15

Cartographic Computer Programming

15.1 LANGUAGES

The control necessary for implementing cartographic data structures, for performing cartographic transformations, and for anything other than applied computer cartography, requires the use of a programming language. Many students find their way into analytical cartography from other disciplines and as such are excluded from a more in-depth understanding of analytical cartography because they do not know how to program. Although much of the material presented in this book can be understood, and even taught, without the use of programming, the advanced student may wish to go further. If your background has not been in computing, the best advice is to learn programming from the experts. This usually means in a computer science department. Learning programming along with a graphics course is not recommended.

When we program, we usually do so in a programming language. Just as we use the English language to express our ideas to others, so we use a programming language to express our ideas to the computer. Computer programming languages perform sequences of instructions. This is the case even though our thoughts are often not sequential. A program moves from step A, to step B, to step C, and the move to step C does not start until B is reached. Human thought can easily branch from A to C, or to X, Y, and Z, or all four simultaneously, based on the loosest of connections.

Usually, a program or set of programs written in a programming language performs a specific task. Such a program is referred to as an *applications program*. The other general division of programming is *systems programming*, in which the programs control the operating system itself. All cartographic programming falls into the applications programming category, although there is much that cartographers have to gain from an understanding of systems programming.

All computer programs perform certain generic processes. Most programs perform *input;* that is, they read data from files or a user, they perform *operations* that transform the data in some way (such as between data structures) and then they produce *output.* The output can be a column of numbers, a new file, a graphic, or a map. The most important function of a computer programming language is to control the operation taking place. The programming language gives complete control over an application; it allows us to say exactly what we want to do, how we want to do it, what data should be processed, and where we want to put the output.

Languages have different levels. At the very lowest level of instruction computers use programs called microcode. Microcode is in binary and is so fundamental to the operations of the computer that without it you cannot even load the operating system or use the computer. Microcode instructions tell a computer where to find the operating system, what kind of computer it is, and how to start the *boot* or the initialization process.

At the next level is the machine language program. This language has to be tailored exclusively for a particular brand of computer, a particular type of memory, and so forth. Programs in machine code (language) talk directly to the hardware. Machine code has the disadvantage that not only are data required to be in binary but the instructions are also. The last of the low-level computer languages is assembly language. Assembler is compiled in the same way as higher-level languages. This means that a program can be entered and stored in a file, then processed into machine language via a large translation program called a compiler. The machine language program is then ready to run.

An assembler gives access to *registers* or parts of the memory into which we can write instructions and numbers. Assembly language uses as building blocks an instruction set, consisting of multicharacter mnemonics standing for individual operations. These operations codes perform simple operations, such as putting a number into memory, or taking the current number that is in one register, and performing a binary AND with the number in the current register. Assembler language programs are highly adapted to the architecture of the computer on which they run and as such are capable of running very fast. In many cases, programs are written in higher-level languages, and their most time-consuming modules are rewritten in assembly language for efficiency.

The remaining levels of computer programming languages fall into what are called high-level languages. High-level languages have named operations, expressions that perform control, and defined data types. Examples of widely distributed higher-level languages are FORTRAN (or FORmula TRANslation language) although it certainly was not the only one, BASIC, and COBOL. Others have found their way into general use. A distinguishing characteristic of these languages is that although they contain the mechanics to write modular programs, they do not make the modular structure an integral part of the language.

One step beyond high-level languages like FORTRAN are the structured languages. Structured languages, like Pascal and Ada, encourage the user to write programs that are modular. Modular programs consist of independent units that can be worked on in isolation from each other. A structured language encourages the splitting up of programs into smaller and smaller pieces. Each small program can be worked on, tested, and optimized independently, and then the modules can be assembled into a whole, working program.

Other structured languages are PL/1, ALGOL, APL, C, and RATFOR. In addition, a number of special-purpose languages that are suited to specific tasks, such as text and list processing, and complex database management exist at this level. Among these are LISP, PROLOG, MODULA, SNOBOL, and the object-oriented languages such as C++ and Smalltalk. Beyond the high-level language are application-level programs or macros within systems with their own internal programming language. Graphics languages exist that can perform manipulation of graphic objects and data structures with a command language rather than a programming language. For example, within AUTOCAD, you can program with LISP, or in many systems you can query a database using SQL (Standard Query Language).

The language used throughout this book is a language called C. C is the third version of a programming language developed at Bell Laboratories by Dennis Richie. Originally written on a DEC PDP-11 for the development of the UNIX operating system, the language is general purpose and terse; it supports advanced control and data structures; and it is free of many of the restrictions placed on other languages by their operating environments.

The language was chosen for this book because C language programs are brief, powerful, and support the complex data structures required for cartography and because C is highly portable to different computers and operating systems. The language was originally set forth in a book, *The C Programming Language,* by Brian Kernighan and Dennis Richie (1978). This book set the early C standard (K and R–C), and is the standard followed in this book. Since then, the American National Standards Institute has endorsed a version of C (ANSI-C). The second edition of Kernighan and Richie's book (1988) contains both standards and highlights the differences between them, which to an experienced programmer are minor and are almost entirely limited to type definitions

The C language gains in brevity by leaving out all special-purpose functions, such as mathematical operators, input/output, and text handling, and instead making it easy to build and use program libraries containing working algorithms for these functions. Thus, for example, where FORTRAN defines a SQRT operator to compute a square root, C instead assumes that the user will link the program with a "mathematical library" containing an implementation of an algorithm for computing the square root of a number. C compilers as a result can be relatively small, and many different compilers, debuggers, and programming aids are available for C, even on microcomputers.

In recent years, a preference has been shown for C++, an extension to the C language. C++ adds to the language support for the concept of a *class*, a necessary requirement for the use of object-oriented programming methods. Although object-oriented programming is possible in many other languages, the proliferation of C++ compilers has made C++ the development language of choice, especially because all C++ compilers include the ANSI C language as a subset, with complete forward compatibility. Students should note that effective C++ programming follows from a familiarity with and experience in the C language. Calls to GKS functions are as simple in C++ as in C, because the primary differences relate to how data are modeled (see Chapter 10).

C++ allows the programmer to group together data and operations, such as the actions produced by different functions, together in a class. The graphic object of a polygon encoded as a RING structure, for example, could have a set of functions that relate to it:

area computation, point-in-polygon testing, or centroid specification, among others. The complete specification of a set of classes or objects of geographic interest remains to be done. A subset of cartographic objects seems more feasible and is likely to follow the specifications of the DLG-E descriptions advanced by the U.S. Geological Survey.

Finally, C++ allows the specification of complex objects such as windows, user dialogs, and events. This means that C++ is particularly useful when the programmer wishes to use the GUI toolbox directly. Windows and X-windows coupled with C++, for example, allow the programmer access to the building blocks of the user interface with which the operating system itself is constructed, giving significant power of control to the programmer. A consequence, for example, is the ability to inherit the characteristics of the operating system's windows by an application after compilation. More extensive use of C++ is significantly increasing the ability of software to execute in the same way under a number of different GUIs and operating systems.

15.2 GOOD COMPUTER PROGRAMS

Given a group of programmers with different levels of experience and a problem, perhaps even a problem solved by a published algorithm, it is most likely that each programmer will write a different program to solve the problem. Some programs will not work, in which case it is easy to tell whether or not the solutions have value. Among the working programs, however, how can we tell which is the best, or even which are good? To be able to do so requires that we can tell a good program when we see one. Fortunately, good computer programs have some distinguishing characteristics; so much so, in fact, that it is possible to have a good program that does not work and a bad program that does work! In this section, we will try to cover some of the factors that make computer programs "good."

To begin with, a good computer program is *readable.* This means that programmers can read the program, working their way through it without spending an undue amount of time trying to figure out what the original programmer meant. "Clean code" is neatly formatted (usually by a "beautifier program") and is well presented. A good computer program is *structured.* A structured program has embedded within itself subprograms that do smaller and smaller tasks, so the main program consists of very general tasks. This program passes program control to more and more specific tasks, each of which works independently of the others, and returns control when the subtask is complete. This "task nesting" can make programs very manageable and can reduce programming time significantly. Structuring within some languages, such as within FORTRAN, is by function and subroutine. In the C language subroutines do not exist, only functions. So in C a program is defined as a function that is itself a set of functions.

The philosophy behind structuring is that programming complexity is worth minimizing. Writing tiny programs that do very specific tasks is comparatively easy, so if we can take a major task and divide it into many small tasks, each of which can be solved by writing a very simple program, programming complexity is minimized. This approach could be called the "divide and conquer" approach because a difficult problem, divided into less difficult problems, and finally into easy problems, expresses the original problem in a way that can be easily solved. Good programs are structured, and the structure is meaningful in the terms of the problem and works as a solution.

Good programs are *concise*, meaning that they do not contain any more instructions than are necessary to get the task done. A major cause of "verbose" programs is the rewriting performed when program maintenance is done. Rather than reworking the solution, the programmer simply bypasses the old instructions and adds new, duplicating whole sections. Modular programming is by definition concise. When modules are maintained, it is easy to remove a task and rework the solution as a new module or function.

Good programs should be *efficient*. Efficiency is usually measured in terms of the time taken for a task to complete. Many simple measures can be taken to make programs efficient, some of which can make vast improvements in execution time at very little cost. Good programs should be *usable*. Programs that are not usable probably have only one user, the author. Worse than this, once the author is finished using the program it will be forgotten. Modular programs encourage usability because modules can be interchanged. A programmer using structured programming rarely starts a new program from scratch. General-purpose functions can be used and reused in a variety of different programs and contexts.

Good programs are *documented* and *maintained*. Documentation, it is arguable, is more important than the program. Documentation can be internal in the form of headers and comments or external in the form of written text, either interactive (such as help screens) or paged. External documentation should be able to function in a user or tutorial capacity for the new user and as reference for the experienced user. Often this means two manuals, a user manual and a reference manual. Good documentation is understandable; it can be read by a novice user and understood.

Good software is maintained. Good software has somewhere near the beginning a version number, and the version number is high and decimal, such as 4.03. Constant updates mean that a programmer, preferably the author, has gone to the trouble of finding out what is wrong with the software and has changed it to make it better. Usually, for small changes the decimal place is increased. For major revisions of a program, the version number is increased.

Finally, good programs *work*. This means that they produce the correct answer, which is not always the answer we expect or that we want. Working programs must be reliable in two ways. First, they should survive changes in the operating system. Second, programs should be internally consistent. For example, a statistical program should give the same results using the same data more than once. This implies reliability in terms of consistency and in terms of resilience to change.

We have already noted the special demands of cartographic and geographic data. The volume of data that need to be handled influence how we write programs that work with geographic data. Geographic programs also have to run on a variety of different types of computers because no one computer is best for every geographic problem.

Cartographic computer programs very often use graphic output rather than sets of statistics or numbers. Cartographic programs use geographic data structures instead of the data structures that have been optimized for general-purpose data as in database management or numerical computing. Some of the more interesting and challenging data structures that exist are those that are specifically designed to deal with map data. Cartographic programs usually have a very different audience from general-purpose programs.

5.3 SOFTWARE DEVELOPMENT METHODS

15.3.1 Graphics, Standards, and GKS

Graphics are extremely important for analytical and computer cartography. Using graphics with a programming language requires some form of interaction between the program and the graphics. This is accomplished using a *language binding*. There are really two different types of graphics systems, each with different ways of achieving a binding. The first type is extremely device specific and involves a direct or indirect message to the graphics board, the special purpose computer that controls the graphic display. Many graphics programs use the memory addresses on the graphics card directly, and send values to the locations that cause the display to react.

For example, on an IBM PC using EGA graphics in mode 6, memory locations starting at address B800 (hexadecimal) and occupying 640 by 200 binary pixels control the screen display. A binary array written to these locations would automatically appear on the display, in fact, very quickly. We are virtually talking directly to the hardware.

Although very fast, programs that are written in this way usually only work on one type of graphics configuration and on one specific computer. Different computers have different screen sizes, different colors, different memory addresses and address ranges, and different ways of mapping a byte onto bits, and they support different numbers of colors, line thicknesses, and so forth. Not only must each graphics program be rewritten for each display device, it must also be rewritten for each computer. In a rapidly changing hardware environment, to be device-specific means almost instant obsolescence.

As a response, many software manufacturers offer "graphics toolkits," which allow some degree of program portability, but are still very device-specific. Many programming languages, such as microcomputer-based versions of languages like C and Pascal, offer graphics extensions or functions that produce graphics.

At the very simplest, a `MOVETO` and a `DRAW` are provided, which allow the user to move to a screen pixel location (the geometry of which is the responsibility of the programmer), with or without drawing a line from the current location. Many toolkits support some sophisticated options, such as graphics text, circles, ellipses, and polygon fills, but the programming level is still fairly close to the machine, and the resulting programs, while a little more portable, are still highly device-specific.

The alternative is to use graphics standards. Two major graphics standards have found their way into analytical and computer cartography. The first of these, CORE, was an outgrowth of a workshop group of SIGGRAPH, the Association for Computing Machinery Special Interest Group on Graphics, in April 1974. The early standard was published in 1977, with completion in 1979. This standard was important in establishing terminology and concepts. The standard was broadly implemented, mostly with FORTRAN bindings, during the 1970s and 1980s. Starting in 1979, an ANSI Technical Committee on Computer Graphics used the CORE system as a model in designing a now well established two-dimensional standard called GKS. Almost all the facilities available in CORE were carried over into GKS and its three-dimensional equivalent, GKS-3D.

CORE showed users the advantages of separating modeling and viewing functions.

This means that a graphical "object" such as a cartographic object, can be defined, given attributes, and so forth, independently of the geometry and properties of the final image, an idea termed *virtual maps* (Moellering, 1983). CORE's weaknesses were the concentration on line-drawing functions, its inability to support a multiuser, workstation environment, and the failure to carry the standard over to include language bindings.

This last weakness meant, for example, that function and subroutine names, the order of arguments in function calls, and even the number of arguments and their meanings were changed by individual software producers. This led to a failure of CORE-based programs to be very portable, one of the principal advantages of using a standard in the first place. The CORE system has not been considered for formal adoption as a standard by any ISO or ANSI committee and will not be in the future because most of its concepts are now embedded in GKS.

The purpose of using a graphics standard is to ensure that a program written using the standard will survive for new systems, will be portable, and will work independently of the specifics of implementation. Using standards, we sacrifice direct access to a device, although GKS provides a high level of specialized access through the generalized device primitive and special escape sequences supported by the language binding. The ability to support such a generic work environment is achieved by separating the graphics into the graphics system, the language binding, and device drivers.

Device drivers are computer programs that convert graphics operations into instructions that communicate directly with the hardware. This requires a device driver for each particular device we are going to use, so that users must add device drivers as new devices are acquired. For the programmer, the trade-off is flexibility for drawing speed. Device independence also means that the same map can appear and look the same on a printer, on an electrostatic plotter, a camera, or a window of a workstation.

As we saw in Chapter 11, with a graphics standard we use three different coordinate systems. We call the coordinate system of the application the *world coordinate system*. Graphics standards use *normalized device coordinates* (NDCs), which are coordinates on a virtual device. A *virtual device* is an idealized planar space on which the map is to be drawn. Under GKS, the lower left corner of the NDC space is at `[0,0]` and the upper right corner is at `[1,1]`. In the CORE system, which supports three dimensions directly, the center of the screen is `[0,0,0]`, the upper right corner is `[1,1,0]`, and the lower left corner is `[-1,-1,0]`.

At the display end, there are workstation or *device coordinates* which are locations on the display surface as referenced using the mechanics of the display system. Under a graphics standard, the specifics of the device coordinates are unnecessary, although we can inquire their values if need be. In addition, once a window and viewport transformation have been established, the programmer can reference all locations with map coordinates, and specific locations on the map such as locations of legends, titles, insets, and so forth, in normalized device coordinates.

Graphics standards also allow maps to be structured. At the lowest level, the map consists of primitives. Primitives can be collected into *segments.* Segments can be named, numbered, and recalled by reference instead of being rebuilt from data. Different transformations can be applied to whole segments. Similarly, higher-level implementations of the standards support the *metafile.*

The metafile is a generic storage space that behaves the same as a device or workstation. Thus to save the outline of an island, a collection of polylines representing the island could be named as a segment, be written into a metafile, and then later during the same program run, or even later by another computer program, be retrieved and be redrawn with different transformations or attributes. It is sometimes possible to retrieve these metafiles from different programs, perhaps even in different languages.

An additional feature of the standards is the ability to cluster groups of attributes into *bundles*. A bundle can be selected and uniformly applied to either segments or primitives. For example, we could define a set of attributes that should apply to map titles, such as boldface fonts, large letters, a specific spacing of letters, a particular color, and text centering. By referencing this bundle, we can apply it to a specific piece of text that is to be a title, rather than changing all the attributes as we go.

A typical computer graphics program sets the attributes after having computed where it wants to draw primitives, opens a segment, simultaneously defines and draws a segment, and then closes the segment. For example, to draw a solid red polygon first we need the boundaries of the polygon, usually from data. Next we assign characteristics, such as a solid red fill. With this done, we simply open a segment, draw the polygon, and finish. Once drawn and defined, the red polygon can be recalled very easily just by referring to its segment identifier.

In Chapter 16, we will see how to write a graphics program using the GKS standard. The GKS standard has become an important part of graphics programming, especially since the standard's adoption by the American Standards Institute and by the International Standards Organization in 1985. The standard includes a functional description of GKS (ISO 7942:1985), the metafile (ISO 8632-1,2,3,4:1987), and bindings for FORTRAN (ISO 8651-1:1988), Pascal (ISO 8651-2:1988), and Ada (ISO 8651-3:1988). The three dimensional extension of the standard was approved by ISO in 1988 (ISO 8805:1988).

The ISO review of the standard involved over 100 scientists and about a 50 person-year labor effort. The standard has been adopted for several books in computer graphics, including Enderle, Kansy, and Pfaff (1984) and Hearn and Baker (1986). Although other standards, discussed in the following section, are often more suitable for specific graphics and even some cartographic applications, GKS remains the benchmark against which all computer cartographic software in the future will be measured.

15.3.2 Higher-Level Graphics Languages and Standards

Standards in the computer graphics industry have been slow to develop, yet offer a large number of advantages to the programmer. A standard expresses a nominal set of requirements, a minimal level of performance, and a mandatory area of compliance and conformance to accepted specifications (NCGA, 1987).

In the United States, it is the American National Standards Institute (ANSI) that reviews and approves standards and represents the United States in the International Standards Organization. The X3H3 committee of ANSI started examining standards for computer graphics in 1979.

The X3H3 committee initiated GKS, the Computer Graphics Metafile (CGM), and

the Programmer's Hierarchical Interactive Graphics Standard (PHIGS), the Computer Graphics Interface (CGI) as well as GKS-3D.

GKS-3D extended GKS to handle the definition and viewing of three-dimensional "wire-frame" objects. The system supports hidden-line and hidden-surface removal in the workstation transformation, but does not support rendering techniques such as shading and light sources. GKS-3D places a significantly enlarged set of demands upon the host computer, particularly the amount of memory used and the processing power required. As such, microcomputer applications may be limited to only the most powerful machines.

The Computer Graphics Metafile is a file format suitable for saving a "snapshot" of graphics in their final form. Access to images stored in a metafile is either sequential or random access, and the image is defined in a device-independent way. The purpose of the CGM is to allow images generated on different graphics systems to be interchanged and regenerated on different computing and display environments. The three parts of CGM contain the character encoding for assisting in network communications, the binary encoding to support transfer between machines with different word sizes and operating systems, and the clear-text encoding to ensure maximum readability for use and debugging

PHIGS, like GKS, is a functional definition of the links between a programming language and a graphics system. PHIGS supports some operations unavailable in GKS, including the ability to nest graphical objects hierarchically (important in modeling applications and CADD) and graphical object database support. PHIGS is especially useful when complex objects are to be constructed from smaller simpler objects; for example, a fan can be constructed by defining a blade and then repeating it with rotation.

PHIGS supports three dimensions and interactive graphics, such as panning and zooming, at the workstation level. The increase in PHIGS's flexibility and complexity, however, means that many workstations, particularly at the lower end of the power scale, are not capable of supporting all the capabilities of a full PHIGS implementation. PHIGS has come to be used, therefore, in high-end interactive workstation applications, particularly in computer-aided design (CAD), architecture, and computer assisted manufacturing (CAM).

The Computer Graphics Interface (CGI) specifies a set of basic elements for the control of data exchange between the device-independent and device-dependent levels in a graphics system. This involves a standardized virtual device interface (VDI). CGI can operate as software to software, in which case it transfers graphics from metafile to binding, or vice versa.

As a software-to-hardware link, CGI allows stored graphics to be output to any supported device independently of the graphics system which generated them. Thus a map stored in CGI could be reproduced using the available display devices or retrieved for analysis via a computer programming link. The most important function of CGI is to allow device-specific graphics systems to produce output that is in accordance with standards and so to support a larger number of devices.

A large number of additional graphics systems exist at this higher level. For window-based applications, the X-windows system from M.I.T. has gained widespread acceptance. In addition, standards for graphical rendering, and even entirely open graphics-based operating systems, are now in the works. It is important to remember that to be of use a standard should be both supported and used. Programs written to any standard will

not have to be rewritten as hardware and software change flavor with the times.

If the early days of computing can be classified as the era of hardware, then the current generation will clearly be the era of software, and above all, programming standards. Links with the standard version of C, now approved at the ANSI level, and with the Spatial Data Transfer Standards, ensure that the duplication of effort and the time wasting of the early days of analytical and computer cartography are over and that cartographers can concentrate upon the more substantive aspects of their discipline, safe in their knowledge of how to produce the map.

15.4 REFERENCES

Enderle, G., K. Kansy, and G. Pfaff (1984). *Computer Graphics Programming: GKS—The Graphics Standard.* Symbolic Computation, New York: Springer-Verlag.

Hearn , D., and M. P. Baker (1986). *Computer Graphics.* Englewood Cliffs: Prentice Hall.

International Standards Organization (1985). *Information Processing Systems—Computer Graphics—Graphical Kernel System (GKS) Functional Description.* ISO Publication No. 7942:1985.

Kernighan, B. W. and D. M. Richie (1978). *The C Programming Language.* Prentice Hall Software Series, Englewood Cliffs: Prentice Hall.

Kernighan, B. W., and D. M. Richie (1988). *The C Programming Language*, 2d ed. Prentice Hall Software Series, Englewood Cliffs, N.J.: Prentice Hall.

Moellering, H. (1983). "Designing Interactive Cartographic Systems Using the Concepts of Real and Virtual Maps. " *Proceedings, AUTOCARTO 6,* Sixth International Symposium on Computer-Assisted Cartography, Ottawa, October 16–21, vol. 2, pp. 53–64.

National Computer Graphics Association (1987). *Standards in the Computer Graphics Industry.* Fairfax, VA: National Computer Graphics Association.