INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center St. • Suite 600 • Berkeley, California 94704-1198 • (510) 643-9153 • FAX (510) 643-7684

Designing and Integrating User Interfaces of Geographic Database Applications

Agnès Voisard*

TR-94-015

March 1994

Abstract

In this paper, we investigate the problem of designing graphical *geographic database user interfaces* (GDUIs) and of integrating them into a *database management system* (DBMS). Geographic applications may vary widely but they all have common aspects due to the spatial component of their data: Geographic data are not standard data and they require appropriate tools for (i) editing them (i.e., display and modify) and (ii) querying them. The conceptual problems encountered in designing GDUIs are partly due to the merger of two independent fields, geographic DBMSs on the one hand, and *graphical user interfaces* (GUIs) on the other hand. Although these areas have evolved considerably during the past ten years, only little effort has been made to understand the problems of connecting them in order to efficiently manipulate geographic data on a display. This issue raises the general problem of coupling a DBMS with specialized modules (in particular, the problem of strong vs. weak integration), and more generally of the role of a DBMS in a specific application. After giving the functionalities that a GDUI should provide, we study the possible conceptual integrations between a GUI and a DBMS. Finally, a map editing model as well as a general and modular GDUI architecture are presented.

*A previous version of this paper has been accepted for presentation at AVI'94, Bari, Italy, June 1994.

Contents

1 Introduction

An efficient human-computer interaction is a task of increasing importance in many new database applications such as CAD/CAM, environmental management, or multimedia. In the area of *geographic information systems* (GISs), little effort has been made to provide both designers and end-users with appropriate tools for manipulating geographic data. We focus here on GISs using a *database management system* (DBMS) as a support. Although DBMSs sometimes provide basic features for handling geographic data, they are usually not designed to manipulate them graphically in an efficient way. In addition, only a few of them [2, 24, 14] offer a customizable and extensible *graphical user interface* (GUI).

As far as geographic applications are concerned, some work has been done in the area of spatial query languages (see [22, 7, 11]) and graphical manipulation [17, 8]. However, to our knowledge, nothing has been done to understand in detail the problems of designing GUIs for geographic applications when connected to a DBMS. One of these problems is that such a design involves geographic DBMS features as well as graphical tools. A lot of progress has been made in both areas independently, but the bridge between them still has to be built.

The goal of this paper is to investigate the problem of designing a *geographic database user interface* (GDUI), and to propose a toolkit-based solution that satisfies a large number of users. This work is based on our experiment: First, GOODS/O_2 [23], a GIS prototype built on top of the object-oriented system O_2 (version V1) [9], whose user interface [28] was developed by extending the O_2 regular interface generator *Looks* [19]. Second, an orthogonal approach consisting of a map editing kernel [20] connected to the geographic DBMS GeO_2 [6] developed on top of the O_2 product [2], whose basis was the map editing model presented in [29] and extended in [30].

GISs are concerned with many different fields, such as planning, resource management, traffic control, etc., which may have an impact on the functionalities of a GUI (data input, cartography, animation, use of graphic tools for statistics, etc.). We focus here on basic functionalities that such an interface should provide, namely map editing and querying. The geographic entities we consider are 2-dimensional (polygons), 1-dimensional (lines) and 0-dimensional entities (points).

This paper is organized as follows. In Section 2, we list the requirements of a GDUI for a better understanding of the problems. In Section 3, we discuss two design approaches radically opposed: In the first one a strong integration of the graphic components into the DBMS is considered, while in the second approach a map editing kernel as separated as possible of the DBMS is designed (weak integration). In Section 4, we propose a model for graphical map manipulation embedded into a general map editing kernel architecture, which shows our favorite solution for coupling a GDUI with a geographic DBMS. Finally, Section 5 draws some conclusions.

2 Map Editing and Querying Requirements

We now present the basic features that a GUI should provide for map editing and querying. The terminology not being standard in this domain, we first start with an informal definition of the terms used throughout this paper. The main objects we consider are *maps* stored in a database (sometimes called "thematic maps" or "themes" or "layers" in the literature). The geometry of a map can have either a vector form or a raster form. We focus here on maps whose geometry is stored in a vector format. Generally speaking, maps are collections of *geographic objects*. For instance, a country is a geographic object. A geographic object has usually two parts, an alphanumeric one, or *description* (e.g., the name and population of a country), and a spatial one corresponding to its geometry (e.g., a polygon or a set of polygons). In addition, a geographic object can be atomic or complex (made of other geographic objects). A city composed of streets and buildings is a complex object. A map is more than a complex geographic object since it includes information on data it contains such as the coordinates system, the source of information, etc. A *geographic database* is a collection of maps. Note the difference between a map as it has just been defined and the common term "map" that denotes a final (and frozen) representation, on paper for instance, of what is called a map here.

Map editing is concerned with two major aspects: (i) displaying and (ii) modifying maps. Displaying maps usually refers first to the display of the spatial part of their geographic objects. The corresponding displayed objects are called *cartographic objects*. The geographic objects' description can be displayed after pointing to cartographic objects on the screen.

As far as map querying is concerned, the problem is more complex than in the case of "standard" queries (purely alphanumeric) since map queries sometimes need graphic parameters. As a consequence, a specific interaction with the end-user has to be considered. For example, before answering the queries "What is the closest gas station from *this point*" and "Is there a city hall in *this area*?", a point or an area have to be drawn on the screen by the end-user.

We give below the requirements for (i) visually representing maps (map display), (ii) manipulating or updating maps (e.g., erase a boundary between two geographic objects) and (iii) interacting with the database (e.g., get the description of a geographic object or perform queries on maps).

2.1 Map Display

2.2 Basic Functionalities

Displaying maps includes the following basic functionalities:

- Display data of type raster (bitmap) or vector (point, line, polygon).

- Display alphanumeric data (description of a geographic object).

2.3 Visual Aspects

For a minimum of map understanding and map manipulation, the following is required:

- Overlay vector maps and raster maps (e.g., satellite images).
- Allow one to change scale (e.g., zoom) on a whole map or on a subset of a map.
- Attach a legend (color, pattern, text, symbol) to maps or subsets of a map.
- Display data with various scale and different representations (mix of the two previous points). A well-known related problem is the problem of generalization [16, 3]. For instance, it allows one to modify the representation of geographic data according to the scale (e.g., for displaying them in a smaller space). This means that new cartographic objects have to be computed. To give examples, the four following points may improve the clarity of a map display:
  - Give a new representation to a geographic object according to the scale. For example, a city that is a polygon in the database can appear as a point or an icon (new cartographic objects) depending on the scale and on design choices. In the worst case, if the scale is too small, the geographic object will disappear unless it has a "semantic importance" [20].
  - Simplify the contour of a 2-dimensional object (remove points on the border), or of a 1-dimensional object (drop points on a line).
  - Simplify the display of geographic objects by using the concept of aggregation when there exists subobjects that represent a partition of the plane. For example, on a map of states, if a state is made of counties, only the contour of the state (the geometric union of the counties' geometry) will be shown.
  - Simplify the display of a complex geographic object. For instance, on a map of cities, just represent a few buildings of each city. This shows that m geographic objects (the buildings that compose a city) can be represented as n cartographic objects (a selection of n buildings among m buildings).

2.4 Windows and Interaction with End-Users

Maps are displayed in windows with which users have a particular interaction. This includes:

- Map display (geometry):
  - Display several maps in a same window without overlaying them.
  - Possibly show a restricted part of a map (the rest can be seen by scrolling).
  - Overlay several views of the same geographic space in a same window (e.g., states, cities and highways).

  Although the first two tasks seem to be GUI issues rather than GIS issues, we will see later on that they need to be handled in a specific way in geographic applications.

- Interaction with the user on the display:

  - Allow an end-user to select objects on the screen (usually with a mouse).
  - Provide the possibility to draw areas on the screen (e.g., a rectangle for a zoom or for a spatial selection, see below).
  - Give the user the possibility to modify objects interactively (e.g., erase a boundary between two polygons).
  - Provide query language facilities, either graphically (e.g., [17]) or through a language (in a text editor) such as SQL extended to spatial concepts ([7]). Such a language has a tight connection with the GUI because of the interactive nature of queries in geographic databases. As said before, GIS queries often deal with objects displayed on the screen, for example: "What is the appropriate subway route to go from *here* (clicked on the screen) to home?" In an object-oriented environment, some queries corresponding to (algebraic) operations on maps such as clipping, windowing, map overlay, etc.(see [22]) can be defined as methods invoked from a GUI. They sometimes need a graphic argument as well (e.g., clipping and windowing need an area drawn on the screen by the user).

- At a meta level, provide (graphic) tools for browsing the database and the map library, for instance to select appropriate maps for a given study.

2.5 Interaction with a Database

Given the previous requirements, it is clear that the interface has to communicate with the database for at least:

- Getting the objects to be displayed: Maps (as a whole), i.e., the geometric part of their geographic objects but also their description (text editor).

- Attaching a representation such as color to a geographic object, depending on the information it contains. This is usually done by reading a *legend* stored in a database independently of maps. Then the corresponding cartographic objects are created.

- Updating objects (either their geometry or their description), i.e., changing the value of a geographic object in the database.

- Querying maps: (i) alphanumerical querying (e.g., SQL in a text editor), and (ii) graphical querying (transmit graphical arguments to queries).

The main communication functions between the user interface and the database are given below. Regarding the notation, in order to remain general we refer to the alphanumeric part of a geographic object as "description" (instead of string, etc.). A description is for instance the string: "Name:France, Population:57, Surface:550000". Functions that refer to database operations are prefixed by `db:`, while user interface functions are prefixed by `ui:` and functions that are the link between the interface and the database are prefixed by `connection:`.

- Display a map:
  `ui:MapDisplay(map)`.

- Get a geographic object (`go`) in a (database) map from the coordinates of a cartographic object (`co`):
  `connection:GetObject(map, co)` → `go`.
  `/* Assuming there may be several maps displayed on the screen */`

- Update a geographic object in a (database) map
  `connection:UpdateObject(map, go)`.

- Apply a (db) function `f` on a map:
  `db:f(map, args)` → `result`, where `args` is a list of either alphanumerical or graphical arguments and `result` the result of a query (same thing).

3 Strong vs. Weak Integration: Two Design Approaches

The user interface of a geographic application can be considered as a kernel *independent* of a DBMS (although it has to be connected to it). This holds for many specific applications, whose particularity is to perform treatments that are usually out of the scope of the DBMS. As an example, we can cite multimedia applications such as hypertext, applications dealing with statistics whose data are persistent, etc. In the following, we refer to the set of specific treatments (statistics, expert system, planning, traffic analysis, etc.) as "external module". In this section, we first consider the general problem of connecting an external module to a database, then we study such a connection in the context of GDUIs. We insist on the fact that our goal is not to develop in detail DBMS aspects such as openness and extensibility, but rather is to describe them with the objective of GDUI design in mind. This is also the reason why we focus more on conceptual aspects than on technical aspects.

3.1 Connecting an External Module to a DBMS

When an external module is used together with a DBMS, one question that arises concerns the division of tasks among the DBMS and the external module. The functionalities of a DBMS are limited and it is sometimes not clear whether some specific tasks should be performed by it or by an external module. Should the DBMS be in charge of performing complicated and expensive algorithms as in the case of statistics, CAD/CAM, planning, etc.? Or should these algorithms be performed by a more appropriate module that is optimized for a given treatment? This question is especially relevant in the case of object-oriented DBMS where there exists a single language both to define and to manipulate data (no "impedance mismatch"). A more general question related to it in this case is: Which methods should really be associated with an object? Unfortunately, we believe that there is no real answer to this question, since it depends on particular problems and on the openness of the systems involved. In the context of geographic databases, which require for example robust geometric algorithms, this issue is particularly important.

There exist basically two different ways to define a specific application within a DBMS environment, i.e., to connect an external module to a DBMS: Either the integration is *strong* (no real external module, Figure 1.a) or it is *weak* (Figure 1.b). Note that the way from strong to weak integration is gradual. For instance, a solution in between is to consider an extensible DBMS (i.e., a DBMS with hooks for extensions at different levels, see [12]), so the DBMS plays the role of an integrator. In the case of strong integration, the potential of the underlying DBMS is exploited as much as possible, and all concepts are embedded within the same homogeneous environment. In particular, only one data model exist, and if the (virtual) external module needs special structures, these structures will be defined using the database model. This approach was chosen in [1] for instance.

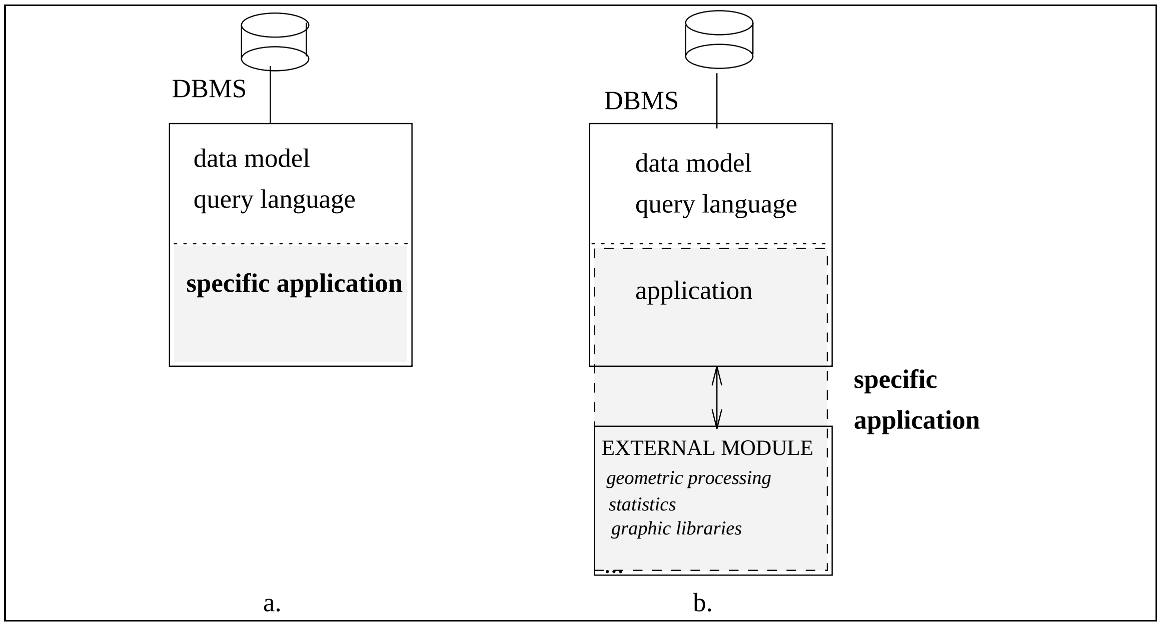

Figure 1: Strong vs. weak integration

In the case of weak integration (Figure 1.b), two independent systems coexist: The DBMS on the one hand and the external module on the other hand, and the main problem is to build the connection between them. As mentioned above, another problem concerns the repartition of tasks among the two systems. In the case of GDUIs for example, it is not clear what should be left to the GUI module and what should be defined in the geographic DBMS. For instance, consider the problem of efficiently finding an object on the screen, and suppose the existence of an external package for displaying graphic objects, for instance the C++ graphic library Ilog Views [13]. Ilog Views defines indices on graphic data (thus in the GUI), although any geographic DBMS kernel now encompasses such a notion. Moreover, it is usually more adequately defined in a DBMS since it is made especially for geographic data. Which index mechanism(s) should then be used? Is it really worthwhile rebuilding index structures at the GUI level (which may slow down the performances)?

3.2 Integrating a Graphical User Interface into a DBMS

Compared to other possible external modules, geographic user interfaces have a tight connection with a DBMS since, among other things, they reflect the contents of the database at a given time and show a view of database objects. A few database environments provide tools, unfortunately not always sufficient, to design a graphic user interface. Then for designing a GDUI, the problem of choosing a type of integration still remains, and the two opposite approaches mentioned above can be considered: Either graphic concepts are embedded into the DBMS environment (such as in [23]) (Figure 2.a), or they are completely independent (such as in [29], Figure 2.b). In Figure 2.b.2, the GUI objects are stored in a database, which is not necessarily the one used to store the geographic objects. In the first solution, the cartographic objects of Section 2 correspond exactly to the database geographic objects, while in the second approach they are totally independent. Hence they could be defined and manipulated using an external graphic package (by defining first a mapping onto the external language that deals with graphic routines).

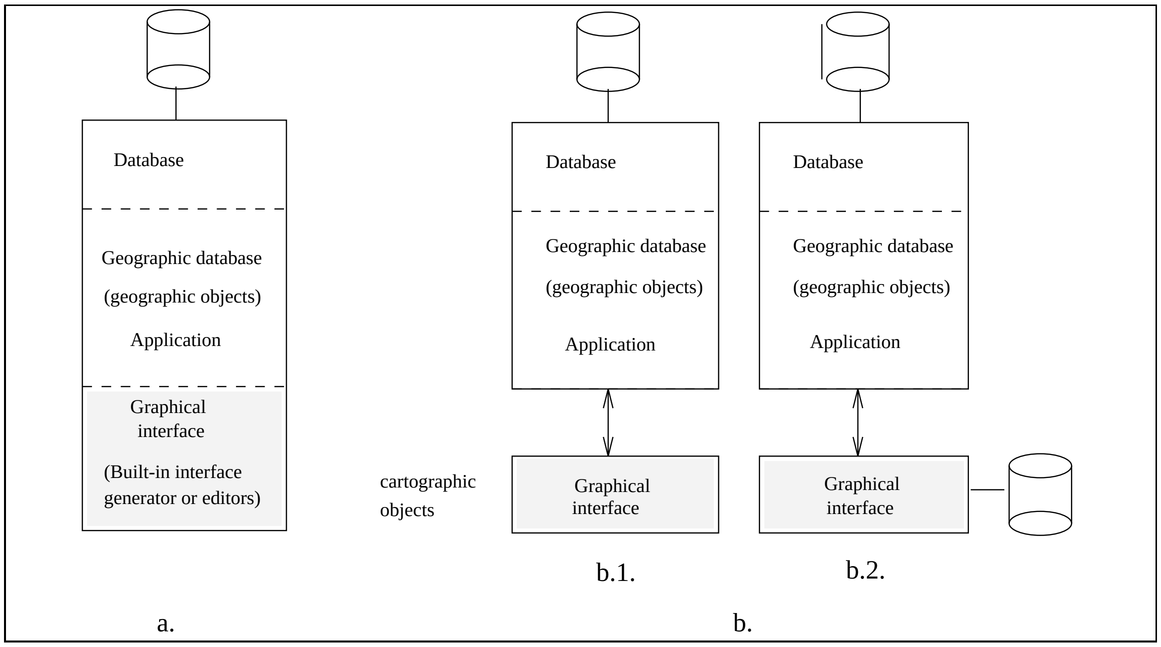

Figure 2: Integrating a user interface with a DBMS

We now detail these two possible integrations of a GDUI into a DBMS, and discuss their advantages and drawbacks while designing a GDUI.

3.2.1 Strong Integration into a DBMS

In this approach (Figure 2.a), there is no real distinction between geographic objects and cartographic objects since cartographic objects are *the* (unique) views of geographic objects. In other words, there is only one environment to deal with, and the geographic application is driven by the DBMS. We distinguish two cases: Either the DBMS provides an interface generator (Case 1), which allows one to design editors for new types of data such as geographic data, or no tools are provided and display routines (map editors) need to be defined in the application (Case 2). As an example of Case 1, we can cite the O_2 prototype [9] with its interface generator *Looks* [19]. *Looks* offers editors for each basic O_2 type (string, integer, bitmap, etc.) and for each O_2 constructor (tuple, set, list) called recursively, thus each O_2

object, even complex, can be displayed. There is only one way to display an object (the idea is "*One* object from the disk to the screen"). On the other hand, the geographic application GeO$_2$ [6] (built using the O$_2$ product [2]) is an example of Case 2: With some GeO$_2$ objects is associated a "display" method programmed by the GeO$_2$ developers.

In Case 1 (existence of an interface generator), the advantage is that the developer's task is obviously facilitated. S/he needs not to consider communication aspects between the interface and the database for getting and updating objects because the framework to do that already exists. In Case 2 (no interface generator), the application must include routines for displaying objects. For instance, in the basic GeO$_2$ system, displaying objects is done via methods on geographic objects that invoke Xlib routines.

However, strong integration suffers from major drawbacks. In both previous cases, the DBMS application gives an object the "order" to display itself on the screen, and everything is done within the database environment which is a complete geographic DBMS. It is clear that this *ad hoc* solution is data model dependent. The interface can be considered as an application on top of the DBMS, which leads to limits if the model or the application change. Moreover, it seems difficult to consider many different databases simultaneously, although this is useful in geographic applications when data from different sources are to be merged (say for instance a set of maps stored at a big scale in some DBMS and another set stored at a small scale in another DBMS). In addition, in geographic applications, it is not unusual to have data stored under different systems (e.g., Unix and MS/DOS).

Regarding map display functionalities, the DBMS environment must have an elaborate knowledge of windows in which objects are to be displayed. If the mechanism for displaying objects (e.g., maps) in windows is "frozen", map display turns out to be problematic. Take for instance the *overlay* of two maps (two maps have to be displayed in the same window). If the DBMS display mechanism is such that there is a bijection between windows and objects to be displayed (as in *Looks*), such an overlay is impossible and the regular display mechanism has to be bypassed. This is also the case when a database operation needs a graphic argument. As an example, take the call for methods function of *Looks*. In "standard" O$_2$ applications, the arguments of methods are either objects or alphanumeric values, hence they are supposed to be entered via a regular text editor. The *Looks* function that handles method calls takes care of (i) getting the argument(s) in a text editor, (ii) checking the type and (iii) transmitting it to the DBMS. Now consider a method *clipping* on maps. The argument of this method is a rectangle drawn on the screen by the end-user and thus cannot be entered via a text editor (or this is not user friendly). Then the entire *Looks* call for methods has to be reprogrammed in order to deal with graphic (interactive) argument in addition to alphanumeric (standard) arguments. This is a tedious task because it concerns low level functions of the interface generator.

The previous examples illustrate the major problem of this approach, namely the lack of flexibility in the display mechanism. This is partly due to the philosophy "A single object from the disk to the screen", which is not suitable for displaying geographic objects. In addition, it makes it tedious to handle generalization where m geographic objects become

n ($m > n$) cartographic objects (see Section 2). Unfortunately, given the lack of choice in the programming tools, the designer is dependent on both the existing graphic tools and the display mechanism and must adapt to the existing philosophy. If no interface generator is provided (Case 2), there is more flexibility in the programming, but no flexibility in the design either, or the programmer has to master the low levels of the DBMS (with possible use of system primitives) together with graphic routines.

3.2.2 Weak Integration: Separated Conception

In this approach (Figure 2.b), the user interface is developed separately and is eventually connected to a particular DBMS, usually in an ad-hoc manner. The user interface can be developed using graphic packages (such as GoPATH [5], IlogViews [13], ...). The application is driven by the interface.

Compared to the previous approach, the advantages here rely on the fact that the interface is adapted to the needs of geographic applications. From a design point of view, there is a total independence between the database and the interface, and the designer needs not use neither a given a display philosophy nor imposed graphic tools. S/he can take advantage of efficient graphic routines defined in a specialized package, which leads to a more powerful interface than in the case of strong integration. New tools can easily be added, such as sophisticated editors for statistics.

The main drawback of this approach is that the connection with the database needs to be defined by the developer (e.g., for getting objects, querying, sending data flows.). It is now widely recognized that coupling systems is cumbersome for many reasons, from a logical level (type mapping) to a physical level. For instance, if the geographic interface is developed from scratch, a low level communication dialog has to be defined (e.g., using Unix sockets). Moreover, if the DBMS changes, the communication layer has to be reprogrammed. To implement one of the first user interface of the Sequoia project [25] built on top of Postgres [26], a *visualization software package* (AVS) was chosen, which required the writing of an AVS-Postgres bridge (obviously very specific). A more modular solution was chosen in the Godot system [10], developed on top of ObjectStore [15], whose GUI uses the GoPATH [5] package together with an "interactive interface" level.

Moreover, as mentioned in the beginning in the case of index structures, it is sometimes not clear what actions should be performed by the DBMS and by the GUI. As another example take the zoom operation. Should it be programmed at the interface or at the database level?

3.3 Conclusion

Given the requirements of geographic applications and the major drawbacks of the two solutions presented above (the lack of flexibility of the first one and the integration problems of the second one), an alternative solution would be a sort of mix corresponding to the second solution *together with well defined general interfaces* with a DBMS. This means that we tackle the problem at a higher level, and that the solution we advocate is to offer tools for developing customizable interfaces within a general database environment. The user interface is developed separately using graphic tools. There still exists a DBMS data model behind the geographic application but this is transparent to the user if there is a module for making the junction between the two worlds. This is the topic of next section.

4 A Modular Approach for Editing Maps

In this section, we propose a general model and a GDUI architecture for editing maps. This proposal is independent of any underlying database model but is supposed to cooperate with one or many. The architecture presented below was used in previous development [20]. After describing the main modules of the GDUI, we come back to map display and we detail some features necessary for "advanced" map display.

4.1 A Model for Editing Maps

Before presenting the editing model itself, we briefly describe maps in the database. It should be stressed that, in order to be independent on any database model, the map model described here is purely conceptual (and is intended to be implemented according to a particular database model). We then present the basic entities of our editing model: Image, stack of images, presentation, mapget. Because our purpose is to present in a simple way a general model that encompasses the requirements given in Section 2, technical specifications are not given here.

4.1.1 Maps in the Database

A *map* is a *collection of geographic objects.* A geographic object is usually derived from an entity of the real world. It can have a spatial part as well as a (alphanumeric) description. For instance, a river is characterized by its name, its flow (both representing its description) and its geometry (a polyline). Let us summarize these basic notions in the following definitions, in which { ... } denotes a collection of objects.

```
map = (information,{geographic-object})
geographic-object =  (description, geometry)
```

A geographic object may be composed of several other geographic objects. For instance, a branch of a river can be isolated to form a geographic object. In this case, a river is a complex object whose branches are atomic objects.

The notion of composition can be introduced in the objects definition itself. This consideration leads to the following more elaborate definition of geographic objects:

```
geographic-object = (description, geometry)  /*atomic geographic object*/
                  | (description,{geographic-object}) /*complex geographic object*/
```

This representation permits to associate a geometry with only atomic objects. One then avoids duplicating the geometry of (complex) objects in the database when it can be inferred from the lowest levels (the atomic objects).

At implementation time, the geometry of geographic objects usually corresponds to spatial data types. These are easily implementable using an extended-relational or an object-oriented DBMS [30]. In general, the following spatial data types are defined: Area (2-dimensional), line (1-dimensional) and point (0-dimensional), and operations such as intersection, adjacency, etc. are associated with these types.

In Section 2, the notion of raster data such as satellite images was mentioned. In the sequel, we shall call the corresponding entities *raster-images*. They are stored in a database as well, but their manipulation within a DBMS environment is out of the scope of this paper.

4.1.2 Images

An *image* is an interface object that corresponds to *a* view of a map to be displayed (note that it has a vector form and has nothing to do with the "raster-images" mentioned above). A (database) map may have many images. An image is constituted of objects derived from a mapping of the database geographic objects with which is associated a type of representation (basically, a type of legend, see Section 4.3.1). For instance, a city of the database can appear as an icon or a point together with a label, depending on the scale and on user choices. The objects of an image are called *cartographic objects*. Since an image requires some time to be computed, it is sometimes worthwhile storing it in a database for other sessions (Figure 2.b.2).

4.1.3 Stack of Images

Views of maps (images) are stacked in windows as shown in Figure 3. The main object considered for this purpose is an *image-stack*. It has a crucial role since it is in particular in charge of the visibility of a map display (see below). The use of such a structure allows a geographic object to have many different representations on the screen. For instance, take the first two layers of Figure 3 (labels and countries) and the geographic object "the country Spain". One can see two views of this object, its name, "Spain", and its geometry (a polygon), and one could even imagine other representations (such as a particular color if this object belongs to the result of a query, etc.).

Operations on stacks such as shift-up or shift-down can be found in [30].

4.1.4 Presentation

A *presentation* is a view of an image-stack at a given time. It can be the drawing of a map with symbols such as labels or icons (Figure 3). It is composed of *graphic objects*, which are not relevant here because they are implementation dependent. In the prototype [20], presentations correspond exactly to images (no notion of stack). They are implemented as X/Motif "pixmaps", i.e., raster structures, for performance reasons (in particular, the

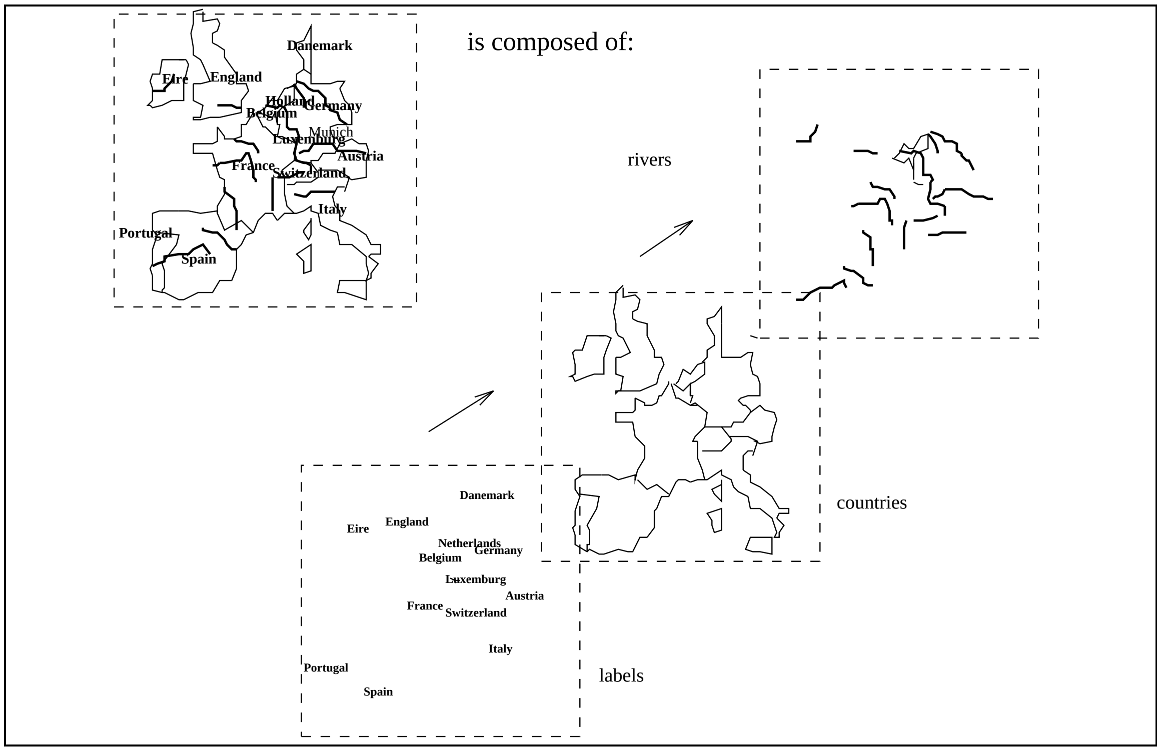

Figure 3: Overlaying maps: Stack of images

scroll and simple zoom operations already exist on pixmaps). However, given the lack of semantics of such a trivial structure, using an efficient vector object is more advisable.

4.1.5 Mapget

A *mapget* (shortcut for "map widget") is a (X) window in which maps are displayed. A mapget can be composed of other mapgets. In this case it is a *compound mapget*, otherwise it is a *simple mapget*. With a simple mapget is associated a presentation that constitutes a resource of the window:

```
mapget = simple-mapget | compound-mapget
simple-mapget = (rectangle, presentation)
compound-mapget = (rectangle, <mapget>part),
```

where `< >part` represents a partition constructor not detailed here. The rectangle in the compound mapget is optional and can be inferred from the rectangles of the mapgets that compose it. Its value can also be checked by integrity constraints.

More precisely, a compound mapget is a tree T whose internal nodes are compound mapgets and leave nodes are simple mapgets (Figure 4). The root of T is the largest window. There is an edge from w to s in T if s is a window included into w. An *application* is a collection of mapget trees.

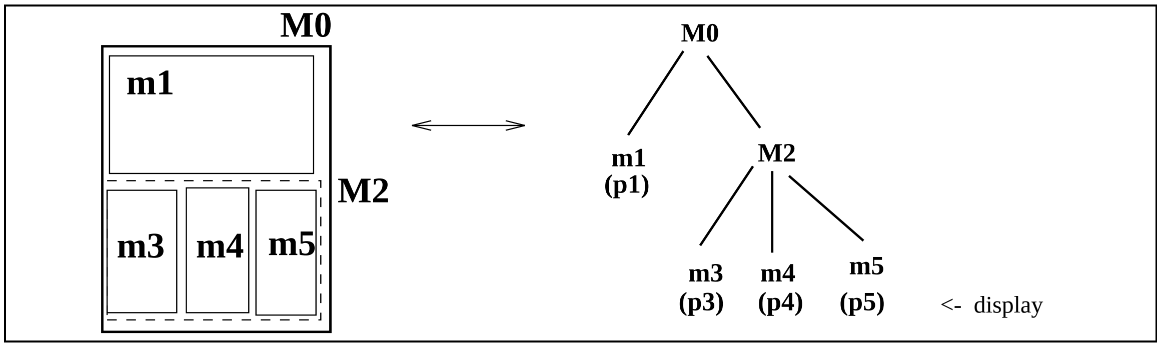

Figure 4: Tree structure for compound mapgets

The definition above could be a general definition of (compound) windows. However, in the case of geographic applications, such windows require special attention. If some operations on windows do not have any effect on their contents (e.g., *move*, *raise*, *hide*), other operations require modifying the contents of a window (e.g., *resize*, *scroll* and *zoom*) and thus a specific definition of these operations needs to be given for map display.

Mapgets that belong to the same tree may have to communicate, for instance in case of modifications or updates. Recall (Section 2) that many different views of maps can be displayed in different windows. On Figure 4, suppose that the map of European states is displayed in mapget $m1$ and that the map of France is displayed in mapget $m3$. Modifying the France object in $m1$ or $m3$ may affect the contents of the other one (basically, the whole process of map display has to be called again). This does not necessarily happen because one of these two maps may be a "working" map, and modifying an object does not necessarily imply updating it (cf. Emacs buffers). Hence it may be possible to define a link between images of different presentations, but this should not be a default solution. A control mechanism is in charge of respecting the consistency among mapgets of a same tree if necessary.

Maps displayed on the screen are overlaid in image-stacks of simple mapgets (Figure 5). They correspond to the clipping of a presentation with the rectangle of a simple mapget.

4.2 Architecture of a Geographic Database User Interface

Figure 6 shows the architecture of a GDUI kernel.

At the support level, two main environments coexist: On the one hand, graphic tools for displaying maps, and, on the other hand, a (geographic) DBMS. The GDUI communicates with both environments and is composed of three modules: **Mapget, Main, Connection**. Mapget is in charge of displaying maps in windows (presentations). Main is in charge of dealing with images and it communicates both with the mapget module and with the connection module.

When a *map* (stored in the database) is to be displayed, it has first to be converted into

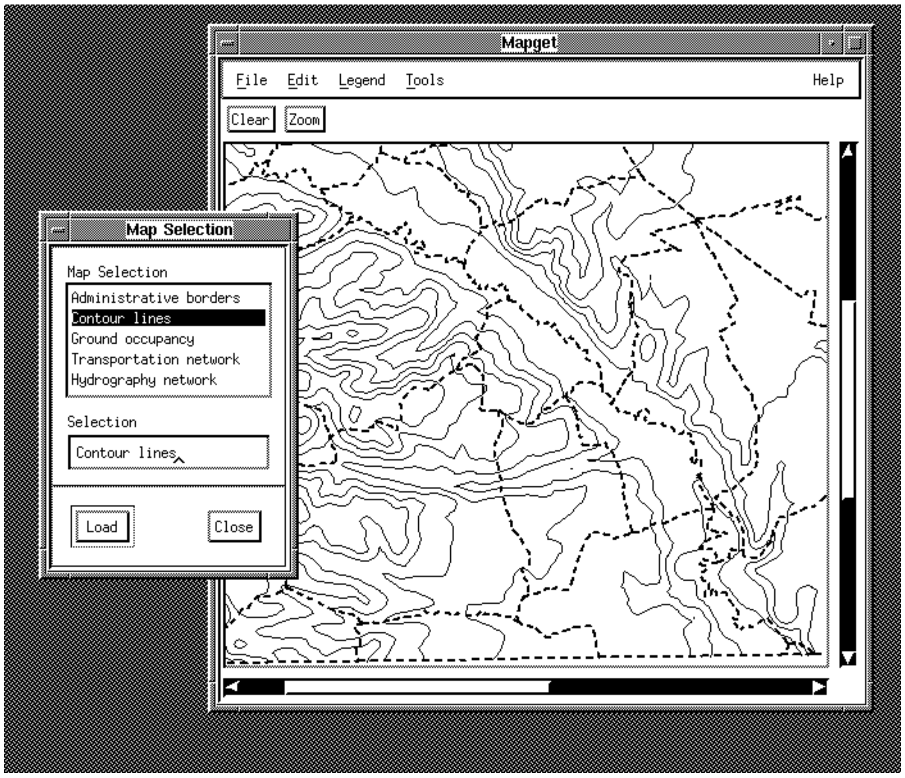

Figure 5: Example of map display

an *image*. This is performed by the connection module, which knows both data models. This image is handled by the main module, kernel of the GDUI. The connection module, which is the interface between the DBMS and the main module, is in particular in charge of handling all exchanges with the DBMS when displaying and querying. In our previous prototype, this was done by *O_2link* [27], which handles exchanges between C++ and the O_2C programming environment using *import* and *export* functions. It is clear that the connection module can be used for other applications (e.g., applications using statistics), in order to make the bridge between an external module and a DBMS. To be generic, the connection module should support two functionalities:

- Communication with the DBMS (queries, data flows).
- Mapping from database objects onto external (GUI) objects.

Regarding the second point, such a mapping has to be defined at a high level and should allow one to switch from one representation (a database model) to the other (the interface model), as illustrated in Figure 7. This can be realized by a layer having a unified view of the two worlds, and could be defined using a language such as the FGD functional language [4]. Note that such a mapping is not an "1-1" mapping, but rather is a "$m - n$" mapping (see Section 2).

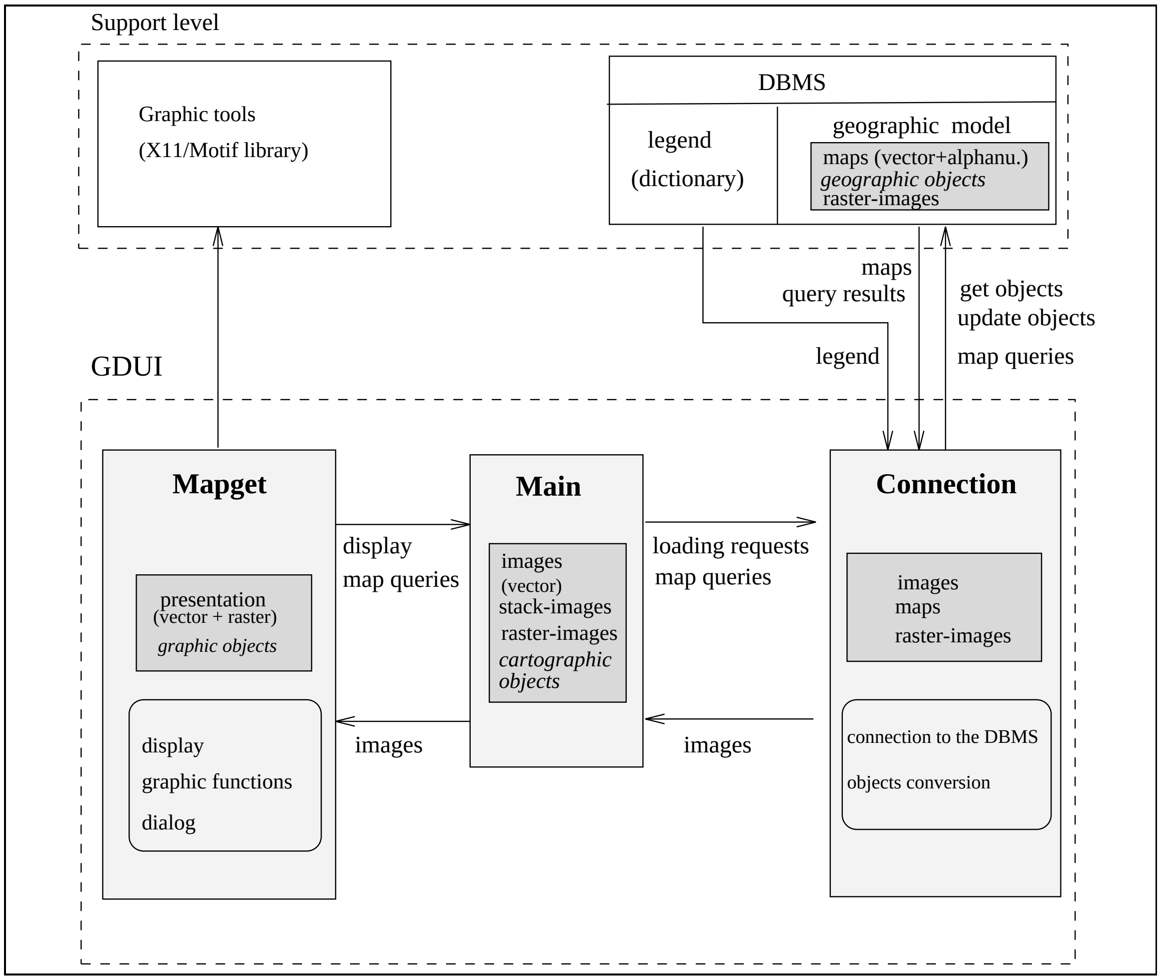

Figure 6: Architecture of a geographic database user interface kernel

4.3 Back to Map Display

The model described above allows one to display maps in mapgets. We now come back to two functionalities: The overlay of maps in a single window and the multi-representation of geographic objects, and explain them using our model. Finally, we illustrate these concepts through a brief presentation of the map display mechanism.

4.3.1 Overlaying Maps

Maps and raster-images are displayed in mapgets with two main constraints: (i) The scale must be the same for all maps (and raster-images) displayed within the same mapget and (ii) the drawing must be readable. In particular, the legend of different map layers should not interfere. It is obviously difficult to automatize the realization of such a constraint because it is related to human perception.

In [29] four types of legend are proposed: *Name* (or label), *symbol* (or icon), *statistics*

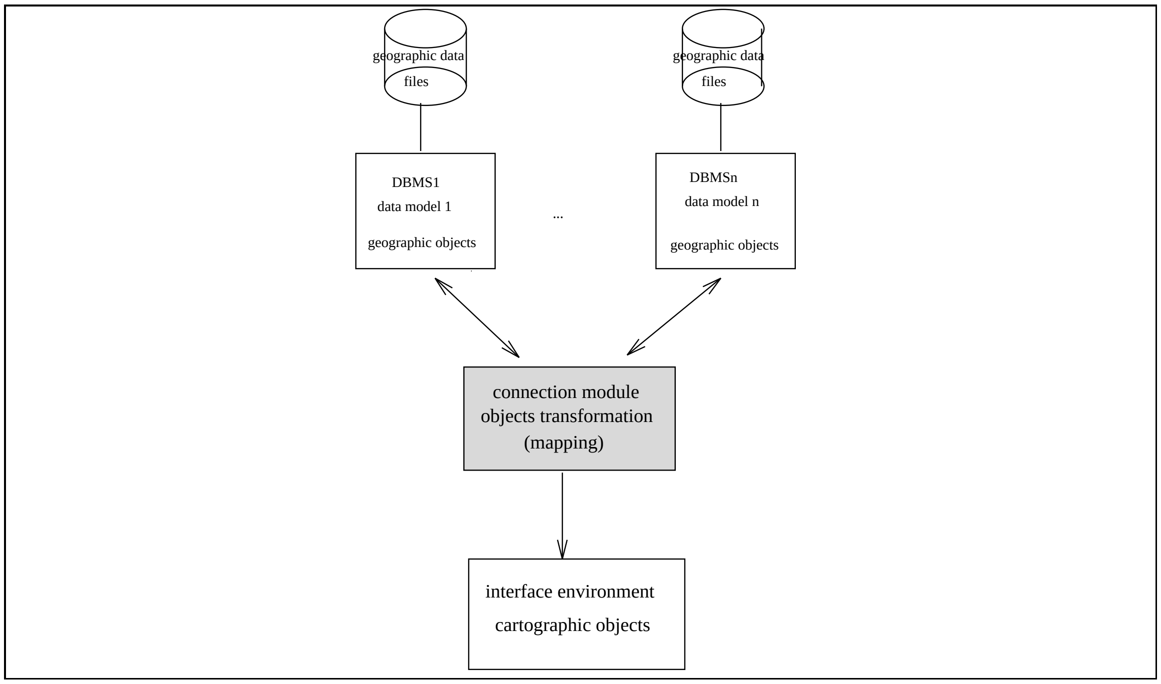

Figure 7: One of the roles of a generic connection module

and *theme* legend. All these legends are stored in the database (for consistency reasons), usually in the form of a dictionary that associates a representation with a type of objects For instance, the drawing of a church stored as a bitmap is associated with the object type "church" in the symbol legend[1]. A legend is meant to be integrated into a cartographic object (basically, this is realized by a join between the description and the legend). Some of these legends can be visualized directly on the final presentation. This is the case for names, symbols and statistics, although the statistics legend is somehow more complicated to handle for other reasons. From a visibility point of view, the only legend that could be problematic is the theme legend since themes cannot be overlaid when the representation associated with a type of object is a color: How can a blue lake be drawn on top of a yellow corn field? To simply solve this problem, we consider that images with a "colored" theme legend put on top of each other are not compatible. But the theme legend does not only deal with colors for the representation of objects. For instance, in Figure 3, the image of the middle is shown with a "non-colored" theme legend (borders of countries): The polygons are not filled.

4.3.2 Multiple Representation Display

In the context of map display, the problem of multiple representation is that one geographic object in the database, e.g., a city, has many possible representations on the screen (e.g., a polygon or a symbol or a point or even a subset of its subobjects such as a set of buildings).

[1]Note that "church" is a value rather than a type. But if this value does not appear explicitly (no attribute "kind-of-object"), then it makes sense to consider it as an object type. One should perhaps say an "object kind" instead.

This is very common in cartography for example, where different map representations are created from the same set of information (e.g., according to different scales). This is an active research topic and not the main focus of this paper, but it is definitely something to keep in mind while designing a GDUI.

These various representations may occur either simultaneously in a mapget (e.g., the icon of a city on the screen together with its name), or at different times (the city becomes a point because the scale has changed). To affect many representations to an object, two cases are considered: (i) Either all the possible representations are stored in one or in many databases (for instance, databases with different scales), or (ii) the new representation has to be computed when an object is to be displayed at a certain scale. For example, an object can become a symbol, or an object whose geometry has been simplifyed (erase points on a polygon, see Section 2), etc. A way to implement this feature is to perform a treatment at the interface level, which may use the symbol legend to affect an object a new representation. Such a treatment works like a filter and is not detailed here. In our model, this is handled when an image is to be displayed. This mechanism is also applied when a "semantic zoom" (as opposed to magnifying glasses) is performed within a mapget.

4.3.3 Map Display Mechanism

To summarize the previous points, map display requires the following steps:

- Build a *stack of images*:
  - Get a map in a database.
  - Convert the map (geographic objects) into an image (cartographic objects) and incorporate a legend into it.
  - Put the image into the stack. Note that many images corresponding to the same geographic space, but having a different legend, can appear in the same stack.
- Define a window (*mapget*) and attach a resource *presentation* to it, which corresponds to the previous image-stack:
  - Define a global size for the presentation
  - For each image of the image-stack:
    - Apply a possible filter on each cartographic object and put it into the presentation.
- *Display* the presentation in the mapget (clipping of the presentation with the mapget's rectangle).

It is worth noting that even though maps are prestored in the form of images (for the sake of independence with the DBMS, consistency, and to save time in other sessions), changes "on-the-fly" are allowed on presentations. These changes may need information stored in a database (e.g., the legend), which is not a problem because of the existence of the connection module.

5 Conclusion

In this paper, we studied the problem of designing the user interface of a geographic application and of integrating it into a DBMS. We first presented the specific needs of geographic database user interfaces (GDUIs), with emphasis on map editing and querying. We then studied the connection problem itself at a conceptual level, and we saw that it is related to the more general problem of connecting an external module to a DBMS. This can be solved in two different ways: Either by strong integration, which means that there is a single homogeneous system to consider and that everything is done according to the DBMS features (this is particularly suitable for object-oriented DBMSs), or by weak integration. In the second case, two separate systems coexist, the DBMS and the external module. For modularity reasons, we are in favor of the second alternative, although we then have to solve the problem of communicating efficiently with the DBMS.

As far as GDUIs are concerned, we believe that a very promising solution is to define a generic graphic interface independently of the DBMS (in particular of the database model), and to provide conceptual tools for mapping both environments. The GUI is defined using an appropriate graphic package. An important problem is then to define properly the most suitable division of tasks among the DBMS and the GUI (in other words, which actions should be performed in the database and which ones should be left to the interface). This problem is related to the classical time-space trade-off, which is especially relevant here because of the size of the objects involved.

We finally proposed a model of a map editing kernel as well as a general architecture for GDUIs. This architecture has been used in a recent prototype [20], using on the one hand the O_2 DBMS [2] and the geographic layer GeO_2 [6], and on the other hand the X11/Motif environment [21, 18]. We now plan to implement the map editing kernel using more sophisticated tools for displaying graphic objects (e.g., Ilog Views [13]), to integrate a powerful multiple representation mechanism, and to finally study the more general problem of integrating within the same environment a GDUI, many geographic DBMSs as well as other specialized systems (statistics package, expert system, etc.). We believe that solving this integration problem represents a crucial step towards efficient geographic data manipulation.

References

[1] B. Amann, V. Christophides, and M. Scholl. HyperPATH/O_2: Integrating Hypermedia Systems with Object-Oriented Database Systems. In *Proc. of the DEXA Intl. Conf.*, 1993.

[2] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object-Oriented Database Systems: The Story of O_2*. Morgan Kaufmann, San Mateo (CA), 1991.

[3] K. E. Brassel and R. Weibel. A Review and Conceptual Framework of Automated Map Generalization. *Intl. Journal on Geographical Information Systems (IJGIS)*, 2(3), 1988.

[4] M. Breunig and A. Perkhoff. Data and System Integration for Geoscientific Data. In *Proc. Intl. Symposium on Spatial Data Handling (SDH)*, 1992.

[5] Bull, France. *GoPATH 1.2.0, Documentation Volume of the Public Release*, 1993.

[6] B. David, L. Raynal, G. Schorter, and V. Mansart. GeO2: Why Objects in a Geographical DBMS? In D. Abel and B. C. Ooi, editors, *Advances in Spatial Databases (SSD'93)*. Lecture Notes in Computer Science No. 692, Springer-Verlag, Berlin, 1993.

[7] M. Egenhofer. *Spatial Query Languages*. PhD thesis, University of Maine, 1991.

[8] M. Egenhofer and A. Frank. Towards a Spatial Query Language: User Interface Considerations. In *Proc. Intl. Conference on Very Large Data Bases (VLDB)*, 1988.

[9] O. Deux et al. The Story of O2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), 1990.

[10] O. Günther and W.-F. Riekert. The Design of GODOT: An Object-Oriented Geographic Information System. *IEEE Data Engineering Bulletin*, 16(3), 1993.

[11] R. H. Güting and M. Schneider. Realm-Based Spatial Data Types: The ROSE Algebra. Technical report, Fernuniversität, Hagen, Germany, 1993.

[12] L. Haas and W. Cody. Exploiting Extensible DBMS in Integrated Geographic Information Systems. In O. Günther and H. J. Schek, editors, *Advances in Spatial Databases (SSD'91)*. Lecture Notes in Computer Science No. 525, Springer-Verlag, Berlin, 1991.

[13] Ilog. Ilog Views Version 1.1, Reference Manual, 1993.

[14] Ingres. Windows L4G. Technical report, Ingres Corporation, 1990.

[15] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore Database System. *Communications of the ACM*, 34(10), 1991.

[16] W. Lichtner. Computer-Assisted Process of Cartographic Generalization in Topographic Maps. *Geoprocessing*, 1, 1979.

[17] M. Mainguenaud and M. A. Portier. CIGALES: A Graphical Query Language for Geographical Information Systems. In *Proc. Intl. Symposium on Spatial Data Handling (SDH)*, 1990.

[18] OSF. Motif 1.0 programmer's guide. OSF Journal, 1989.

[19] D. Plateau, R. Cazalens, D. Lévêque, J.C. Mamou, and B. Poyet. Building User Interfaces with the LOOKS Hyper-Objects System. In *Proc. Eurographics Workshop on Object-Oriented Graphics*, 1990.

[20] P. Rigaux, M. Scholl, and A. Voisard. A Map Editing Kernel Implementation: Application to Multiple Scale Display. In A. Frank and I. Campari, editors, *Spatial Information Theory (COSIT'93)*. Lecture Notes in Computer Science No. 716, Springer-Verlag, Berlin, 1993.

[21] R.W. Scheifler and J. Gettys. The X Window System. *ACM Transactions on Graphics*, 5(2):79–109, 1986.

[22] M. Scholl and A. Voisard. Thematic Map Modeling. In A. Buchmann et al, editor, *Design and Implementation of Large Spatial Databases (SSD'89)*. Lecture Notes in Computer Science No. 409, Springer-Verlag, Berlin, 1989.

[23] M. Scholl and A. Voisard. Object-Oriented Database Systems for Geographic Applications: An Experiment With O_2. In *The O_2Book*, F. Bancilhon, C. Delobel, P. Kanellakis (Eds.), Morgan Kaufmann, San Mateo, California, 1992.

[24] S. Seitz and P. Schank. Picasso Widget Writer's Guide. Technical report, UC Berkeley - ERL, 1990.

[25] M. Stonebraker, J. Frew, and J. Dozier. The Sequoia 2000 Project. In D. Abel and B. C. Ooi, editors, *Advances in Spatial Databases (SSD'93)*. Lecture Notes in Computer Science No. 692, Springer-Verlag, Berlin, 1993.

[26] M. Stonebraker and L. A. Rowe. The Design of Postgres. In *Proc. ACM SIGACT-SIGMOD*, pages 340–355, 1986.

[27] O2 Technology. O_2 Link, 1992. Chapter 11 of the O_2 Documentation.

[28] A. Voisard. *Programming Map Editors in GOODS/O_2 using Looks*. Internal report, projet Verso, INRIA. In French, 1990.

[29] A. Voisard. Towards a Toolbox for Geographic User Interfaces. In O. Günther and H. J. Schek, editors, *Advances in Spatial Databases (SSD'91)*. Lecture Notes in Computer Science No. 525, Springer-Verlag, Berlin, 1991.

[30] A. Voisard. *Geographic Databases: From the Data Model to the User Interface*. PhD thesis, University of Paris at Orsay and INRIA, 1992. In French.

Appendix: Specifications of the Editing Model

We give below some more precise specifications of our proposal as we plan to implement it. This part is meant to complete Section 4 and we restrict our attention to the most relevant objects and operations. Since the model we propose is part of a (geographic) database environment, we describe the database level as well. Specifications at the connection level are also given. The following specifications are written in a object-oriented pseudo code. For each class we give (i) the definition of the class and (ii) the associated methods. The receiver of the method is denoted by *self*. @ denotes the application of a method. Access to an attribut is denoted →.

5.1 Database Level

5.1.1 Maps

```
/* below is a simple generic model that is intended to be implemented
   using a DBMS model */
class Map = [ map-info: MapInformation,
              setgeographicobjects: { GeographicObject } ]

/* MapInformation represents information at a metalevel, for instance
   the coordinates system, the projection used, and some other general
   features of a map such as the biggest coordinates, etc */

class MapInformation = [ xmax-map: int, ymax-map: int]

class GeographicObject = [ description: list(Attribut),
                           atom-or-complex: AtomOrcomplex ]

class AtomOrComplex;

class Atom inherits Atom-or-complex = [geo: Geometry]

class Complex inherits Atom-or-complex = [ setgeographicobjects: {AtomOrComplex}]

/* Generic description of an alphanumeric attribute */
class Attribut = [att-name: String];

class IntAttribut inherits Attribut = [value: integer];

class StringAttribut inherits Attribut = [value: string];

class Geometry = [type-geometry: string];

class Zone inherits Geometry = [setpolygon: {Polygon}]
class Line inherits Geometry = [setseglines: {LineSeg}]

class Points inherits Geometry = [ setpoint: {Point}]

class Point = [ x: real, y: real]
class LineSeg =  [ beg: Point, end: Point]
class Polygon = [ listlineseg: list(Lineseg)]

method adjacent (g: Geometry) on class Geometry: boolean;
/* method adjacent can be redefined at lower levels, for instance
in ``method ZZadjacent (Z: Zone) on class Zone'', ``ZLadjacent (L: Line)'',
etc.
*/
```

5.1.2 Legend

```
/* theme, name, symbols */

class Legend = [type-legend: string]

/* ThemeLegend:
   for each theme give the representation of each attribute value
   ``detail level vs. metalevel'' */
class ThemeLegend inherits Legend: {[theme: {[attribute-value: string
                                               representation: RepTheme]}
                                     ]}

/* Textlegend:
   For each text to appear on the screen give the font, etc. */
class TextLegend inherits Legend: {[which-attname: string,
                                    /* which-attname optional bc it
                                       appears in an image  */
                                    text: RepresentationTex]}

/* SymbolLegend:
   With each application-relevant ``kind of object'' such as bridge,
  church, etc.   is associated a symbol
  ``metalevel  vs. detail level'' */
class SymbolLegend inherits Legend: {[attribute-value: string,
                                      symbol: RepresentationBitmap]}

/* A-Representation is the common class to all representations;
   it's empty here but could have attributes such as where
   (which DB) to find a representation, etc. */
class A-Representation;
class RepTheme inherits A-Representation = [rep: [color: string,
                                                  pattern: string
                                                  linestyle: string
                                                  thickness: int]]
class RepText inherits A-Representation = [rep: string,
                                           font:string]
class RepBitmap inherits A-Representation = [ rep: Bitmap]
```

5.2 Editing Level: Main Module

5.2.1 Images

```
/* An image corresponds to one map or to a subset of a map */
class Image  = [ OID-Map: Map,
                 theme: Attribute,
                 a-type-legend: Legend,
                 i-cartographic-object: {CartographicObject}]

/* correspondence geographic object - cartographic object:
   m geographic object - 1 cartographic object
   m            ``     - n    ``
   1            ``     - m    ``    does not make sense */

class CartographicObject;

class SimpleCartographicObject inherits CartographicObject =
                                  [highLighted: boolean,
                                  geogrobjets: list (OID-GeographicObject),
                                  representation; A-Representation,
                                  /*according to the type of legend */
                                  geo: GeometryUI]

class ComplexCartographicObject inherits CartographicObject =
                                 [set-carto-object: {CartographicObject}]

/* ``descr-attribute: Attribute'' could be stored in a  cartographic-object,
   but we assume that it is obtained by a call to the DB
   (use of OID-GeographicObject).
   list(OID-GeographicObject) helps describing the m-1 relationship
   (the m-n relationship is given by the set constructor at the
   cartographic-object level) */


/* GeometryUI: Geometry at the interface level */
```

```
class GeometryUI = [type-geometry: string];

class ZoneUI inherits GeometryUI = [setpolygon: {PolygonUI}]

class LineUI inherits GeometryUI = [setseglines: {LineSegUI}]

class PointsUI inherits GeometryUI = [setpoint: {PointUI}]

class PointUI inherits GeometryUI = [ x: real, y: real]
class LineSegUI inherits GeometryUI =  [ beg: PointUI, end: PointUI]
class PolygonUI inherits GeometryUI = [ listlineseg: list(LinesegUI)]
```

5.2.2 Stack-Images

```
class StackImages = [gene-info: GeneralImageInformation,
                     the-stack: list(Image)];

/* GeneralImageInformation is similar to MapInformation and
   contains general data on images. Some of these data are useful for
   scalling down, etc. */
/* A lot of information is already in the image (e.g., the
   legend-type) and it is not repeated here */
/* In [Voisard92], a layer was transparent or opaque. We don't have
   this notion here, but one could easily add it. Then the stack would be
   a list of pairs (i: Image, transparent-or-opaque: boolean). */

class GeneralImageInformation: [max-x: real, max-y: real]


method AddImage (i: Image) on class StackImages: StackImages;
method RemoveImage (i: Image) on class StackImages: StackImages;

method ShiftDown() on class StackImages: StackImages;
body:
/* We assume te existence of basic operations on list */
    oldtop= first(self->the-stack);
    first(self->the-stack) = first(tail(self->the-stack);
    self->the-stack = append(self->the-stack,oldtop);

method ShiftUp() on class StackImages: StackImages;
body:
    newtop= first(reverse(self->the-stack));
    self->the-stack = append(newtop,(tail(self->the-stack));
    RemoveImage(first(reverse(self->the-stack))@self;
```

5.3 Connection Level

```
/* At least 2 alternatives for creating an image:
   Either create-image on class Map,
   or create-image on class Image.
   in any case, the OID of the map it corresponds to must be known */

method Connection:create-image-from-map(theme: string) on class Map: Image;
body:
      i=new(image);
      (i -> OID-Map) = self;
      the-setgeographicobjects = self->setgeographicobjects;
      for each atom-or-complex-geographicobject
       in the-setgeographicobjects {
        carto-objects=create-cartographic-objects@atom-or-complex-geographicobject}

      put-legend(alegend)@i;
      return(i);
end body create-image-from-map;


method Connection:put-legend(a-legend: Legend) on class Image: Image;
/* fills out the field ``representation'' of each
SimpleCartographicObject, according to a particular legend */

method create-cartographic-objects in class AtomOrComplex;
/* fills out the geometry field of cartographic objetcs */
body:
    while not-empty (self)
    setcartoobjects= new(CartographicObject)
    for each atom-or-complex in self {
            setcartoobjects=union(setcartoobjects,
            /* fills out the spatial values of the simple cartographic objects */
            switch (atom-or-complex -> Geometry -> type):
            /* Trivial   Mapping */
            - ``Point'' {
                             vo = new(cartographicobject);
                            (vo-> geo-> type) = ``PointUI'';
                            (vo->x = self->x; vo->y = self->y);}
            - ``SetPoint'' { for each pi in geographicobject -> Geometry;{
                                vo = new(cartographicobject);
                                (vo-> geo-> type) = ``SetPoint'';
                                (vo->x = pi->x; vo->y = pi->y);}}
      - ‘‘Arc’’
      -    ...
      - ‘‘SetPolygon’’ { ..... }
vo-> representation=[];} /* will be filled out when the legend will
                              be read */
```

Mapget

```
class Mapget = [ rectangle: Rectangle,
                  mapget: Mapget]

class SimpleMapget inherits Mapget = [presentation: Presentation]
class CoumpondMapget inherits Mapget = [mapgetpartition: list(Mapget)]

class Presentation: [an-image-stack: StackImages}

method zoom-in-mapget (zone: Rectangle, factor: int) on class Mapget:Mapget
method scroll-in-mapget(axis: string, direction: int,  nbsetps: int)
on class Mapget:Mapget
/* The body of the two previous methods is graphic dependent */
```