An Object-Oriented Approach to the Design of Geographic Information Systems

Peter van Oosterom and Jan van den Bos

Department of Computer Science, University of Leiden
P.O. Box 9512, 2500 RA Leiden, The Netherlands
Email: OOSTEROM@HLERUL5.BITNET

October 10, 1989

Abstract

The applicability of the Object-Oriented (OO) approach to Geographic Information Systems (GISs) is analyzed. In software engineering, the OO approach as a design model, has been proven to produce quality software. It appears that GISs might also benefit from the OO approach. However, a GIS also imposes special (*e.g.,* spatial) requirements, inclusion of which in the OO model has to be investigated. The proposed solution tries to meet these special requirements by incorporating two data structures: the R-tree and the Binary Line Generalization (BLG) tree. The latter is a novel data structure introduced in this document.

1 Introduction

The Object-Oriented (OO) approach can be useful during several phases of the development of a Geographic Information System (GIS): information and requirements analysis, system design and implementation. The approach also offers good tools for the database and user interface sub-systems of an information system. The interest in Object-Oriented GISs appears to be growing[6, 8, 15]. The OO approach has been proven a good method for these purposes in other information and software systems. It leads to quality software which is extendible and reusable[14]. The reuse of software itself tends to improve the quality because reuse implies testing and, if necessary, debugging. The OO approach also results in good user interfaces, including both visual and non-visual aspects. The explanation for the latter is that if the objects are well chosen, the user can easily form a mental model of the system.

In some sections of this document the OO approach will be illustrated with the case *Presentation of Census Data.* Each of these sections consists of two parts: one with the principles of the OO approach and one with the case. The case will be illustrated with objects described in the language PROCOL, a Protocol-Constrained Concurrent Object-Oriented Language[19]. The next section summarizes the aspects of PROCOL, which are relevant to this document.

In this document the emphasis is on the design of GISs and their databases. However, design must be preceded by information analysis to obtain the requirements. Section 3 contains some notes on Object-Oriented information analysis for a GIS. The design aspects concerning the database and related search problems are discussed in Section 4. Section 5 deals with avoiding data redundancy and also introduces topology. This is accomplished by the *chain* object.

Geographic generalization is the topic of Sections 6 and 7. The term generalization is intended in its cartographic sense and not in the sense as often used by computer scientists in the inheritance hierarchy of the data/object model. The BLG-tree, a data structure for line generalization, is discussed in depth in Section 6. Section 7 is about other generalization techniques in the context of the OO approach.

2 PROCOL

This section gives a short summary of PROCOL; further details can be found in [19]. PROCOL is a simple parallel Object-Oriented language supporting a distributed, incremental and dynamic object environment. PROCOL does not support inheritance because this exposes the internals of an object. So in principle, inheritance runs counter to the idea of information hiding. To enable reuse, PROCOL offers delegation, an equally powerful concept. In the remainder of this paper we will neither use inheritance nor delegation.

PROCOL offers two communication primitives: *send* and *request*. A send is based on a one-way message. The request is comparable to the type of communication in ADA: remote procedure call with (possibly) early return. The advantage of send over request is that the objects are only bound during actual message transfer and not during the processing of the message. This short duration object binding promotes parallelism. A PROCOL object definition consists of the following sections:

OBJ	name and attributes
Description	natural language description
Declare	local type defs, data, functions
Init	section executed once at creation
Protocol	(sender-message-action)-expression
Cleanup	section executed once at deletion
Actions	definitions of actions
EndOBJ	name

An object offers a number of services to other objects. In PROCOL the services are specified as *Actions*. The name of an action is known externally. The body of an action may include, besides standard C-code, send-statements and request-statements which have respectively the following syntax:

DestinationObject.ActionName(msg)
DestinationObject.ActionName(msg)$\rightarrow$*(mes)*

ActionName is the name of the action in *DestinationObject* to which a message *msg* is sent for processing. The message *msg* is comparable to a set of parameters. The values returned by the request are deposited in the variables indicated in the list *mes*. The body of an Action can further contain the PROCOL primitives *new* and *del*, which create a new instance of an object and delete an existing instance of an object, respectively. The latter is only allowed by the creator of the object. The PROCOL primitive *Creator* contains the identification of the creator of the current instance. The bodies of the Init and Cleanup sections have the same form as the bodies of the Actions.

PROCOL constrains possible communications, and thus access to the object's actions, by a *Protocol*. The communication leading to access has to obey an explicit protocol in each object. The use of such protocols fosters structured, safer and potentially verifiable communication between objects. The protocol has the form of an expression over *interaction* terms, the counterparts of send-statements and request-statements. During the life of an object this expression is matched over and over again, that is, the protocol is repeated. The form of an interaction term is:

SourceObject(msg) $\rightarrow$ *ActionName*

The semantics is that upon receipt of message *msg*, from *SourceObject*, *ActionName* will be executed. *SourceObject* can be specified as: object instance, object type (class), and *ANY* object. In the receiving object, the PROCOL primitive *Sender* contains the identification of the communication partner. Expressions over interaction terms are constructed with the following four operators (in this table E and F themselves stand for interaction terms or for expressions):

E + F	*selection:*	E or F is selected
E ; F	*sequence:*	E is followed by F
E *	*repetition:*	Zero of more times E
φ : E	*guard:*	E only if φ is true

3 Finding the objects: information analysis

3.1 Some principles

The problem area must be analyzed, just as in other cases of system development. The application area must be well understood and the task of the new system within it must be clear. This results in a set of requirements for the new system.

The functional requirements and the data affected must be defined in terms of objects types. This means that during the information analysis special attention must be paid to what might possibly be the objects in the target system[19]. There is one golden rule for determining the objects of the functional requirements: *the entities the user is working with and thinking about.* These objects are representations of entities in the real world. After we identify an object, we ask ourselves what do we want this object to do for us (services) in the context of our application. If we can't come up with useful services, this entity is not worthy of being an object in our system.

The resulting object should be reasonably coherent and autonomous: strong internal coupling, weak external coupling. An object type (class) is described by a set of attributes and their associated operations (also called actions or methods). The data can only be accessed from the outside by means of the services. In fact this property is called information hiding, and is the basis for the quality of OO software.

In a GIS there must be at least one object type with a geometric attribute. The type of this geometric attribute fixes the basic form of the graphical representation of the object. There are two categories of geometric attributes: raster and vector. The vector type can be subdivided in: point, polyline and polygon.

In an interactive information system the interface must be user-friendly and the required operations must also be performed fast (interactivity). In the case of a GIS this implies that special attention has to be paid to the spatial organization of the objects. Another very important requirement for an interactive GIS is that the user should be able to look at the data (for example thematic maps) at several levels of detail. There are several reasons for this:

- If too much information is presented to the user at one time, it will harm the efficiency in perceiving the relevant information. Old saying: "You can't see the forest for the trees."
- Unnecessary detail will slow down the drawing on the display. This is especially true for sub-pixel sized drawing primitives because these are hardly visible and thus time is wasted.

The user looking at a small-scale (coarse) map, must not be bothered with too much detail. Only if he or she *zooms in*, the additional detail should be added. This operation will be called *logical zooming*, in contrast to normal zooming which only enlarges. This must at least have two effects. First, the objects which were already visible at the smaller scale map are now to be drawn in finer detail. Second, the objects which were not visible at the smaller scale map, may now become visible. The rule of thumb on *constant pictorial information density*: the total amount of information displayed on one screen is about the same for all scales. A data structure that incorporates both a spatial organization and detail levels is called a *reactive* data structure[21].

Figure 1: Municipalities in The Netherlands

3.2 The case

In the case *Presentation of Census Data*, the objects typically are the administrative units for which the census data are collected. Figure 1 shows the smallest available administrative units in The Netherlands called municipalities (Dutch: *gemeenten*). So, municipalities are objects in the requirements. The geometric attribute of the object municipality is of type polygon. This polygon corresponds with the boundary line of the municipality.

In The Netherlands there are five levels of administrative units for which census data are collected. These are called, listed from small to large: municipality, economic geographic region (Dutch: *economisch geografisch gebied*), nodal region (Dutch: *nodale gebied*), corop region (Dutch: *corop gebied*) and province (Dutch: *provincie*). These units are hierarchically organized. The provinces, the largest administrative units, are shown in Figure 2. The different administrative units are natural candidates for the division in detail levels.

The different administrative units form the object types in the functional requirement for this application. Besides the geometric attribute the objects also have attributes describing the relationships with the other administrative units and attributes containing the census data (*e.g.*, the number of inhabitants) and the name of the unit. Note that this is a very "GISsy" application because all objects have a geometric attribute. The relationship between the administrative units has a *part-whole* character and not a *class hierarchy* character. In the OO approach one is tempted to use class hierarchy because this is supported well by the inheritance mechanism. In the case *Presentation of Census Data*, the use of class hierarchy would be a modeling error.

Until now, we have discussed only the data-part of the objects. The operation-part of the objects contains procedures for:

- Displaying itself in several ways, depending on the selected census data.
- Returning census data. It is possible that the required data are not stored at the current level, but that they are stored in objects lower in the part-whole hierarchy. In that case the census data are collected and aggregated.
- Changing census data. This is only directly possible if the data are actually stored at this level.

These operations are described here in a generalized manner and have to be present for all actual object types, with exception of the "border cases." For instance, it is impossible for a municipality to collect data from objects lower in the hierarchy because it is already the lowest object type in the hierarchy. A part of the PROCOL code describing the object type Economic Geographic Region (EGR) is given below. In this example the object types, MUN (municipality) and NODAL, and the data type polygon are already known.

/ somewhere:*
OBJ NODAL ...

Figure 2: Provinces in The Netherlands

OBJ MUN ...
**/*

```
OBJ          EGR (geom, name, nrof..., parent)
             /* Economic Geographic Region */
             polygon geom;
             string  name;
             int     nrof...;/* census data */
             NODAL   parent;

Description  Part of the object EGR

Declare      struct list {
                     MUN          child;
                     struct list  *next;
             } first, curr;
             void Aggregate (...) { ...};
             boolean EGRComplete;

Init         parent.AddEGR;
             first = curr = NULL;
             EGRComplete = FALSE;

Cleanup      parent.DelEGR;

Protocol     not EGRComplete:
                     MUN() → AddMUN +
             EGRComplete:
                     ( MUN() → DelMUN +
                     ANY() → Display +
                     ANY() → RetrieveData +
                     ANY() → ChangeData
                     )

Actions      AddMUN = { ...}
             DelMUN = { ...}
             Display = { ...}
             RetrieveData = { ...}
             ChangeData = { ...}

EndOBJ       EGR.
```

The other object types are similar. However, the top level objects (type Province) do not have an attribute parent. The top level objects are created first, then the second level objects, and so on. The Init section of EGR specifies that when a new instance is created, an AddEGR message is send to its parent. This parent, a nodal region, then adds the EGR to its list of EGRs. Note that the parent is not the same object as the PROCOL primitive *Creator*. In a normal situation an object has received all the "Add" messages from its children, before it receives any other message type. This is ensured in the Protocol by the use of the *guard* construction. The variable EGRComplete is set TRUE if all municipalities that belong to the EGR have sent their registration message, otherwise it has the value FALSE. EGRComplete, which is initially FALSE, is used to control the access to parts of the Protocol. EGRComplete can be set in the body of the AddMUN and DelNUM actions.

4 Object management and search problems

4.1 Some principles

In the previous section we saw that in general an object consists of a data-part and an operation-part. Normally, the object instances are only present when the program is executing. The data must be loaded into the (new) objects when the program is initialized from a file or a database system. Just before the program stops the data must be saved again. This is an inconvenient method, especially in the case of GISs in which there are huge amounts of data that are not all needed in every session. An OO step in the right direction is to store the objects themselves, including the data (attributes, local variables and protocol status). We will call this operation *save*. After an object is saved, it may be removed with the PROCOL primitive *del* and loaded again with the operation *load*.

The suggestion to store the objects themselves is not as simple as one might expect. This is because objects usually contain references (in attributes or local variables) to other objects. A reference to an object is an *id* (identification of the right type), which the operating system assigned to that object when it was created with the PROCOL primitive *new*. These id's are stored in attributes and variables of the right type and also in the PROCOL primitives *Creator* and *Sender*. We cannot save an object in a useful manner unless we also save the related objects. This is also true for the load operation.

A better solution to this problem is to have some kind of *Object Management System* (OMS) that takes care of *persistent* objects. If during the execution of a program an object wants to communicate with an object that is not yet present, the OMS tries to find this object and load it. It is the responsibility of the OMS to keep the (references in the) object system consistent. The function of the PROCOL sections *Init* and *Cleanup* must be reviewed for persistent objects. There should probably be a mechanism to indicate in which persistent objects one is interested. This limits the scope, so the chance that the OMS finds the wrong communication partner decreases. It will be clear that further research is needed in this area.

The objects as specified in the previous section are unsuited for search operations. If we want to solve the query: "How many inhabitants has the municipality with the name 'Leiden'?", we have to look at all the instances of the MUN object type until we have found the right one. This is an $O(n)$-algorithm. However, this problem can be solved with an $O(log(n))$-algorithm, if a binary search is used. In a relational database, efficient search is implemented by a B-tree[2] for the *primary key* and for all other keys on which an *index* is put. The B-tree has many useful properties, such as: it stays balanced under updates, it is adapted to paging (multiway branching instead of binary) and has a high occupancy rate.

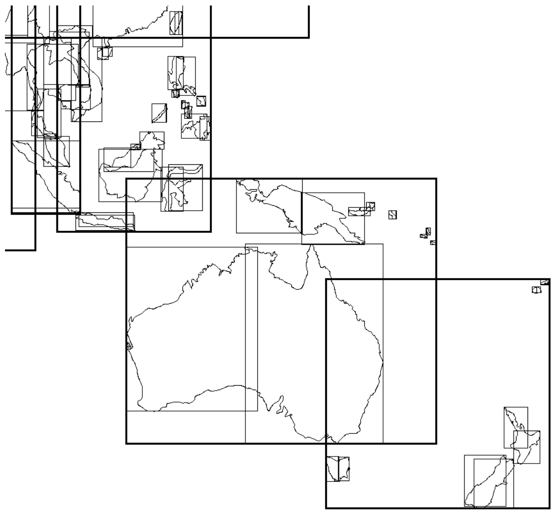

Figure 3: Rectangles in R-tree

The B-tree solution in an OO environment is established by a set of auxiliary objects. These objects do not contain the actual data, but contain references to the objects with the actual data. This must be part of the OMS and, if possible, transparent to the "application" objects. Note that the OMS itself can be implemented in PROCOL as a set of objects.

The searching problem also applies to the geometric data. If no spatial structure is used, then queries such as "Give all municipalities within rectangle X" are hard to answer. A spatial data structure which is suited for the OO environment is the R-tree[11]. This is because the R-tree already deals with objects; it only adds a minimal bounding rectangle (MBR) and then it tries to group the MBRs which lie close to each other. This grouping process is reflected in a tree structure, which in turn may be used for searching. Figure 3 shows the rectangles created by the R-tree for a polyline map of the world (zoomed in on Australia). There are two levels of rectangles: the ones drawn with the slightly thinner line width correspond with the leaf nodes. Several test results[7, 9] indicate that the R-tree is a very efficient spatial data structure. Other spatial data structures[20], for example kd-trees, quadtrees, bsp-trees, and grid-files, are more difficult to integrate in the OO environment because they cut the geographic objects into pieces. This is against the spirit of the OO approach, which tries to make complete "units," with meaning to the user.

4.2 The case

In PROCOL binary trees can be implemented in two ways. The first method stores the whole tree in one search object. The second stores each node of the tree in a separate instance of the search object. The latter introduces overhead by creating a lot of search objects (nodes). However, it has the advantage of being suited for parallel processing in PROCOL because the search objects can run on parallel processors. This is useful for range queries: "Give all municipalities with more than 10.000 and less than 20.000 inhabitants."

There must be a separate search tree for each attribute (for which efficient searches are required) on each detail level. This must be made clear to the OMS. So, if we are interested in the number of inhabitants, we have to build search trees on the detail levels of municipalities, economic geographic regions, nodal regions, corop regions and provinces.

In the situation that the value of an attribute changes, after the search tree is created for that attribute, the tree may become incorrect. This is solved if an object sends a message to the search tree object(s) in the OMS, just after the attribute has changed. Upon receipt of this message the search tree adjusts itself.

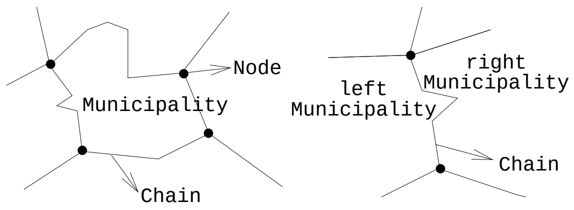

Figure 4: Chains and nodes

5 Avoiding redundancy and introducing topology

5.1 Some principles

An additional (implementation) requirement is that redundant data storage is undesirable. This applies to both geometric and non-geometric data. There is one exception to this requirement: attributes which are difficult to aggregate and which are often required, should be stored redundantly. In this situation special protocols and actions must guarantee the consistency of the data.

5.2 The case

In our case the census data are only stored in the administrative units in which the data were collected. The collection and aggregation of census data is very simple. Just retrieve the data from the children and add it.

The type polygon as used in the attribute *geom* of administrative units is not yet specified. If it is a list of coordinates (x, y), then the attribute geom introduces redundancy in several ways. At a fixed detail level, neighbors have common borders. This implies that all coordinates are stored at least twice. There is also redundancy in the attribute geom between the detail levels. A border of a province is also a border of corop regions, and so on. It is possible that the same coordinate is stored up to 10 times.

It is obvious that this is an undesirable situation. We introduce a new object type named *CHAIN*. A chain is a part of the border of a municipality that contains a *node* only at the begin and end point. A node is a point where three or more municipalities meet (see Figure 4). These are the normal definitions in the topological data model[17, 3]. The attributes of the chain object are: a sequence (array) of coordinates and references to a left and a right municipality. A part of the PROCOL code describing the object type chain is given below.

```
/* somewhere:
typedef struct { float x,y;} Point;
typedef enum { LEFT, RIGHT } Position;
typedef ... BLG;
*/
```

```
OBJ          CHAIN(nop, points, left, right)
             int      nop; /* nr of points */
             Point    *points;
             MUN      left, right;

Description  Part of the object CHAIN

Declare      BLG tree, Build(...) { ... };

Init         left.AddCHAIN(LEFT,right);
             right.AddCHAIN(RIGHT,left);
             tree = Build(nop, points);

Protocol     ...

Actions      ...

EndOBJ       CHAIN.
```

```
OBJ          MUN(name,..., parent)
             string   name;
             EGR      parent;

Description  Part of the object MUN

Declare      struct list {
                      CHAIN        chain;
                      struct list  *next;
             } first, curr;
             MUN      neighbor;
             Position pos;
             boolean BoundaryComplete;

Init         parent.AddMUN;
             BoundaryComplete = FALSE;

Protocol     /* Lowest object in hierarchy */
             not BoundaryComplete:
                      (CHAIN(pos,neighbor)
                      → AddCHAIN) +
             BoundaryComplete:
                      (ANY() → Display +
                      ANY() → RetrieveData +
                      ANY() → ChangeData
                      )

Actions      AddCHAIN = {
             .../* add to list */
             parent.UpCHAIN(Sender,neighbor);
             }
             Display = { ...}
             RetrieveData = { ...}
             ChangeData = { ...}
EndOBJ       MUN.
```

The attribute *geom* is removed from the five administrative units. Instead of this attribute there is a local variable which contains a list of (references to) chains. Initially, the list is empty because the administrative units are created before the chains. The Display action of an administrative unit should not be called before all chains are created. This can be controlled with a proper Protocol.

The chains are created after the administrative units have been created. In the Init section of the chain object a message is sent to the AddCHAIN action of the municipality left. This message tells that the municipality is to the LEFT of the chain and that its neighbor is the municipality right. A similar message is sent to the municipality right. After these messages have been sent, the BLG-tree (Section 6) of the chain is built and assigned to the local variable tree.

The municipality receiving an AddCHAIN message knows which chain did send the message. This chain is added to the list of chains. Finally the municipality sends a message to the UpCHAIN action of its father (an EGR object), with parameters: the chain (contained in the PROCOL primitive *Sender*), and the neighbor municipality. The EGR object has also a list of chains. Upon receipt of the message from the municipality, it checks whether the neighbor is also in the list of municipalities. If this is true nothing is done (internal boundary), else the chain is added to the list of chains and again a message is sent to the UpCHAIN action of its father (a NODAL object). This process is repeated on all detail levels if necessary.

Notice that, with the new object chain, we have also introduced a topological structure[3]. The chain can be used to determine the neighbors of an administrative units. This is possible at each detail level.

An other interesting issue concerning redundancy is whether the area of an administrative unit is stored. The area can be derived from the polygon (list of chains), but this is a non-trivial operation. The area is not only useful as such, but also in the Display of relative information. For example, the population density is the number of inhabitants divided by the area. Because the area is so often needed and difficult to compute, we allow some redundancy here. To guarantee consistency, a border must send a message to the administrative units if it changes. The administrative units then update their area values.

6 Line generalization

When dealing with small-scale maps (large regions) only global boundaries will be displayed. However, without specific measures, the boundary is drawn by the CHAIN object with too much detail because all points of the chain are used. This detail will be lost on this small-scale due to the limited resolution of the display and the drawing will take an unnecessary long period of time. It is better to use fewer points. This can be achieved by the *k-th point algorithm*, which only uses every k-th point of the original chain when drawing it. The first and the last points of a chain are always used. This is to ensure that the chains remain connected to each other in the nodes. This algorithm can be performed "on the fly" because it is very simple. The k can be adjusted to suit the specified scale. However, this method has some disadvantages:

- The shape of the chain is not optimally represented. Some of the line characteristics may be lost if the original chains contain very sharp bends or long straight line segments.
- If two neighboring administrative units are filled, for example in case of a choropleth, and the k-th point algorithm is applied on the contour, then these polygons may not fit. The contour contains the renumbered points of several chains.

Therefore, a better line generalization algorithm has to be used, for instance the *Douglas-Peucker algorithm*[5]. These types of algorithms are more time consuming, so it is wise to compute the generalization information for each chain in a preprocessing step. The result is stored in the CHAIN object in, for instance, a line generalization tree[12]. This tree has the disadvantage that it introduces a discrete number of detail levels, which must be determined in advance. The number of children of a node in a line generalization tree is not fixed. This implies that the tree is not binary, which would be preferable for the implementation.

Strip trees[1] and Arc trees[10] are binary trees that represent curves (in a 2D-plane) in a hierarchical manner with increasing accuracy in the lower levels of the tree. These data structures are designed for arbitrary curves and not for simple polylines (the geometric attribute of a chain). We introduce a data structure that combines the good properties of the mentioned structures. We call this the *Binary Line Generalization Tree* (BLG-tree).

6.1 The BLG-tree

The original polyline consists of the points p_1 through p_n The most coarse approximation of this polyline is the line segment $[p_1, p_n]$. The

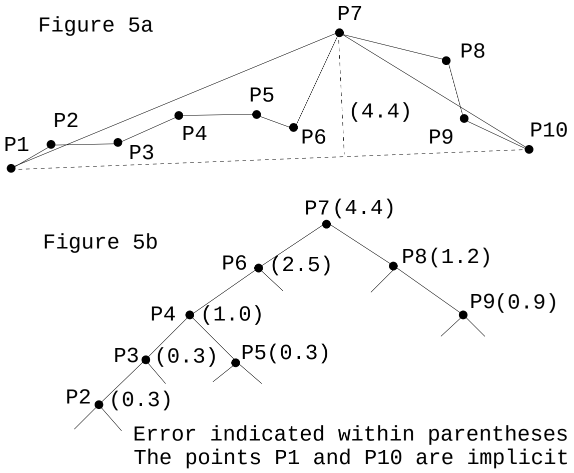

Figure 5: A polyline and its BLG-tree

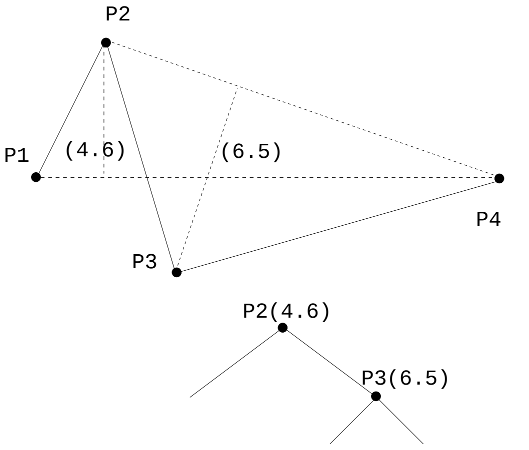

Figure 6: Increasing error in BLG-tree

point of the original polyline, that has the largest distance to this line segment, determines the error for this approximation. Assume that this is point p_k with distance d, see Figure 5a. p_k and d are stored in the root of the BLG-tree, which represents the line segment $[p_1, p_n]$. The next approximation is formed by the two line segments $[p_1, p_k]$ and $[p_k, p_n]$. The root of the BLG-tree contains two pointers to the nodes that correspond with these line segments. In the "normal" situation this is a more accurate representation.

The line segments $[p_1, p_k]$ and $[p_k, p_n]$ can be treated in the same manner with respect to their part of the original polyline as the line segment $[p_1, p_n]$ to the whole polyline. Again, the error of the approximation by a line segment can be determined by the point with the largest distance. And again, this point and distance are stored in a node of the tree which represents a line segment. This process is repeated until the error (distance) is 0. If the original polyline does not contain three consecutive points that lie on a straight line, then the BLG-tree will contain all points of the original polyline. The BLG-tree incorporates an exact representation of the original polyline. The BLG-tree is a static structure with respect to inserting, deleting and changing points that define the original polyline. The BLG-tree of the polyline of Figure 5a is shown in Figure 5b. In most cases the distance values stored in the nodes will become smaller when descending the tree. Unfortunately, this is not always the case, as shown in Figure 6. It is not a monotonic decreasing series of values.

The BLG-tree is used during the display of a polyline at a certain scale. One can determine the maximum error that is allowed at this scale. During the traversal of the tree, one does not have to go any deeper in the tree once the required accuracy is met. The BLG-tree can also be used for other purposes, for example:

- Estimating the area of a region enclosed by a number of chains.
- Estimating the intersection(s) of two chains. This is a useful operation during the calculation of a map overlay (polygon overlay).

6.2 Error analysis

As indicated above, the BLG-tree is used for efficiently manipulating digital map data at multiple scales. At smaller scales this will introduce some inaccuracies. During the error analysis, it is assumed that the most detailed data, stored in the BLG-tree form, is the exact representation of the mapped phenomenon. Three topics will be discussed in this context: position, circumference and area of a region enclosed by a number of chains. The BLG-tree is traversed

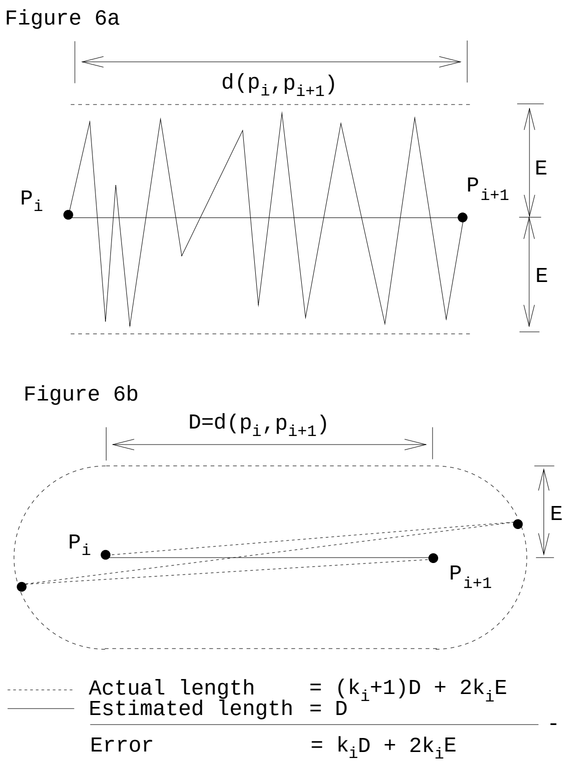

Figure 7: Error in circumference estimation

until a node is reached with an error below a certain threshold, say E. This is always possible because the error value of a leaf node is 0. This traversal results in an approximation of the region by polygon P, defined by points p_1 through p_n, with coordinates (x_i, y_i) for i from 1 to n and $p_{n+1} = p_1$. The estimated circumference C and area A based on polygon P are[20]:

$$C = \sum_{i=1}^{n} d(p_i, p_{i+1})$$

$$A = \frac{1}{2}\sum_{i=1}^{n}(x_i y_{i+1} - y_i x_{i+1})$$

with d the distance between two points. By definition, the *position* of the region is always within a distance E. If E (in world coordinates) is chosen to correspond with the width of a pixel on the screen, then this will result in a good display function. One can trade efficiency for accuracy and choose E to correspond with the width of three pixels. The accuracy is less, but the display is faster because fewer points are used.

The estimate of the circumference will always be too small. This is because for every skipped point, two line segments are replaced by one line segment, which has a smaller length than the sum of the replaced line segments. It is hard to say something about the accuracy of the estimation of the *circumference* C because the boundary can be *zig-zag* shaped (Figure 7a). If the total number of points in the exact representation of the region is known, the worst case error can be calculated. On approximation line segment $[p_i, p_{i+1}]$, with k_i points in between in the exact representation, the largest possible error is (Figure 7b):

$$k_i d(p_i, p_{i+1}) + 2k_i E$$

Summed over all the approximation line segments this results in: $KC + 2KE$, with C the estimate of the circumference and $K = \sum_{i=1}^{n} k_i$. This is an inaccurate estimate because the error can be more than K times as large as the estimate itself. The BLG-tree is therefore not suited to make estimates of the circumference. It should be noted that in "normal" cases the estimate may be quite fair in comparison with the worst case.

The estimate of the *area* A can be calculated with reasonable accuracy. The largest error per approximation line segment $[p_i, p_{i+1}]$ is (Figure 8a):

$$2Ed(p_i, p_{i+1}) + \pi E^2$$

Summed over all line segments this gives: $2EC + n\pi E^2$, with n the number of points in the approximation polygon P. A better calculation of the worst case error can be made if one realizes that it is impossible for all approximation line segments to have their worst case situation at the same time. For a convex polygon, see Figure 8b, the largest possible error is: $EC + \pi E^2$, which is about twice as good as the previous one. The same, though less obvious, is valid for a concave polygon.

6.3 Join of BLG-trees

On small-scale maps it is possible that a region is represented only by the begin and end points of the corresponding chains. As the chains are

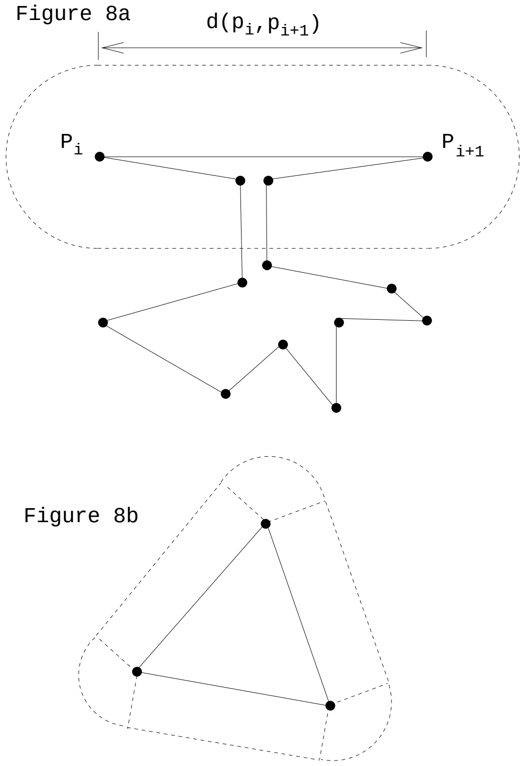

Figure 8: Error in area estimation

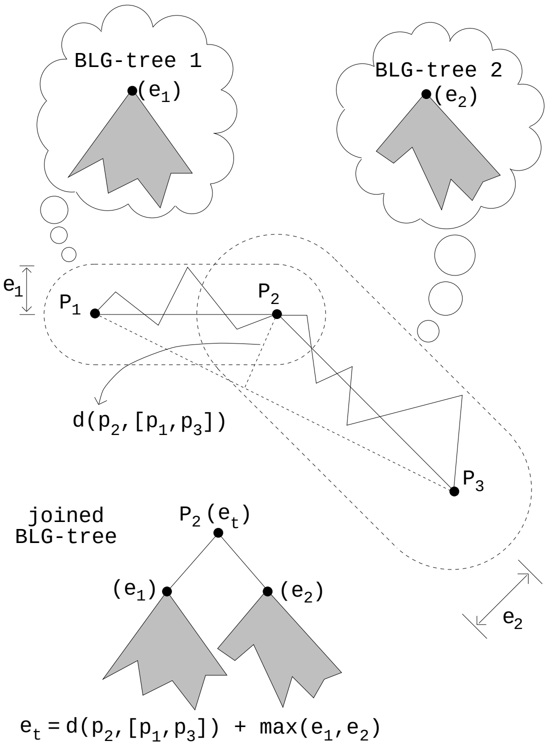

Figure 9: Join of BLG-trees

defined on the lowest level (see Section 5), this representation can be too detailed. The BLG-tree should be built for the whole polygon of the region instead of a BLG-tree for each part of the boundary. However, if the results are stored on region basis, then this will introduce redundant data storage. Contrary, if the BLG-tree is not stored, then a time consuming line generalization algorithm has to be executed over and over again.

The solution for this dilemma is the dynamic join of BLG-trees, which are stored in the chains. Figure 9 shows how two BLG-trees which belong to consecutive parts of the boundary are joined. The error in the top node of the resulting BLG-tree is not calculated exactly, but estimated on basis of the errors in the two sub-trees:

$$e_t = d(p_2, [p_1, p_3]) + \max(e_1, e_2)$$

where $d(p_2,[p_1,p_3])$ is the distance from point p_2 to the line $[p_1,p_3]$ and e_t, e_1 and e_2 are the error values in the top nodes of the joined and the two sub BLG-trees. The estimate of the error e_t is too large, but this is not a problem. The big advantage is that it can be computed very fast.

The pairwise join of BLG-trees is repeated until the whole region is represented by one BLG-tree. Which pair is selected for a join depends on the sum of the error values in the top nodes of the two BLG-trees. The pair with the lowest sum is joined. The building of a BLG-tree for a region is a simple process; only a few joins are needed, one less than the number of chains by which the region is represented. Therefore, the BLG-tree for the region can be computed each time it is needed. Using this tree an appropriate representation for the region can be made on every scale.

7 More on generalization

Geographic generalization, that is, the process of transforming large-scale maps (with detailed data) into small-scale maps, is supported in several places in the design of a GIS in this document:

- Things can be removed. For example: a border between two municipalities in the same province is not drawn, when displaying a small-scale map.
- An object can be drawn with less resolution. For example: A (part of the) border of a country is drawn with only 5 points on a small-scale map. The same part is drawn with 50 points on a large-scale map.

Generalization is a complex process of which some parts (for example line generalization) are suited to be performed by a computer and others are not. Nickerson[16] shows that very good results can be achieved with a rule based system for generalization of maps that consist of linear features. Shea and McMaster[18] give some guidelines for when and how to generalize. Brassel and Weibel[4] present a framework for automated map generalization. Mark[13] states that the nature of the phenomenon must be taken into account during the line generalization. This means that it is possible that a different technique is required for a line representing a road than for a line representing a river.

We will shortly examine some generalization techniques, and comment on the fact whether they can be supported by the OO approach to GIS. This does not imply that a certain part of the generalization can be automated, only that the result of the generalization process is reflected (stored) in the system. The major generalization techniques are:

- A *change* of the geometric representation of an object. A city on a small-scale map is represented by a point (marker) and on a large-scale map by a polygon. Though not described in this document, it is possible to store multiple representations of an entity in one object. At the time this object is displayed, it decides which representation is used on basis of the current scale and resolution of the output device.
- An important object is *exaggerated* on a small-scale map, *e.g.* an important road is drawn wider than it actually is in reality. It is not very difficult to implement this type of generalization, if the importance of the object is stored. A simple adaptation of the Display action should do the trick (with a procedure call to "set line width").
- An object is slightly *displaced* on a small-scale map because otherwise it would coincide with another object. For example, a city is moved because a road is exaggerated. This could be an annoying thing. The displacement can be stored, but this depends on the scale. A solution might be a procedure which determines the displacement. But this is a very difficult process, which involves the subjective human taste or preferences. These may be simulated to a certain extent by an expert system, but this is not within the scope of this document. An other solution is to store a number of displacements, each of which is valid for a certain range of the scale. The number of ranges (and their sizes) varies from object to object.
- Two or more objects are *grouped* and represented by one graphical primitive. For example, two cities which lie close to each other are drawn on a small-scale map as one city. This is the most radical type of generalization because it involves the creation of a new object, with connections to the contributing objects. Nevertheless, there is no reason why this should not be possible. The resulting GIS will become more complex.

8 Conclusion

Besides an Object-Oriented programming language, PROCOL is also a parallel programming language. It is possible that the objects run in parallel on multiple processors. This can be useful in several operations or actions. The aggregation of census data for different provinces can be done in parallel. The displaying of the administrative units can also be done in parallel.

The design of the case studied in this document is based on the principle of data abstraction. Inheritance, a powerful feature in most OO programming languages, is not used in this document. However, there is at least one situation in which it could be very useful. In Section 3, five object types were introduced of which only one, the economic geographic region, was described. The others, province, corop region, nodal region, and municipality, can be described as similar. That is, large parts of the object type are exactly the same. This can be handled by an *abstract* object type (generalizing class) called *Administrative Unit*. The actual object types inherit large parts of this abstract object type and only the specific parts are local. This approach is also better for the maintenance of the system.

Of course, the OO approach does not solve every problem that is encountered during the design of a system. For example, the use of the R-tree and the BLG-tree is necessary to meet the requirements of a reactive information system. The properties of these data structures are also valid outside the OO approach. Nevertheless, the OO approach offers a good design environment that is also well suited for GISs. In order to test the quality of the design, the case *Presentation of Census Data* will be implemented. General purpose GIS object libraries will be built and used in the case. This will tell more about the applicability of the OO approach to the whole development process of a GIS.

9 Acknowledgements

Our thanks go to TNO Physics and Electronics Laboratory,[1] the employer of Peter van Oosterom, for giving him the opportunity to perform this research at the Department of Computer Science, University of Leiden. Andre Smits and Peter Essens of TNO made many valuable suggestions in reviewing this document.

[1] TNO Physics and Electronics Laboratory, P.O. Box 96864, 2509 JG The Hague, The Netherlands.

References

[1] Dana H. Ballard. Strip trees: A hierarchical representation for curves. *Communications of the ACM*, 24(5):310–321, May 1981.

[2] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1973.

[3] Gerard Boudriault. Topology in the TIGER file. In *AUTO-CARTO 8*, pages 258–269, 1987.

[4] Kurt E. Brassel and Robert Weibel. A review and conceptual framework of automated map generalization. *International Journal of Geographical Information Systems*, 2(3):229–244, 1988.

[5] D.H. Douglas and T.K. Peucker. Algorithms for the reduction of points required to represent a digitized line or its caricature. *Canadian Cartographer*, 10:112–122, 1973.

[6] Max J. Egenhofer and Andrew U. Frank. Object-oriented modeling in GIS: Inheritance and Propagation. In *Auto-Carto 9, Baltimore*, pages 588–598, April 1989.

[7] Christos Faloutsos, Timos Sellis, and Nick Roussopoulos. Analysis of object oriented spatial access methods. *ACM SIGMOD (Management of Data)*, 16(3):426–439, December 1987.

[8] Mark N. Gahegan and Stuart A. Roberts. An intelligent, object-oriented geographical information system. *International Journal of Geographical Information Systems*, 2(2):101–110, 1988.

[9] Diane Greene. An implementation and performance analysis of spatial data access methods. In *IEEE Data Engineering Conference*, pages 606–615, 1989.

[10] Oliver Günther. *Efficient Structures for Geometric Data Management*. Number 337 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1988.

[11] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. *ACM SIGMOD*, 13:47–57, 1984.

[12] Christopher B. Jones and Ian M. Abraham. Line generalization in a global cartographic database. *Cartographica*, 24(3):32–45, 1987.

[13] David M. Mark. Conceptual basis for geographic line generalization. In *Auto-Carto 9, Baltimore*, pages 68–77, April 1989.

[14] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, London, 1988.

[15] L. Mohan and R.L. Kashyap. An object-oriented knowledge representation for spatial information. *IEEE Transactions on Software Engineering*, 14(5):675–681, May 1988.

[16] Bradford G. Nickerson. *Automatic Cartographic Generalization For Linear Features.* PhD thesis, Rensselaer Polytechnic Institute, Troy, New York, April 1987.

[17] Thomas K. Peucker and Nicholas Chrisman. Cartographic data structures. *The American Cartographer*, 2(1):55–69, 1975.

[18] K. Stuart Shea and Robert B. McMaster. Cartographic generalization in a digital environment: When and how to generalize. In *Auto-Carto 9, Baltimore*, pages 56–67, April 1989.

[19] Jan van den Bos. PROCOL – A protocol-constrained concurrent object-oriented language. Special Issue on Concurrent Object Languages, Workshop Concurrency, OOPSLA '88, San Diego. *SigPlan Notices*, 24(4):149–151, April 1989.

[20] Peter van Oosterom. Spatial data structures in Geographic Information Systems. In *NCGA's Mapping and Geographic Information Systems, Orlando, Florida*, pages 104–118, September 1988.

[21] Peter van Oosterom. A Reactive Data Structure for Geographic Information Systems. In *Auto-Carto 9, Baltimore*, pages 665–674, April 1989.