PRINCIPLES FOR A NEW EDITION OF THE DIGITAL ELEVATION MODELING SYSTEM SCOP

L. Molnar

Institute of Photogrammetry and Remote Sensing, The Vienna University of Technology, Vienna, Austria
ISPRS Commission IV

Abstract:

DEMs as information (sub)systems storing vectors, raster, image pixels, and attributes. Communicating with other databases, and accessing DEM information, via (TOP)SQL. Treating topology in terms of relationality; dealing with true three-dimensionality. Main functions. Combining vectorial algoritms with those of raster geometry. Patchwise processes with stages varying locally. Controlling DEM processing via object oriented programming in a multitasking, and eventually distributed system. Machine independent base modules. Modes for real-time processing, sub-system (slave or driver) mode, batch processing mode. Employing parallel programming including massive parallelity.

KEY WORDS: Digital Terrain Models, System Design

INTRODUCTION

A university institute has the two interconnected tasks of teaching and of research. A new dimension has been added to these tasks by some leading personalities of the academic world of photogrammetry, in the first line by Prof.Dr. Ackermann at the Stuttgart University, followed by Prof.Dr. Kraus at the Vienna University of Technology, and by many others. According to this new dimension, results of recent university research are made commercially available to the world of production in the form of computer programs or modules - rather than just as theoretical papers and research reports. These programs should have the merit of applying new technological methods. They can hardly be, however, on the level of versatility of the available commercial products. Recently, in view of some major developments in computer science, the contrast between merits and disadvantages of such research programs became by far too large - concerning, in the first line, *object oriented design*, *graphical user interfaces*, *interactive graphics*, etc.

Current versions of the digital elevation modeling package SCOP[1] represent a splendid example of the aforementioned contrast. Therefore, and to refine the package in many other respects, work has been started on creating a new, open frame to hold some of the current modules and all future ones. Principles proposed for this architecture constitute the subject of this paper. At the time of this writing, these principles are not yet approved in any way as final. In many points, they represent but a personal view.

THE "CLASS-ORIENTED" PROGRAM ARCHITECTURE

Sought is a highly modular, easily extendable architecture to be implemented on most major platforms, to combine some existing modules with the new ones, and to be controlled, on the one hand, by graphical user interfaces, and on the other - in batch mode, on mainframes, or just as an option - by command language.

The existing source of SCOP and of related programs represent up to 200.000 lines of proven FORTRAN code, one through ten years old. Three quarters of it could and should - indeed: must - be rescued.

Due to its *FORTRAN-centered, cross-platform programming expertise*, the SCOP team is by now extremely proficient in creating new modules to the existing system. The price of giving up this expertise all together could be, I am afraid, about one year per programmer. So let's examine concisely whether there is sufficient technical reason to do so.

There are two major recent developments in computer programming affecting the application of FORTRAN: first, the increasing necessity to address different software and hardware resources such as *graphical user interfaces (GUIs), graphics libraries, external databases (embedded SQL, GIS), or additional hardware*. The problem is not just that FORTRAN does not provide any standard means for such purposes. More importantly, different libraries, device drivers, software kits, and the similar are nowadays supplied for C only. These problems are then usually solved by mixed language programming.

The second point concerns *object oriented programming*. It facilitates a deeper reaching modularity, a way of thinking coming much closer to that of the user, it yields a highly modular and highly extendable architecture, and it may aid greatly the transition to distributed and parallel computing. On a fine-grained, SMALLTALK-level, with object oriented operating systems and compilers for all major platforms, and with widespread processor support, distributed intelligence, and parallel processing - on this level, object oriented programming is not yet with us. For this, I think, even developers have to wait for maybe five years. However, a hybrid solution with a large-grained object structure mixed with procedural parts is rapidly gaining importance: the C++ solution and the similar ones, based on structures designated as ***classes** to encapsulate both data and the methods (algorithms) operating on them*.

Externally, a (complex) class controlled by messages resembles very much a (small) program controlled by directives of a command language. (Small) programs can thus be considered and employed as some common denominator between classical procedural programs, on the one hand, and (complex) classes of a C++ type object oriented program. As parts of an integrated system, such programs could be termed *processes*. This expression emphasizes dynamic so much that for most of us it is hard to think of a process as of data, with the code integrated. So after rejecting a series of terms (e.g. *complex object*, or just *complex*),I found the name *autonomous class* to suit best. An architecture then, integrating such classes, will be termed a class-oriented

[1] INPHO GmBH, Stuttgart, and Institute of Photogrammetry and Remote Sensing (IPRS) of the Vienna University of Technology

one. Object-oriented on the surface, it is capable of integrating modules belonging to both the procedural and the object oriented world.

Implementing Autonomous Classes

An autonomous class is, in the most general terms, a complex object incorporating (and encapsulating) data, instructions, and hardware. Full implementation of such classes is possible in distributed processing (network; specialized hardware on PC boards). But member programs integrated into a system and running in parallel in a multitasking environment represent the most typical, though not full, implementation of autonomous classes.

The *main programming languages to be used are C++ and FORTRAN.* Within one autonomous class, mixing them is to be avoided as far as possible (for reasons of portability). The choice of the language depends on:

- the class being *machine bound* (preference for C++),
- the class performing *mainly numerics* (mild preference for FORTRAN),
- the *availability of re-usable routines* (usually FORTRAN),
- the *experience of the programmer* (in most cases FORTRAN).

An autonomous class is invoked by the user, or by other classes. It may command over any number of autonomous sub-classes thus building a family of autonomous classes. To enable the *invocation of an autonomous class in different environments*, different heading statements have to be provided by conditional compilation ("ifdef"-s). In FORTRAN, a PROGRAM statement enables SPAWN-type invocation in a multitasking environment. An alternative SUBROUTINE statement allows to build large linked systems, e.g., on mainframes, or the invocation of classes stored in dynamic link libraries (DLLs). In C++, similar techniques are possible in using corresponding CLASS statements and names. In the future, special headers for Object Request Brokers (ORBs) may become necessary, so to enable invocation by some Distributed Object Management technology.

Communication among classes is via *messages* (strings). These are reminiscent of simplified directives of a command language. Simplicity is partly due to polymorphism: the object knows how to react to the same message in different cases, it itself knows best the stage and ways of processing, etc. - In the case of the autonomous class for topological database management (TOPDB), messages acquire the syntax of TOPSQL (Loitsch, Molnar, 1991). Autonomous classes should be able to handle simultaneous requests (I/O queues, buffering, priority handling, managing asynchronous communication with other classes, record locking, periodic checking of external requests to pause or abort, etc.). Multithreaded operating systems (OS/2) provide fine means to keep an autonomous class alert to messages while working on a task. *Data communication* should proceed either via the database management class (TOPDB), or via binary files, memory pipes, shared memory, and similar means. Synchronization via semaphores, flags, etc. becomes very important - e.g. if one class is processing a patch (tile) of data, and the database class is simultaneously retrieving data for the next patch.

The developer should clearly differentiate between classes
- mainly machine bound, and
- practically machine independent,

and furthermore, concerning invocation:
- public classes, to be invoked
  - by the user,
  - by (other classes of) the system, and
- private ones, being subclasses to some other class.

For a system integrating autonomous classes, "lazy" processing is characteristic: without user requests, there happens nothing, and in answering requests, there happens the minimum necessary. This *principle of laziness* is a sometimes shortsighted but generally advantageous principle of *economy*.

APPLYING THE CLASS-ORIENTED ARCHITECTURE TO BUILD A DTM SYSTEM

Instead of an Introduction:
DTMX, and Hints on an Object-Oriented View of DTMs

DTMX was to be a modular system of software tools operating on a (transitory) database, designed *to provide a highly flexible interface between DTMs of different structure and origin* (Molnar, 1984). It never materialized as a whole but its definitions of a sequential data exchange format became widely used in Austria (data in this format cover many times the territory of the country).

The database of DTMX was to contain *patches of data with a code characterizing the level of processing* achieved on them ("lazy" processing). Examples of such levels are 'raw data', 'data edited', 'DTM grid', 'contour lines interpolated', etc. Modules have been foreseen to perform the upward transition from one level of processing to the next. With some system on the input side yielding data of one level (e.g. raw data), and the system on the output side requesting another level (e.g. contour lines), DTMX was thought to perform this transition automatically by piping internally the corresponding tools.

So, the design of DTMX did have the potential to result in a fully-fledged DTM-ing system. Looking at it in an object-oriented way, *a DTM is a complex object encapsulating both data and processing code*, uniting information with intelligence. DTMX corresponds to this definition. - It is the origin of some major principles to follow here.

Classifying Autonomous Classes of the DTM

According to their role, the autonomous classes of the DTM can be classified as
- *FRAME (or Management) classes* such as Management of User Interactions (including on-line context-sensitive help), Database Management, Project Management including Error and Message History, Graphics Management, and others;
- *KERNEL (or Application) classes* such as Input Data, DTM Surfaces, Graphic Sheets, Isolines/Zones/Distributions, Views, the products of DTM-Algebra, of DTM Classification, of DTM Exploration - and numerous other classes.

In thinking of an autonomous class (e.g. in planning one), the main consideration should be given to its instances (i.e. to its data records); the methods (algorithms) operating on them should be considered as means. This corresponds to the aspects of the user, and to an object-oriented analysis and modeling of the task. In this sense, notes on some of the classes of the DTM follow.

FRAME (MANAGEMENT) CLASSES

Management of User Interactions

There is, indeed, only a limited number of different actions a user needs to perform. He needs to identify some options (words) or some position (x,y), to end pausing states (e.g. by typing a carriage return), or to enter some strings or numbers as values. From this point of view, it is easy to construct an interface transferring information from the user interface to the autonomous classes.

The problem is the other, the prompting side. Early single-line promptings developed to *graphical user interfaces (GUIs)* presenting the user with a wealth of details such as windows with drop down menues, scroll bars, dialogue boxes, radio buttons, etc. *IBM's Systems Application Architecture (SAA)* specifications created a widely accepted standard (OS/2 Presentation Manager, MS Windows, X-Window, DEC-Window, and many others). Recently, there are cross-platform tools commercially available providing a single interface to all these systems, and realizing an acceptable compromise of the common denominator to them.

Development, or sometimes fashion, do not stand still, however. With regard to Smalltalk and Macintosh type user interfaces, the SAA standard appears to be antiquated. And I do not doubt that further stages are following in a rapid sequence, considering, e.g., the exploding importance of multimedia technology.

It is, therefore, imperative to separate the GUI from the classes to be controlled by it. There is an interface necessary, somewhat similar to the *resource file statements* as originally formulated by IBM for the PM. Interpreting such statements, the autonomous class 'Management of User Interactions' can compose the graphical user interface to appear on the screen in using the latest cross-platform development tools and libraries. This way, the user interface can be kept to correspond, though with severe compromize, to the fashion of the time.

The class 'Management of User Interactions' is to support the user interface: where possible, a GUI, and everywhere, a *command language interface* - as a choice, and as a necessity for *batch processing*. Via this interface, the user can invoke the major classes, and keep more than one of them active at a time. This class is overwhelmingly machine bound for it has much to do with the GUI, with foreign libraries, with graphics and device drivers, and, in invoking other autonomous classes, with operating system calls. Therefore, it has to be implemented in C++. It is the tip of the iceberg - it is the ROOT or MAIN class of the DTM system.

Database Management

Database management plays a central and multiple role in this system's architecture: it is managing data for all the different classes. Besides of dealing with sets of original (input) data including digital images and overlays of the DTM surface, it should carry tables and matrices with overviews of previous and current stages and peculiarities of the processing, of data changes within individual patches (tiles) with date and time, objects of graphics output, matrices and tables of error and message history, 3-D enclaves, and much more. The class *'Database Management' is not directly accessible by the user;* such access is only granted via other classes. These last ones are considered to *logically encapsulate* the data managed by them.

There is every reason to go on with the SCOP philosophy to reflect various aspects of terrain characteristics by building multiple models of the same area rather than having more than one function value at each location of a single model. This allows to have independent structures and densities of the models covering the same area, e.g. those reflecting elevations belonging to different epochs, or soil quality, or indeed any characteristics capable of sorting on an ascending scale.

For this purpose, in the system as proposed, database management should handle multiple overlapping terrain models structured independently of each other (this is solved in SCOP's current DTMLIB routines already). These can serve then either as overlays, or as operands in expressions of DTM Algebra resulting in operations identical or similar to those provided in the current SCOP.INTERSECT module.

Most probably, the class 'Database Management' is to comprise more than one sub-class to fit different purposes. TOPDB and DTMLIB to be described next will be integrated as such sub-classes. A further sub-class to manage graphic objects may become necessary.

TOPDB (Loitsch, Molnar, 1991) is a fully *relational database management system* capable of dealing with data, attributes, and arrays of these (including pixel arrays). TOPDB is specifically designed to be efficient in performing *spatial and topological queries* according to 2-D criteria on large amounts of data distributed in a "two and a half" dimensional space. It is *treating topology in terms of relationality*. It is controlled by *TOPSQL*, a major subset of the Structured Query Language (SQL) extended by topological operators. TOPDB tables can be created and discarded on the fly, at run time. TOPDB has hardly any limitations concerning the amount of data to be treated. The major sources of its efficiency are *multiple continuously balanced binary trees, and extensive buffering*. - Further development is necessary concerning TOPDB's capabilities to function as a server in a multitasking and/or networked environment. - TOPDB has been developed in FORTRAN by using the totally incredible compiler and workbench created by A. Koestli of the SCOP team in Stuttgart (Koestli, 1990).

DTMLIB is a main library of the current edition of SCOP. It contains the database management system for the random access DTM. It is *designed to access multiple individually structured DTMs concurrently*. It contains routines available to user-written programs (FORTRAN). DTMLIB should be further developed to deal with irregular patches of the surface rather than just with regular computing units. It should be extended to work in concert with TOPDB, especially with regard to vector type data such as break lines, borderlines, highs and lows, etc., to be managed in the new system by TOPDB only.

Project Management

This class is supervising the entire project. It is to manage records of data security, of archiving (saving), of limiting polygons, matrices of coverage, etc. The Error and Message Management belongs here, as well, both to send and to administer these messages, and to build up a history of them, for analysis, and for summaries. - Most records are stored on TOPDB tables.

Graphics Management

Graphics Management is to receive data via interprocess data communication (memory pipes, shared memory areas, binary files). These data are to be displayed,

plotted, or stored as graphics objects in the 'Database Management'.

The best way to organize the data communication is an asynchronous stream on memory pipes, so to allow the sending and the receiving classes to act in parallel. Data items should be coded as meaningful objects rather than as graphics primitives, e.g. as 'contour line section' rather than 'spline curve section', or 'contour label' rather than 'string'. The Graphic Management should "know" how to represent the objects on the basis of set-up tables and user definitions.

Storing graphics objects in some sub-class of 'Database Management' serves two purposes. First, such storage allows for an *on-screen customization of graphics output* (e.g. of a map sheet with different contents). And second, it can serve purposes of *"what-if" analyses* where variants of the output have to be compared among themselves on the screen. For this, specialized displaying methods have to be used, and graphic displays to be compared should be constructed in ways best fitting this purpose (e.g., probability-distribution type isolines (Kraus, 1992) in different colors might yield great advantages when mixed additively).

The class 'Graphics Management' plays important roles in interactive graphics. Displays and user entries have much to do with the Management of User Interactions. User actions should be received from it, and reported back to the requesting class.

KERNEL (APPLICATION) CLASSES

Notes on some selected important and characteristic classes follow here. There will be many more of them. Most importantly, this series can easily be extended by adding new autonomous application classes.

The Class 'Input Data'

The flexibility of database management by TOPDB allows for great freedom in handling different *data types*. A classification according to current needs could be: *On-terrain data* in the form of points at random, grids, lines (breaklines, formlines, borderlines), geometric figures, enclaves with analytical surfaces and 3-D objects, etc., *off-terrain data* (e.g. church crosses), *data defined in the (x,y) plane only*, furthermore *geometric constraints, digital images*, and different *attributes* to the above. Because data can be retrieved by any of the usual relational queries, and, in addition and most importantly, according to spatial and topological criteria, it is easy to create flexible methods (tools) operating on the data.

The *methods* to this class are numerous indeed, so in implementing them, a large family will have to be created with many sub-classes. The major methods necessary are:

- *Import/Export*, on the one hand via software interfaces from and to different systems such as GIS, ArcInfo, Intergraph Microstation, or AUTOCAD, and on the other, from and to sequential data formats (file interfaces) such as the ones of DTMX, those of current versions of SCOP: WINPUT and KA001, furthermore from and to pixel data standards, and others;
- Different *transformations,* including absolute orientation;
- *Editor* for vector type data, with
  Methods of Data Exploration and Analysis:
  Analytic, including *Progressive Sampling* to be applied both on-line (data acquisition), and off-line (checking),
  - *Numeric*, including *Line Networking*, and handling contradictions in the *Overlapping Zones* of photogram-metric models (both of them semi-automatic processes with interactive user support),
  - *Visual*, including *Data Displays* and *Data Plots*,
  - and a complex system providing means (via invoking other classes) of prospecting contour lines, perspective views, pixel-coded views, and the similar within local areas of interest (*"What-if" Analysis*);

  Listing and Updating, with addressing
  - by means of *interactive graphics*,
  - via *alphanumeric entries* (inevitable for batch processing).

  Patching (replacing patches of data);
- *Editor for Raster (Pixel) Data*;
- *Changes' Management*, in the first line to facilitate automatic updating of previously derived products (such as the DTM surface or contour lines) by re-processing the tiles (data and algorithm patches) involved.

The Class 'DTM Surfaces'.

In this architecture, the concept *DTM Surface* is an elusive one. It is a means to achieve a goal, a *transitory*, intermediate state of the information contained in the original ("input") data on its way to be expressed in another form - e.g. as contour lines or slope vectors. It may not be needed in some cases at all, or it may be needed locally only, e.g. along a profile axis, and it may take on different forms, such as a grid or an analytical expression. It is up to the user, whether to have this surface representation just transient (as scratch), to keep it for the sake of some processes and then discard it, or to store it *permanent*ly as a regular, classical DTM structure.

Notwithstanding all this, whether transient or permanent, we need the means DTM in most cases, and, so to attain fine quality, we should be able to work on it directly and locally: reality is not always willing to submit to generalised schemes. I often remember Dr. Yoeli fighting on these grounds against replacing original data by all-purpose DTM grids (unpublished).

'DTM Surfaces' should be, according to this discussion, just some sub-class (or even just a method) to such classes as 'Isolines', or 'Slope Vectors'. But because it is so overwhelmingly important, common to most application classes, and sometimes even a goal in itself (in case, e.g., of country-wide permanent DTMs), regular and usually discrete surface representation continues to be a corner-stone of DTM-technology, and as such, it yields one of the largest and most important classes in this architecture, as well.

The class 'DTM Surfaces' is thought to provide space for a growing number of competing and co-operating algorithm sub-classes. *Algorithm Tiling* is a task of the mother class. Enclaves with analytic surfaces and 3-D objects on the input data are pre-defined tiles. These tiles are belonging to the sub-class for Special Enclaves. Algorithm tiling for the rest of the terrain may become a quite complex process combining data exploration with user interactions. Areas surrounded by break lines, and without any other data within them, should be identified and classified as enclaves with analytic surface. It must be of special concern to provide for smoothness of surface representation on both sides of the tiling limits (just as it is along the edges of computing units).

Surface Exploration and Analysis is to become a major group of methods of the class 'DTM Surfaces'. Applying these methods, the user should be able to find optional ways of surface representation in problematic areas, and to convince himself and others about the quality of it. *"What-if" Analysis* should enable him to apply different algorithms to the same set of tiles and to compare results numerically, analytically, and visually, e.g. by viewing shadows or shading in oblique light. Different methods of *Quality Analysis* should be at the user's disposal, e.g. transient and local data transformations (translation, rotation) to check on isotropy.

Algorithm Sub-Classes are to be very independent large black boxes logically encapsulating data for a tile of the surface, local co-ordinate systems, (a combination of) interpolation algorithms, *private surface representation* (e.g. hexagonal tiling, or analytical forms), *functions to service inquiries* concerning this surface (z, components of f' and f" at a given location, the isoline at a given elevation, ray tracing, etc.), and in some cases even hardware (e.g. on a board). They should employ *parallelity* in their co-operation with each other (tiling), and with the Database Management class. Multithreading may be pursued (e.g. reading the data stream and setting up equations or matrices).

Modules realizing the linear prediction algorithm in the current version of SCOP can be adapted to become one of the algorithm sub-classes of the new edition.

Algorithms should fulfill, at least in co-operation, many requirements. Concerning *data types,* they should operate on points distributed at random and be able to deal with large variations in data density. They should exploit additional information carried by special data such as breaklines and structure (form) lines, both with or without elevation, contour lines, highs and lows. If possible, geometric information provided explicitly (e.g. normal vector components) should be used.

Concerning *quality*, filtering of noise in data should be solved adequately. Attributes to each data element carrying a-priori accuracy characteristics should be taken into account. Outlier analysis (blunder detection) is important. However, an automatic blunder elimination should be avoided (Wild, 1983). Algorithm sub-classes should provide spatially distributed a-posteriori quality characteristics.

Concerning the *mathematics applied*, algorithms may be widely different. There will be a series of *vectorial* ones, such as the linear prediction mentioned (Assmus, Kraus, 1974). Solutions belonging to *raster geometry* will follow; they may rapidly gain importance - special hardware for them is on its way to become standard on PC-s, due to the growing interest in multimedia technology (array processors: Next Station, IBM PS/2, Apple Macintosh; massive parallelity may follow). To this realm belong neuronal nets with heuristic solutions, or, more realistically, *systolic arrays.* Hybrid algorithms may exploit advantages of both worlds, combining, for instance, the topologic capabilities of systolic arrays with the numeric precision of vectorial solutions.

The DTM Sub-Class 'Special Enclaves' with Analytic Surfaces and 3-D Objects

Applying true 3-D algorithms all over a DTM places tremendous burden on most computer resources and therefore they should be used only when and where inevitable. Terrain canopies, sometimes even vertical walls, and 3-D objects scattered over the surface represent problems for "2,5 D" solutions defining locations just by a pair of (x,y) co-ordinates and carrying the third dimension (z) rather like an attribute of the point. Here, a hybrid solution is proposed, applying true *3-dimensionality only within special enclaves surrounded by breaklines.*

As already mentioned, special enclaves are tiles of data with pre-defined "2,5" or 3 dimensional algorithms to operate on them. A growing choice of such algorithms should be provided, and identified by some names or codes to enable referencing them in data sets.

Instances (data records) of this sub-class can be classified as:

- data provided for the enclave to be approximated
  - by *analytic surfaces* such as a horizontal plane or a hyperbolic surface;
  - by *restrained surfaces* such as minimal surfaces or such to fit river basins (Kalmar, 1991)
- 3-D objects
  - *terrain canopies, vertical walls,*
  - *3-D constructs* such as buildings or bridges.

In the case of 3-D objects, a surface should be defined to represent, when needed, the continuation of the terrain surface passing the object. For this, the ways as used outside of the enclave should be used. Instead of defining the continuation surface, small enclaves can be coded as 'disregardable' for cases when the 3-D object is to be ignored (e.g. on a fast perspective view, or more characteristically, on an overlay with contour lines).

In special cases, enclaves can become very large to carry, for instance, (parts of) a city model. For such purposes, complex software should be adapted and included in this system as another sub-class (Kager, Loidolt, 1989).

Special enclaves should service the same requests as other algorithm sub-classes do. Additional options could become necessary in this respect, e.g. of providing the perspective image of the object according to specifications or providing special ways of access to the data for the purpose of editing them.

Short Notes on Some Other Classes

The class *Graphics Sheets* is to build up presentation-quality graphics stored, for the time of on-screen customization, as TOPDB tables. While frame composition is a direct task of this class, to fill the sheet with contents is via messages to other classes such as Isolines/Zones/Distributions, or Views. Upon such invocations, the user can interact with those classes. The graphics stored is then edited and customized by invoking the class 'Graphics Management'.

Isolines/Zones/Distributions is an example of a class usually invoked from the class Graphics Sheets, it can however be invoked directly, as well. This second case becomes necessary when results are required in forms other than graphics output - e.g. as a stream or file of numeric values.

The class *Views* is for vector and raster (pixel) type visualizations of the DTM including the input data, as well (e.g. a perspective view of the latter). The class should be capable of superimposing different visualizations where applicable (Ecker, 1991 and Hochstoeger, Ecker, 1990).

DTM Algebra applies as operands different DTMs of the same area carrying different epochs of the same information (e.g. elevation), or different types of information (e.g. slope or soil quality models). There may be used very complex expressions. This is one of the functions of the current SCOP.INTERSECT module (Sigle, 1991), parts of which should be adapted as a method of this class. *Classification* is to incorporate the second important function of SCOP.INTERSECT allowing to classify contents of a DTM according to a network of polygons of any complexity (e.g., to classify a slope model according to cadastral boundaries). *DTM Exploration* is for compiling (deriving) different products of a DTM, by methods such as reported in (Rieger, 1992).

It is of mandatory importance to represent results of different methods not just as graphics or listings but in forms capable of further processing - such as, again, DTM structures with the derived quantities as z values. Slope models are a good example of this, but even visibility of the surface from a specified point of space (a by-product of the hidden line algorithm of perspective views) can be represented as a DTM structure. In this case, the elevation difference (negative, 0, or positive) should be represented as z, to be added to the elevation of the DTM location so to become visible or invisible, respectively (Hochstoeger, 1991).

SYSTEM INTEGRATION

The classes as described will be integrated into two different versions of the DTM system:

- a full-featured version with graphical user interface (GUI), command language, interactive graphics, etc., available for some of the most important platforms (such as PC workstations under OS/2 2.* and under WINDOWS NT, some major UNIX workstations, and maybe DEC VMS), and
- a major machine-independent subset of the DTM system, mostly for mainframes, with just the command language as user interface.

Both versions will run in three modes of operation:

- interactive mode,
- batch mode controlled by command procedures, and
- driver or slave mode, to service requests of a host system, without being seen by the user in any ways (him seeing just the user interface of the host system), and sending all output to that system.

Integrating the DTM system into a greater application software environment concerning both input and output is of crucial importance. A new edition of the Topographic Information and Archiving Software (TOPIAS) based on TOPDB is to be integrated into the DTM package so to play the role of some foreign secretary. Messages in the syntax of standard SQL, or better still, of TOPSQL should enable communication in *both* directions. *Servicing (TOP)SQL requests by the DTM system* yields a good example of "lazy processing": requesting (SELECTing) an elevation at a location (x,y) will result in a message to the class DTM. This will check whether there is an interpolated surface ready at the location; if not, a message will check with the class of input data whether interpolation is possible, and so on. After some "small talk" among the classes exchanging messages and data, the result will be deduced and sent back (as INSERT statements) via TOPIAS to the requesting system (which could be a user-written program, as well).

The driver or slave mode of operation provides fine means of integration with *geographic information systems (GIS)*, with systems to serve *analytical plotters* (e.g. to support data acquisition: progressive sampling, editing, image overlay of different DTM products), or with *interactive graphics systems*.

And furthermore, there remain such means of integration as DXF and other file format standards. *Compatibility with earlier versions of SCOP* must be ensured by supporting not only the corresponding I/O formats but also both databases (DAF and RDH).

SUMMARY

An architecture for application software is described, as proposed for the new edition of the DTM package SCOP. It consists of fairly independent modules ('autonomous classes'), integrated within an object-oriented frame. Autonomous classes can be implemented, in addition to object oriented programming, in classical procedural ways. This compromise has major merits at the current state of the software engineering practice: it provides fair portability, allows for integrating modules of earlier programs, from the user's perspective it carries clear signs of object orientation resulting in user friendliness, and the system is easily extendable by further autonomous classes. Some selected classes of the DTM system are described. Further important application classes are going to be created as important by-products of dissertations, the commercial benefits providing the student with means for his studies, and the application helping to spread new technology.

Acknowledgment

This study has been financed by Fonds to Promote Scientific Research of the Austrian government, Project Nr P7385-GEO.

References

Assmus, E., Kraus, K., 1974. Die Interpolation nach kleinsten Quadraten; Praediktionswerte Simulierter Beispiele und ihre Genauigkeiten. Deutsche Geodaetische Komission, Reihe A, H.76.

Ecker, R., 1991. Rastergraphische Visualisierungen mittels digitaler Gelaendemodelle. Geowiss. Mitteilungen der Studienrichtung Vermessungswesen der TU Wien, Heft 38.

Hochstoeger, F. 1989. Ein Beitrag zur Anwendung und Visualisierung digitaler Gelaendemodelle. Geowiss. Mitteilungen der Studienrichtung Vermessungswesen der TU Wien, Heft 34.

Hochstoeger F., Ecker R., 1990. Application and Visualization Techniques for Digital Terrain Models. International Archives of Photogrammetry and Remote Sensing, Komm. IV, Vol. 28, Part 4, Tsukuba, Japan.

Kager H., Loidolt P., 1989. Photomontagen im Hochbau. Vermessung, Photogrammetrie, Kulturtechnik, Nr.3.

Kalmar, J., 1991. Diverse Interpolationsverfahren. Institut fuer Photogrammetrie und Fernerkundung, TU Wien. *Intern*.

Koestli, A., 1988. Manual for BC FORTRAN. BC Software, Bad Canstatt bei Stuttgart.

Kraus, K., 1992. Analysis of Geographic Data and Visualisation of Their Quality. Presented paper, ISPRS Congress, Washington,D.C.

Loitsch, J., Molnar L., 1991. A Relational Database Management System with Topological Elements and Topological Operators. Proceedings Spatial Data 2000, Dept. of Photogrammetry and Surveying, University College London.

Molnar, L., 1984. DTM-Verwaltung: DTMX. Institut fuer Photogrammetrie und Fernerkundung, TU Wien. *Intern.*

Rieger W., Automated River Line and Catchment Area Extraction from DTM Data. Presented paper, ISPRS Congress, Washington,D.C.

Sigle, M., 1991. Die Erstellung von Bodenerosions-gefaehrdungskarten auf der Basis eines DGM. GIS 4, pp 2-7.

Wild E., 1983. Die Praediktion mit Gewichtsfunktionen und deren Anwendungen zur Beschreibung von Gelaendeflaechen bei topographischer Gelaendeaufnahme. Deutsche Geodaetische Kommission, Reihe C, Heft 277, Muenchen.