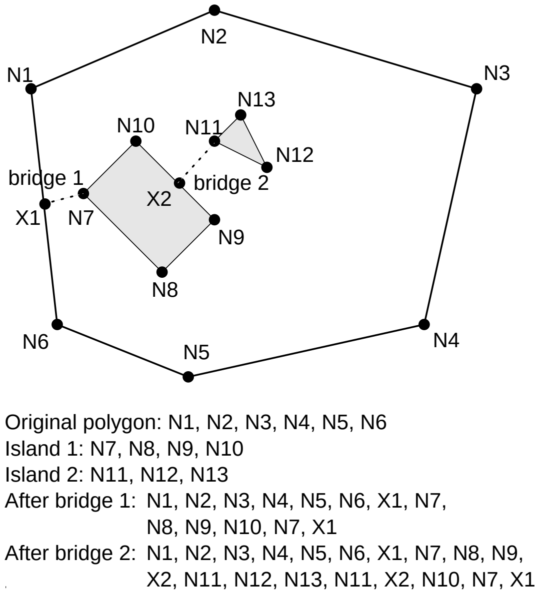


Fig. 5: Remove islands from polygon

of its left and right area boxes `abox`; see Fig. 3. Two simple spatial queries are now sufficient: Retrieve the edges where its `abox` overlaps the screen rectangle. Repeat this for the areas.

Fig. 4 shows the data model for an area partitioning in GEO$^{++}$. The `areas` class stores the references to the boundaries in a variable length array: `b_ids`. The `geo_dyninfo` entry with `relname = "areas"` tells the topological drawing function `bin_tpgn2_shape` to use this array of references. The references in the array are in the right order to form the polygon and if they carry a negative sign then the edge has to be reversed.

Besides normal polygons we also want to handle polygons with holes or *islands*. In the list of references to edges, the start of a new island is marked with a `0` separator. The `DeIsland` module removes islands by creating 'bridges' from interior to exterior boundaries. If there are several islands then they are processed in the order of shortest distance; see Fig. 5.

### 2.2 Other topologies

We also want to handle non polygonal topology, e.g., a linear network used in applications that deal with pipes, cables, or roads. These networks do not pose a visualization problem: nodes are displayed as markers and edges as lines. It is convenient for the analysis functions to retrieve this topology from the database instead of recomputing it. Several types of network topology are useful: planar/non planar, directed/non directed edges, disallowing certain turns at a node, etc.

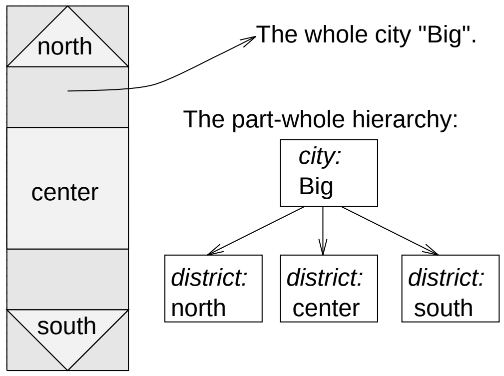


Fig. 6: The part-whole hierarchy

Another type of topology is a part-whole hierarchy; e.g., a city may consist of districts; see Fig. 6. Using the relational approach it may not be clear that users should also visualize the districts when looking at cities. If the `geo_topol` meta-class signals GEO$^{++}$ that a city contains sub-objects, then a smart drawing module (SmartTopolShape) may draw the districts without an explicit user query. Fig. 7 a-g shows the part-whole hierarchy model. It is possible to introduce redundancy into the data model by including references from children to parents. Note that something similar is possible in the polygonal topological model: the edges may contain references to the left and right regions. In our case the district class is extended with a reference to the city (which is extended with a sequence number) and `geo_topol` gets an extra entry; see Fig. 7 h-i.

Topology references are not limited to one level of indirection. For example, the `district` class could be modified to be part of an area partitioning and to contain references to edges instead of a `POLYGON2`. So, when the `city` is displayed, the `districts` have to be retrieved. To display a `district`, the edges have to be retrieved.

Fig. 7: The part-whole modeling

```
a. create city (name=text, parts=int4[], boundary=POLYGON2)
b. create district (name=text, nr=int4, region=POLYGON2)
c. append district (name="north",  nr=3,region="(3:0,3,2,3,1,4)"::POLYGON2)
d. append district (name="center", nr=1,region="(4:0,0,2,0,2,2,0,2)"::POLYGON2)
e. append district (name="south",  nr=8,region="(3:0,-1,2,-1,1,-2)"::POLYGON2)
f. append city (name="Big", parts={1,3,8},
                              boundary="(4:0,-2, 2,-2, 2,4, 0,4)"::POLYGON2)
g. append geo_topol (ind_relname="city", ind_relattr="parts",
     topoltype="hierarchy: parent to child",
     ref_count="0",   /* variable nr of references */
     ref_relname="district", ref_relid="nr",
     ref_relvis="region", ref_relbbox="None")
-------------------------------------------------------------------------------
h. create city (name=text, nr=int4, parts=int4[], boundary=POLYGON2)
i. create district (name=text, nr=int4, belongs_to=int4, region=POLYGON2)
j. append geo_topol (ind_relname="district", ind_relattr="belongs_to",
     topoltype="hierarchy: child to parent",
     ref_count="1",   /* district belongs to 1 city */
     ref_relname="city", ref_relid="nr",
     ref_relvis="boundary", ref_relbbox="None")
```

## 3 Builder Interface

The spatial analysis and database components of a GIS are often available as batch commands. These can be combined by means of operating system scripts or with a GIS specific scripting language, which in addition may allow the developer to customize the interface by giving access to graphical user interface (GUI) components like buttons, sliders and menubars.

In addition, most systems give access to the GIS components by means of a *C* (Kernighan & Ritchie, 1978; Purdum, 1985) or *C*$^{++}$ (Stroustrup, 1986) library which allows the development of applications by programmers. Database examples are the *Postgres* `libpq` library of C interface routines and commercial DBMSs like Ingres or Oracle, that offer embedded SQL preprocessors whose output is linked with a C library. GUI examples are the *ET*$^{++}$ (Weinand, Gamma, & Marty, 1988, 1989) C$^{++}$ library (used for the construction of GEO$^{++}$) which offers a very rich set of GUI components and data structures and the XView (Heller, 1990) library which gives access to GUI blocks.

However, the C$^{++}$ language is not suited for loosely coupling components by means of a simple intuitive language. For this we would prefer a dedicated interpreted language and this is the reason that most GIS systems offer such a scripting language. These languages are often not extensible by the user and do not allow the integration of third-party software in a coherent interpreted framework. For integration of this software one has to revert to C or C$^{++}$ programming and the use of operating system scripts; e.g., Unix shell scripts.

The shell scripting language has proven over time to be very effective in the implementation of new commands from primitive building blocks. The popularity of advanced GUIs and database systems is however currently limiting the arena in which the shell scripting language can be applied, due to its stream-oriented and sequential nature. The use of GUIs calls for a more message-oriented approach and in addition we want to make use of the huge installed base of C, C$^{++}$ and other language libraries, all in a coherent framework. We developed an interpreted extensible object-oriented language, called Builder, which allows access to compiled C and C$^{++}$ libraries. Summarizing, we state as our design goal:

> *We would like to apply the shell programming methodology on objects which interface to, or are implemented by means of, existing* C, C$^{++}$ *and other language libraries.*

The Builder language has the following features:

- It has interfaces to C and C$^{++}$ code or ex-

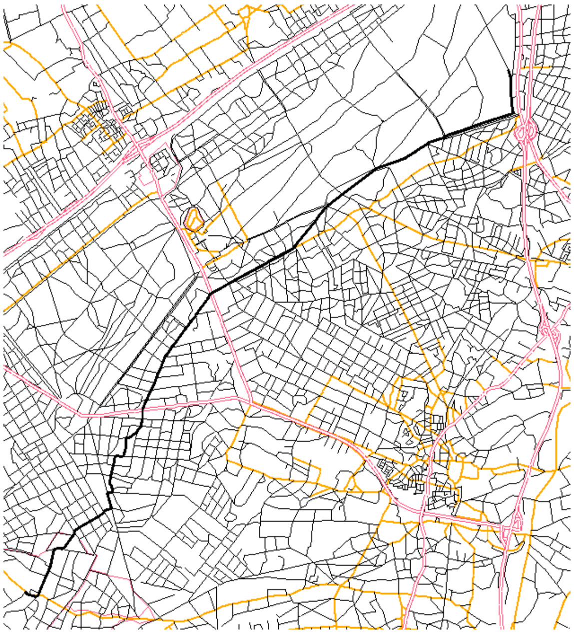

Fig. 8: Shortest path in a road-network

isting libraries. This allows implementation of operations in an efficient compiled language when needed.

- It is interpreted to allow for rapid prototyping and eliminates the 'edit, compile, link' cycle when using existing building blocks.
- It has object-oriented features such as classes, inheritance, messages (methods), and polymorphism.
- It has conventional programming constructs like variables, loops, and 'if then else'.
- It Integrates building blocks in one address space (i.e., a single executing program) by means of runtime loading of the binary object files implementing the building blocks. This allows efficient access to shared data, e.g., needed for spatial analyses. The traditional communication in shell scripts by means of files or sequential pipe lines creates a performance penalty and results in one way communication and difficult integration with GUI objects.

The last point can be illustrated with our pre-Builder language GEO$^{++}$ implementation of shortest path (SP) analysis. In the old situation, the analysis program first had to read all the road data from the database. This data transfer was the bottleneck in the analysis as can be seen in the timings of Table 1. Note that this read step had to be repeated for each analysis, because the batch modules have no persistent memory. The SP computation in the larger data set is visualized in the map-fragment; see Fig. 8. In this large set, the actual computation time depends on the length of the optimal path since we have implemented Dijkstra's algorithm (1959). Note that the overhead in both test cases was huge compared to the actual time for the SP analyses; about one order of magnitude.

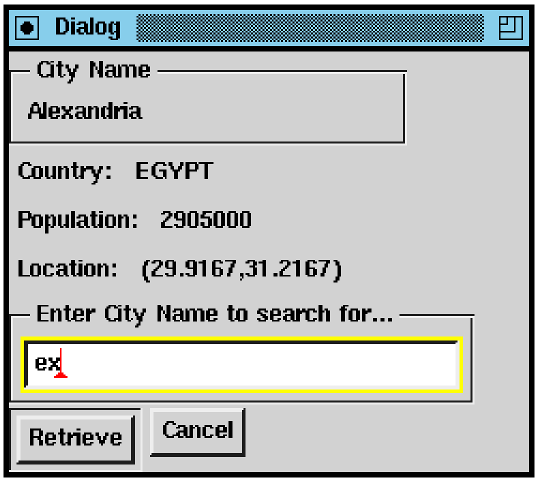


Fig. 9: Screen dump of a sample interface

We have currently implemented Builder blocks which interface to:

- The Postgres and Oracle DBMSs.
- The Unix operating system.
- Various C library functions and utilities for string operations and arithmetic.
- Network communication by means of Remote Procedure calls.
- ET$^{++}$ GUI components like buttons, menubars, icons, sliders, dialog boxes, and menubars.

Table 1: Shortest path analysis benchmark (elapsed time in seconds on SparcStation 10)

| #edges | read | compute | visualize |
|---|---|---|---|
| $\sim 1.000$ | 6 | $< 1$ | 5 |
| $\sim 24.000$ | 95 | $1 - 12$ | 5 |

- The core GEO++ system which allows:
  - Modification of the main menubar.
  - Placement of objects on the map with which the user may interact.
  - Access to the database tuple editor and data browser.
  - Zooming/panning to specific spatial objects.
  - Refetching specific layers from the database and modifying queries.
  - Notifying users by means of the alert message line.
  - ....
- Specific spatial analyses modules.

We illustrate the BUILDER environment by describing a simple application which retrieves information from a Postgres database. The user inputs a (wildcard) specification of a city name. The system then retrieves a matching tuple from the `cities` relation in the `geo` database. The GUI is shown in Fig. 9 and the BUILDER specification is shown in Fig. 10. We cannot explain the full semantics of the BUILDER language in this tight space, but see (Vijlbrief, 1993) for details. The examples should be intuitive however. See also section 4.2 for some of the GUI components.

The interface between the BUILDER Postgres block and the underlying implementation is specified in the `Postgres.build` file as:

```
class Postgres {
  implementation "$BUILD_BLOCKS/Postgres.o"
  constructor    MakePostgres
}
```

This specifies that the class `Postgres` is implemented in the binary file `Postgres.o` which is generated by compiling the C++ code shown (in part) in the Appendix. The BUILDER interpreter will load this file at runtime and it will call the C++ function `MakePostgres` when an instance of the `Postgres` class must be created.

## 4 Complex Analysis Example

In this section, we describe a type of complex analyses, cross-country movement (CCM) (Subsection 4.1) and its GUI specification in the BUILDER language (Subsection 4.2).

### 4.1 CCM Analysis

The CCM problem is also known as the Weighted Region (Least Cost Path) Problem (Mitchell & Papadimitrou, 1991; Smith, Peng, & Gahinet, 1989). The unit cost of traversing a given region (polygon) is uniform within the region, but varies between regions and is based on soil, vegetation, etc. Finding an optimal path, i.e., locating a corridor from a given source location to a given destination (Church, Loban, & Lombard, 1992; Goodchild, 1977; Huber & Church, 1985) can be used for traveling, but also for planning highways, railways, pipelines, and other transport systems. The cost function is based on optimization criteria such as time, safety, fuel usage, impact, length, etc. Note that the CCM-problem is very different from the more common route planning in a linear road network, because the cost of using an element (polygon) is not fixed.

There exist raster and vector-based algorithms for this problem. The vector-based approach of Mitchell and Papadimitriou (1991) has been implemented. The first step in the vector algorithm is to apply a constrained Delaunay triangulation (Chew, 1987; Lee & Schachter, 1980) to the polygonal map data. Then, a wave-front propagation technique is used to trace the optimal paths from the source throughout the terrain. During this step the topology of the triangulation is heavily used.

Further, several raster-based algorithms have been implemented, which are all based on creating a graph with nodes (e.g., raster cells) and implicit edges, i.e., the possible movements between the nodes. Then the Dijkstra (1959) graph search algorithm is used to find the optimal path. Other search techniques, e.g., based on heuristics (A*-algorithm (Hart *et al.*, 1968)), could be used. The main difference between the raster algorithms is the assignment of nodes and edges: *4- to 128-connected rasters*; *extended raster*; *quadtree* (Samet, 1989) based-rasters which save memory space. The new extended raster method extends the number of possible angles to move within one individual grid cell. More details can be found in (van Bemmelen, Quak, van Hekken, & van Oosterom, 1993).

Fig. 10: BUILDER code for city_dialog

```
#include "blocks/BuilderBlocks.build" // Builder files are C-preprocessed!
#include "blocks/Postgres.build"

object postgres : Postgres("geo")
object city_dialog : Dialog("mode=modeless") {
  VBox {
      name : Border("City Name") { TextItem("width=200", "text=") }
      HBox { TextItem("text=Country:")
             country : TextItem("width=200", "text=") }
      HBox { TextItem("text=Population:")
             population : TextItem("width=200", "text=") }
      HBox { TextItem("text=Location:")
             location : TextItem("width=200", "text=") } }
  Border("Enter City Name to search for...") { cn : TextField }
  HBox { ActionButton("label=Retrieve", "message=retrieve", "default=1")
         ActionButton("label=Cancel", "message=quit") }

  on quit do send utils Call("exit", 0)
  on retrieve do {
    c= send self.cn GetText ; where= send utils Sprintf("cities.name ~ \"%s\"", c)
    send postgres Iter("cities", where, "name", "country", "population", "geo_loc")
    if send postgres Next {
      tmp= send postgres GetField(1) ; send self.name SetText(tmp)
      tmp= send postgres GetField(2) ; send self.country SetText(tmp)
      tmp= send postgres GetField(3) ; send self.population SetText(tmp)
      tmp= send postgres GetField(4) ; send self.location SetText(tmp)
      if send postgres Next    send utils ShowAlert("More than 1 city found!")
    } else
      send utils ShowAlert("No city found!")
    send postgres Done
  }
}
send city_dialog Open
```

### 4.2 Builder Interface

The upper-left part of Fig. 11 shows that the user can choose between the vector or raster CCM method from the GEO++ menubar. The lower-right part of Fig. 11 shows the 'Raster CCM Form' which allows graphical selection of multiple sources, the computation method (more-connected, extended raster, or quadtree), and several other parameters. The GUI used to select sources (top part of the 'Raster CCM Form') has been created with the BUILDER code in Fig. 12.

The interface specification consists of the layout of GUI primitives (labels, buttons, lists, etc.) and the handlers (methods) for messages generated by these primitives, e.g., `ActionButtons`. Only two methods of the `rastccm_dialog` are included here: `source_add`, to add sources to the list, and `source_rm_all`, to clear the list of sources.

The `CollView` object is used to manage a collection of items; in this case the coordinates of the sources. A `Scroller` adds scrollbars to its contained object (in this case a `CollView`) when this is needed. `ActionButton`, `CollView`, and `Scroller` are BUILDER blocks based on the ET++ library. `VBox`, `HBox`, and `Border` are ET++ objects used for layout purposes.

The `send` command is used to communicate within and between objects. The `geo` object (GEO++ core) is no exception; e.g., the `SelectXY` message (see Fig. 12) lets the user select a coordinate on the displayed map. Also, the CCM analysis module is integrated this way (not shown in fragment). Markers, indicating selected source locations on the map display, are managed by a special object `source_marker`.

The user starts the execution of the CCM analysis by pressing the `Compute` button; see Fig. 11. Thus, the current CCM-application is com-

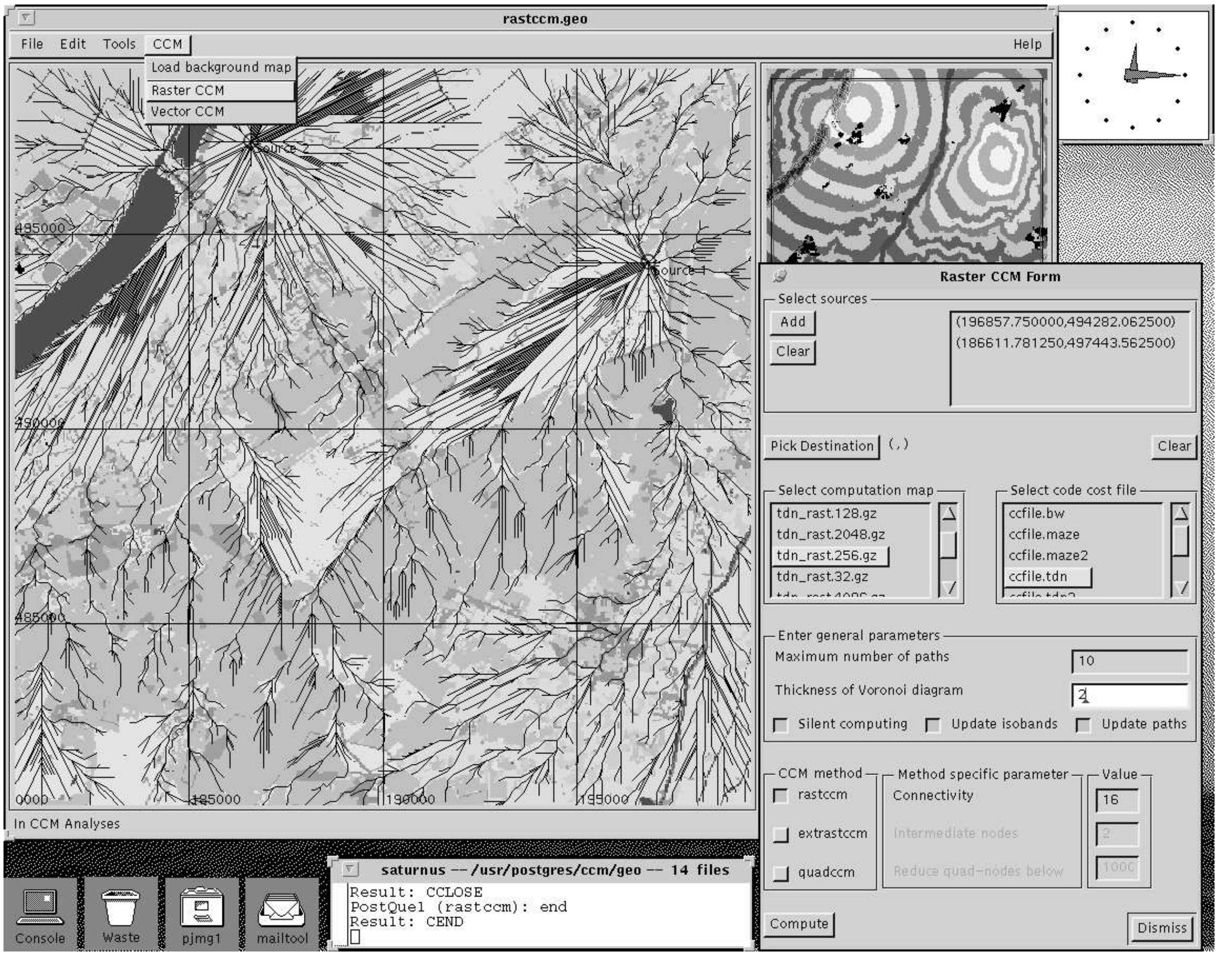


Fig. 11: CCM Analysis: pull-down menu, Raster CCM Form, and map display

pletely integrated with respect to the GUI and the data.

## 5 Conclusion

We have described a general method for storing and visualizing topological data in a GIS. This is very important as topology is used in many different analysis functions.

The BUILDER environment allows the integration of all aspects of a complex (GIS) system to be integrated in a really object-oriented environment. One of its key features is the extensibility, which allows the integration of third-party software in a coherent interpreted framework.

It also enables the implementation of tightly coupled analysis modules with respect to the data and the GUI. This is beneficial for the performance, because data transfers are reduced.


## Acknowledgements

Many valuable comments and suggestions on a preliminary version of this paper were made by Paul Strooper and two anonymous referees.


## References


Akman, V., Franklin, W. R., Kankanhalli, M., & Narayanaswami, C. (1989). Geometric computing and uniform grid technique. *Computer Aided Design*, *21(7)*, 410–420.

Chew, L. P. (1987). Constrained Delaunay Triangulations. *ACM*, *?*, 215–222.

Church, R. L., Loban, S. R., & Lombard, K. (1992). An Interface for Exploring Spatial Alternatives for a Corridor Location Problem. *Computers and Geosciences*, *18(8)*, 1095–1105.

Clementini, E., Felice, P. D., & van Oost-

Fig. 12: BUILDER code for rastccm_dialog

```
object rastccm_dialog = Dialog("label=Raster CCM Form","mode=modeless") {
  // Interface layout....
  VBox ("align=HLeft|HExpand","gap=20") {
      Border("Select sources") { HBox("align=HGapExpand") {
        VBox ("align=HLeft|HExpand") {
          b_add    : ActionButton("label=Add", "message=source_add")
          b_rm_all : ActionButton("label=Clear", "message=source_rm_all") }
        Scroller("height=60", "width=200") {
          cv_source : CollView { TextItem("text=No source") } } } }
    // Other bordered HBox's for computation method, parameters are omitted....
  }
  // Specification of actions triggered by messages
  on source_rm_all do {
    nr_sources = 0
    send source_marker remove_all
    send self.cv_source Empty
    send self.comp_but Disable(1)
    send self.b_rm_all Disable(1)
  }
  on source_add do {
    nr_sources = eval(nr_sources,"+",1)
    my_name= sprintf("Source %d", nr_sources)
    send geo SetMessage("Give source location")
    source_loc = send geo SelectXY(&worldx, &worldy, &screenx, &screeny)
    send source_marker add(my_name, screenx, screeny)
    send self.cv_source AddTextLine(source_loc)
    send self.comp_but Enable(1)
    send self.b_rm_all Enable(1)
    send geo SetMessage(ccm_std_message)
  }
  // Other actions are omitted....
}
```


erom, P. (1993). A small set of formal topological relationships suitable for end-user interaction. in *SSD'93: The Third International Symposium on Large Spatial Databases, Singapore*, pp. 277–295 Berlin. Springer-Verlag.

de Hoop, S., & van Oosterom, P. (1992). Storage and Manipulation of Topology in Postgres. in *Proceedings EGIS'92: Third European Conference on Geographical Information Systems*, pp. 1324–1336. EGIS Foundation.

de Hoop, S., van Oosterom, P., & Molenaar, M. (1993). Topological Querying of Multiple Map Layers. in *COSIT'93, Elba Island, Italy*, pp. 139–157 Berlin. Springer-Verlag.

Dijkstra, E. W. (1959). A note on two problems in connection with graphs. *Numerische Mathematik 1*, *1*, 269–271.

Dougenik, J. (1979). WHIRLPOOL: A Geometric Processor for Polygon Coverage Data. in *Auto-Carto 4*, pp. 304–311.

Egenhofer, M. J., & Franzosa, R. D. (1991). Point-set topological spatial relations. *International Journal of Geographical Information Systems*, *5(2)*, 161–174.

Fisher, P. F. (1993). Algorithm and implementation uncertainty in viewshed analyses. *International Journal of Geographical Information Systems*, *7(5)*, 331–347.

Floriani, L. D., & Magillo, P. (1993). Computing Visibility Maps on a Digital Terrain Model. in *COSIT'93, Elba Island, Italy*, pp. 248–269 Berlin. Springer-Verlag.

Goodchild, M., Haining, R., & Wise, S. (1992). Integrating GIS and spatial data analysis: problems and possibilities. *International Journal of Geographical Information Systems*, *6(5)*, 407–423.

Goodchild, M. F. (1977). An evaluation of lattice solutions to the problem of corridor location. *Environment and Planning A*, *9*, 727–738.

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

Hart, P. E., Nilsson, N. J., & Raphael, R. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems, Science and Cybernetics*, *SSC-4(2)*, 100–107.

Heller, D. (1990). *XView Programming Manual*. O'Reilly & Associates, Inc.

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

Huber, D. L., & Church, R. L. (1985). Transmission Corridor Location Modeling. *Environmental Engineering ASCE*, *111(2)*, 114–130.

Kernighan, B. W., & Ritchie, D. M. (1978). *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ.

Lee, D. T., & Schachter, B. J. (1980). Two Algorithms for Constructing a Delaunay Triangulation. *International Journal of Computer and Information Sciences*, *9(3)*, 219–242.

Mitchell, J. S. B., & Papadimitrou, C. H. (1991). The Weighted Region Problem: Finding Shortest Paths Through a Weighted Planar Subdivision. *Journal of the Association for Computing Machinery*, *38(1)*, 18–73.

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

Purdum, J. J. (1985). *C Programming Guide, Second Edition*. Que Corporation, Indianapolis.

Robinson, V. B., & Tom, H. (1993). Towards SQL Database Language Extensions for Geographic Information Systems. Tech. rep. NISTIR 5258, National Institute of Standards and Technology.

Samet, H. (1989). *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, Mass.

Smith, T. R., Peng, C., & Gahinet, P. (1989). Asynchronous, Iterative, and Parallel Procedures for Solving the Weighted-Region Least Cost Path Problem. *Geographical Analysis*, *21(2)*, 147–166.

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.

van Bemmelen, J., Quak, W., van Hekken, M., & van Oosterom, P. (1993). Vector vs. Raster-Based Algorithms for Cross Country Movement Planning. in *Auto-Carto 11*, pp. 304–317.

van Oosterom, P., & Schenkelaars, V. (1993). Design and Implementation of a Multi-Scale GIS. in *Proceedings EGIS'93: Fourth European Conference on Geographical Information Systems*, pp. 712–722. EGIS Foundation.

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. EGIS Foundation.

van Oosterom, P. (1993). *Reactive Data Structures for Geographic Information Systems*. Oxford University Press, Oxford.

van Oosterom, P. (1994). An R-tree based map-overlay algorithm. in *Proceedings EGIS/MARI'94: Fifth European Conference on Geographical Information Systems*. EGIS Foundation. To be published.

van Roessel, J. W. (1990). Attribute Propagation and Line Segment Classification in Plane-Sweep Overlay. in *4th International Symposium on Spatial Data Handling, Zürich*, pp. 127–140 Columbus, OH. International Geographical Union IGU.

Vijlbrief, T., & van Oosterom, P. (1992). The GEO++ System: An Extensible GIS. in *Proceedings of the 5th International Sympo-*

*sium on Spatial Data Handling, Charleston, South Carolina*, pp. 40–50 Columbus, OH. International Geographical Union IGU.

Vijlbrief, T. (1993). The Builder development environment (DRAFT). .

Watson, D. F. (1992). *Contouring: A Guide to the Analysis and Display of Spatial Data*. Pergamon Press, New York.

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.

## Appendix: C++code Postgres.cc

```
#include "Block.h"

class PostgresBlock: public Block {
  char          *database, **last_q;
  QueryIter     *qi;
  int           nr_fields;
public:
  PostgresBlock(char *db): qi(0), database(strsave(db)) { }
  int Handle(Block *sender, TypedMessage *m, TypedArg *ret_val);
};

int PostgresBlock::Handle(Block *sender, TypedMessage *m,
  TypedArg *ret_val) {
  if (IsMessage(m, "Iter")) {
    // C++ Code for message Iter deleted
  } else if (IsMessage(m, "Next")) {
    // C++ Code for message Next deleted
  } else if (IsMessage(m, "GetField")) {
    int field_sel= m->GetArg(1) - 1;
    if (last_q == 0 || field_sel >= nr_fields || field_sel < 0)
      BuildWarning("Postgres GetField: Invalid selector");
    else
      *ret_val= TypedArg(strsave(last_q[field_sel]), TRUE);
  } else if (IsMessage(m, "Done")) { SafeDelete(qi);
  } else if (IsMessage(m, "Exec")) { PQexec(m->GetStr(1));
  } else return Block::Handle(sender, m, ret_val);
  return TRUE;
}

extern "C" Block *MakePostgres(int n, TypedArg a[]) {
  if (n != 1) { BuildFatal(
    "Postgres Block needs a database name as argument");
    return 0; }
  return new PostgresBlock(a[0]);
}
```