May 2005

Mapalester:Powerful, East-to-Use GIS Software Under Development

Brent Hecht
Macalester College

"Mapalester: Simple, Powerful GIS Software Under Development"

MAPalester

GIS software *powerful › easy-to-use › free*

Geography Honors Project
Computer Science Capstone

Brent Hecht

Advisor: Dr. Laura Smith, Geography
Comp. Sci. First Reader: Dr. Susan Fox
Comp. Sci. Second Reader: Dr. Libby Shoop

1. Introduction

1.1. Purpose of the Mapalester Project

Over the past nine months, I have been developing a new GIS software application named Mapalester, after my soon-to-be alma mater. The goals of the research project with Mapalester as its product are defined by the three target markets of the product: K-12 education, non-profits, and individual/personal use. I identified these markets as markets that had one or more needs for GIS software that is/are currently unmet by the body of currently-available GIS software. These unmet needs vary widely among the markets, and I struggled at first to identify a reasonable set of development goals that would equip Mapalester with the functionality that would meet all three of these markets' needs. In the end, however, I was able to construct four broad development goals to which I aligned my design and programming efforts. First, Mapalester needs to be powerful. In other words, if Mapalester is not able have some basic set of GIS functionality, Mapalester will be more or less useless to all three of Mapalester's target markets. Second, Mapalester must be easy-to-use. Third, Mapalester must work on both the Mac and PC platforms. Finally, Mapalester must be available for free.

In the subsequent parts of the introduction, I will discuss the specific unmet GIS needs of each of the target markets and will explain how these four development goals – when fully implemented in Mapalester – will enable Mapalester to meet those needs. A diagram of the needs and markets is found in figure 1.1a. To confine this discussion to merely the introduction of this paper is to falsely reduce the complexity and, in many cases, the immensely troubling nature of the unmet GIS needs in the three target markets. However, the focus of this project – at least at this stage – is not to explore in depth the effects of a powerful, easy-to-use, Mac-compatible, and free GIS on these target markets. Rather, this project is by and large a GIScience software engineering project. The subsequent discussion is intended to provide some context for my GIScience software engineering work. A very valuable direction of future research would be to examine these target markets and development goals in greater detail. My hope is that this future

research will eventually be enabled with empirical data from observing the effects of Mapalester after its release.

Target Market	Unmet Need
K-12 Education	Software to increase adoption of GIS as a method of teaching existing curriculum
Non-profits	Software to empower non-profits with the same GIS capabilities as for-profit companies and government.
Personal Use	Software to facilitate the long-overdue transition of GIS from the lab to the consumer desktop.

Figure 1.1a – The unmet needs of Mapalester's target markets.

Moreover, before this more detailed discussion of the reasoning behind Mapalester's basic design, it is important to note that embedded in the goals and target markets of Mapalester is an inherent commitment to helping leveling the social playing field by removing socioeconomic barriers to the incredibly powerful tool that is GIS. This commitment is similar to that present in an Al Gore speech cited by Elwood (2002)

> "[GIS will] help communities help themselves by putting more control, more information, more decision-making power into the hands of families, communities, and regions to give them all the freedom and flexibility they need to reclaim their own unique place in the world." (State Cartographers Office 1998)

1.1.1 Analysis of the K-12 Education Target Market

At first, it may seem ridiculous to think of Geographic Information Systems in the context of primary and secondary education. Today, GIS is often thought of as a complex, professional- and academic-level tool. After all, GIS is taught at the undergraduate level, and masters degrees in GIS are becoming widely available in most countries with

graduate-level Geography programs. Indeed, for most of its uses, GIS is a very powerful – and correspondingly complex – tool. However, as documented by Kerski, Baker, Bednarz, and others, GIS can have many helpful applications in primary and secondary education curriculum. "Since the First National Conference on the Educational Application of Geographic Information Systems in 1994, researchers and educators have repeatedly identified the merits....of including GIS in elementary and secondary classrooms." (Baker, Bednarz 2004) Kerski skillfully distinguishes the applications of GIS that are helpful at a K-12 level from the applications at higher levels of education through a simple categorization method (emphases are my own):

> "GIS is used in three major ways in education at the elementary, secondary, and university level. First, teaching about GIS dominates at the university level, where courses in methods and theory of GIS are taught in geography, engineering, business, environmental studies, geology, and in other disciplines. Every major university and most community colleges in the USA host a GIS program. Second, teaching *with* GIS is emphasized at the elementary and secondary level, where GIS is increasingly used to teach concepts and skills in earth science, geography, chemistry, biological science, history, and mathematics courses. Finally, GIS is used as a fundamental research tool in all institutes of higher education in geography, demography, geology, and other disciplines." (Kerski 2004)

Despite these widely acknowledged benefits and well-defined role of GIS in the K-12 market, all those with knowledge of GIS in education agree that efforts to incorporate GIS into the K-12 curriculum have mostly failed thus far. In the context of the infamous "Innovation Adoption Curve," Baker and Bernardz note that GIS in education has not made it past the early adopter phase in K-12 education. In other words, "GIS education is still struggling to win a wider audience among those educators who serve as role models and opinion formers for the majority of teachers, characterized by respectable early adopters" (Baker, Bednarz 2004).

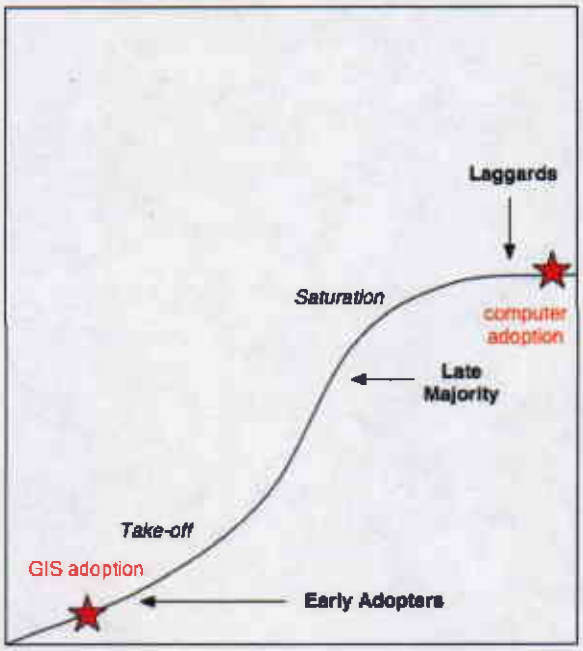


Figure 1.1.1a – The Innovation Adoption Curve for GIS vs. computers in K-12 education.

Therefore, stated broadly, the unmet need of the K-12 education GIS market a GIS technology that will enable schools to successfully teach *with* GIS, forcing the progression of GIS technology along the innovation adoption curve in the K-12 market. In other words, the goal for Mapalester is to facilitate the incorporation of GIS into the K-12 curriculum. Each of the four development goals mentioned above is designed to empower Mapalester with the characteristics and features necessary to fill this void in the K-12 GIS market. In the subsequent paragraphs, I will elaborate upon the exact manner in which the four design goals of Mapalester help to meet this need.

The first design goal – that Mapalester must be a reasonably powerful GIS – is an absolutely essential prerequisite for any GIS software that attempts to help incorporate GIS into the curriculum of K-12 education. In order for Mapalester to be an option for K-12 GIS, it must have the entire set of basic GIS functionality required in the use of GIS in the K-12 curriculum. Rather then identify the exact set of functions necessary for K-12 education, I have assumed that K-12 requires the same basic set of functionality as the two other target markets. I have defined this basic set of functionality as all of the features that I was taught in two semesters of undergraduate-level GIS. This includes features ranging from basic raster and vector GIS tasks to raster-based spatial analysis to geocoding. After

this basic set is completed, it would be an interesting extension of research to try to fine tune Mapalester's functionality for the needs of the education market. I have done some initial thinking – based on the literature and on my own assumptions – on what this fine-tuning would entail, a description of which can be found in section 4.6.

Judging from existing literature, the most important design goal for the K-12 market is ease-of-use, my second design goal. From the literature, it seems that the largest barriers to incorporation of GIS into the K-12 market lie in the complexity of currently available GIS software. This complexity inhibits the adoption of GIS in nearly all phases of the education process – from in-service professional development to the development and implementation of lesson plans to students' learning of the curriculum through GIS.

The third design goal is almost exclusively for the K-12 education market. According to Quality Education Data (QED), in 2003, 28 percent of the installed base of the computers in K-12 schools in the United States is made up of Apple Macintoshes. While more recent data suggests that Dell is far outselling Apple in this market (41 percent to 14 percent), Macs will remain a large portion of the K-12 education market for a long time to come. Moreover, Apple has made regaining the lead in the K-12 education market a high priority in recent months, and has had some success at doing so. As such, the design goal that Mapalester must be able to work on both PCs and Macs is an essential design goal for the education market.

When first investigating the K-12 GIS market and its unmet needs, I assumed that the most important design goal would be extremely low cost. However, at least according to the K-12 GIS literature, cost does not seem to be the pre-eminent barrier to adoption of GIS into the curriculum. Rather, as mentioned above, the complexity of use of currently available GIS software seems to be the largest such barrier. In fact, in Kerski's seminal paper on GIS in the K-12 market, he does not identify cost at all as a significant reason why GIS has not made it past the early adopter stage in the innovation adoption curve. This omission probably revolves around his reliance on a survey sample that only included registered owners of the top three GIS applications who listed their occupation as K-12 educators. A helpful path of future research would be to delve into the immense body of literature covering the "digital divide" in education and to apply findings from this literature to a GIS context. Kerski does mention, however, that only 5 percent of

American high schools have access to GIS software. As such, while the "free availability" development goal may not have much of an effect for these 5 percent and the likely smaller percentage of American middle schools and schools located in other countries that implement GIS, it will play at least some role in GIS adoption in the vast majority of this target market. It is no doubt a lot easier to convince a school administration to invest in GIS when it costs nothing to do so. Figure 1.1.1b shows the approximate cost of implementing GIS using in a K-12 environment. Note that the prices are essentially rental fees; the software must be renewed each year for the price seen in figure 1.1.1b.

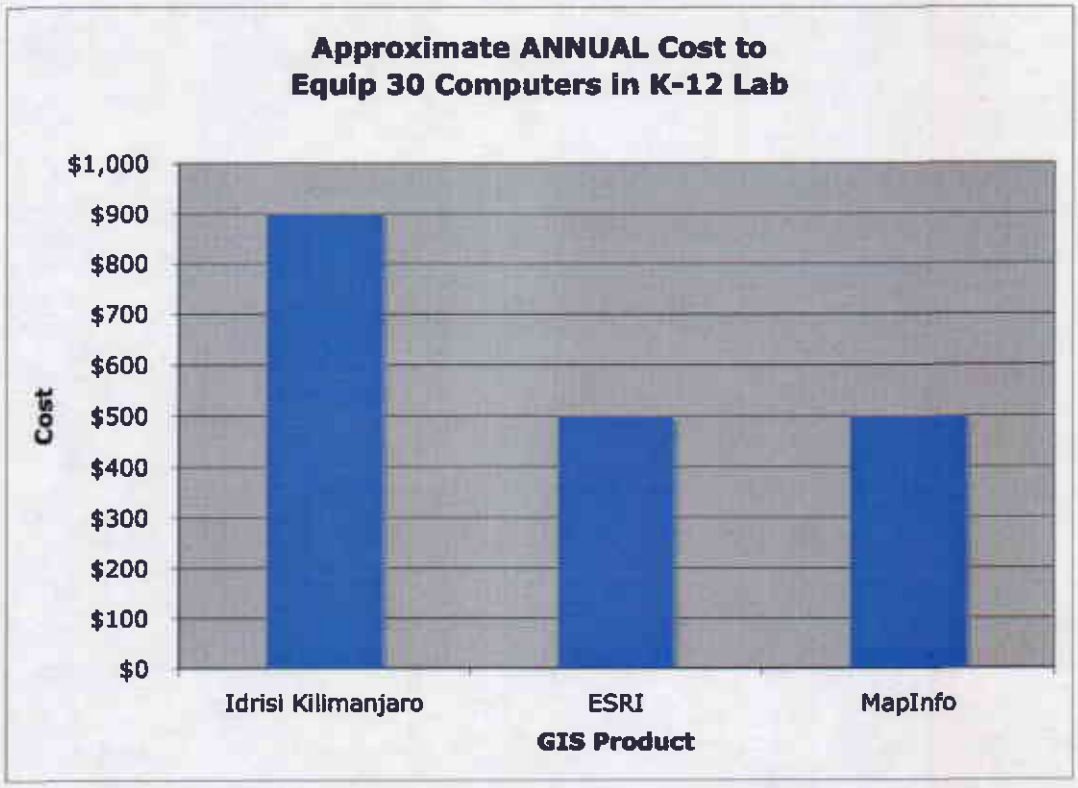


Figure 1.1.1b – Cost to equip 30 computers in a K-12 computer lab or classroom for one year. Clark Labs, developer of Idrisi, has a different pricing scheme than the ESRI (Environmental Systems Research Institute) and MapInfo, and the $900 figure is an estimate based on the many pricing plans offered by Clark Labs.

If we localize the innovation adoption curve to a sample individual school, it is easy to see that it would be impossible to make it past the early adoption phase without the administration making the very basic element of GIS adoption – the GIS software itself - available to the early and late majority population of teachers. Indeed, in a personal

interview, Sara Damon, a teacher at Stillwater High School in Stillwater, MN indicated that while there are relatively steep discounts available for schools that want to buy GIS software, a free software package would be a huge boon to schools and school districts that have not yet invested in GIS software.

1.1.2 Analysis of the Non-Profits Target Market

According to Elwood (2002), authors in the GIS literature have found that "information technologies and GIS...benefit [non-profits and community groups] by expanding participation in decision-making processes and by increasing their political power." (Elwood 2002). Although Elwood identifies some nuanced problems with their analyses, Elwood more or less agrees with these authors' conclusions. However, Elwood also notes that for GIS to allow for true empowerment of non-profits[1], there must be some way to free the non-profits that use GIS from the external power structures that often accompany the use of GIS. For instance, for many non-profits to utilize GIS in their activities, they must either obtain funding to purchase GIS software or they have to rely on GIS "shops" and colleges and universities. Thanks to its core development goals, Mapalester will be able to simultaneously benefit the non-profit community in the ways identified above and free non-profits from the external oversight and control that usually accompanies GIS adoption.

Because of my dedication to making Mapalester into a "powerful" GIS software application, Mapalester should be able to benefit non-profits in the ways identified in the GIS literature by Elwood. As long as Mapalester can provide most, if not all, of the GIS features and functionality required by non-profits, Mapalester will inherently result in these benefits, according to the literature.

Mapalester provides GIS power to non-profits without the negative side effects of currently available commercial GIS software applications through the "free" and "easy-to-use" development goals. Obviously, by making Mapalester a free download, non-profits wishing to utilize GIS no longer will have to rely on outside funding sources.

---

[1] I will group together non-profits and community groups under the umbrella title of non-profits for the remainder of the paper.

However, another barrier to GIS usage in non-profits is likely the complexity of existing GIS software. Even free GIS software is useless to a non-profit unless there is someone in the organization who knows how to use it. Lack of GIS experience could drive non-profits back into the arms of external groups, even if free GIS software is available. Therefore, similar to the benefits of the "ease-of-use" goal for the K-12 education market, Mapalester should both decrease the amount of training necessary to use GIS and, in the process, increase the number of people in a given organization who are capable of using GIS. Hopefully, Mapalester will be able to do so enough that a large number of non-profits will be able to internalize their GIS operations.

1.1.3 Analysis of the Personal Use Target Market

Despite having been widely used for over thirty years, GIS has yet to make the jump from the business office, government center, and laboratory of academia to the desktop of the consumer. Simply stated, the goal of Mapalester in the context of this target market is to be the springboard on which GIS can make that jump. The two main reasons that GIS cannot be used on the personal computer are price and compatibility. The "free" and "Mac-compatible" development goals directly target these causes. Also, the "powerful" development goal is a prerequisite for the other two goals. Ease-of-use does not generally play a role for the GIS fan who wants to use GIS for her or his personal use. In the subsequent paragraphs, I will discuss the impact of the "free" and "Mac-compatible" development goals on this target market.

Before I engage in this discussion, however, an examination of hardware requirements of GIS in the context of the average consumer's personal computer is appropriate. In other words, one possible reason that GIS could not be used on personal computers is that personal computers generally do not have the "horsepower" to handle GIS. This possible reason, however, is completely rebuffed by the table in figure 1.1.3a. This table shows that there is nothing special about GIS that makes it require extra hardware compared with a sample of standard consumer PC applications. The table also shows that nearly all home PCs made in the last several years or so should be able to run

GIS satisfactorily. In other words, the installed base of consumer PCs is more than ready for GIS.

	Operating System	Processor	Memory	Storage
iTunes 4.7	2000 or XP	500MHz	256MB	n/a
Office Professional 2003	2000 or XP	233MHz	128MB	400MB
ArcGIS 9	NT, 2000, or XP	800MHz	256MB	n/a
Idrisi	98/ME, NT, 2000 or XP	n/a	128MB	600MB
MapInfo 7.8	98/ME, NT, 2000 or XP	n/a	32MB	103MB
Dell's $299 PC	*XP*	*2.4GHz*	*256MB*	*40GB*

Figure 1.1.3a – System requirements for a sample of consumer software applications and the leading GIS software applications. The current bottom-line PC configuration from Dell is included for comparison.

Figure 1.1.3b displays the price of the leading GIS software applications for the average consumer. The cost of several industry-standard consumer software applications are provided for comparison. It is easy to see that, at best, GIS software ranks among the most expensive of consumer software applications, and at worst, well outside the consumer software price range. Note that ESRI (Environmental Systems Research Institute) charges three times as much for ArcGIS 9 with Spatial Analyst as Adobe does for its entire suite of creative applications that includes such powerful tools as Photoshop, Illustrator, InDesign, GoLive, and Acrobat. Obviously, cost is a very important barrier to the adoption of GIS on the consumer desktop. As such, the "free" design goal will make

Mapalester a very appealing product within this target market.

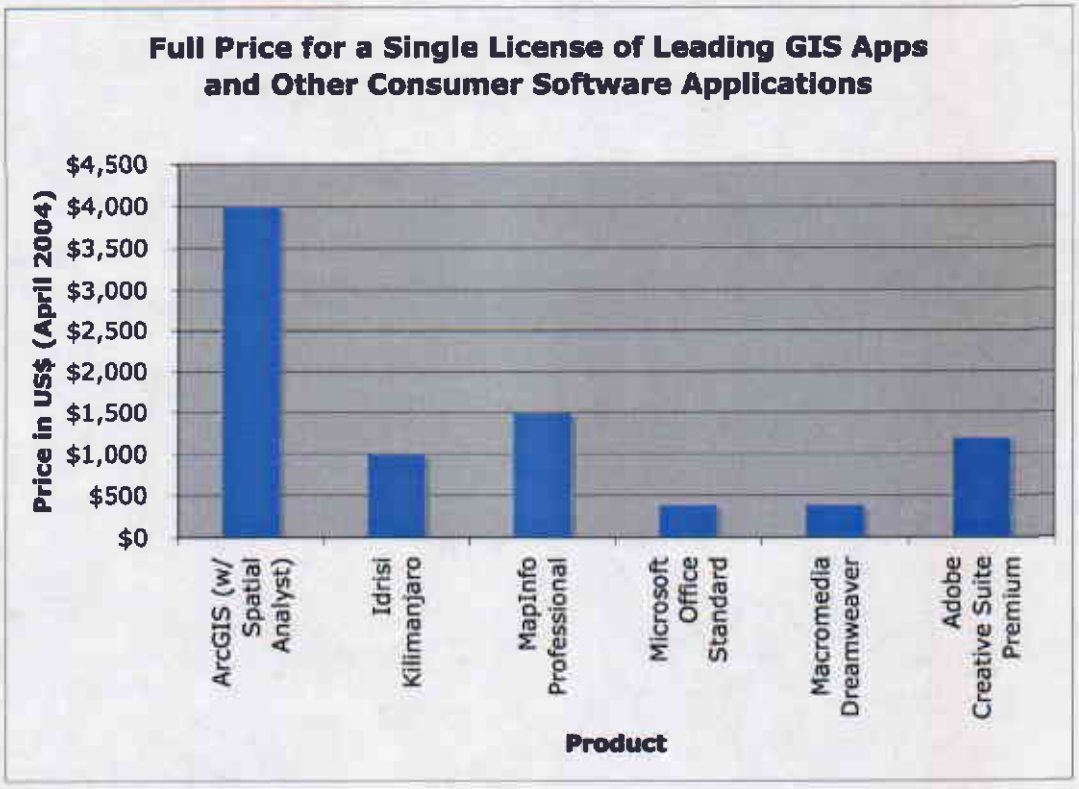


Figure 1.1.3b – Prices were gathered from a variety of sources including the manufacturer and the manufacturer's price on Amazon.com.

Even for students, for whom software is cheaper thanks to academic licensing and who will likely make up an important portion of the personal use target market, Mapalester's price will be very appealing. Figure 1.1.3c is the student pricing analogy to figure 1.1.3b. Despite the significant student discounts provided by some of the GIS providers, Mapalester will still save students at least $250. In addition, students using Mapalester will not be subject to some of the restrictive usage policies that often accompany student discounts. For instance, the academic discount for REALbasic, the software used to program much of Mapalester, demands that no commercial software be produced with the copy of the software purchased with the academic discount.

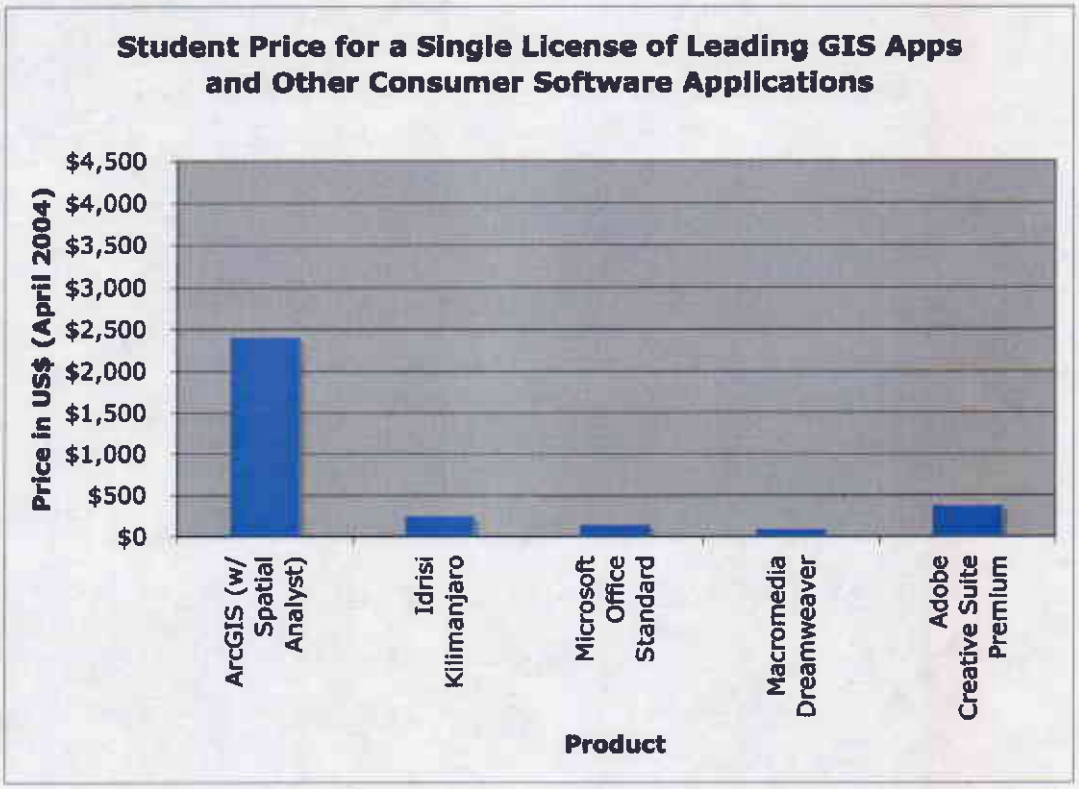


Figure 1.1.3c – Prices were gathered from a variety of sources including the manufacturer and the manufacturer's price on Gradware.com, a leading distributor of student-discounted software. MapInfo student discount pricing was not easily available.

The "Mac-compatible" development goal is also essential to the personal use target market. As of this writing, there are no viable, modern GIS software applications available for the Macintosh platform. At one point, ESRI developed GIS software for the Mac, but the last version was released with Mac OS compatibility is ArcView 3.0. MAPublisher ($999, Avenza Software) is still regularly updated and released, but it is distributed as a plug-in for Adobe Illustrator and Macromedia Freehand. In other words, MAPublisher is intended almost entirely for cartographic purposes, not GIS purposes. As such, the Macintosh GIS market is essentially open for Mapalester's taking. According to market research firm IDC (Dalrymple 2005), Apple had about 2.88 percent of the United States desktop marketshare and 4.99 percent of the United States portable computer marketshare in the fourth quarter of 2004. The corresponding figures for Apple's global marketshare were 1.75 percent and 2.93 percent.

1.2 Introduction to the GIScience Engineering of Mapalester

Now that the context for Mapalester has been established, most the remainder of the paper is dedicated to the GIScience of Mapalester. The paper is divided into four parts. The first section is this introduction to the paper. In the second section, I will discuss the basic engineering structure of Mapalester, the decisions behind this structure, and my reasoning behind each of the decisions. The focus of the third section is a description of the heavily object-oriented structure of Mapalester. The fourth section is dedicated to detailed analyses of the computational geography, computer science, and software engineering behind some of the key functionality of Mapalester. This section, which covers both completed and work-in-progress functionality, is quite large and is subdivided into subchapters, each of which covers a specific feature or feature set. Finally, the fifth section will conclude the paper with an analysis of the success of the Mapalester research project thus far and a discussion of the project's future.

However, before launching into a discussion of technical aspects of Mapalester, it is helpful, due to the computer science context of this paper, to engage in a brief discussion on the history, use, promise, and current state of GIS technology.

1.3 Introduction to GIS

In the shortest terms possible, a GIS, or Geographic Information System, provides a "technology and method" (Kerski 2004) to "visualize, analyze, and display" (ESRI 2005) spatial data, or data that can be linked to a location. Some estimates suggest that up to 80 percent of all data has such a component (ESRI 2005). Much in the same way that graphs help people visualize non-spatial data, a GIS allows its users to view and analyze data linked to a location within its spatial context. Applications for GIS exist in nearly every single discipline within the social and natural sciences, as well as in a large number of commercial, government, and non-profit operations. Obviously, with such a large and diverse group of users, important examples of applications of GIS are quite common. Fire and police departments around the world are using GIS to improve and

make more efficient their provision of emergency services. Political campaigns use GIS to analyze socioeconomic spatial data in order to more accurately target their limited funding. Similarly, the major national news services all used GIS to aid in predicting the turn-out of the 2004 U.S. presidential election. Thousands of companies have used GIS to assist in location planning and delivery management. The United States military uses GIS for a vast array of tactical activities. More relevant to the target markets of Mapalester, non-profit organizations that can afford GIS are using it to empower themselves with a means of visualizing and analyzing data in way they were never able to before. Civic Builders, a non-profit in New York that is dedicated to locating new charter schools where they will serve the New York community the best, has used GIS to "greatly facilitate their analyses." (Keohane 2005) Also in New York City, The Robin Hood Foundation, which is dedicated to helping rid the city of poverty, "initiated a major redirection of resources" (CMAP 2005) after using GIS to analyze current funding sites. Moreover, CMAP, or Community Mapping Assistance Project, is a NYPIRG project entirely dedicated to helping non-profits empower themselves with GIS. The immense potential of GIS has been making news lately. In 2004, the U.S. Department of Labor listed GIS as one of the three "most important emerging and evolving fields" (Gewin 2004). The other two listed were biotechnology and nanotechnology.

How exactly does GIS enable such a diverse array of uses? In the context of this paper, the best way to answer this question is through several examples. A business would use GIS to help locate a new store by combining data layers of all of the variables it would consider important to such a decision and then finding the site that optimizes all of these variables. For example, a person wishing to open a small retail business that sold ethnic foodstuffs in a large city would get ethnicity data by block group from the U.S. census and zoning data from a local zoning authority, and would use GIS to find the optimum location for her or his store in the city. A larger company or franchise would probably have a much larger array of data to optimize. In another example, for a study on Christian contemporary music, I used GIS to significantly enhance my knowledge of the subject. By creating a database of over 500 large concerts by famous Christian contemporary musicians and referencing those concerts by their location in a process called "geocoding," I was able to develop a map of the density of the popularity of

Christian contemporary music. Using the same methodology – but with already-existing spatially-referenced data sets – I created maps for a variety of demographic variables and churches of a variety of denominations. I then ran regressions on these density grids and the Christian contemporary music density grid, and was able to determine which of these factors had a causal relationship with the location of Christian music concerts. Although GIS is most often used with data in the context of small and large-scale locations on the surface of the Earth, it has also been employed in such diverse applications as studying the human brain (Zaslavsky et. al. 2004). Any data that can be defined as attributes of a location, no matter how large-scale or unrelated to the traditional discipline of geography (like information about areas of the brain), can be visualized, analyzed, and displayed in a GIS.

Geographic Information Systems are commonly defined in the literature as having four components: hardware, software, data, and personnel. The personnel in the system ask the geographic question, the hardware and software provide the technology for answering that question, and the data provides the information. Mapalester provides the software and, to some extent, the data parts of the system; the user of Mapalester is expected to supply any data not included with Mapalester, hardware, and the personnel part of the system. GIS is commonly mistakenly thought to refer to "geographic information software," but in fact, the software is only one of the four essential parts of any geographic information system. Similarly, GIS is often confused with GPS, or Global Positioning Systems. Although GPS units are one of several ways in which the spatial component of data is collected, GPS is a field separate from that of GIS.

2. Basic Engineering of Mapalester

2.1 Introduction to the Engineering of Mapalester

The most critical aspect of the core engineering structure of Mapalester is the division of coding into two very distinct parts with a very specific and limited interface between the two. Most of the front-end work and some of the algorithmic code is in a programming language called REALbasic. The majority of the algorithmic code, as well as all of the lower-level functionality,is in C++. The REALBasic integrated development environment (IDE) offers an impressive but somewhat limited C++ plug-in interface that allows most C++ code to run in the REALBasic front-end, but the communication between the REALbasic interface and the C++ code is restricted by the application programming interface (API) for the REALbasic plug-in. The interface is also limited by the abstraction-oriented development style inherent to plug-in architectures.

In the remainder of this section, I will provide a brief description of REALbasic and the REALbasic plug-in API. I will also discuss the reasoning behind the decision to use this REALbasic/C++ two-part architecture given the original goals and target markets of Mapalester. This discussion will include a response to criticisms over my choice not to use Java for the project. I will also discuss the specific challenges encountered due to the architecture.

2.2 What is REALbasic?

In the words of REAL Software, the developer of REALbasic, "REALbasic is the powerful, easy-to-use tool for creating your own software for Macintosh, Windows, and Linux." In other words, REALbasic is an integrated development environment (IDE) that allows programmers to generate native code for Macintosh, Windows, and Linux from a single set of code in the REALbasic language. Probably the closest analogy to the REALbasic language and IDE is Microsoft's VisualBasic language and IDE, although elements of Java (especially the heavily object-oriented focus) and C++ can be found in the REALbasic language as well. For those readers who are unfamiliar with these products, figure 2.2a shows code that sums the squares of the first 10 integers and returns

the value to the user. While REALbasic and VisualBasic have nearly identical code in this case, they diverge significantly in more complicated algorithms and when objects become involved.

VisualBasic

```
Sub first10Integers()
   Dim i, sum As Integer

   sum = 0
   For i = 0 To 9
      sum = sum + i ^ 2
   Next

   MsgBox (sum)

End Sub
```

REALbasic

```
Sub first10Integers()
   Dim i, sum As Integer

   sum = 0
   For i = 0 To 9
      sum = sum + i ^ 2
   Next

   MsgBox (str(sum))

End Sub
```

C++

```
#include <iostream>

int main (int argc, char * const argv[]) {

      int sum, i;
      for (i = 0; i < 10; i++){
            sum = sum + i*i;
      }

      std::cout << sum;
      return 0;
}
```

Java

```
public class SumSquares {

   public static void main (String args[]) {

         int i, sum;
         sum = 0;
         for (i = 0; i < 10; i++){
               sum = sum + i*i;
         }


System.out.println(Integer.toString(sum));
   }
}
```

Figure 2.2a – The four sets of code all do the same thing, but they are written in different languages.

2.3 Why RB/C++?

One of the primary engineering goals of Mapalester is to make it cross-platform. The reasoning behind this goal arises primarily out of the need of the K-12 schools target market. The goal is also rooted in two elements not as central to the mission of the research project. First, there is no viable, modern GIS available for the Mac besides a

non-user friendly port of the UNIX-based GRASS open source system that is notoriously difficult to use. Second, while I am by no means inexperienced at using and developing software on a PC, my design sensibilities are much more suited to Macintosh development.

The cross-platform development goal left me with three options for my development platform. First, I could attempt to produce a maximally portable core set of code and then develop the portions that are platform dependent (interface, drawing engine) for each platform. Second, I could use Java, a language with which I was very familiar. Third, I could use REALbasic, and its C++ plug-in architecture if necessary. I eliminated the first option quickly, as I was not familiar enough with the Mac or Windows APIs to not force most of my development time into learning those APIs. Also, as this was a funded research project, I needed to have something to show for my work as I progressed, not just a pile of portable code without an interface or an interface without much GIS functionality. This left me with the choice of either Java or REALbasic. In the end, I chose REALbasic over Java for the following reasons. First, Java is notoriously slow and while REALbasic is also sluggish compared to some other languages, the C++ plug-in architecture would allow me some leeway. Many functions in GIS require the processing and display of millions of pieces of data in microseconds, and thus speed is a very important factor. Second, with REALbasic, I could develop interfaces that were native to each platform, whereas Java interfaces always look somewhat foreign and awkward. Considering that ease-of-use is one of the three primary goals of Mapalester, a comfortable interface is a very important factor.

However, before considering interface familiarity as a factor, I went looking for any Java-based programs that would be used extensively by any non-profit, K-12 school, or personal users. After all, if these users are familiar with the quirks of the Java interface, then Java might provide an equally comfortable interface for Mapalester's target users. While it is hard to predict the activities of the personal user category, the only application that I could find that I assumed that my other target markets would use that was programmed in Java was Limewire, a music-sharing program. REAL Software realizes that it shares a certain market with Java and publishes a "REALbasic vs. Java" comparison on its website. On this site, it lists the problems with Java's interfaces in

more detail. REAL Software is correct to point out that "unlike Java, REALbasic provides fully native controls with native behavior on all supported platforms." It also states that "REALbasic compiles to native machine code for each platform eliminating the need for virtual machines, special installers and environmental variable settings" and that "REALbasic provides access to the unique features of each platform such as ActiveX and the Registry on Windows and Apple Events and Keychain on Mac OS X." For these two reasons, despite being more familiar with Java, I decided to choose the REALbasic route.

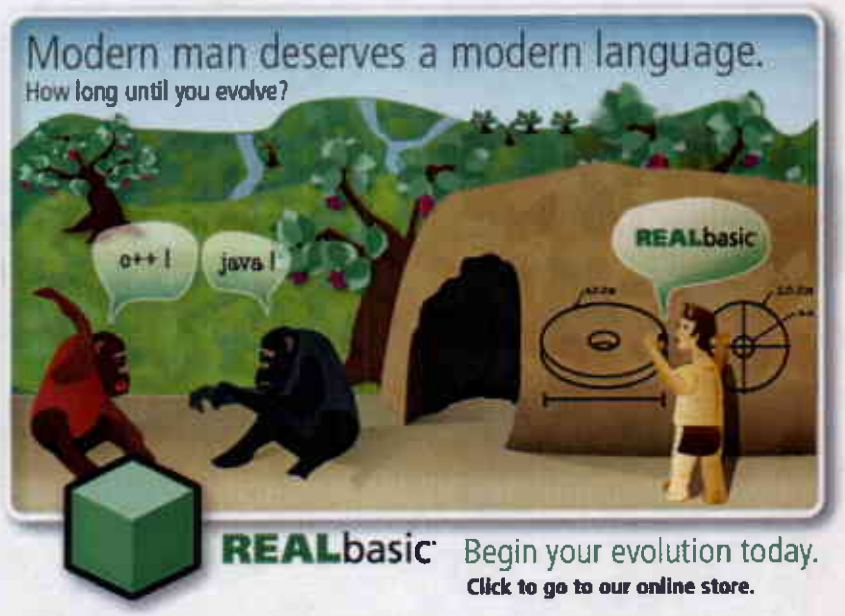


Figure 2.3a – REAL Software has a high opinion of its project compared with competing languages. It was the target markets of Mapalester that determined my choice of programming language more than any sort of judgment of the languages themselves. Image from http://www.realsoftware.com/realbasic/compare/

2.4 The challenges of my chosen architecture

I encountered three main problems due to my choice of engineering architecture, although I believe that the collection of obstacles would be greater in number and difficult if I had chosen a different architecture. In order of difficulty the problems were (1) I needed to refamiliarize myself with REALbasic and learn nearly every aspect of the language, (2) I needed to negotiate what algorithm and object-oriented content would be

in the C++ plug-in format and what would be in the pure REALbasic format, and (3) I needed to learn C++.

While I had engaged in minor programming projects using REALbasic prior to Mapalester, I had never worked extensively in REALbasic. In addition, it had been quite a while since I had used the REALbasic integrated development environment. As such, prior to starting on the Mapalester project, I spent several weeks working with the REALbasic environment and refamiliarizing myself with the language. I also spent much time in the REALbasic language reference guide, a text that became a great resource as the project continued. When I started programming Mapalester, I did not have the depth of knowledge in REALbasic to complete the REALbasic portion, but I had the background and familiarity with the necessary resources to learn most anything I needed.

The second problem, deciding what I would implement in REALbasic and what I would implement in C++, plagued me for a long time. Essentially, the REALbasic plug-in API allows me to create a Mapalester API with which to work in REALbasic. But what do I implement in the API and what do I leave in REALbasic? My initial guideline involved the factors of speed and low-level access. Due to the fact that REALbasic performs automatic garbage collection, its code tends to run slower than native C++ code in the REALbasic format. As such, my initial ventures into C++ coding were motivated by the running speed of several important functions, such as projections (section 3.4) and data drawing (section 3.2). Additionally, REALbasic struggles with recursive algorithms (algorithms that call themselves) that reach depths of around 50 or more function calls. Since many of the important algorithms are recursive, it was faster to run these algorithms in C++ code than in REALbasic. Theoretically, since all of these algorithms are tail-recursive, I could have made iterative versions of all of them. In fact, I did implement iterative versions of some recursive algorithms, most notably, the line generalization algorithms (section 4.3). However, it became quite tedious and difficult to make these conversions, especially since I could run the recursive versions in C++ and achieve acceptable performance.

I soon discovered that I could speed up some algorithms, especially those that load in and process data files, by accessing memory and disk data at a low level. However, REALbasic offers no byte-level or bit-level access to binary data, either on

memory or on disk. The lowest-level access REALbasic provides is at the word level. As such, the level of memory and disk access became an important factor for the programming language decision as well.

It turned out that these two factors – speed and low-level access – in combination with Mapalester's object oriented structure, led to a simple and very important rule for code separation that I now use as a successful "rule of thumb". As is discussed in greater detail in section 3.3, Mapalester considers spatial information on the shape level and on the data layer/dataset level. The shape level deals with individual shapes (points, polygons, etc) and their vertices. The data layer level represents an entire set of shapes such as, for instance, the set of polygons that represent counties in Minnesota. Conveniently, all of the algorithms that have needed C++ speed thus far only concern data related to the shape level. Likewise, the low-level access functions, while possible to implement at the data layer level, can also be run at the shape level. As such, I established a firm rule in order to avoid switching code back and forth between the "Mapalester API" and the REALbasic use of that API. All classes and associated algorithms that deal with the shape level must be coded in C++. All classes and associated algorithms that deal with the data layer as a whole must be programmed in REALbasic. The REALbasic C++ plug-in interface is used to navigate between the two layers. This navigation is quite complicated, and too intricate and tangential to describe here in detail.

It is interesting to note that even in topics that are seemingly relegated exclusively to the realm of software engineering, the basics of geography become a factor. Prior to establishing the aforementioned rule, my code was split between REALbasic and C++ through educated guessing and intuition. Once I grounded this software engineering in a GIScience context, the choice of programming language become much simpler and well defined.

The final and most difficult challenge related to the chosen development architecture involved learning C++. While this process is by no means worthy of a long description in this context, it is important to note that even just the C++ portion of Mapalester has been the largest programming project I have ever attempted in a language other than REALbasic, or VisualBasic. A large portion of development time was spent

researching and learning appropriate memory management techniques, gaining a functional understanding of pointers to pointers to pointers (etc), and comprehending templates.

3. The Object-Oriented Structure of Mapalester

3.1 An Overview of the Object-Oriented Structure

In its current state, Mapalester is comprised of over 10,000 lines of code in four languages (REALbasic, C++, SQL, XML) that are organized into an elaborate class structure. In order to help the reader make sense of this structure and better understand the overall architecture of Mapalester, I have divided the class structure into several subsections. The subsections operate mostly independently, but have key interactions with each other. The four subsections, which I will discuss in the subsequent parts of this chapter, are the interface system, the data interaction system, the projection system, and the database system. In each part, I will describe and analyze the architecture and basic functionality of the subsection of the class structure being discussed. I will also detail the subsection's primary interactions with other subsections.

3.2 The Interface System

Figure 3.2a. The object-oriented nature of the interface of Mapalester can be seen in the above screenshot. Notice the replication of windows and maps. Due to use of the object-oriented architecture, such replication required almost no additional coding.

The REALbasic IDE contains a large variety of interface elements organized in an efficient and logical class structure (Brandt 2004). In order to take advantage of the benefits of this hierarchy while still modifying the user interface elements to fit the needs of Mapalester, I primarily used a subclassing methodology for most of my interface classes. In total, Mapalester currently has more than twenty individual classes that are subclassed from REALbasic's interface classes. Most of these classes provide a very specific functionality to a standard interface element. For example, Mapalester has many bevel buttons that, when pressed, bring open the operating system's color chooser and display the color chosen as a button icon. To implement this functionality in a portable and reusable manner, I have simply subclassed REALbasic's bevel button and input the necessary code into the draw() and action() methods of the subclass.

There are, however, several very important classes in the interface class structure that have extensive algorithms contained within them: the GISWindow class, the GISMapElementList, and the GISFormatWindow. The GISWindow is the most dominant feature in the user's experience with Mapalester. Each GISWindow contains elements that interact with all three other class subsections, as well as elements that access other parts of the interface class substructure. From GISWindows, the number of instances of which is only limited by available memory, users can access the spatial data database (section 3.5), perform many GIS operations, and engage in layout, input and export operations (section 3.3). Moreover, each GISWindow contains an instance of the GISMapElementList (later in this section). The GISFormatWindow is the only other major site of program control (later in this section).

In addition to its key role in user interaction, the GISWindow also serves as the document unit of Mapalester and is roughly analogous to a "project" in ESRI's GIS products. When a user performs a load or save operation, it is the GISWindow that takes control. Using an XML-based format, the GISWindow recursively goes through all of its members that need to save information, requesting that each member save its information in a nested section of the resulting XML document. The reverse is done when the user wishes to load a saved XML file. These members include everything from

BackgroundLayerCanvas (see section 3.3) and, when completed, all of its GISLayoutElements (see section 3.3). Not all of the save functionality is complete, but the basic system is established and is functioning for the GISWindow class itself, along with a subset of its members. In figure 3.2b, I have included an example of the XML content of a saved GISWindow. The structure of the XML file reflects well the object-oriented structure of the interface system and other parts of the program. It is also important to note that when Mapalester is released, this structure will be released as an open file format. The extension of files in this format is ".mmap."


```
<?xml version="1.0" encoding="UTF-8"?><plist
version="1.0"><dict><key>gisWindow</key><dict><key>Left</key><integer>334</integer><key>T
op</key><integer>55</integer><key>Width</key><integer>853</integer><key>Height</key><integ
er>694</integer><key>Background Layer</key><dict><key>Fill
Color</key><dict><key>R</key><integer>255</integer><key>G</key><integer>255</integer><key
>B</key><integer>255</integer></dict><key>Border
Width</key><integer>0</integer><key>Border
Color</key><dict><key>R</key><integer>0</integer><key>G</key><integer>0</integer><key>B<
/key><integer>0</integer></dict><key>Use Color Fill</key><true/><key>Picture
Transparency</key><real>1</real><key>Picture Display
Method</key><integer>2</integer></dict><key>LIbrary
Browser</key><dict><key>Map</key><dict><key>1</key><integer>14</integer><key>2</key><int
eger>20</integer><key>3</key><integer>15</integer><key>4</key><integer>22</integer><key>5
</key><integer>26</integer><key>6</key><integer>18</integer><key>7</key><integer>11</intege
r><key>8</key><integer>28</integer><key>9</key><integer>29</integer><key>10</key><integer
>30</integer><key>11</key><integer>31</integer><key>12</key><integer>24</integer><key>13<
/key><integer>2</integer><key>14</key><integer>3</integer><key>15</key><integer>4</integer>
<key>16</key><integer>5</integer><key>17</key><integer>6</integer><key>18</key><integer>7
</integer><key>19</key><integer>8</integer><key>20</key><integer>9</integer><key>21</key>
<integer>10</integer><key>22</key><integer>12</integer><key>23</key><integer>13</integer><
key>24</key><integer>16</integer><key>25</key><integer>17</integer><key>26</key><integer>
19</integer><key>27</key><integer>21</integer><key>28</key><integer>25</integer><key>29</k
ey><integer>27</integer></dict><key>Widths</key><dict><key>1</key><integer>155</integer><k
ey>2</key><integer>76</integer><key>3</key><integer>84</integer><key>4</key><integer>94</i
nteger><key>5</key><integer>54</integer><key>6</key><integer>82</integer><key>7</key><inte
ger>52</integer><key>8</key><integer>114</integer><key>9</key><integer>109</integer><key>
10</key><integer>104</integer><key>11</key><integer>106</integer><key>12</key><integer>17
5</integer><key>13</key><integer>80</integer><key>14</key><integer>101</integer><key>15</k
ey><integer>129</integer><key>16</key><integer>138</integer><key>17</key><integer>144</int
eger><key>18</key><integer>130</integer><key>19</key><integer>83</integer><key>20</key><i
nteger>78</integer><key>21</key><integer>65</integer><key>22</key><integer>50</integer><ke
y>23</key><integer>50</integer><key>24</key><integer>50</integer><key>25</key><integer>50
</integer><key>26</key><integer>50</integer><key>27</key><integer>100</integer><key>28</ke
y><integer>124</integer><key>29</key><integer>47</integer></dict></dict></dict></dict></plist
>
```


Figure 3.2b. This XML code represents a Mapalester saved document.

The primary function of the GISMapElementList, which is only instantiated inside of GISWindows, is to provide access to a list of the VectorThemes of the currently selected GISCanvas. From this list, via the GISFormatWindow, users can make critical changes to the content and display of each VectorTheme. Additionally, it is through the GISMapElementList that users are able to manipulate the ordering of the data layers, a key aspect of GIS functionality. To enable this ordering functionality, the GISMapElementList contains a heavily modified implementation of a bubble sort algorithm that interacts with both the host GISMapElementList and the affected GISCanvas. Users are also able to enable and disable VectorThemes through the GISMapElement list, functionality which comprises yet another cornerstone of GIS.

The GISFormatWindow is a Mapalester-only feature. As a GIS lab assistant, it is my perception that one of the greatest failings of currently available GIS software is the hiding of a vast array of important features in non-context sensitive windows that can only be accessed from deep within menus or through obscure buttons on the toolbar. This failing, which is partially a result of the extension/module-based development of many GIS packages, has two major ramifications. First and foremost, inexperienced users are often unaware of important and powerful operations and settings that can be applied in a given context. For the education market, this ramification is particularly important. Wilder et. al. (2004), for instance, note that teachers learning GIS during in-service sessions can tend to view GIS as simply a "digital map" device from which they can print reference maps. Second, problems are often harder to solve as the important information is not immediately available to the user. While some GIS packages have made strides forward in solving these problems through the use of the contextual menu, I believe that the GISFormatWindow does a better job at providing quick and context-sensitive access to all of the important operations, settings, and information. It does so by making its large number of panels visible or invisible based on the currently selected element of the user interface. For instance, if the user has a GISCanvas selected, the GISFormatWindow would only display the panels that function with a GISCanvas. If the user selected items in a GISMapElement list, the GISFormatWindow displays all of the panels with controls that relate to the modification or display of VectorThemes. If the user has selected seemingly disparate user interface elements, the GISFormatWindow

only displays panels relevant to all selected elements. In other words, GISFormatWindow displays the panels in the set resulting from the intersection of the panels relevant to the selected elements. Both Microsoft Word 2004 for the Mac and OmniGraffle by OmniSoft have similar, but not identical, features.

The GISFormatWindow's functionality is implemented through the extensive use of the class interface construct. Each and every panel that can be displayed in the GISFormatWindow has a corresponding class interface. For instance, the ProjectionPanel is linked to the ProjectionPanelInterface, which is implemented by the GISCanvas. Similarly, the BackgroundPanel corresponds with the BackgroundPanelInterface, which is implemented by both the GISCanvas and the BackgroundLayerCanvas. Each interface requires that classes that implement it have the full set of the methods relating to the options in the panel. For instance, classes that implement the BackgroundPanelInterface must be able to change the content of the background display of the class. The beauty of the class interface methodology is that the definition of what is the "background display" is entirely up to the context of the class. The class must simply have a method called setBackgroundDisplay() (and several others) that can be called from the panel.

The five panels that are finished are the ProjectionPanelInterface, which is the interface to the coordinate system functionality, the PicturePanelInterface, which is exclusively dedicated to features in the PictureElement class (section 3.3); the NamePanelInterface; which is central to nearly all aspects of the interface that use the GISFormatWindow; and the BackgroundPanelInterface, which is described above. Currently under development is the LayerFormatPanelInterface from which users will be able to make a large portion of basic changes to layer symbology. Many more interfaces are planned and will be implemented as I implement the features that the interfaces will contain.

How does the GISFormatWindow know what user interface elements are selected, and thus what panels to display? The forefront GISWindow is in charge of maintaining a stack of all selected elements of the user interface. This means that the GISWindow also decides what is meant by "selected." For instance, if nothing is selected, the GISWindow places the instance of the BackgroundLayerClass on the stack,

because this is the most intuitive choice based on the interface. Also, if the user selects a member of the list in the GISMapElementList, the GISWindow pops all other elements of the stack and pushes the corresponding VectorTheme on the stack because this is, given the interface, intuitively what should occur. Each time the stack changes, the GISWindow sends a message to the GISFormatWindow, which is in effect a global variable as only one instance of it may occur. The GISFormatWindow then examines the stack. When the GISFormatWindow finds an instance of a class that does not implement an interface corresponding to a panel, it makes that panel invisible. The panels that remain are the panels that are implemented by all of the selected elements and are thus, context-sensitive.

Other key classes related to user interaction with Mapalester are the MenuBar class and the contextual menu classes. At the moment, all of these classes offer a rather mandate set of features and a correspondingly standard set of algorithms. However, when Mapalester is released, they will be semi-criticial to user interaction with Mapalester. Both the menu bar and contextual menus will provide access to important features inaccessible through other means. For example, the insertion of a GISCanvas class into the BackgroundLayerCanvas (section 3.3) – otherwise known as adding an additional map frame to the document – will be a function exclusively available through the menu bar and contextual menus. Also, because both the menu bar and contextual menus have context-sensitive content, the classes that provide their functionality will probably share code with the GISFormatWindow in a manner that is yet to be established.

3.3 The Data Interaction System

To the user, all data interaction occurs within the context of the GISCanvas, the class that provides the connection between the interface class structure and the data interaction class structure. I will begin by discussing this class and its relatives, and will then delve deeper into the class structure behind the functionality of what actually is displayed in the GISCanvas.

The GISCanvas is a subclass of the GISLayoutElement class. In each GISWindow (section 3.2), the BackgroundLayerCanvas (section 3.2) maintains an array

of GISLayoutElements placed inside the BackgroundLayerCanvas. Currently, there are only two classes of the type GISLayoutElement: the GISCanvas and the PictureElement class. The PictureElement class represents a picture placed in the BackgroundLayerCanvas by the user. As I implement more functionality in Mapalester, the number of classes of the type GISLayoutElement will drastically increase. For instance, soon under development will be a LegendElement and MapScaleElement, which will be responsible for handling functionality related to the display and manipulation of map legends and map scales, respectively. Figure 3.3a depicts the relationship between all of the aforementioned classes in a semantic network.

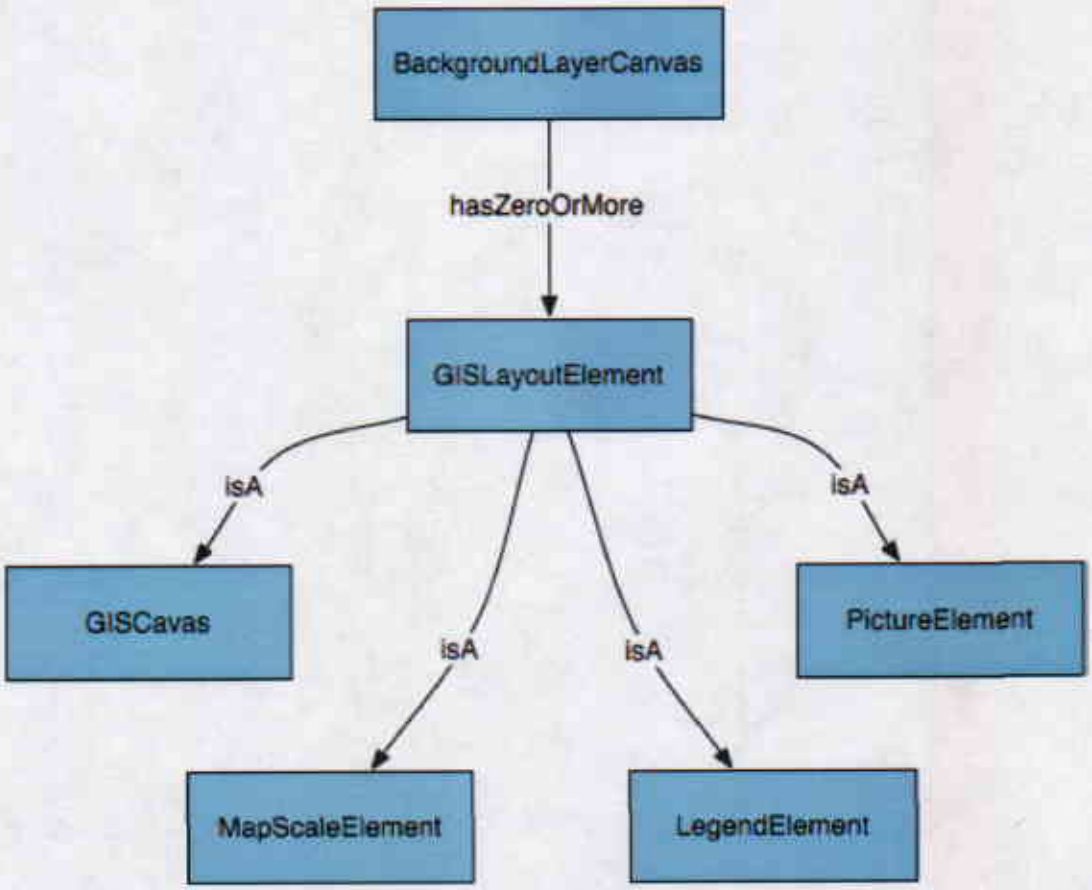


Figure 3.3a – The interface between the data interaction system and the interface class structure. MapScaleElement and LegendElement have yet to be implemented are are depicted as samples of future classes of type GISLayoutElement.

A BackgroundLayerCanvas with all of the aforementioned GISLayoutElements (and any future ones) will be able to be printed and exported. As a side note, this means that I have chosen to eschew the layout view/data view construct common to ESRI products.

Now that the context in which the GISCanvas class operates has been established, it is necessary to explain how the GISCanvas displays spatial data. The largest challenge in designing the GISCanvas was making its functionality universal to all types of GIS data that will eventually be supported. The answer to this challenge was making the GISCanvas interact with the data it contains only on the layer level. Basically, each GISCanvas has an array of VectorThemes, each of which are responsible for returning to the GISCanvas an image (essentially, just a picture – like any JPEG) of what it looks like given the extent of the GISCanvas. When I implement the RasterTheme class, this class will also be responsible for simply returning an image of what it looks like in the given extent. When the GISCanvas's extent changes, say, because the user has zoomed in/out or panned, the GISCanvas requests new pictures from each of the VectorThemes (or, later, RasterThemes as well) contained within its VectorTheme stack. Essentially, every time the GISCanvas redraws, all it does is pancake all of the returned images from the VectorThemes on top of each in the correct order. This order corresponds to that which the user manipulates using the GISMapElementList class (section 3.2). Layers that have been "turned off" in the GISMapElementList are skipped in this "pancaking" process.

How do the VectorThemes draw an accurate image of themselves in any given extent? Each instance of the VectorTheme class is responsible for asking its C++ parent class to query all of its VectorObjects (PointObjects, PolylineObjects, or PolygonObjects) to create the appropriate display. This query will eventually utilize an R-tree index (see section 4.2) which is attached to every VectorTheme, but for now uses the brute force approach outlined in section 4.2. Note that this interaction between the VectorTheme and the corresponding VectorObjects is the single most important interface between the C++ plug-in code and the REALbasic code. In fact, as touched upon above, the VectorTheme is actually a subclass of a VectorThemeBase class implemented in the C++ plug-in. This subclassing is the construct by which the two code bases interact. When the REALbasic VectorTheme needs to update its cached picture, it simply calls the update() function,

which has been programmed in the C++ superclass and only deals with C++ elements of Mapalester.

Once this update() function, which is provided a two-dimensional extent as a parameter, has identified which VectorObjects it needs to draw using the spatial indexing methods as noted above, the actual drawing process is quite simple. Since all coordinate systems are treated as two-dimensional Cartesian planes in Mapalester (see section 3.4), the update() function first makes a new image the same size in *pixels* as the GISCanvas() in which the VectorObject is located. The function next determines the scale of the image in the form of a unit/pixel ratio. This unit can be any angular or linear unit because by the time the drawing process has begun and the update() function has been called, the projection engine has made all necessary arrangements to convert all of the VectorObjects and their constituent points in all the VectorThemes to the correct two-dimensional coordinate system for the GISCanvas. Next, using the scale as a guide, the update() function draws all the VectorObjects contained within the extent onto the image. Finally, the image is returned to GISCanvas. This process is shown graphically in figure 3.3c. Figure 3.3b shows the whole data interface system class structure "behind" the GISCanvas in a semantic network.

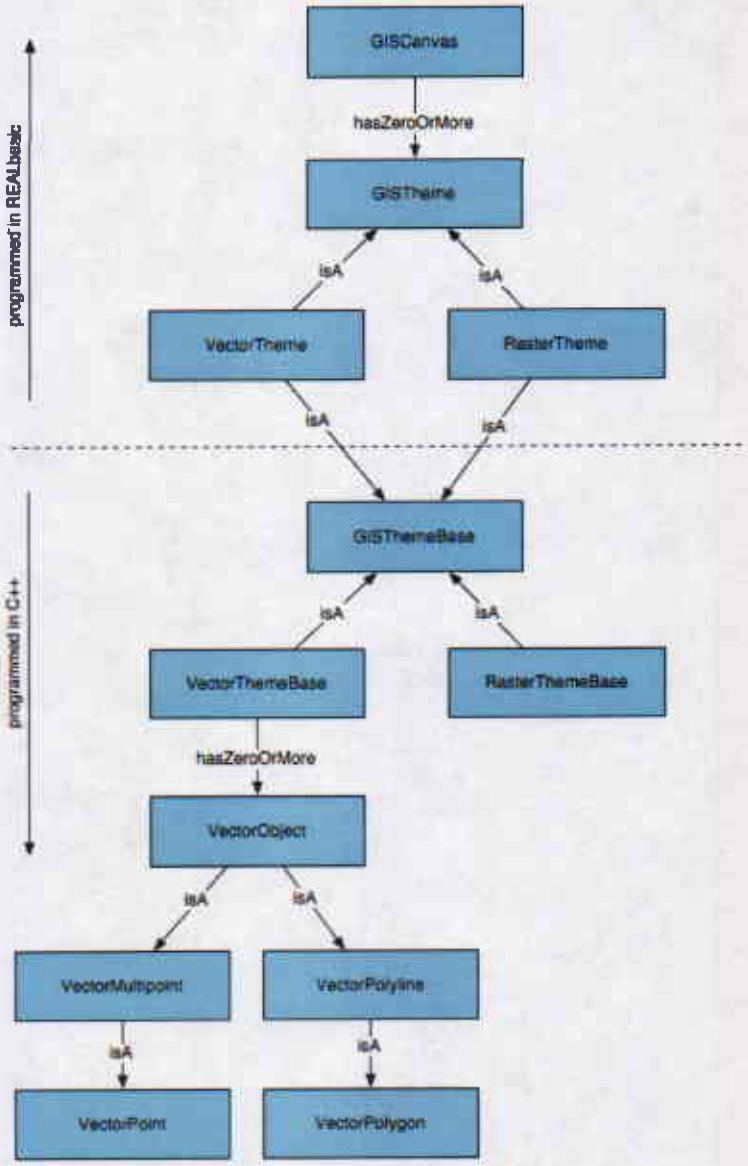


Figure 3.3b – A semantic network depicting the data interaction system that operates "behind" a GISCanvas. Note the division of programming between C++ and REALbasic. Also note that VectorPoints and VectorPolygons are both subclasses of other VectorObjects. I have implemented these two classes in this manner in order to maximally share code. In effect, a VectorPoint is a VectorMultiPoint with just one point in it. Similarly, a VectorPolygon is a special case of a VectorPolyline in which the first and the last point are the same. Implementing the four basic geometric types in a hierarchical manner has saved me from writing thousands of lines of code throughout the program.

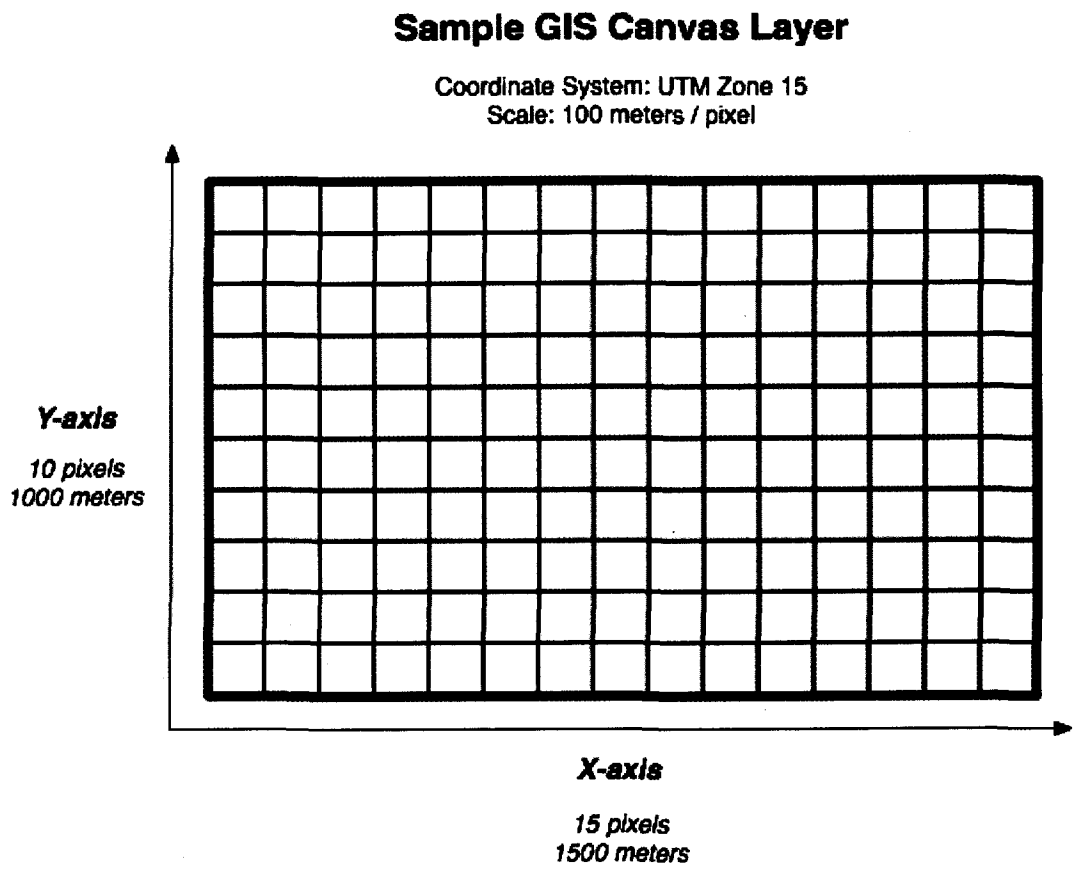


Figure 3.3c – The basics of the actual drawing part of the update() function.

3.4 The Projection System

Every spatial data set has some sort of coordinate system. This coordinate system can be a projected coordinate system or an unprojected coordinate system. Spatial data in a projected coordinate system have coordinates that are in some sort of x,y coordinate system where x and y represent values in a Cartesian plane. Projected data have been transformed using one of many projection equations, such as that for the Mercator projection or the Lambert Conformal Conic projection. A common unit for projected coordinate systems is the meter or the kilometer; all projected coordinate systems utilize units of length. Spatial data in unprojected coordinate systems have coordinates that represent degrees on some ellipsoid (or spheroid). A common unit for unprojected

coordinate systems is decimal degrees of longitude and latitude; all unprojected coordinate systems utilize angular units. Unprojected coordinate systems are sometimes incorrectly referred to as a data's "datum." Every projected coordinate system has an underlying unprojected coordinate system or "datum", as the x and y values must be generated from original degree values.

In order to be a viable GIS, Mapalester must have a robust projection system that can convert between all of the coordinate systems in standard use. This is an immense task. There are over thirty common projections, each with its own highly complex equation. Many of the equations accept five or more variables in addition to the standard latitude and longitude coordinate parameters. Additionally, there are an equally large number of unprojected coordinate systems based on a variety of ellipsoid approximations of the Earth. In order to display data accurately in the same view frame, Mapalester must be able to convert data in any coordinate system to each and every of the other coordinate systems. In order to implement such a projection system, code must be organized efficiently in a simple but robust class structure.

The structure I eventually settled on for Mapalester is based on my research into coordinate systems and my investigation into how coordinate information is stored, such as in the ".prj" file discussed in section 4.1.4. Because all coordinate systems have some sort of projection – even if that is "unprojected" – the primary class of the projection system is the ProjCS class. The ProjCS class is responsible for changing the projection of the coordinate system whereas the GeoCS class takes care of converting the underlying unprojected coordinate system, a process that mostly involves the science of "datum conversion" but has other important aspects as well. No datum conversion functionality has been implemented in Mapalester. As such, an error of about 800ft at large scales is possible while currently viewing data in Mapalester. There are four more classes that make up the projection system: the Spheroid class, the PrimeM(eridian) class, the Datum class, and the Unit class. Each is responsible for converting the elements of the coordinate system that correspond to its name. The basic relationships among the classes is depicted in figure 3.4a in the form of semantic network.

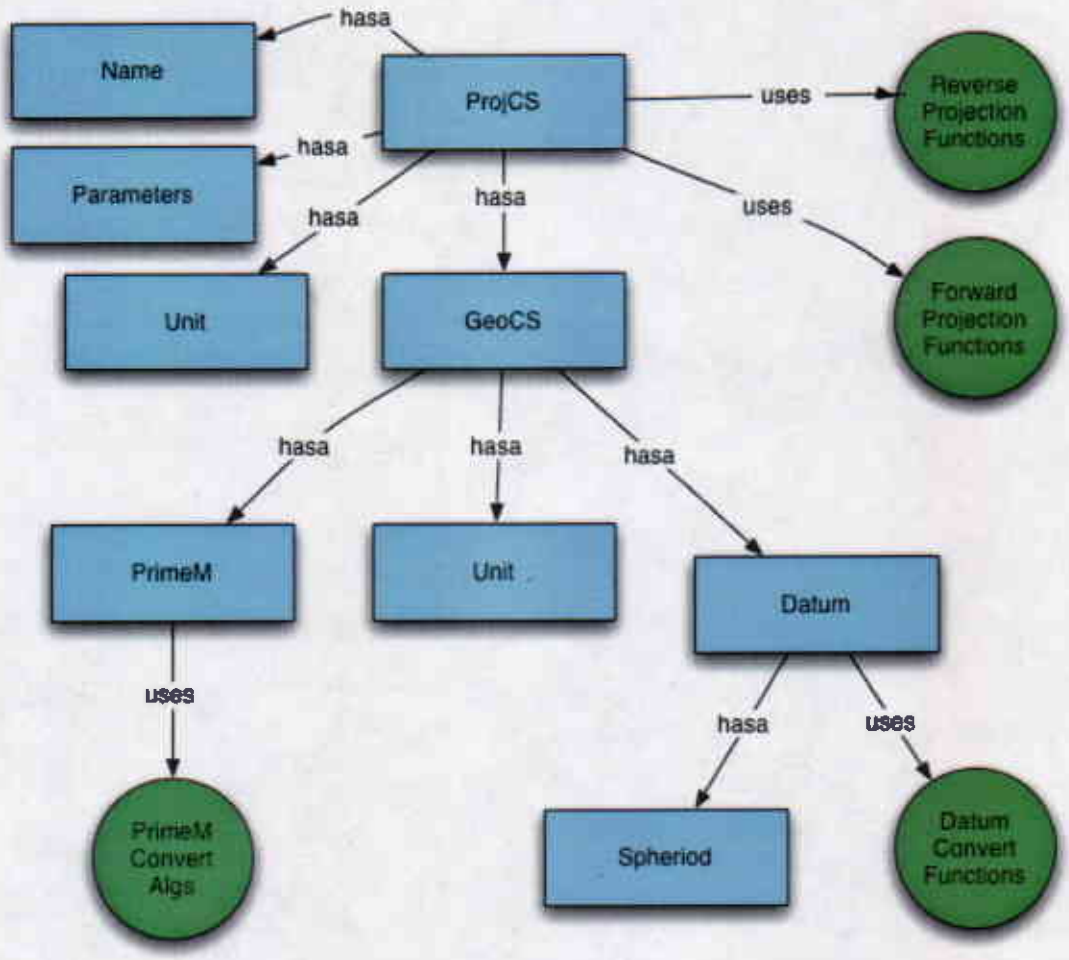


Figure 3.4a – A simplified semantic network of the relationships among the classes of the projection system. Note that the projection system makes extensive use of C++-coded functions, but no C++ coded classes. This is a major departure from the original version of the projection system, which had all of the code in REALbasic. The REALbasic code, however, proved too slow for the more complicated functions and algorithms, and thus, a switch to C++ was necessary. This change was also made to further enforce the rule that REALbasic never have access to the VectorObjects of a VectorTheme. All of the algorithms shown require vertex-level access to the spatial information of a VectorObject, and thus must be in C++ code according to the rule. The C++ coding is not complete.

Every VectorTheme has an instance of a ProjCS class. In addition, the GISCanvas class also has a ProjCS member. In order to accurately display the information (or display the information at all in many cases), the GISCanvas's ProjCS and the VectorTheme's ProjCS must match completely. Meeting this condition is the primary function of the projection system and is an excellent demonstration of the effectiveness of the system's object-oriented structure.

The process of meeting the condition is as follows: when a VectorTheme is loaded into a GISCanvas, the GISCanvas passes the VectorTheme's ProjCS instance to the convert() function of its own ProjCS instance. The convert() function is the command center for the projection conversion. If the projection, parameters of the projection, or the GeoCS of the inputted ProjCS differs from that native to the GISCanvas, the convert() function tells the inputted ProjCS to unproject itself. Before doing so, if the inputted ProjCS has a unit other than meters, the inputted data's unit must be changed, as all the constants in the forward and reverse projection algorithms are in meters. The inputted ProjCS then calls the appropriate reverse projection equation function (section 4.6). If the GeoCS differs, the convert() function then passes control to the convert() function of the ProjCS's GeoCS. In the GeoCS's convert() function, the datum (and its spheroid) and the primeM of the inputted ProjCS's GeoCS are converted using the datum conversion functions and the prime meridian conversion algorithms (section 4.5). The angular unit of the GeoCS may also be converted in the GeoCS convert() function, if necessary. The GISCanvas' GeoCS's convert() function then returns control to the GISCanvas' ProjCS() function. If the inputted spatial data was unprojected using the reverse equations, then the data must be projected using the forward projection algorithms of the GISCanvas's ProjCS. Finally, it may be necessary to convert the unit of the forward projected data to match that of the GISCanvas' ProjCS. The two coordinate systems will then match.

3.5 The Database System

There are two important determinants of the object-oriented structure of the database system in Mapalester. The first is the dual uses of the system at a fundamental level. The database structure is not only used to manage attribute data inherent to every GIS file, but also to manage those files themselves. Section 4.4 is dedicated to the description of this second functionality. Not only did this dualism in intended function reinforce the need for a clean and efficient object-oriented structure, but it also shaped the design of the structure. The second major determinant of the structure of the database objects is my choice of the Valentina database by Paradigma Software for the core

database infrastructure[2]. Implementation of the iTunes-like GIS data file management system, as well as provision of support for attribute databases, has been greatly facilitated by my choice of the Valentina database infrastructure as the framework for Mapalester's object-oriented database structure. As an object-oriented relational database framework, Valentina provides all of the low-level database functionality that would be too tangential to GIS to program for this project. Valentina is available for a large number of programming languages and it provides amazing support for REALbasic. In the subsequent paragraphs, I will discuss the database class structure of Mapalester in the context of these two determinants.

The class structure provided by Valentina for REALbasic forms the basis of Mapalester's database class structure. There are several types of classes in the Valentina framework. The VDataBase class represents a complete database. In its current state of development, Mapalester only interacts with a single complete database, the library database file located in the same folder as the application. The library database file holds three different types of tables – the library browser table, the projection browser table, and all the attribute data tables associated with the files available through the library browser. The library browser table handles the iTunes-like library functionality discussed in section 4.4. The projection browser stores all the coordinate system information ever loaded into the program. In general, there will be as many of the third type of tables as there are records in the library browser. In the case of shapefiles in the library browser table, for instance, the associated table is copied directly from the DBF portion of the shapefile. Each of these tables, no matter what type is represented by either the VBaseObject class or a subclass when loaded into the program,. In order to give each of these classes functionality specific to Mapalester, it was necessary to subclass nearly all of the classes provided by Valentina. The VDataBase subclass used in Mapalester, for example, is the appropriately named LibraryVDatabase. The subclasses of the VBaseObject that are used handle the two special tables in the LibraryVDatabase: are the library browser table (BrowserVBaseObject) and the projection browser table

[2] Note: in late March, 2005, Paradigma Software released Valentina 2.0, a very major upgrade to the database framework that promises "incredible speed multipliers" (Paradigma Software 2005) over former versions of Valentina. The current version of Mapalester does not implement this database, but future versions will.

(ProjectionVBaseObject). The tables linked to the records in the BrowserVBaseObject are accessed using the core VBaseObject.

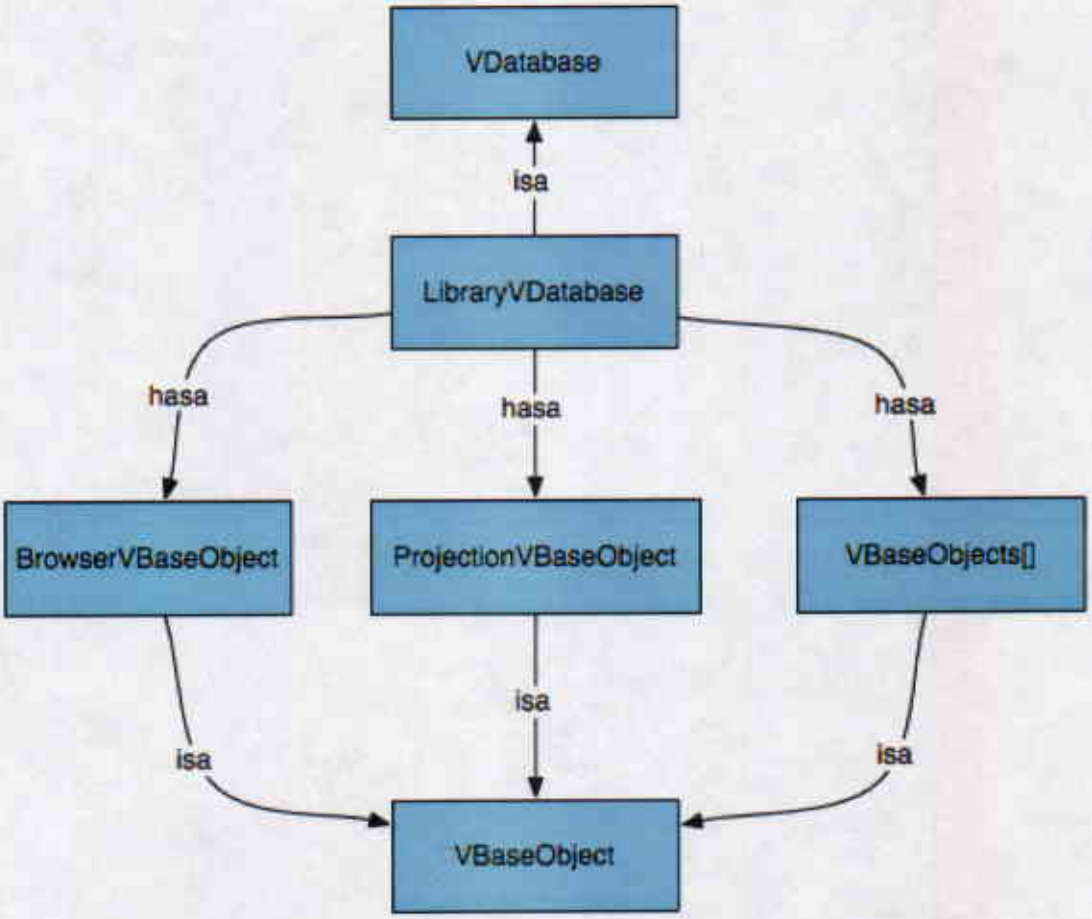


Figure 3.5a – A semantic network of the database class structure above the VField level.

Each field in each of the objects of the VBaseObject type is derived from one of Valentina's many field classes. Each of these field classes handles a specific type of field; there are classes for Boolean field types (VBoolean), date field types (VDate), all sorts of number field types (double, integer, long, etc.), and string field types (VString). More interestingly, Valentina also provides classes for the blob (VBlob), object pointer (VObjectPtr) and text data types (VText). The VBlob field allows Mapalester to store large chunks of data in the database. This is particularly important because Mapalester stores all of the spatial information for each record in the BrowserVBaseObject in a VBlob field. Essentially, Mapalester stores all of the data in the ".shp" file of a shapefile

in this field. The VObjectPtr enables the object-relational functionality in Valentina and is used in Mapalester to link each record in the BrowserVBaseObject to the record in the ProjectionVBaseObject table that stores the information relating to the coordinate system in which the record in the BrowserVBaseObject is stored. Finally, the VText field is a subclass of the VBlob field intended specifically to store text. A complete schema of the two specific tables are found in figure 3.5b and c. The reasoning behind the inclusion of each of the fields in the BrowserVBaseObject is found in the description of the library functionality in section 4.6. Details on the fields in the ProjectionVBaseObject can be found in section 3.4.

Name	Type	Size (bytes)
TableNumber	VLong	32
AbstractField	VText	8192
ContactInfoField	VText	1024
GatherEndDateField	VDate	64
GatherBeginDateField	VDate	64
RelevancyBeginField	VDate	64
RelevancyEndField	VDate	64
RestrictedField	VText	128
LegalField	VText	128
GeneralField	VText	8192
EditionField	VText	8192
URLField	VText	1024
KeywordsField	VText	1024
NameField	VString	1024
PublisherField	VText	8192
DataSourcesField	VText	1024
DataGroupField	VText	8192
GenreField	VText	8192
IDCodeField	VText	1024
ContentTypeField	VShort	16
VectorRasterField	VBoolean	1
FileFormatField	VShort	16
SpatialDataField	VBlob	variable
DataAddedField	VDateTime	112
FileNameField	VString	256
FileSizeField	VLong	32
RecordNumberField	VLong	32
minXField	VDouble	64
maxXField	VDouble	64
minYField	VDouble	64

maxYField	VDouble	64
projectionField	VObjectPtr	32

Figure 3.5b – The schema for the BrowserVBaseObject.

Name	Type	Size (bytes)
PEString	VText	1024
ProjectedCoordinateSystem	VString	100
Projection	VString	100
GeographicCoordinateSystem	VString	100
Datum	VString	100
AngularUnit	VString	100
LengthUnit	VString	100

Figure 3.5c – The schema for the ProjectionVBaseObject.

No schema for the third type of table is specified because schemas will vary greatly depending on the attribute database of the GIS file being represented. However, the schemas of this type of table are currently restricted to fields incorporated in the DBF file format because the only type of attribute database currently supported is that of the shapefile, which is in the DBF file format. The DBF file format only supports fields of the types "character" (interpreted as the VString type), "numeric" (interpreted as the VLong type), "floating point binary numeric" (interpreted as the VDouble type), and "logical" (interpreted as the VBoolean type). As an example, the schema for an attribute database of a shapefile containing counties in the United States is given in figure 3.5c.

Name	Type	Size (bytes)
Name	VString	32
State_Name	VString	25
State_FIPS	VString	2
Cnty_FIPS	VString	3
FIPS	VString	5
Area	VDouble	64
Pop1990	VDouble	64
Pop1999	VDouble	64

Figure 3.5d – An example schema for the attribute database of a shapefile containing counties in the United States.

Because the database system is quite intricate and difficult to explain out of its programmatic context, figure 3.5f explains what happens when a user adds a GIS file to the library (see section 4.6) in a highly abstracted manner.

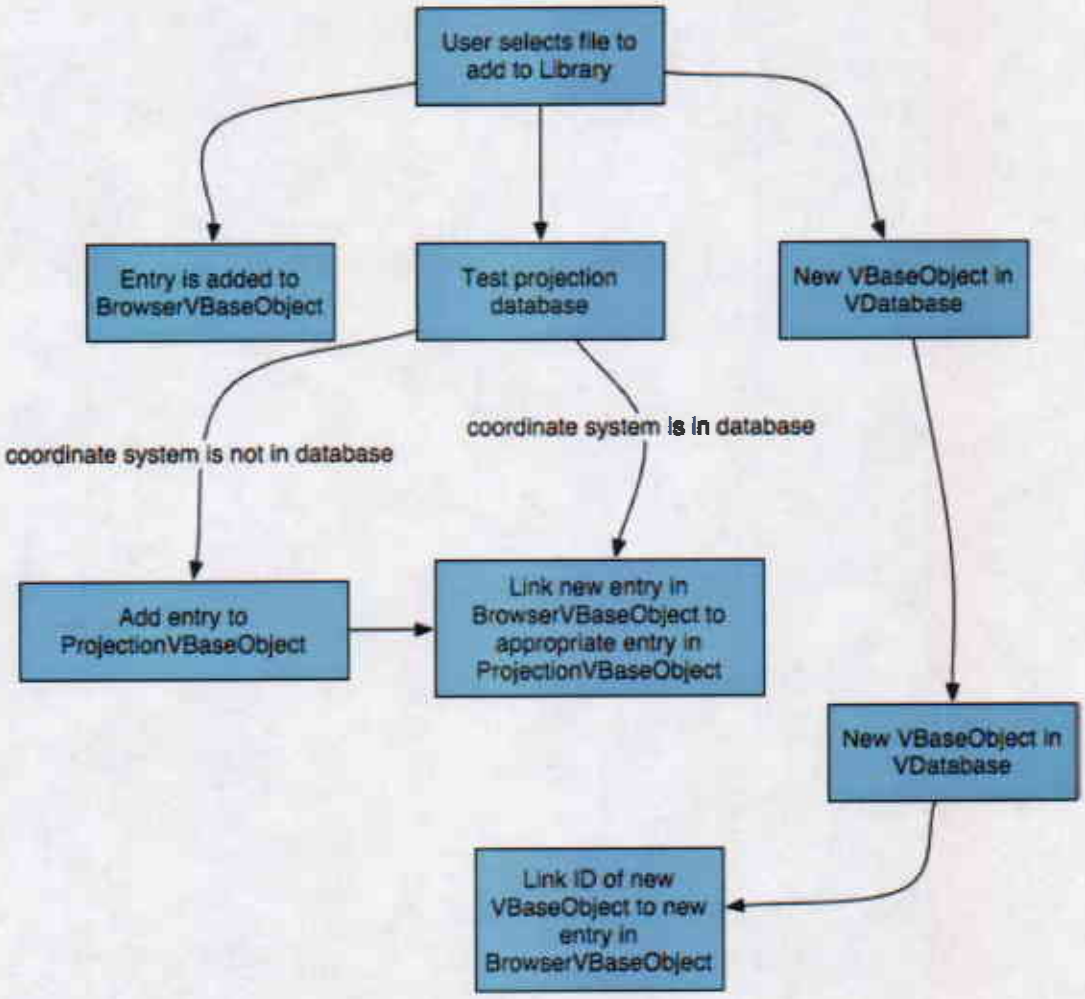


Figure 3.5f – The database class structure in action!

How does the user interact with the database class structure? In addition to the set of classes for storing database information, there is a less elaborate set of classes for viewing and modifying that information. This set of classes provides the intersection

between the database class structure and the interface class structure, and the remaining portion of the section is dedicated to its description.

Like the foundation of the database class structure, the database interface class structure is based on a third-party product. In this case, the product is DataGrid by Einhuger Software, which is distributed in the form of a REALbasic plug-in. DataGrid is a small set of REALbasic classes designed to provide programmers with an easy way to implement the visualization and modification of Valentina databases[3].

Due to the third-party nature of the database interface system's framework, the database interface class structure is also heavily reliant upon subclassing. Primarily, the three important classes in this structure are all subclasses (either directly or indirectly through another subclass) of the DataGrid class. Figure 3.5g shows the class structure of the database interface system in more detail.

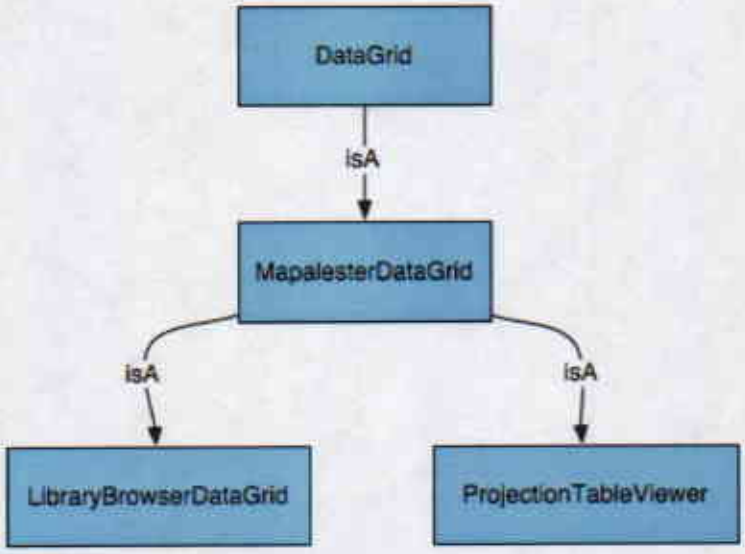


Figure 3.5g – The class structure of Mapalester's database interface system.

[3] My reasoning for adopting a third-party solution is identical to my reasoning for adopting the Valentina database: it would not have been a good use of my GIS honors project time to spend several weeks to recreate a Valentina display outlet, probably with worse results that\n I could get with DataGrid.

The MapalesterDataGrid class provides all of the functionality shared between the LibraryBrowserDataGrid and the ProjectionTableViewer, which at the moment, is about 90 percent of all the functionality in these two classes. All of these classes interact with the LibraryVDatabase class through the use of SQL (structured query language) queries. These queries return a class in Valentina for RealBasic called VCursor, which contains all of the database information relevant to the given query. In Mapalester's current state, most of the queries that are conducted are simple "return all" from a specific table queries. Since the class structure corresponds directly with the table stucture of the LibraryVDatabase, whenever each class needs to display data from the table to which it corresponds, it simply performs a "return all" query on that table. Figure 3.5h explains this process graphically.

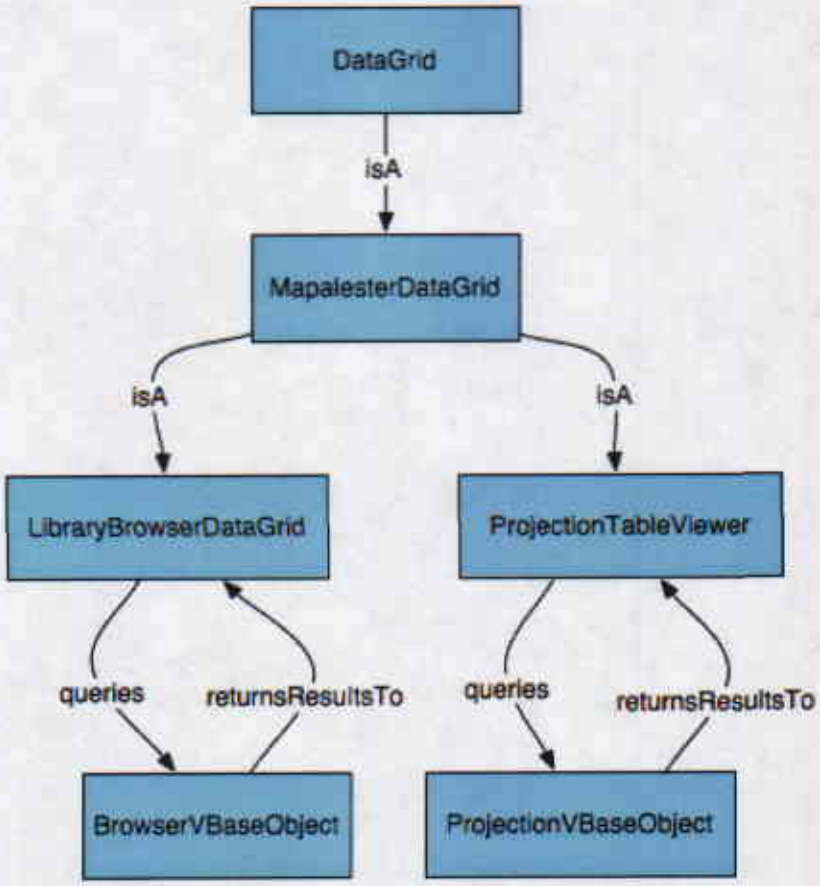


Figure 3.5h – A graphical view of the use of SQL as an interface between the database and the database interface class structures.

As I implement more functionality into Mapalester, these queries will become more complicated. For instance, the library will soon offer a context-sensitive spatial search feature. In other words, the library will be able to operate in a mode in which it only shows data relevant to the special extent being viewed in the selected GISCanvas. For example, if someone is zoomed-in to the Twin Cities metro area, the library will hide files that only include information about Europe or Asia, etc. This will obviously require greater complexity of search queries, possibly requiring me to implement a subset of spatial SQL via a C++ plug-in to Mapalester. Similarly, as I implement simpler, text-based searches for the LibraryDataGrid and the ProjectionDataGrid, much more extensive SQL queries (hidden to the user through an easy-to-use interface) will be necessary.

4. The Design and Implementation of Mapalester's Feature Set

4.1. Support of a variety of file formats

A primary characteristic of available GIS data is its extremely heterogeneous nature. GIS data come in a variety of file formats, and even data in the same file format can vary significantly in representation. GIS file formats can be loosely divided into two groups, vector and raster. This section will focus on vector data, as it is the most difficult to implement and has been the focus of the Mapalester's file format support thus far. Additionally, it should be noted that the term "file format" is used loosely in this context, as the specification for the currently dominant GIS vector data file format, Environmental Systems Research Institute's (ESRI) shapefile, describes three separate files that make up each shapefile. Moreover, since the shapefile specification was published in 1998 (ESRI. 1998), many new GIS features have necessitated the addition of more files to the de facto specification, resulting in situations in which each shapefile can includes upward of nine individual files, each supporting its own specific set of features.

Any GIS software package that aims to be "easy to use" must make the user completely unaware of the complexity of GIS file formats. Ideally, a Mapalester user should not need to know what type of file format she or he is using, or even if the file is a vector or raster file. I have made some progress toward this goal. As of this writing, Mapalester supports the shapefile, which, as mentioned above, is the most common file format for vector GIS data. Mapalester also supports the ".prj" expansion of the shapefile specification. A ".prj" file contains the metadata for the coordinate system of the corresponding shapefile, making ".prj" files vital to the accurate display of spatial information. I have also made progress toward supporting several other file formats, including the formerly ubiquitous Arc/Info exchange format, or ".e00" file, and the up-and-coming XML-based file formats. However, I have only done initial research into raster file format support.

In the first part of this section, I will explain issues common to supporting any spatial data file format. In the second part, I will do the same for vector data formats. I

will discuss the technical issues and techniques involved with supporting shapefiles in the third part. The fourth part will cover similar issues for the ".prj" extension.

4.1.1 General Issues Related to File Format Support

In order to support a spatial data file format, Mapalester must be able to convert data in the file format to Mapalester's internal data format. A consistent and permanent internal data format enables me to easily add new data formats as the project progresses and, later on, as they become newly available. In other words, the internal structure enables an architecture with a type of abstraction similar to that of plug-in architectures of programs like REALbasic and Photoshop. Because there are few similarities between representations of vector and raster data, I have chosen to give Mapalester a two-part internal representation, with one part dealing with vector information and the other handling raster data. A diagram of the file format support architecture can be found in figure 4.1.1a.

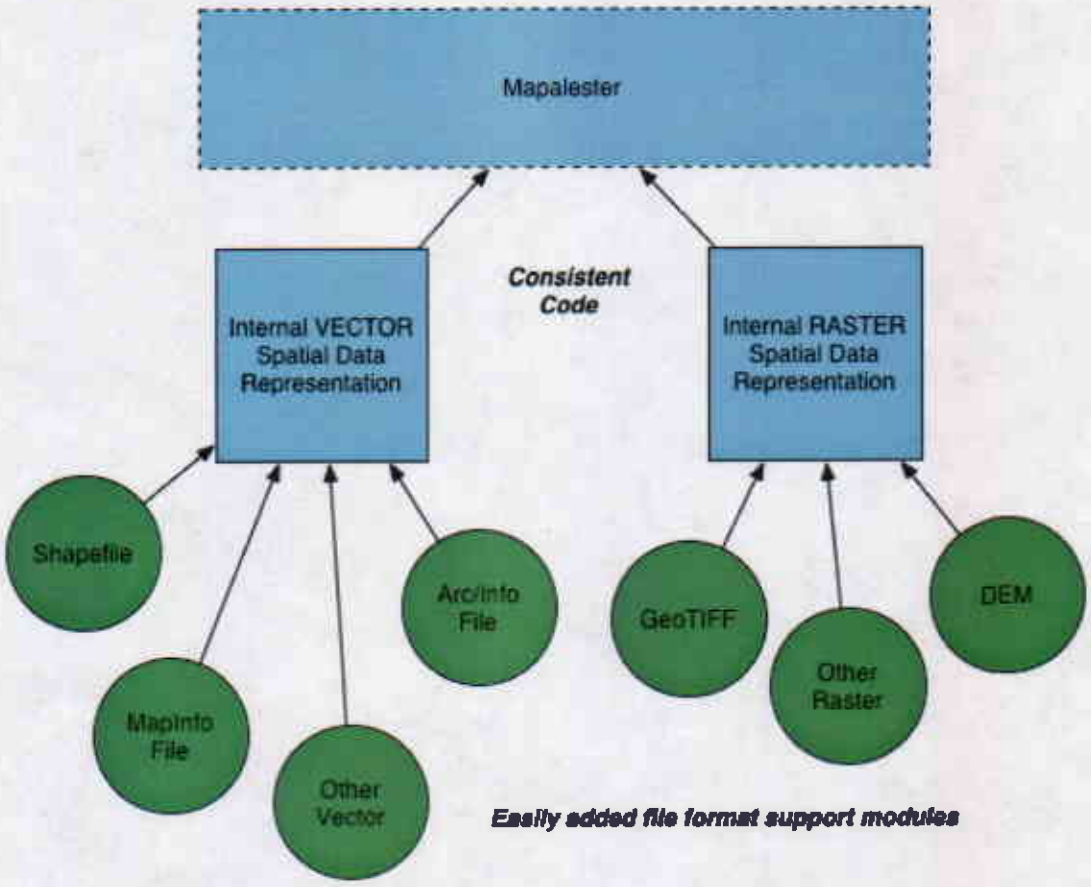


Figure 4.1.1a – A generalized diagram of Mapalester's file format support architecture.

As mentioned in the introduction, every spatial unit of GIS data - vector or raster - has two parts: spatial information and attribute information. Mapalester must understand both of these types of information for both vector and raster at two levels. The higher of the two levels is the object-oriented structure discussed in preceding sections. The lower of the levels is word-level representation stored in memory or on disk, depending on the situation. In the current implementation, the word-level representation is always in memory. However, I have been careful to use only algorithms and structures that work just as well with data from a disk, or from a combination of disk and memory. For now, given the relatively small sizes of the data

sets likely to be used by the target markets, disk-based data access is not a huge concern. More details on disk and memory access of data can be found in the discussion of indexing later in the paper.

4.1.2 General Issues Related to Vector File Format Support

Because shapefiles are the dominant and most common vector file format, I have based the word-level vector internal representation very closely on the shapefile specification. This applies for both spatial and attribute information. The similarity between the two representations allows Mapalester to very quickly convert shapefiles into the internal representation, thus minimizing processor cost for the most common of such necessary conversions. The shapefile specification will be discussed in detail in the subsequent section. However, for now, it is important to note that the internal representation is an exact copy of the shapefile specification, with two major modifications. First, while the shapefile specification requires a mixing and matching of high and low byte order words in the spatial information portion, all words in the internal representation must be in the byte order that is native to the host platform. In other words, in the Mac OS X version, all words must by high-to-low, and in the Windows version, all words must be low-to-high. To leave the byte order in its original state would mean having to perform a byte order conversion every time any data stored in a non-native byte order is accessed, significantly slowing down critical processes. Second, the internal representation ignores all shape headers and shape type specifications in spatial information. This allows the use of these bytes for other purposes, such as linked list address information. All of the important information in the record header can be stored in the object-oriented high-level representation, where it is more conveniently accessed by Mapalester. While the shape type specifications found before every shape record in a shapefile are theoretically necessary, ESRI makes clear that shapefiles should not include more than one type of shape. (ESRI 1998). See figure 4.1.2a for a comparison between the shapefile representation of the spatial information of a polyline and the representation of the same data in Mapalester's internal format. As of this writing, the internal specification for attribute information is an exact copy of the shapefile's attribute

specification. This means that Mapalester uses the xBase file format for its internal attribution information structure. See the subsequent section on shapefiles for more information.

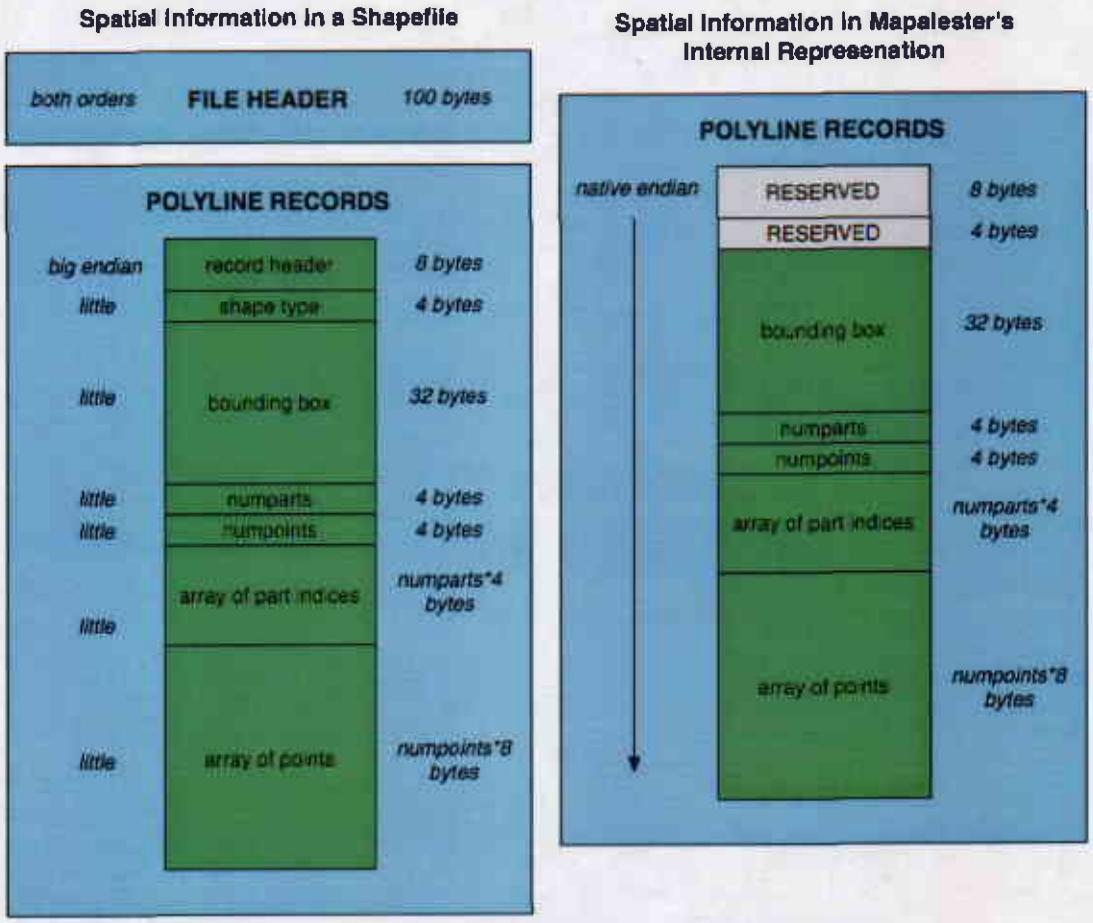


Figure 4.1.2a – The left side of this diagram shows the internal layout of the spatial part of a shapefile. The corresponding diagram on the right demonstrates the layout of the spatial part of the same sample shapefile in Mapalester's internal spatial data format. A much more detailed explanation of this format can be found in the next part of this section.

Once the word-level configuration of the data matches that of the Mapalester internal specification, the second layer of file format support – the object-oriented level – takes over. While this level is described in much more detail in the preceding sections of the paper dedicated to the object-oriented structure of Mapalester, I will briefly discuss here how Mapalester builds the objects based on the word-level configuration. Because the internal specification holds with the shapefile's requirement that each file have only

one type of shape, Mapalester starts building objects by creating the subclass of VectorTheme (PolylineTheme, PointTheme, etc.) appropriate to the data in the file just loaded. It then sweeps through the word-level specification, adding a new object of the subclass of VectorObject appropriate to the shape type of the data for each new record encountered. The constructor for VectorObjects requires a pointer to the start of the spatial data and a pointer to the attribute data in the word-level specification, so as Mapalester sweeps through the new data, it provides those pointers. Mapalester also adds the appropriate bounding box information to each VectorObject. Finally, Mapalester provides a linked list of the new VectorObjects to the new VectorTheme, which then creates an RTree index of the new objects.

4.1.3 Shapefile Support

The shapefile (.shp) specification was developed and made public by ESRI in 1998. The company created the format in response to criticisms of the shapefile's predecessor, the Arc/Info exchange file format. The Arc/Info format is based on a topological framework, which is rooted in an entirely different spatial school of thought than that of the shapefile, which is strictly nontopological in nature. The topological structure will be discussed in a future section covering the Arc/Info format, but for now, all that is important is that the topological structure of spatial data storage is inherently expensive in both storage and in processing speeds of common tasks (ESRI 1998). As a result, in the shapefile, ESRI adopted a shape-based structure in which each shape is comprised of one or more vector coordinates (ESRI 1998). This structure has many performance and storage benefits.

> "Because shapefiles do not have the processing overhead of a topological data structure, they have advantages over other data sources such as faster drawing speed and edit ability...They also typically require less disk space and are easier to read and write." (ESRI 1998)

The shapefile specification describes three separate files for each shapefile. The first and most important file is the ".shp" file, which contains all of the spatial information for each shapefile. To put it simply, the ".shp" file is what makes shapefiles geographic. A shapefile can describe information for one of fourteen types of shapes that

range from zero-dimensional to three-dimensional. However, Mapalester only works with four of the thirteen shapetypes for the following three reasons: (1) Mapalester does not yet support three-dimensional data, (2) only a portion of the one- and two-dimensional shape types are used in practice, (3) and zero-dimensional shape types are never used. It is the author's belief that restricting support to the four shapetypes – points, polylines, polygons, and multipoints – will result in very little if any functionality loss for users in Mapalester's target markets. A description of the four supported shape types can be found in figures 4.1.3a-d; these are critical to many aspects of Mapalester's GIS capabilities due to Mapalester's reliance on the shapefile for its internal representation of spatial information.

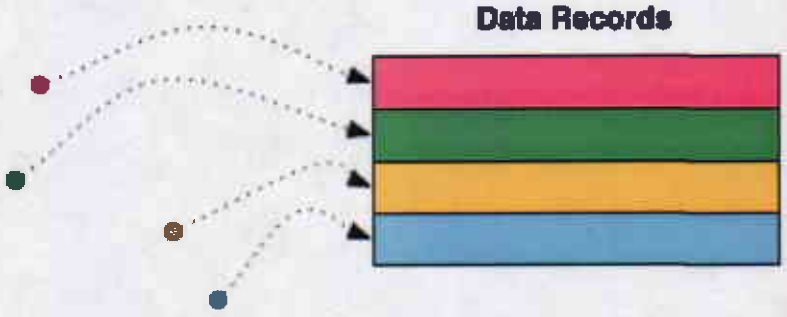


Figure 4.1.3a – Four spatial objects of the POINT shape type are shown above. Note that each point has its own data record.

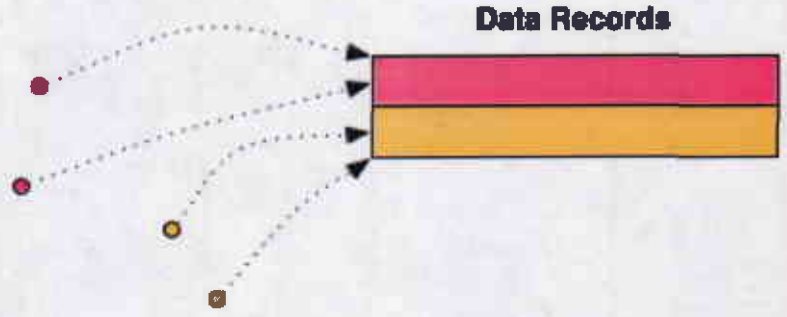


Figure 4.1.3b – Two spatial objects of the MULTIPOINT shape type are shown above. Each multipoint record contains one or more points. Multipoints are useful when multiple points have the same attribute information. For example, a file of stores in a town could be divided into multipoint records, each of a certain store type (laundry, restaurant, Curves, etc.)

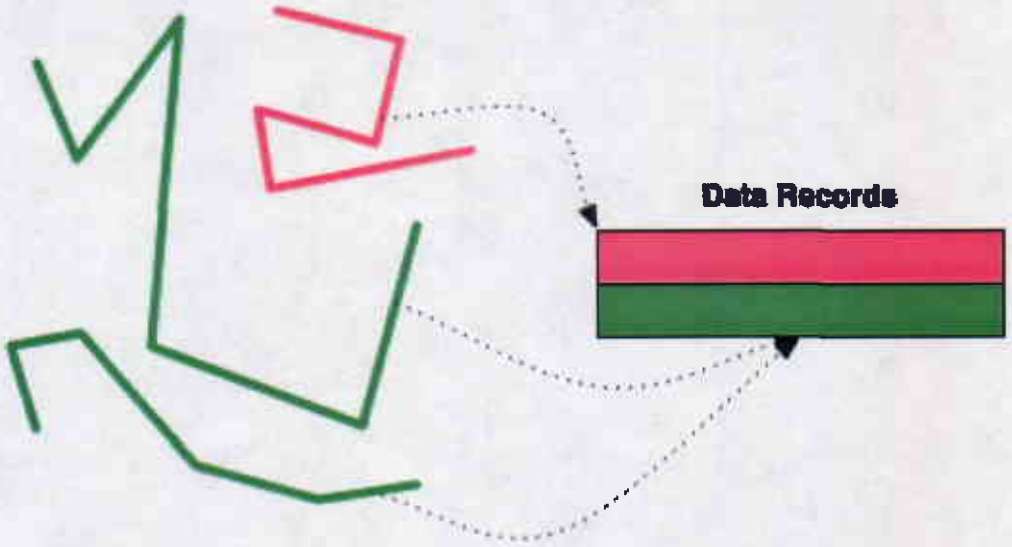


Figure 4.1.3c – Two spatial objects of the POLYLINE shape type are shown above. Note a polyline is not the same geometric concept as a line. Each polyline is made up of one or more line segments. Also notice that each polyline can be made of one or more parts (ESRI, 1998). A part is defined as a series of one or more line segments (ESRI, 1998). Parts are generally only defined when two line segment series are disjoint but must refer to the same attribute record. A single polyline can have millions of parts that all refer to the same record in the attribute information database, although in practice polylines rarely have more than twenty or so parts. At first, many people wonder why the concept of parts is necessary. A good example of a multiple-parts polyline record would be a file of toll roads on the East Coast of the United States. Toll roads often have no-toll sections, but a good polyline shapefile of these roads would keep all the toll parts of a the same road in the same polyline. A description in the case of polygons can be found in the subsequent diagram. Also, take another look at figure 4.1.2a; it should be easier to understand why the structure of the shapefile is the way that it is now that polylines have been described.

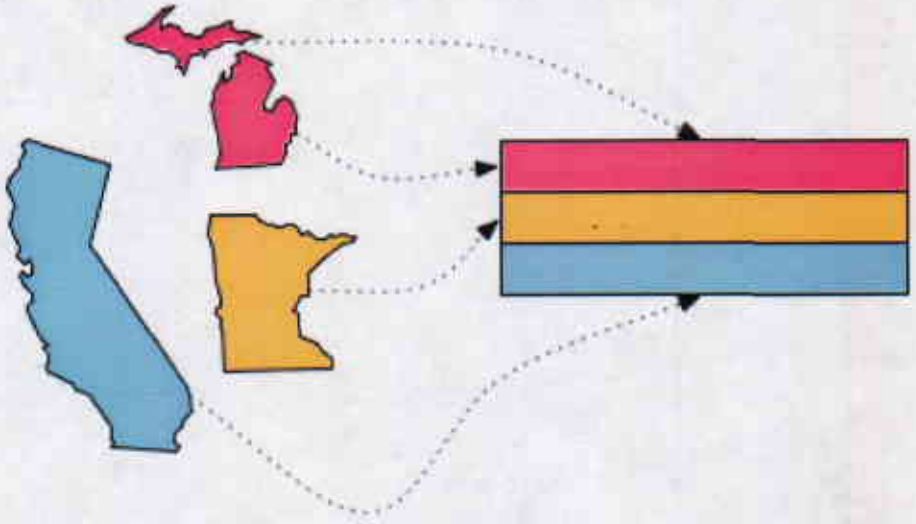
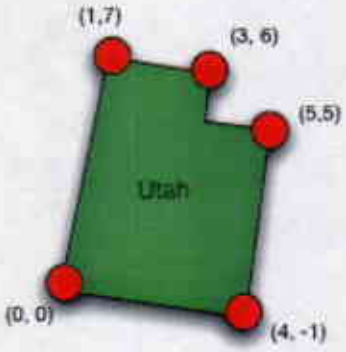


Figure 4.1.3d – Three objects of the POLYGON type are shown above. ESRI describes shapefile polygons as "one or more rings" (ESRI 1998). The ring concept is analogous to the part concept in polylines. Note that the two parts of Michigan are separate rings, but are part of the same polygon record. The points in a polygon are stored in an order such that if someone were to walk along the lines connecting the vertices, the interior of the polygon would be on her or his right. Interestingly, the spatial information for a polygon is stored identically to that of the polyline; the last vertex in each ring is simply assumed to connect to the first vertex.

The second file, the ".dbf" file, is a database file that contains all of the attribute information for each of the shapes. The ".dbf" file is in the standard xBase format, and each record in the file corresponds to a shape in the ".shp" file. Finally, the ".shx" file is an index of the ".shp" file. Each record in the ".shx" file points to the beginning of a record in the ".shp" file. Because Mapalester does its own spatial indexing, the ".shx" file is of limited use, although I have included code in Mapalester to read and utilize this file if the need arises in later development. A diagram of how the spatial information is split between the three files of a shapefile is shown in figure 4.1.3e.



Representation of Binary Data in the ".shp" file

...	...						
30600	Record for Nevada...						
31692	Record for Washington...						
34592	**Polygon**	**(0,0)**	**(4,-1)**	**(5,5)**	**(5,5)**	**(3,6)**	**(1,7)**
35678	Record for Minnesota...						
...	...						

Representation of Binary Data in the ".shx" file

...	...
204	31692
212	**34592**
...	...

Representation of Binary Data in the ".dbf" file

...	...			
(204/8)	Record for Washington...			
(212/8)	**Utah**	**Salt Lake City**	**3,500,000**	**Mormon**
...	...			

Figure 4.1.3e – Spatial object in a sample shapefile and its data distribution across the three required files in the shapefile specification.

All three files are binary files. The word-level layout of the ".shp" and ".shx" are described in the aforementioned ESRI whitepaper. The ".dbf" format is a member of the xBase complex of files, whose structure is modeled exactly off that of the dBASE format designed by Ashton-Tate, and later continued by Borland (Bachmann 2000). The word-level structure of the xBase format is public and is widely available online through a number of sources (Bachmann 2000, others). In the C++ plug-in portion of Mapalester, I have developed robust file readers for each of two of these three files (the DBF file reader

is currently under development, although a REALbasic version of the reader has been implemented as a temporary measure). File writers are a trivial expansion of the reader functionality.

Because, as mentioned above, the internal data representation is very similar to the shapefile specification, the word-level transformation from shapefile to internal data is quick and simple. The most difficult part of this transformation involves the conversion from multiple byte-orders to the byte order native to the platform. As discussed in previous sections, this is one of the few processes that is platform-specific. For Intel processors (Windows), all byte-orders must be switched to little endian. For IBM Power PC (Mac OS X), all byte orders must be switched to big endian. For Mac OS X version of Mapalester, the three files mandated by the shapefile specification must be run through the byte order swapper. Because xBase files are already little endian, the Windows version must only process the .shx and .shp files.

Performing this switch upon the first opening of the shapefile is much less expensive in processor cost over the long run than leaving the data in its original format and switching the byte order on the fly over and over again. The pseudocode for the brief algorithm used to swap byte orders can be found in figure 4.1.3f. It must be applied on all files that need byte order conversion.

function MAKE-CORRECT-WORD-ORDER (*ptr_to_start_of_data, ptr_to_end_of_data*)
input: *ptr_to_start_of_data* is a pointer to the first byte of the data file loaded into memory.
ptr_to_end_of_data is a pointer to the last byte of the data file loaded into memory

for each *long_ptr* or *double_ptt* in the shapefile specification **between** *ptr_to_start_of_data* and *ptr_to_end_of_data*
    if *is_Mac* = **true and specSaysIsLittleEndian(***long_ptr* or *double_ptr*) = **true then**
        SWAP-ENDIAN(*long_ptr*, 4) or SWAP-ENDIAN(*double_ptr*, 8)
    if is_PC = **true and specSaysIsBigEndian((***long_ptr* or *double_ptr*) = **true then**
        SWAP-ENDIAN(*long_ptr*, 4) or SWAP-ENDIAN(*double_ptr*, 8)

return

```
function SWAP-ENDIAN (firstByte, length_to_swap)
inputs:     firstByte is the address of the first byte of the series of bytes that is to be
            swapped
            Length_to_swap is the length of the contiguous series of bytes that is to be
            swapped

ptr = firstByte
for i = 0 to length_to_swap/2 – 1
        temp  = value_of(ptr + i)        // pointer addition is defined as increasing the pointer by
                                         one byte; this is  implemented in C++ by using char
                                         pointers
        value_of(ptr + i) = value_of(ptr + length_to_swap – i – 1)
        value_of(ptr + length_to_swap – i – 1) = temp

return
```

Figure 4.1.3f – Pseudocode for the MAKE-CORRECT-WORD-ORDER algorithm used to convert byte orders.

MAKE-CORRECT-WORD-ORDER is a fast algorithm; it is $O(n)$ if n is considered to be the number of byte series where conversion is necessary. If we consider the number of bytes to be swapped as n, the algorithm is only $O(n/2)$! The algorithm is fast in practice as well. It converts millions of four-byte (long) and eight-byte (double) sequences in a negligible amount of time that is barely noticeable to the user, even on the Mac OS X platform on which more conversions must be performed because the majority of data in the shapefile is stored in the little endian byte order. Only on abnormally large shapefiles on the Mac OS X platform does MAKE-CORRECT-WORD-ORDER take more than one second. This is not a problem, however, as MAKE-CORRECT-WORD-ORDER will only be performed once for every shapefile loaded into Mapalester thanks to the database structure described in previous sections of the paper.

4.1.4 Support for The ".Prj" Extension

The ".prj" extension of the shapefile specification includes critical metadata describing the coordinate system information for the shapefile with the same non-extension name in the same folder. In order to support the ".prj" extension, Mapalester must be able to convert the data in ".prj" files to Mapalester's object-oriented projection engine described in previous portions of this paper.

Unlike the ".shp," ".shx," and ".dbf" files of the shapefile specification, the ".prj" file is a text file. Because of the high variability of text encodings and REALbasic's built-in ability to handle such encodings, the parser for the ".prj" file is programmed in REALbasic code. (The projection engine itself, as mentioned above, will soon be converted into C++.) While text files in REALbasic may be easier to read and write due to the lack of binary data problems described above, the parser of the ".prj" files involves much more computer science than the ".shx", ".shp," and ".dbf" readers. Before discussing the details of the parser, however, it is first necessary to describe the structure of a standard ".prj" file.

The ".prj" file, due to the metadata it describes, exhibits a much more complicated structure than that of the aforementioned files. While the ".shx", ".shp", and ".dbf" files can be described with a simple linear definition of bytes and what they represent, the ".prj" file describes inherently hierarchical information, and must thus have a more convoluted structure. Similar to the specification of many programming languages, the ".prj" specification is based around an EBNF (extended Bachus-Naur form) grammar (ESRI, 2000). Figure 4.1.4a depicts this definition using a very slightly modified syntax of pure BNF. Terminal symbols are depicted in bold.

<PRJ_FILE> ::= <PROJ_CS> | <GEO_CS>
<PROJ_CS> ::= **PROJCS[**<NAME>,<GEO_CS>,<PROJECTION>,<PARAMETER>,<UNIT>**]**
<GEO_CS> ::= **GEOCS[**<NAME>,<DATUM>,<PRIMEM>, <UNIT>**]**
<DATUM> ::= **DATUM[**<NAME>,<SPHEROID>**]**
<SPHERIOD> ::= **SPHEROID[**<NAME>,<DIGIT>,<DIGIT>**]**
<PRIMEM> ::= **PRIMEM[**<NAME>, <DIGIT>**]**
<NAME> ::= "<ALPHA>"
<PROJECTION> ::= **PROJECTION[**<NAME>**]**
<PARAMETER> ::= ∈ | **PARAMETER[**<NAME>, <ALPHA>**]** | **PARAMETER[**<NAME>,
<ALPHA>**]**,<PARAMETER>
<UNIT> ::= **UNIT[**<NAME>, <ALPHA>**]**
<ALPHA> ::= **any character except quotes | any character except quotes** <ALPHA>
<DIGIT> ::= <TERM_DIGIT> | <TERM_DIGIT><DIGIT>
<TERM_DIGIT> ::= **0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | .**

Figure 4.1.4a - The slightly extended BNF for the ".prj" file specification. Based on ESRI's ".prj" specification (ESRI 2000).

Because of the similarity between the ".prj" file specification and that of programming languages, Mapalester's ".prj" file reader uses techniques similar to those used in programming language compilers. The reader first "tokenizes" the text from the ".prj" file using a discrete finite automata derived (DFA) from the one in figure 4.1.4b.

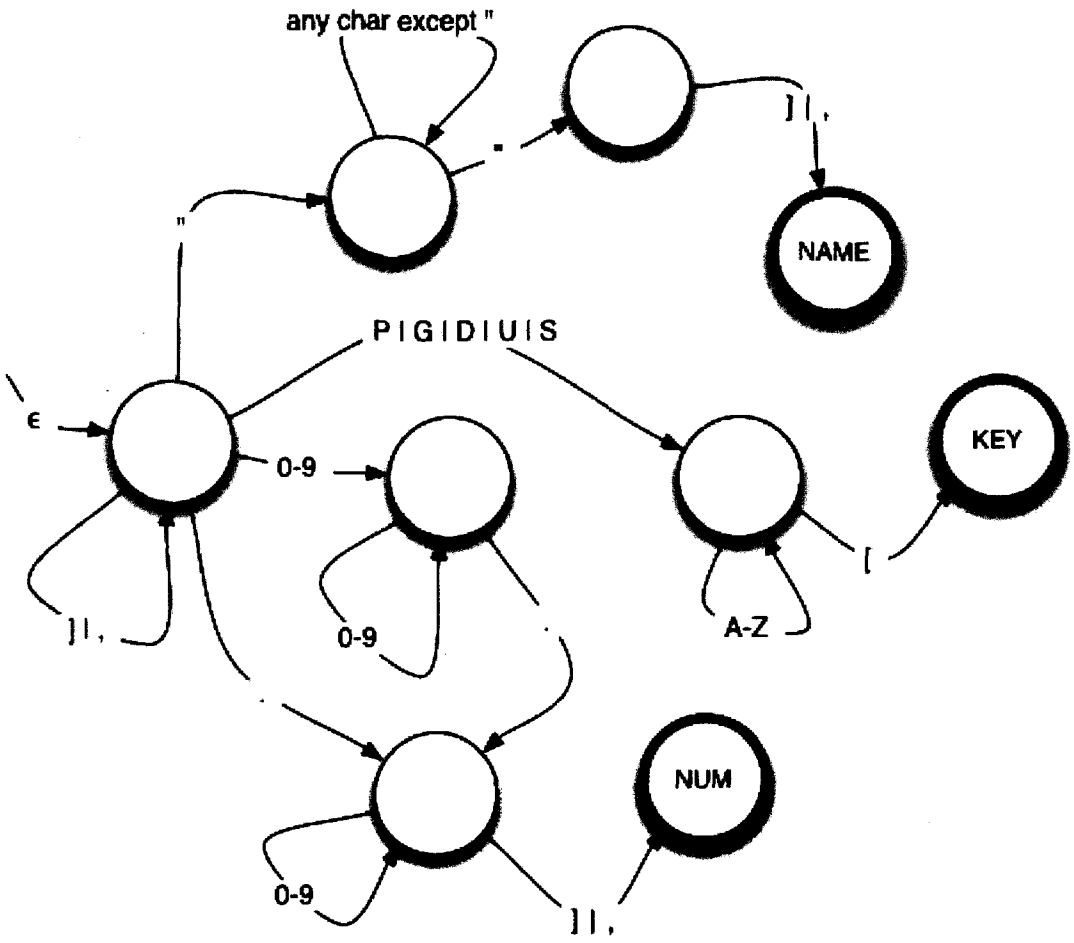


Figure 4.1.4b – The above discrete finite automata is implemented in the REALbasic ".prj" file parser. Notice that the right bracket and comma characters are the only true token delimiters.

The program then applies the grammar described in the EBNF in figure 4.1.4a to organize all of the information in the tokens. This organization has the end result of

having an instance of the ProjCS class, which is described earlier in the paper, with all of the necessary parameters to understand the coordinate system in the ".prj" file. An example of an inputted string from a ".prj" file and the resulting ProjCS class instance can be found in figure 4.1.4c.

PROJCS["NAD_1983_UTM_Zone_15N",GEOGCS["GCS_North_American_1983",DATUM["D_North_American_1983",SPHEROID["GRS_1980",6378137.0,298.257222101]],PRIMEM["Greenwich",0.0],UNIT["Degree",0.0174532925199433]],PROJECTION["Transverse_Mercator"],PARAMETER["False_Easting",500000.0],PARAMETER["False_Northing",0.0],PARAMETER["Central_Meridian",93.0],PARAMETER["Scale_Factor",0.9996],PARAMETER["Latitude_Of_Origin",0.0],UNIT["Meter",1.0]]

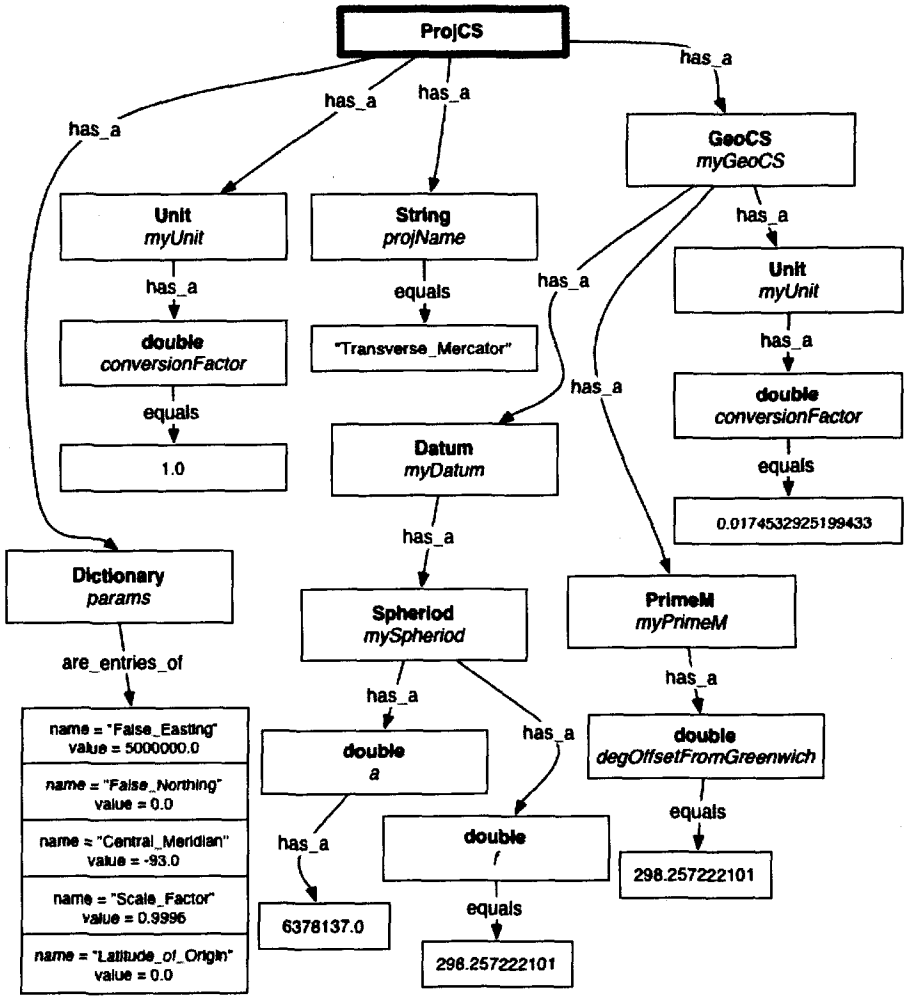


Figure 4.1.4c – The figure above is the resulting class structure generated from the ".prj" file containing text above the figure.

It is interesting to note the similarity between the class structure of the projection system and the categories in the BNF grammar. As mentioned in preceding sections, the class structure for the projection engine is based on the nature of the data that the class structure represents, and the ".prj" grammar very well represents the nature of the data it structures. This means that one could learn about projection mathematics via the grammar and someone who understands projection mathematics would understand the grammar immediately. For example, a spheroid can be defined by the major axis and the flattening, both of which are represented in the BNF. The details of projection mathematics are beyond the scope of this section, so I will not discuss them further. See sections 3.4 and 4.6 for more information.

4.2 Spatial Indexing

In any GIS software package, a search through spatially referenced data for a particular spatial location is an extremely common task. For instance, any time a user pans or zooms in an instance of Mapalester's GISCanvas (section 3.3), Mapalester must search through all of the data layers in the GISCanvas to identify which objects to draw in the canvas and which fall outside of the canvas's extent. While it may be trivial to search through a small number of spatial objects, GIS software often must scan through thousands or millions of spatial objects in order to find all objects that fall within a specified zone. As such, the issue of how to most efficiently perform these searches arises. This section is dedicated to several spatial indexing approaches used in Mapalester, each of which corresponds with a different technique for performing these searches. Each part of this section describes a different indexing approach and discusses the time and storage costs of the approach. At the time of this writing, most of the development efforts are in this portion of the program.

4.2.1 The Brute Force Approach

The positives and negatives of the brute force approach are easy to describe. The biggest advantage is that it is extremely easy to implement. It also can be somewhat storage/memory efficient, depending on the specifics of the implementation. The sole, but important, drawback is that it is very slow compared to other approaches, at least in theory. The brute force approach to spatial indexing is simple: there is no index. When a spatial search is performed, the search algorithm must simply go through all of the available and relevant spatial objects, checking each object to see if it falls within the spatial parameters of the search. Obviously, this is an $O(n)$ process. When n is small, this is not a problem, but when n is, say, a million or more – a common happenstance in the world of GIS – slowdowns occur.

The brute force approach was the first spatial indexing method used in Mapalester. It was implemented as a temporary measure, as some sort of spatial search method is a necessary stepping-stone to many other GIS processes. Surprisingly, however, performance was satisfactory for all but the largest datasets in the collection of sample vector datasets designed to echo the set of datasets likely to be used by my target markets.

I believe this gap between the theoretical slowness and the relative speed in real world tests is explained by my extensive use of the effective "bounding box" heuristic in the implementation of the brute force method. The bounding box of a spatial object is defined by the maximum and minimum coordinate for each dimension of the object. In the context of the current, 2D-only version of Mapalester, this means that the bounding box is the rectangle defined by the minimum x-coordinate, maximum x-coordinate, minimum y-coordinate, and maximum y-coordinate. See figure 4.2.1a for an illustration.

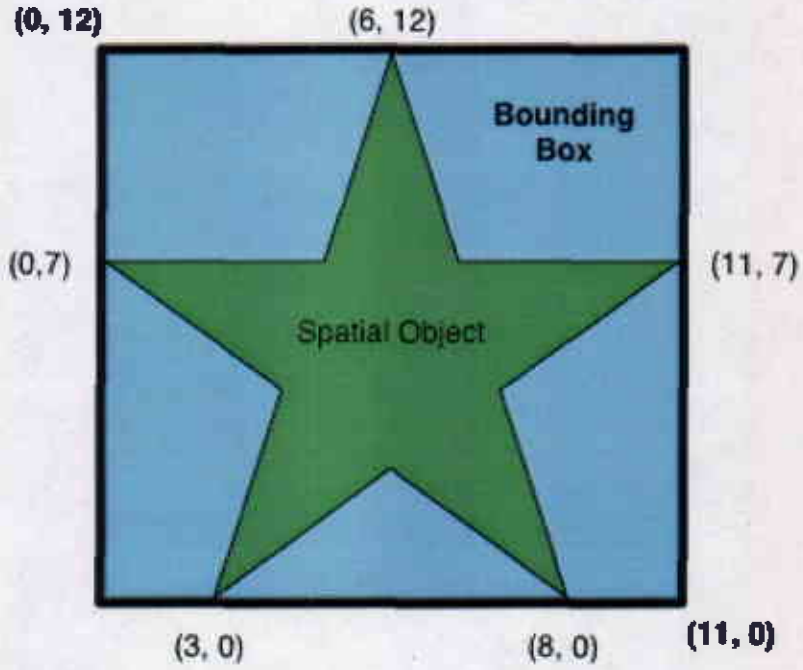


Figure 4.2.1a – The bounding box of a sample spatial object. Note that the box can be defined by the upper left and lower right vertices. The x-coordinate of the upper left vertex is the minimum x-coordinate of the spatial object. The y-coordinate is the maximum y-coordinate of the spatial object. For the lower right vertex, the x-coordinate is defined by the maximum x-coordinate of the object and the y-coordinate is defined by the minimum y-coordinate of the object.

For all of the spatial indexing approaches, the bounding box is used as an approximation for spatial objects when running the search algorithm. While this heuristic can result in false positives, as depicted in figure 4.2.1b, it will never cause any false negatives.

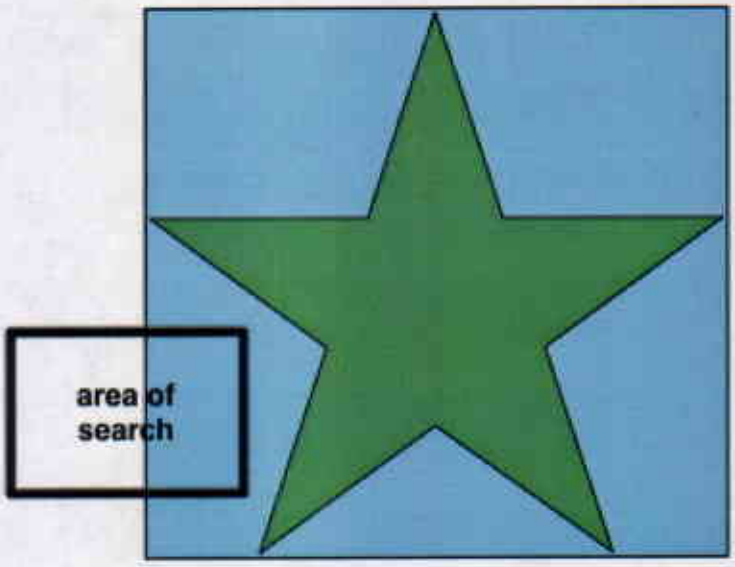


Figure 4.2.1b – Despite the fact that the rectangle area of search overlaps with the bounding box of the spatial object from figure 4.2.1a, the area of search does not actually overlap with the spatial object itself. The spatial object will be returned as a match for the search, but it will be a false positive.

The false positives are a small price to pay considering the massive reduction in calculations per shape that results. If the bounding box were not used, all of the search algorithms, no matter to which indexing approach they correspond, would have to use a polygon intersection test algorithm in the case of polygonal and polylinear spatial objects. Each iteration of this algorithm would be $O(4p)$ (O'Rourke 2002), where p is the number of points in the polygon or polyline and assuming the search area is always a rectangle (with four vertices). This would make the search algorithm associated with the brute force indexing approach $O(4pn)$ in the case of polygon and polyline datasets , where p is the maximum number of vertices of the polygons/polylines in the dataset and n is the number of polygons/polylines in the dataset. Even the faster search algorithms associated with the other spatial indexing approaches, in which n is replaced with smaller functions of n, would stall extensively under such a heavy computation load. It is interesting to note that the bounding box heuristic, due to its optimistic nature, is quite similar to the admissible heuristic for heuristic search algorithms like A*.

Most likely, it is the impressive reduction in calculations per n that causes the brute force search algorithm to perform so well against its competitors for all but the

largest datasets. Essentially, the number of calculations necessary to check if a rectangular area of search overlaps with a bounding box is so few that it takes hundreds of thousands of such comparisons in order for the time taken to be noticeable. Figure 4.2.1c is a psuedocode version of the very simple algorithm needed to perform the comparison.

```
function OVERLAP?(rect1, rect2) returns Boolean
inputs:     rect1 is the search rectangle
            rect2 is the bounding box of the spatial object being queried
if maxX(rect1) < minX(rect2)
        return false
if minX(rect1) > maxX(rect2)
        return false
if maxY(rect1) < minY(rect2)
        return false
if minY(rect1) > maxY(rect2)
        return false
return true
```

Figure 4.2.1c – The algorithm to determine if two rectangles (in this case, the search rectangle and the bounding box of a spatial object) overlap. This algorithm is so simple in theory and in practice that it is implemented in single two-line inline function in the C++ plug-in portion of Mapalester.

4.2.2 The R-Tree Approach

In order to provide good performance for the key low-level functionality that is spatial searching for all file sizes, it was necessary to abandon the brute force approach and research other approaches to spatial indexing for future implementation in Mapalester. The seminal paper by Guttman (1984) forms the foundation of most spatial indexing research still done today. In this paper, Guttman defines a construct he calls the "R-tree." While there have been many proposed improvements to the R-tree in recent years, I decided that Guttman's construct would be a good place to start and implemented

R-tree indexing. Because the R-tree deals with the spatial objects that make up data layers (below the VectorTheme level, see section 3.3), and because R-trees require the extensive use of recursive algorithms, all R-tree programming was done in the C++ language.

The basic idea of the R-tree is simple. The tree is comprised of three distinct objects: non-leaf nodes, leaf nodes, and spatial objects. Every object in the tree has a bounding box identical to those described in the preceding section. Every object in the tree's bounding box is contained within the bounding box of its parent. Additionally, Guttman (1984) identifies six more technical and specific properties of the R-tree as follows (adapted to fit context):

(1) Every leaf node contains between m and M spatial objects unless it is the root, where m is the minimum number of entries per node, M is the maximum number of entries per node, and m is greater than or equal to one half of M.
(2) Every spatial object has a bounding box that is the smallest rectangle that spatially contains the object.
(3) Every non-leaf node has between m and M entries unless it is the root
(4) Every non-leaf node and leaf-node has a bounding box that is the smallest such box that contains all of the children of the non-leaf node or leaf node.
(5) The root has at least two children unless it is a leaf.
(6) All leaves appear on the same level.

All of the algorithms used to operate on the R-tree are designed to maintain and query a tree that conforms exactly to the aforementioned specific and broad requirements. Obviously, since the R-tree is designed to make spatial searches more efficient, the most important of such algorithms is the search algorithm. This algorithm happens to be the simplest of all the algorithms connected with the R-tree. In short, the algorithm scans recursively through the tree starting with the root and returns all the spatial objects within

the bounding box input into the search algorithm. The algorithm is described in more detail in the following psuedocode:

```
function SEARCH(bounding_box) returns list of spatial objects
inputs:        bounding_box is the input of the spatial search

    SEARCH-REC(root, bounding_box, list)
    return list

function SEARCH-REC(node, bounding_box, list)
inputs:        bounding_box is the input of the spatial search
               node is the node to be examined in this spatial search
               list is the "global" list to which all positive results will be added

    for each child in node
        if overlaps(bounding_box, getBoundingBoxOf(child)) = true then
            if isLeafNode(child) = false then
                SEARCH-REC(child, bounding_box)
            else
                append(list, child)
            end if
        end if
    end if
```

Figure 4.2.2a – The R-tree search algorithm

While the R-tree works beautifully and quickly even on polygon and polyline files, the massive number of objects to insert in the R-tree with even a small point file quickly overloads the R-tree capabilities. Point entries are loaded into R-tree by making their bounding boxes degenerate rectangles whose extent is a single point. While polygon and polyline files may have more points total due to the massive number of vertices for many polygons and polylines, they generally have less than 10,000 or so polygons or polylines, which is the level at which bounding boxes are established, and thus the level at which the R-tree indexes. As mentioned above, the R-tree is a data structure whose processes get increasingly more expensive time-wise as more and more items are

inserted. As a result, in preliminary tests, the R-tree was up to twenty times slower than the brute force approach for a point file with around 165,000 points. It is not uncommon for point files to have that many points or more.

As such, it was necessary to find a way to reduce the number of bounding boxes indexed for point datasets. I attempted three approaches to accomplish this goal, all of which either had little impact on the number of bounding boxes indexed or made bounding boxes that were so large that they eliminated most of the benefits of using the R-tree to begin with. The first approach simply grouped *k* points that were adjacently stored in the points file. For all values of *k* tested, the resulting bounding boxes were either too large or the insert algorithm was too slow, or both. Obviously, I needed a more intelligent approach. I next looked into point clustering algorithms in an effort to more accurately group the points and thus minimize bounding box rectangles. The most common type of clustering algorithm is the k-means algorithm, but this algorithm is $O(n^n)$ in the worst case. While it is often much better in practice, it would involve running through all of the points at least several times and comparing each point with the number of clusters desired. Such an algorithm would take more time than it takes to insert the points individually. As such, I implemented a standard variant on the k-means algorithm, often referred to as the one-pass k-means algorithm. The algorithm, modified for the context of Mapalester, is depicted in pseudocode in figure 4.2.2b.

```
function ONE-PASS-K-MEANS(points, epsilon, clusters) returns list of clusters
inputs:     points is the set of points to cluster
            epsilon is the maximum distance a point can be from the geometric mean of a
            cluster
            clusters is an array of clusters (implemented as MultipointObjects), initially empty

for each p in points
    foundCluster = false
    for each c in clusters
        if distance(geometricMeanOf(c), p) < epsilon
            addPointToCluster(c, p)
            adjustMeanOfCluster(c)
            foundCluster = true
```

```
    if foundCluster = false
        c1 = makeNewCluster()
        addPointToCluster(c,p)
        adjustMeanOfCluster(c)
        addToArray(c, c1)
```

Figure 4.2.2b – A psuedocode implementation of a one-pass k-means algorithm.

In the worst case – when no point is within *epsilon* of another point – this algorithm is $O(n^2)$, a significant improvement over the basic k-means algorithm. Additionally, the worst case is very unlikely to occur if *epsilon* is chosen wisely. Note, however, that this algorithm has much less accuracy than the basic k-means algorithm as points are not reassigned as the geometric means shift. However, because this algorithm walks the line between speed and accuracy very well, it was worth trying in Mapalester. Unfortunately, despite a large number of tests, I was unable to find a context-sensitive equation for *epsilon* that would not either cause the near worst-case performance to occur (and thus also not reduce the number of bounding boxes to index) or that would not generate bounding boxes that were so large that there was almost no point in using spatial indexing. The final approach to clustering that I attempted was to combine the first two approaches into my own algorithm. This algorithm's psuedocode appears in figure 4.2.2c.

```
function SLOPPY-K-MEANS(points, epsilon, clusters) returns list of clusters
inputs:     points is the set of points to cluster
            epsilon is the maximum distance a point can be from the geometric mean of a
            cluster
            clusters is an array of clusters (implemented as MultipointObjects), initially empty
c1 = makeNewCluster()
for each p in points
    if distance(geometricMeanOf(c), p) < epsilon
        addPointToCluster(c, p)
        adjustMeanOfCluster(c)
        foundCluster = true
    else
```

```
        c1 = makeNewCluster()
        addPointToCluster(c,p)
        adjustMeanOfCluster(c)
        addToArray(c, c1)

return clusters
```

Figure 4.2.2c – A synthesis of the trivial and intelligent one-pass point clustering algorithms.

Unfortunately, for all logical *epsilon* values attempted, this algorithm resulted in far too many clusters, which meant that the number of bounding boxes was too great to have much of an impact on the index insertion time. While I have not given up on the combination of the R-tree with the point clustering approach, I have switched my attention to the implementation of the R*-tree, which claims to handle points much more efficiently than the basic R-tree. The R*-tree seems to be the most popular 2D-focused derivative of the R-tree (Manolopoulous et. al, 2003), so I am hopeful that it will provide good results.

4.3 Line Simplification

Line simplification algorithms are a portion of the computational answer to the age-old art of generalization in cartography. Early cartographers had to determine the level of detail in polylines (both those that stand on their own and those that are parts of polygons) based on the scale of the map they were generating. For example, maps of the entire United States generally do not need the detail of every tiny nook and cranny of the eastern coastline. The idea behind line simplification algorithms is to automate this process for the huge increases in available spatial data. In their book Generalization in Digital Cartography, McMaster and Shea (1992) describe line generalization algorithms with the following definition: "As a general definition, simplification algorithms weed from the line redundant or unnecessary coordinate pairs based on some geometric criterion, such as distance between points or displacement from a centerline." (McMaster and Shea 1992).

McMaster and Shea identify four main benefits of line simplification in digital cartography: (1) reduced plotting time for (now antiquated) digital plotters, (2) reduced storage space, (3) faster vector to raster conversion, and (4) faster vector processing for operations such as translation, rotation, and rescaling. However, in my research, I have found that, in the time since McMaster and Shea wrote their book, processing speeds have increased enough to nullify most of the benefits of simplification. In fact, in my experience, many of the algorithms presented in their book *add* computation costs and result in slower display and data processing speeds. As a result, Mapalester implements only the most trivial of line simplification algorithms. The first part of this section identifies the results of multiple implementations of the classic Douglas-Peucker algorithm. The second part describes the trivial algorithm actually used in Mapalester. Finally, the section closes with a description of the possibilities that lie with an enhanced version of the Douglas-Peucker line simplification presented by Hershberger and Snoeyink.

Before discussing the efficacy of various line-generalization algorithms, however, it is important to discuss a recently discovered use of line-generalization algorithms for which an accurately generalized line, and not speed of algorithm, is the most important factor. This use – reducing the size of extremely detailed vector data layers to a digital capacity that is distributable, say, on the Internet – is an extremely high-level function, however and may not even make the first version of Mapalester. The most famous application of this use of line generalization has been the production of reasonably sized U.S. census boundary files from the TIGER database (U.S. Census 2004).

4.3.1 Douglas-Peucker

One of the hallmarks of early GIS software and one of the most famous algorithms in computational cartography is the Douglas-Peucker line simplification algorithm. I initially assumed that an implementation of the algorithm would be a necessary element of Mapalester. Additionally, early implementations of Mapalester suffered from slow graphics display speeds, a problem that presumably would be solved by having less detail to display, which is the result of line simplification algorithms. I

initially implemented a recursive version of the algorithm, described by the pseudocode in figure 4.3.1a, which is adapted from Sunday (2002). Figure 4.3.1b shows the effect of Douglas-Peucker on a sample line.

```
function BEGIN-SIMPLIFY(tolerance, line) returns generalized line
inputs:     tolerance is the approximation tolerance for the algorithm, which in the case of
            the Mapalester implementation, is measured in pixels
            line is the line to be generalized and contains a list of all of the points in the line.

DP-SIMPLIFY(tolerance, line, firstVertexIn(line), lastVertexIn(line))
for each vertex in line
        if isMarked(vertex) = true then
                add(vertex, newLine)
        end if
next
return newLine

function DP-SIMPLIFY(tolerance, line, a, b)
        if a = b then
                return
        end if
        lineAB = getSegmentOfLine(line, a, b)
        for each vertex in lineAB
                if distanceFrom(vertex, lineAB) > maxDistanceSoFar
                        maxVertexSoFar = vertex
                        maxDistanceSoFar = distanceFrom(vertex, lineAB)
                end if
        next
        if maxDistanceSoFar > tolerance then
                DP-SIMPLIFY(tolerance, line, a, maxVertexSoFar)
                DP-SIMPLIFY(tolerance, line, maxVertexSoFar, b)
        end if
```

Figure 4.3.1a – High-level psuedocode for the Douglas-Peucker algorithm.

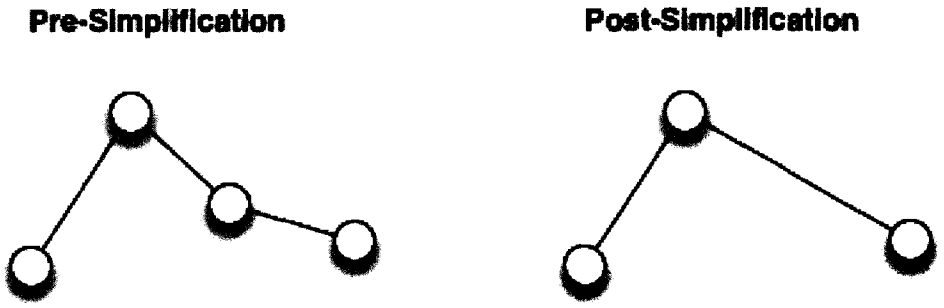


Figure 4.3.1b – The result of Douglas-Peucker on an example polyline with four vertices .

While simple and short, this algorithm has surprisingly high processor costs. The step that requires the most computation is illustrated by the distanceFrom function in the psuedocode. To determine the distance of a point from a line, it is necessary to use vector arithmetic, which slows the algorithm significantly. In addition, the distanceFrom calculation must be performed n^2 times in the worst case, where n is the number of vertices in the input polyline. (The algorithm is O(n*m), where n is the number of vertices in the input polyline and m is the number of vertices in the output polyline. The aforementioned case is that in which no simplification occurs due to all vertex distances exceeding the tolerance.) Moreover, for a large data layer, this algorithm must be performed on thousands, or even tens of thousands, of polylines.

Taking the computational requirements of Douglas-Peucker into consideration, it is no surprise that the time costs of performing the algorithm outweighed the computational benefits of simplification, at least in the context of data display. The performance of the algorithm is highly dependent on the tolerance value that is chosen. However, for all tolerance values that resulted in reasonably accurate display, the algorithm added time to the data display process.

In an effort to decrease the computational costs of the algorithm, I implemented an iterative version of the algorithm. To do this, I employed standard techniques used to convert tail-recursive algorithms like Douglas-Peucker. However, even the iterative version proved more costly in the context of data display than no generalization at all.

4.3.2 The Trivial Approach

Due to the failure of Douglas-Peucker to speed up data display and several other operations, I instead implemented a trivial line simplification approach whenever simplification was needed for speed reasons. For instance, in the case of data display, the most expensive operation is the actual drawing of pixels. As such, my trivial simplification data display algorithm is as follows:

```
function TRIVIAL-SIMPLIFY(line)
inputs:     line is the line to be generalized and contains a list of all of the points in the line.
    lastPoint = firstVertexIn(line)
    for each vertex in line
        point = getAppropriatePlotPoint(line)
        if point does not equal lastPoint then
            drawLine(lastPoint, point)
        end if
        lastPoint = point
    next
```

Figure 4.3.2a – The trivial line simplification algorithm implemented in Mapalester.

4.3.3 The Hershberger/Snoeyink Speed-Up

In 1994, Hershberger and Snoeyink proposed an improvement on the classic Douglas-Puecker algorithm that improves the worst case performance from $O(n^2)$ to $O(n*\log(n))$. The algorithm is identical to Douglas-Puecker except it replaces the distance calculation step identified above as a key performance issue in Douglas-Puecker with a more advanced method of distance calculation that uses convex hulls. The implementation cost of this speed-up is much higher and Hershberger and Snoeyink themselves admit that "the statistical properties of cartographic data usually means that the straight-forward implementation is slightly faster than the convex hull implementation." (Hershberger, 1994) While Hershberger and Snoeyink claim that their simplification algorithm "may be interesting" for parallel or interactive applications, none

of these applications currently exist in Mapalester. As such, no attempt to implement the speed-up in Mapalester will be made in the near future.

4.4 iTunes-like Database Functionality

Through my teaching assistant work, I determined a major weakness of existing GIS software to be its reliance on Windows Explorer as the dominant data file organization framework. The user is expected to more or less maintain a system of folders and files that organizes her or his data. There are two main problems with this framework, however. First, especially in a lab context, it is easy for this system to be invalidated by a careless or creative user. Second, the user may miss certain similarities and patterns in some of her or his data sets that could be identified by a more intelligent data file organization system. As a result, I set out to develop a new system of data file organization. Utilizing similar methodology to the manner in which I developed the GISFormatWindow – identifying what has worked in other programs and modifying it for my purposes - I singled out two possible models for providing the user with the best possible data file management features.

The first of these models was found in photo management software, the second in music jukebox software. How do photo and music files relate to GIS data files? The answer to this question lies in the fact that photo, music, and GIS files all carry a large amount of metadata. Whereas a photo may have the capture date, the capture location, and the resolution; and a music file may have the artist, album, and genre; a GIS file has the data author, field descriptions, legal restrictions, edition, data group, and a great number of other metadata fields. I was not pleased with how existing photo management software allows its users to navigate through the metadata of photo files, primarily because users of photo management software place a much higher value on previews of the data along with display of attribute data than do GIS users. While previews are a satisfactory supplemental feature, they need not be a core element of GIS data file management. Music jukebox software applications, therefore, provide the better model for GIS data. Out of a field of several excellent case studies, I identified the wildly popular iTunes software made by Apple as having what I believe to be the best support of

metadata. I made this judgment for two reasons. First, iTunes by default loads music files into a central database when they are first opened, and does not require original files after that point. This frees the user from having to use Windows Explorer or the Mac OS X Finder to organize data, and assures that the user's only interaction with music files is through the iTunes metadata browser, not through location on the hard drive. In other words, the user never needs to know where she or he stored a music file, just, for example, who sings the song. Second, I believe iTunes strikes a good balance between simplicity of interface and data management power. Primarily through a combination of the use of the "playlist" and "library" concepts, iTunes provides relatively quick access to music files while still supporting stringent organization and searching for music by metadata fields.

Mapalester's GIS data file management system is heavily influenced by that in iTunes. An additional benefit to the adoption of iTunes' system is that users familiar with iTunes will feel right at home in Mapalester. The basis for the GIS file management system is complete; users can view and edit metadata of loaded GIS files, for example. The metadata fields supported in the library feature were determined through the study of a sample of GIS file metadata and through consultation with Professor Carol Gersmehl. While it is very easy to add/delete/modify the fields, the fields are likely stay as they are for the first release (see section 3.4 for more details).

Support for searching and data "playlists" will be simple to implement and will be in the first released version.

4.5 Prime Meridian Conversion or, "The GIS that Acts Like a Globe, Not a Map"

A significant amount of development time has been dedicated to implementing a feature that allows users to interact with Mapalester as they would with a globe, replacing the map-based interactivity found in current GIS software. More specifically, this feature, which is nicknamed "GISpin," enables users to interact with the world as a continuous surface; users panning extents will never encounter an "edge" of the world, so to speak. A key result of GISpin is that Mapalester will not require the world to be "split" at a certain meridian. This "split" is very familiar to map viewers, and has been

identified in personal interviews with teachers as a means of marginalizing the portrayal of certain areas of the world. For instance, most maps made in the United States split the globe at the meridian that runs between Alaska and the eastern reaches of Russia. This creates the impression that geographic processes are not continuous across the split meridian. Given that one of Mapalester's target markets is the education market, it is important for Mapalester to avoid supporting this impression.

GISpin is not yet complete, but will appear in the final version of Mapalester. Implementation of the feature has been an interesting mix of projection and datum mathematics and computer science techniques. This section is dedicated to describing these techniques. Because of the nature of spatial data, implementation of this feature is essentially comprised of two very different components. The first component, which has been completed but not yet incorporated into the Mapalester interface, involves implementing the functionality for point data layers. The second, and more difficult, does the same for polyline and polygon data layers.

4.5.1 Point Support in GISpin

In principle, supporting points in "GISpin" is quite simple. There are many technical, detail-oriented implementation difficulties surrounding the incorporation of point GISpin into the projection system, but these are outside the scope of this paper. As such, the simple theory will be the only element of the point support discussed. Similarly, I will limit the discussion to data in "unprojected" or "geographic" coordinate systems, because these systems use commonly known angular units like longitude and latitude. It is important to note, however, that the same techniques can be adapted to projected coordinate systems, with one major difference discussed at the end of this subsection.

In their native form, unprojected coordinates have a latitude and longitude relative to the equator and a given prime meridian, respectively. While the equator is defined as the intersection between plane x and the surface of the earth where x is perpendicular to the line segment that contains both poles (non-magnetic) and intersects this line at its midpoint, the prime meridian is an arbitrary construct. In fact, the currently predominate

prime meridian – the meridian that passes through Greenwich, England – was not agreed upon as the default prime meridian until the International Meridian Conference in 1884, and certain other prime meridians are still used on occasion.

It is this arbitrary nature of the prime meridian from which GISpin derives its power. Stated simply, GISpin makes the prime meridian the center of the current view extent and adjusts the longitudinal coordinate of each point accordingly. In this way, if the current view extent center is in eastern Russia, around 179.5° East longitude using the Greenwich meridian, Mapalester is able to easily project the points in, say, Alaska, on the right of the 180° line, accurately representing where they exist in the real world. Mapalester does this by establishing the 0° line as the 179.5° East line, and adjusting all other points accordingly. In current GIS software, these points would appear on the opposite side of the possible view extent, just as in a world map.

It is important to consider the issues involved in transferring this methodology to projected coordinate systems. The most important of these issues involves the infinite distortion inherent to the periphery of the range of many projection equations. For instance, in the standard Mercator projection, the south and north poles are projected to infinity. While this particular case is not problematic to GISpin because the latitudinal display restriction can be maintained while rotating longitudes, the corresponding drawback in transverse Mercator (which can be described as Mercator flipped on it side), is highly problematic. Left unchecked, this problem could result in users scrolling through infinitely extended extents at the periphery of the range of the projection equation. This problem has an interesting solution, which has the convenient side effect of helping to eliminate much of the confusion surrounding projections for new users. Essentially, Mapalester simply changes the central meridian of the projection equation to the center of the view frame. In the case of equations with two central meridians, Mapalester uses the meridians that divide the view frame into three equal parts. This methodology has the effect of reprojecting all points in the view frame to maximize the benefits of the chosen projection, averting a common and hard-to-catch problem for new GIS users. Interestingly, if this methodology is extended, it would be very easy to implement an auto-projection system in Mapalester that would choose the best projection for any given view frame. However, such a system would require a large amount of

human-computer interaction research or the construction of an expert system in order to determine the best projection for every category and scale of view frame extent. Nonetheless, it would provide an interesting avenue for graduate school research.

4.5.2 Polyline and Polygon Support in GISpin

The concepts behind polyline and polygon support in GISpin are identical to that for points. After all, polylines and polygons are nothing but a series of ordered points that are connected with lines. However, the manner in which computers construct and fill polylines and connect the points in polylines, creates an enormous road block to implementing support for these geometric forms in GISpin. This problem appears only when the view extent is large enough that a polyline or polygon that appears on the right side of the view frame also appears on the left side of the frame. For instance, such a situation would occur if the view frame is the maximum possible while using GISpin – the whole world – and the center of the frame is aligned such that the United States is cut off on the right side. In these situations, when Mapalester tries to draw a line segment from the last vertex on the right side to the first vertex on the left side (or vice versa), instead of drawing a line to the edge of the view frame on the right side and beginning again on the left, Mapalester just draws a line all the way across the view frame. Obviously, although the context for this bug is somewhat uncommon, the bug is still a major showstopper.

After attempting many trivial and complicated workarounds, I determined the solution to this bug to lie in one of the several line and polygon "splitting" algorithms available. Such algorithms will generate the set of polylines or polygons created by the bisection of a polyline or polygon by a line. Mapalester would draw the polylines and polygons output by this algorithm instead of the original polylines and polygons described in the data layer. Unfortunately, these algorithms have high implementation costs, especially when speed is such an enormous factor. As such, I have not yet been successful implementing these algorithms and, as a result, Mapalester still uses a map-based view frame.

Once I have completed the implementation of these polyline and polygon bisection algorithms, they will have uses well outside GISpin. For example, as mentioned above, a small number of data sets come in geographic coordinate systems that incorporate a prime meridian other than the Greenwich meridian. In order to convert such data sets to display correctly in a data frame with any other prime meridian, I will need to use the algorithms to reconfigure the original coordinates of such data sets. Additionally, the algorithm implementations will come in handy when I am implementing such standard GIS features as layer intersections and clipping.

4.6 Projection Conversion Functions

While I have discussed the structure and function of the coordinate system structure in Mapalester in previous sections, I have yet to explain in detail the actual process of projection conversion. This process is conceptually simpler than the structure of the coordinate system, but just as complicated to implement. Projection conversion functions can be separated into two broad categories, those that "project" and "unproject." Each map projection has one of each. The functions that "project", often referred to as "forward direction" projection functions, take two angular-unit inputs (the longitude and latitude, often abbreviated as λ and ϕ, respectively) and return two length unit outputs (Cartesian coordinates usually in meters, often abbreviated as x and y, respectively). The functions that "unproject," often called "reverse direction" projection functions, take length coordinates as inputs and return angular units as outputs. The functions, regardless of direction, often take other parameters as well. Common supplementary parameters include the central meridian, latitude of origin, and standard parallels.

Projection conversion functions can also be categorized based on whether or not they deal with the Earth as a spherical or ellipsoidal object. Those that view the Earth as a spheroid are only used in very small-scale maps. The vast majority of commonly-used projection conversion functions are those of the ellipsoidal persuasion.

When completed Mapalester will support the set of projection conversion functions found in table 4.6a. The bolded functions are those that are already completed. The table is based on the equivalent table for ESRI's ArcGIS 9. Since data is sometimes

distributed in obscure projections, I have made supporting the widest possible collection of projection conversion functions a high priority. A user not being able to input a certain data set because the data set is projected in an unsupported projection violates both the ease-of-use and powerful development goals.

NAME	DIRECTION	SHAPE
Aitoff	both	sphere
Albers Equal Area Conic	both	ellipsoid
Behrmann	both	sphere
Bonne	both	ellipsoid
Cassini	both	ellipsoid
Craster Parabolic	both	ellipsoid
Cylindrical Equal Area	both	ellipsoid
Double Stereographic	both	ellipsoid
Eckert I - VI	both	sphere
Equidistant Conic	both	ellipsoid
Equidistant Cylindrical	both	sphere
Flat Polar Quartic	both	sphere
Gall Stereographic	both	sphere
Gauss-Kruger	**both**	**ellipsoid**
Gnomonic	both	sphere
Hammer-Aitoff	both	sphere
Hotine Two Point	both	ellipsoid
Hotine Azimuth	both	ellipsoid
Krovak	both	ellipsoid
Lambert Azimuthal Equal Area	both	ellipsoid
Lambert Conformal Conic	**both**	**ellipsoid**
Loximuthal	both	sphere
Mercator	**both**	**ellipsoid**
Miller Cylindrical	both	sphere
Mollweide	both	shpere
New Zealand Map Grid	both	ellipsoid
Orthographic	both	sphere
Plate Carree	both	sphere
Polyconic	both	ellipsoid
Quartic-Authalic	both	sphere
Robinson	both	ellipsoid
Sinusoidal	both	ellipsoid
Stereographic	both	ellipsoid
Stereographic North Pole	both	ellipsoid
Stereographic South Pole	both	ellipsoid
Times	both	sphere
Transverse Mercator	**both**	**ellipsoid**
Two-Point Equidistant	both	sphere
Van der Grinten I	both	sphere

Vertical near-side perspective	both	sphere
Winkel I	both	sphere
Winkel II	both	sphere
Winkel Tripel	both	sphere

Figure 4.6a – Table of supported projections. Bold projections are finished.

Fortunately, I have set up the structure of the coordinate system engine in such a way that adding new projection conversion functions is relatively easy. All that is required is to essentially give the new projection a name, enter the function into the code, and optimize the function. Projection conversion functions are nothing more than formulas. As such, I simply programmatically encode these formulas in REALbasic or C++ code (The final version will only use C++ code as all the data that needs to be projected exists at a level below that of the map layer, and thus must be coded in C++ plug-in format. See section 3.3 for more details). My main source for the formulas has been the seminal map projection book Map Projections – A Working Manual by John Snyder of the USGS. An example of a formula from the book and its REALbasic code equivalent can be found in figure 4.6b.

Transverse Mercator Forward Projection Conversion Formula for Spheroids: Synder Formula

$$x = k_o N[A + (1 - T + C)A^3/6 + (5 - 18T + T^2 + 72C + 56e'^2)A^5/120]$$
$$y = k_o\{M - M_o + N\tan\phi[A^2/2 + (5 - T + 9C + 4C^2)$$

$$e'^2 = e^2(1-e^2)$$

$$N = a/\sqrt{(1-e^2\sin^2\phi)}$$

$$T = \tan^2\phi$$

$$C = e'^2\cos^2\phi$$

$$A = (\lambda - \lambda_o)\cos\phi$$

$$M = a[(1-\frac{e^2}{4}-\frac{3e^4}{64}-\frac{5e^6}{256}-...)\phi-(\frac{3e^2}{8}+\frac{3e^4}{32}+\frac{45e^6}{1024}+...)\sin 2\phi+$$

$$(\frac{15e^4}{256}-\frac{45e^6}{1024}-...)\sin 4\phi-(\frac{35e^6}{3072}+...)\sin 6\phi+...]$$

k_o = the scale on the central meridian. This is a constant and is input into the formula. For the UTM projection, this equal 0.9996, so this is the value always in Mapalester.

e = the eccentricity of the ellipsoid. This is defined by the datum of the GeoCS class of the same ProjCS class.

a = the semi-major axis of the ellipsoid. This is defined by the datum of the GeoCS class of the same ProjCS class.

Mo = "M calculated for ϕ_o, the latitude crossing the central meridian λ_o at the origin of the x,y coordinates." (Snyder 1987) The central meridian is also an input into the projection formula. For UTM uses, this is the center of the given UTM zone.

Transverse Mercator Forward Projection Conversion Formula for Spheroids: REALbasic Code Derived from Synder Formula

```
    // pre-calculate all the stuff you can
dim falseE as double = params.value("False_Easting") * myUnit.conversionFactor
dim falseN as double = params.value("False_Northing") * myUnit.conversionFactor
dim lam0 as double = params.value("Central_Meridian") * myGeoCs.myUnit.conversionFactor
dim k0 as double = params.value("Scale_Factor")
dim phi0 as double = params.value("Latitude_Of_Origin") * myGeoCs.myUnit.conversionFactor
dim Mo as double = a * ((1-e2/4 - 3*e4/64 - 5*e6/256)*phi0-(3*e2/8 + 3*e4/32 +
45*e6/1024)*sin(2*phi0)+ (15*e4/256 + 45*e6/1024)*sin(4*phi0) - (35*e6/3072)*sin(6*phi0))
dim term1 as double = (1-e2/4 - 3*e4/64 - 5*e6/256)
dim term2 as double = (3*e2/8 + 3*e4/32 + 45*e6/1024)
dim term3 as double = (15*e4/256 + 45*e6/1024)
dim term4 as double = (35*e6/3072)

dim N, T, C, bigA, Mp as double
dim cosY, tanY as double

dim bound as integer = uBound(m)
dim tempBound as integer
dim i,j as integer
```

```
for i = 1 to bound
 tempBound = uBound(m(i).x)
 for j = 1 to tempBound

  m(i).projX(j) = m(i).projX(j) * myGeoCs.myUnit.conversionFactor
  m(i).projY(j) = m(i).projY(j) * myGeoCs.myUnit.conversionFactor

  cosY = cos(m(i).projY(j))
  tanY = tan(m(i).projY(j))

      // handle special cases
  if m(i).projX(j) - lam0 >= πDiv2 then
   m(i).projX(j) = πDiv2 - SMALLEST_NUMBER + lam0
  elseif m(i).projX(j) - lam0 <= -πDiv2 then
   m(i).projX(j) = - πDiv2 + SMALLEST_NUMBER + lam0
  end if

  if m(i).projY(j) < πDiv2 and m(i).projY(j) > -πDiv2 then
   N = a/sqrt(1-e2*(sin(m(i).projY(j)))^2)
   T = tanY^2
   C = eprimesq*cosY^2
   bigA = (m(i).projX(j) - lam0)*cosY
   Mp = a * (term1*m(i).projY(j)-term2*sin(2*m(i).projY(j))_
     + term3*sin(4*m(i).projY(j)) - term4*sin(6*m(i).projY(j)))

   m(i).projX(j) = k0 * N * ( bigA + (1 - T + C)*(bigA^3)/6 + (5 - 18* T + T^2 + 72*C +
58*eprimesq)*(bigA^5)/120)
   m(i).projY(j) = k0 * (Mp - Mo + N * tanY*((bigA^2)/2 + (5 - T + 9*C + 4*C^2) *(bigA^4)/24 + (61 -
58*T + T^2 +600*C - 330*eprimesq)*(bigA^6)/720))
   m(i).projX(j) = m(i).projX(j)*myUnit.conversionFactor + falseE
   m(i).projY(j) = m(i).projY(j)*myUnit.conversionFactor + falseN
  else
   Mp = a * ((1-e2/4 - 3*e4/64 - 5*e6/256)*m(i).projY(j)- _
     (3*e2/8 + 3*e4/32 + 45*e6/1024)*sin(2*m(i).projY(j))_
     + (15*e4/256 + 45*e6/1024)*sin(4*m(i).projY(j)) - (35*e6/3072)*sin(6*m(i).projY(j)))
   m(i).projX(j) = 0
   m(i).projY(j) = k0*(Mp - Mo)
   m(i).projX(j) = m(i).projX(j)*myUnit.conversionFactor + falseE
   m(i).projY(j) = m(i).projY(j)*myUnit.conversionFactor + falseN
  end if
 next
next
```

Figure 4.6b – The Transverse Mercator Forward Equation for Spheroids, and the corresponding REALbasic code in Mapalester. Note that the formula does not take into consideration false easting and northings, but the code does, for the purposes of support for the UTM coordinate systems.

Note that the layout of the formula in REALbasic differs somewhat from that presented by Snyder. Because these functions, in the case of a large polygon file, for instance, are run on millions of points in sequence, speed of operation is an essential

consideration while programming. Consequently, in keeping with the "speed is harder than function" experience of this software engineering project, getting the projection function working correctly usually is less difficult than getting the projection function working at its maximum possible efficiency. One approach that I often used to speed up the function was to pre-calculate all of the constants for the projection of any given map layer prior to projecting all the vector objects in that map layer. The main goal is to take as much out of the iterative loop as possible. In non-jargon terms, the goal is to only calculate once the portions of the formula that need to be calculated only once. The projection functions, as presented by Synder, frequently calculate these portions of the formula with each point conversion.

One especially difficult aspect of projection function programming is the handling of the special case coordinate input. Many of the projection conversion functions make heavy use of trigonometric functions that have special or undefined values with certain inputs. For example, the tan function used in figure 4.6b means that Mapalester must handle the cases when the input angular values are either $\pi/2$ or $-\pi/2$. If I do not, the entire function will fail. In this case, I have chosen to make any input that is $\pi/2$ or $-\pi/2$ the closest number possible to that value by adding or subtracting the smallest number recognizable by a standard 32-bit personal computer (0.000000000000001).

4.7 A Small Subset of Other Planned Features

The features and functionality identified in the previous parts of section four of are only a subset of the features and functionality that have been currently developed and are an even smaller subset of the features that will appear in the final version of Mapalester. Moreover, the features and functionality identified above are in large part aimed at the "powerful" development goal identified in the introduction. In this final part, I will describe a series of features and functionality that will appear in the final version of Mapalester, but for which I have done little implementation work thus far. I will also pay careful attention to identifying the reasons behind including these features the context of the target markets and development goals identified in the introduction.

4.7.1 Incorporation of the National Map Application Programming Interface (API)

One of the largest challenges to developing a free GIS software program is to find data to distribute with the program. Even with the limited amount of free data available, there are all sorts of data distribution issues. In order to fully meet my development goals of producing a free (goal four) and powerful (goal one) GIS software application, it is essential that I provide a substantial amount of free data with the GIS. The easiest way to provide that data is to implement the National Map API. I discovered the National Map API during a presentation by the United States Geologic Survey (USGS) on its National Map web depository of spatial data. Implementation of the National Map API will allow Mapalester users access to all data inside the National Map from within Mapalester's interface; to the user, the data may as well reside on the user's hard drive. The data in the National Map ranges from geological to hydrological to demographic to transportation to administrative.

4.7.2 Providing Easy Access to Census Data

For the same reason that I have chosen to implement the National Map API, I have also made making U.S. census data easily available a high priority feature. It has been my experience as a GIS teaching assistant, and my perception from the GIS applications literature, that some of the desirable data for non-profit and K-12 education GIS applications are U.S. census data. My goal for Mapalester is to make accessing census data as easy as possible. The idea is that users should never have to visit the U.S. census website itself; they should be able to get all the census data they need from within Mapalester's interface.

4.7.3 Data File-Sharing Using the Gnutella Network

In order to make even more free data available to Mapalester users, I hope to implement a version of the Gnutella network within Mapalester. Through the network, Mapalester users would be able to share the data sets that they create with other users around the world. For instance, I would be able to share the point shapefiles of Christian

contemporary music concert locations that I created from primary sources for a recent research project on the geography of Christian contemporary music, as well as the point shapefiles of Bruce Springsteen concert locations I created for fun. Similarly, students in a middle school science class could share GPS points that they collected along with associated attribute data.

4.7.4 Internationalization

As I have programmed Mapalester, I have been careful to lay down the framework to make it as easy as possible to produce versions of Mapalester in languages other than English. I have done this by making heavy use of constants instead of directly programming English language phrases into the source code. (A constant is essentially a variable that can change its value based on the language of the operating system on which Mapalester is running.) This will allow me to send someone a spreadsheet of all the words and phrases that I need translated, have them translate that spreadsheet, and directly input the translated phrases into Mapalester. In other words, once I finish the English version of Mapalester, all I will need is a translator for any given language to make a version of Mapalester for that language. Hillegass (2002) suggests that software with an international audience be translated into at least English, French, Spanish, German, Dutch, Italian, and Japanese. I also hope to produce versions in Mandarin, Cantonese, Taiwanese, Russian, and Serbian.

Support for multiple languages certainly bolsters the ease-of-use development goal and all of its benefits to Mapalester's target markets. In addition, the distribution of Mapalester in multiple languages will not only benefit Mapalester's target markets, but it will also *grow* the markets. Whereas the English version of Mapalester can only appeal to the three target markets in English-speaking countries, support for other languages will make Mapalester accessible to the target markets in other countries.

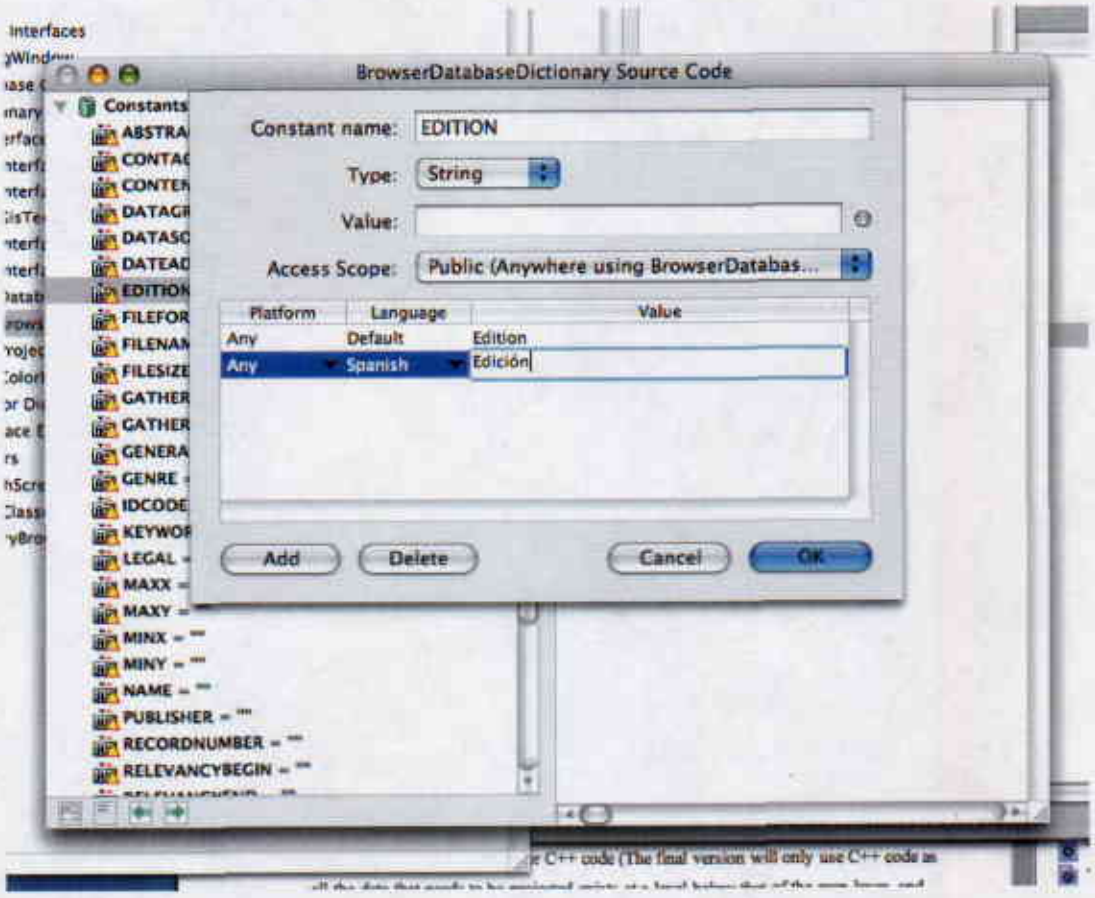


Figure 4.7.4a – A screenshot of the REALbasic constants interface shows how simple it is to add new languages to Mapalester. If the operating system on which Mapalester is loaded is in Spanish mode, Mapalester will display "Edición" instead of "Edition" in the Library Info Browser window.

5.1 Conclusion

After nine months of attempting a project so challenging in both scale and difficulty, I feel that I can conclude this paper with a reasonable amount of pride. Granted, I did not meet the project's original goal of releasing a product by this time. However, I believe I have made more than satisfactory progress towards doing so given the great number of unexpected challenges I have encountered along the way. I feel confident that the project is in a state where it can be finished in a moderately small amount of time. In addition, presenting the software and discussing it with members of the target market have led me to believe that my hypotheses about the unmet needs of the

target markets are true. I already have a small number of people eagerly awaiting the finished product.

On an individual level, the diverse array of knowledge and skills I gained during this project will prove useful as I continue my career in GIScience. I know understand the concepts behind coordinate systems well beyond the functional knowledge I had before. I can write code in C++. I understand database systems and SQL. I know the shapefile format and the reasons behind ESRI defining this way. My knowledge of DFLs and BNFs, learned in the most theoretical of computer science classes, came into practical use. I am now familiar with the trials and travails of computer software engineering, not just computer science. The list goes on. This project has been a grand adventure into both of my majors and has been a fitting and synthesizing finale to my bifocal college career.

I find myself at this juncture very excited about the possibility of other students heading down the same path. I believe that my progress in an attempt to recreate a very expensive package of commercial software is evidence that, with a couple of modifications, a similar effort in the future could be successful in the time allotted for this project. This is a significant statement as I am essentially arguing that students with a computer science background have the empowerment potential of producing free and powerful software for groups with needs left unmet by the commercial market. In the subsequent few paragraphs, I will identify and explicate the aforementioned changes that I think will make similar projects more "profitable" in the future. These paragraphs can also be read as funding recommendations for Keck grants and other summer funding

My strongest recommendation for engaging in a project like my own is to make sure that there is more than one student working on the project. If the project is of the same scale as Mapalester, two students should be sufficient, provided they are willing to put in the time and effort necessary. With two students on the job, a software project like Mapalester would probably make it more than twice as far, as collaborative computer science projects tend to get past frustrating and work-stopping moments better than solo projects.

The second recommendation is to make sure that there is at least a semi-expert in the theories and techniques necessary for *making* the software, not just *using* the

software. In my case, I was expecting to make it past the programming of the basics of GIS much faster than I did, and then rely on Dr. Laura Smith for her knowledge of spatial statistics to implement higher-level GIS functions. Ideally, it would have been helpful to also have a professor proficient in the computer science elements of basic GIScience to get me through the trouble spots.

Finally, students who wish to engage in similar projects should, if possible, enroll in at least an intro software engineering course at one of the ACTC schools. I had something similar to this course during my time at UC San Diego, and I found elements of it to be very helpful. I am sure further education on the engineering aspects of computer science would have provided even more insight into how to approach the problem of a massive programming project.

5.2 Acknowledgements

I would like to thank the following people for their help on my project: Dr. Laura Smith, for her advising on both a professional and personal level; Jovana Trkulja, Paul Singh, and Cole Akeson, for keeping me sane throughout my exciting, productive, and at times frustrating summer on the Keck Grant; the Keck-Bigelow Foundation, for funding much of my research; and Professor Carol Gersmehl, for providing me the GIS skills and passion necessary to engage a project like Mapalester.

6.1 Bibliography

Bachmann, Erik. Xbase File Format Description. 2000 Available from http://www.pgts.com.au/download/public/xbase.htm (last accessed 1/27 2005).

Bachorski, Andy, Andy Fuchs, Bill Mounce, Brian Blood, and et. al. 2003. *VALENTINA Database Kernel.* Paradigma Software, .

———2003. *VALENTINA for REALbasic Reference.* Paradigma Software, .

———2003. *VALENTINA SQL.* Paradigma Software, .

———2000. *VALENTINA for REALbasic Tutorial.* Paradigma Software, .

Baker, Thomas R., and Sarah W. Bednarz. 2004. Lessons Learned from Reviewing Research in GIS Education. *Journal of Geography* 102:231.

Beckmann, Norbert, Hans-Peter Kriegel, Ralf Schneider, and Berhard Seeger. 1990. The R*-Tree: An Efficient and Robust Access Method for Points and rectangles. *SIGMOD Conference* 1:322.

Brandt, Dave. 2004. *REALbasic Language Reference.* Austin, TX: REAL Software, In.c.

Brandt, David. 2004. *REALbasic User's Guide.* Austin, TX: REAL Software, Inc.

CMAP. CMAP Case Study: Robin Hood Foundation. Available from http://www.cmap.nypirg.org/case_studies/CS2/default.asp (last accessed 3/31 2005).

Dalrymple, Jim. 2005. Apple desktop market share on the rise; will the Mac mini, iPod help? *MacCentral.*

Department of Defense. 1984. *World Geodetic System 1984: Its Definition and Relationships with Local Geodetic Systems.*Report Number, NIMA TR8350.2.

Elwood, Sarah. 2002. GIS use in community planning: a multidimensional analysis of empowerment.

ESRI. 2000. *HowTo: Create projection metadata (.prj) files for shapefiles.* Redlands, CA: ESRI, Report Number, 14056.

———1998. ESRI Shapefile Technical Description: An ESRI White Paper.

———*ArcGIS 8: Supported Coordinate Systems and Geographic Transformations.* Redlands, CA: ESRI, .

Estier, Theodore. About BNF Notation. In University of Geneva [database online]. Available from http://cui.unige.ch/db-research/Enseignement/analyseinfo/AboutBNF.html (last accessed 1/31 2005).

Gewin, Virginia. 2004. Mapping Opportunities. *Nature* 427:376.

Guttman, Antonin. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. *SIGMOD Conference* 1:47.

Hecht, Brent. 2004. *Classifying GIS Polyline Data Using Neural Networks.*

———2004. *Mapalester, Empowerment, and Radical Democratic Citizenship.*

Hecht, Brent, and Ben Johson. 2003. Final Project for CS325 - Compilers: A Compiler for Cabal.

Hershberger, John, and Jack Snoeyink. 1992. An O(n*log(n)) Implementation of the Douglas-Peucker Algorithm for Line Simplification. *Proceedings of the 5th International Symposium on Spatial Data Handling* 1:134.

Hillegass, Aaron. 2002. *Cocoa Programming for Mac OS X*. Boston, MA: Addison Wesley.

Keohane, Georgia L., and CMAP. CMAP Case Study: Civic Builders. Available from http://www.cmap.nypirg.org/case_studies/CS1/default.asp (last accessed 3/31 2005).

Kerski, Joseph, J. 2004. Analyzing the Earth With Geographic Information Systems. *National Speleological Society News.*

Leeser, Miriam. Variants on the K-Means Algorithm. 1999 Available from http://www.ece.neu.edu/groups/rpl/projects/kmeans/variants.html (last accessed 1/10/2005 2005).

McGrew, J. C., and Charles B. Monroe. 2000. *An Introduction to Statistical Problem Solving in Geography*. United States of America: McGraw-Hill.

McMaster, Robert B., and K. S. Shea. 1992. *Generalization in Digital Cartography*. Washington, D.C.: Association of American Geographers.

Mikalajunas, Peter. 1998. DBF File Structure.

Neuburg, Matt. 2001. *REALbasic: The Definitive Guide*. Sebastopol, CA: O'Reilly.

O'Rourke, Joseph. 1998. *Computational Geometry in C*. Cambridge, UK: Cambridge University Press.

REAL Software. REALbasic vs. Java. In REAL Software, Inc. [database online]. Available from http://www.realsoftware.com/realbasic/compare/java/ (last accessed 3/31 2005).

Russell, Stuart, and Peter Norvig. 2003. *Artificial Intelligence: A Modern Approach*. Upper Saddle River, NJ: Prentice Hall.

Snyder, John P. 1987. *Map Projections - A Working Manual*. Washington, D.C.: United States Government Printing Office.

State Cartographer's Office. 1998. *Wisconsin Mapping Bulletin*.Report Number, 24.

Sunday, Dan. Polyline Simplification. 2002 Available from http://geometryalgorithms.com/Archive/algorithm_0205/algorithm_0205.htm (last accessed 6/01 2004).

Theodoridis, Yannis. R-tree-Portal. 1/26/2005 Available from http://www.rtreeportal.org/ (last accessed 9/15 2004).

U.S. Census Bureau Geography Division, Cartographic Operations Branch. Scale, Generalization, and Limitations of the Cartographic Boundary Files. In U.S. Census [database online]. 2004 Available from http://www.census.gov/geo/www/cob/scale.html (last accessed 3/25/2005 2005).

United States Geological Survey. 2003. *Implementation Plan for The National Map.* Washington, D.C.: United States Departemtn of the Interior, Report Number, 1.0.

Wilder, Anna, Jonathan D. Brinkerhoff, and Teresa M. Higgins. 2004. Geographic Information Technologies + Project-Based Science: Contextualized Professional Development Approach. *Journal of Geography* 102:255.

Wise, Stephen. 2002. *GIS Basics.* London, England: Taylor & Francis.