SITE CONTROLLER: A system for computer-aided civil engineering and construction.

by

Philip Greenspun

Abstract

A revolution in earthmoving, a $100 billion industry, can be achieved with three components: the GPS location system, sensors and computers in earthmoving vehicles, and SITE CONTROLLER, a central computer system that maintains design data and directs operations. The first two components are widely available; I built SITE CONTROLLER to complete the triangle and describe it here.

Civil engineering challenges computer scientists in the following areas: computational geometry, large spatial databases, floating-point artihmetic, software reliability, management of complexity, and real-time control. SITE CONTROLLER demonstrates that most of these challenges may be surmounted by the use of state-of-the-art algorithms, object databases, software development tools, and code-generation techniques. The system works well enough that Caterpillar was able to use SITE CONTROLLER to supervise operations of a 160-ton autonomous truck.

SITE CONTROLLER assists civil engineers in the design, estimation, and construction of earthworks, including hazardous waste site remediation. The core of SITE CONTROLLER is a site modelling system that represents existing and prospective terrain shapes, road and property boundaries, hydrology, important features such as trees, utility lines, and general user notations. Around this core are analysis, simulation, and vehicle control tools. Integrating these modules into one program enables civil engineers and contractors to use a single interface and database throughout the life of a project.

This area is exciting because so much of the infrastructure is in place. A small effort by computer scientists could cut the cost of earthmoving in half, enabling poor countries to build roads and rich countries to clean up hazardous waste.

This report is a revised version of a thesis submitted to the Department of Electrical Engineering and Computer Science in February, 1993, in partial fulfillment of the requirements for the degree of Master of Science.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for this research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-92-J-4097 and by the National Science Foundation under grant number MIP-9001651.

In Memory of George, 1983-1991.

Acknowledgments

M.I.T. in general and the A.I. Lab in particular are remarkably supportive environments. When I started this project, I was wandering around the 7th floor looking for a copy of an old SIGGRAPH proceedings containing an article on triangulation. A stranger asked me why I needed it and I explained that I wanted to write terrain modelling software using triangulated surveyed points. He turned out to be Jim Little, who'd solved this very problem in the 1970's, and his guidance in the months that followed saved me weeks in the library.

Greg Baecher and Dwight Sangrey lent me their vision of the future of civil engineering. Joel Peterson and the rest of the folks at Caterpillar gave me faith that industry can innovate and that useful work can be done in 47°C (116°F) heat.

My advisor and friend Gerry Sussman demonstrated that one can reach Nerdvana by inquiring for one's entire life. Andy Berlin, always my idea of the model graduate student, sagaciously prevented me from spending the rest of *my* life on this particular report. Henry Wu taught me how to build real circuits. Kleanthes Koniaris suggested changes that would have substantially spiced up the text—I only wish Bill Clinton had taken office earlier so that I might have adopted more of his suggestions without fear of prosecution. The rest of the 4th floor provided me with friendship, entertainment, and assistance of every kind.

Fairness compels me to acknowledge above all the cruel discipline of the marketplace. For without it, I might not have realized my *métier* as a graduate student.

Table of Contents

1. What is SITE CONTROLLER?

Advances in computer systems, computational geometry algorithms, and location systems have made computer-aided earthmoving feasible. Compared with traditional techniques, computer control should cut the cost of earthmoving by 50-75%. The three principal components required to effect a revolution in earthmoving are the following: a central computer system that maintains design data and directs operations; a location system that can determine the position of earthmoving vehicles in real time; and on-vehicle computers that assist the operator and/or directly control implements.

Location systems and on-vehicle computers are available today. I built SITE CONTROLLER, an example of the critical missing component, in order to underline the fact that computer-aided earthmoving is feasible today. SITE CONTROLLER is a Common Lisp program that assists civil engineers in the design, estimation, and construction phases of earthmoving projects. The core of SITE CONTROLLER is a site modelling system that represents the following: existing and prospective terrain shapes; road and property boundaries; hydrology; important features such as trees and utility lines; and general user notations. Around this core are analysis, simulation, and vehicle control tools.

Terrain is represented by surfaces interpolated through triangulated sets of points of known elevation. The most basic surface model simply postulates a plane for each triangle, resulting in a faceted surface like that of a cut gemstone. When smooth contour lines are desired, a C^1 surface model is computed by making each triangle a nine-parameter polynomial surface and matching up first derivatives at triangle borders. Existing topography can be entered in the form of surveyed points while prospective topography is captured from user-sketched contour lines.

Analysis software consists primarily of procedures that calculate the interaction between existing and prospective surfaces: Where would one have to excavate and fill to achieve a specified roadbed? How much earth would have to be trucked onto the site to build this foundation? What would the site look like after certain design ideas have been adopted?

Simulation software allows the user to plan earthmoving activities, then watch them progress in simulation and see the evolution of terrain shape. This provides a basis for accurate estimation of construction costs. Hand-in-glove with the simulation software is real-time vehicle communication and control. Site model information to aid equipment operators is sent over radio modems while vehicle location and operation information (e.g., engine temperature) comes back. SITE CONTROLLER keeps a record of activity so that any particular operation may be replayed on-screen in the future.

My experience building and using SITE CONTROLLER provides valuable lessons for the future:

- a substantial percentage of bugs may be traced to the use of floating-point arithmetic and untyped geometric operations. Reliable systems will require formal methods of error control or exact arithmetic and modern approaches to geometric programming.
- high-level programming languages, such as Lisp, and the best available software development tools allow a practical system to be built with only moderate effort.
- reasonably intelligent software decomposition with a modern object system provides enough flexibility for extensions into unforeseen areas of civil engineering.
- state-of-the-art computational geometry algorithms are barely adequate for coping with the constantly evolving surfaces created by operating bulldozers. I present one solution to the problem of representing surfaces cut by bulldozers.
- comprehensive site models are the key to making a useful system and one must be careful to avoid the pitfalls of computer-aided design (CAD) systems where raw geometry is used to represent real-world objects that have important non-geometrical characteristics.
- integrated computer-aided civil engineering systems will strain object database systems to their limits in the areas of performance, version control, synchronization and integrity.

Civil engineering and construction are enormous areas of human endeavor that have remained almost completely unsupported by computers. SITE CONTROLLER proves that a revolution affecting hundreds of billions of dollars of projects might be achieved by a small group of computer scientists.

2. SITE CONTROLLER and a Typical Project

This section shows how SITE CONTROLLER is useful in all phases of an earthmoving project.

2.1. Describing the Virgin Site

Rebecca, a civil engineer, sits down at a computer running SITE CONTROLLER. She is then presented with a collection of non-overlapping windows that cover the screen. Moving the mouse over the largest window, she clicks to get help and up pops a menu. This menu describes all the commands accessible in that window. A novice would choose commands from the menu, but an expert like Rebecca can invoke any command directly by holding down the correct shift keys and mouse buttons.

Rebecca clicks the mouse to "choose a site to edit in this view" and a menu pops up with the names of all the sites plus an option to "define a new site," which she selects. A dialog box pops up. After filling in values such as the site name and possibly overriding defaults for such parameters as contouring interval, Rebecca is ready to use the new site.

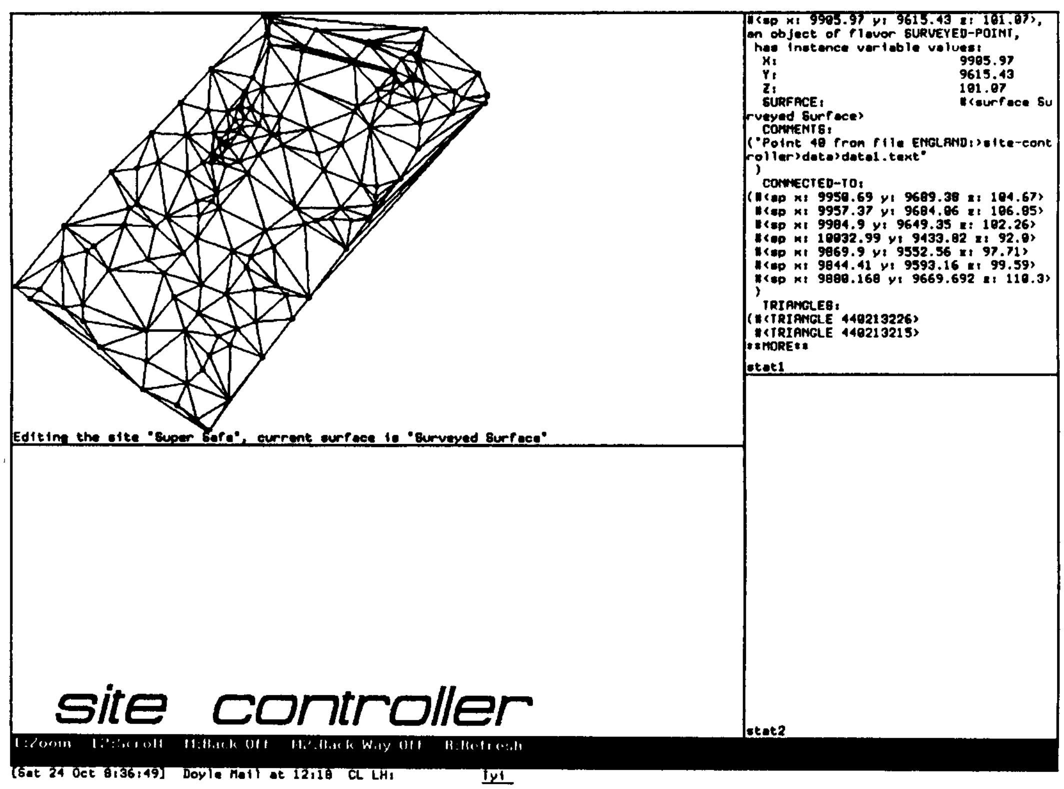

Figure 2.1 Typical SITE CONTROLLER screen showing surveyed points of known elevation connected in a Delaunay triangulation. The statistics window on the upper right shows detailed data about one surveyed point.

Rebecca already has an ASCII file of surface topography data, either from photogrammetry, satellite maps, or standard surveying. Another mouse command turns that ASCII file into a list of SURVEYED-POINTs in the new site's default SURFACE, *Surveyed Surface*[1]. SITE CONTROLLER now knows the elevation of *Surveyed Surface* at, say, 150 points. Rebecca has an old contour map of the area and wants to make sure that there are no glaring errors in her data. So she clicks the mouse to see a contour map of the surface, i.e., the intersections of the surface with regularly-spaced planes of constant elevation.

Intersections between *Surveyed Surface* and horizontal planes will occur at arbitrary locations, yet SITE CONTROLLER only knows the elevation of the surface at 150 discrete locations. Thus, some kind of continuous surface must be interpolated through those points. SITE CONTROLLER constructs the Delaunay triangulation of the 150 surveyed points (see Figure 2.1), a tiling of the site with triangles such that the vertex of each triangle is a point of known elevation. The Delaunay triangulation is notable because it results in triangles that are close to equilateral (as opposed to long and skinny). Since three points determine a plane, a natural piecewise-planar faceted surface model springs immediately from the triangulation. This kind of C^0 surface would look somewhat like a cut diamond, with sharp edges between facets.

Rebecca can now see that the contours do not match those in her existing map, and that, in particular, there is a 500 meter high, 2 meter wide tower in the middle of the site. A click of the mouse suffices to superimpose the contour map in blue and the Delaunay triangulation in red. Rebecca rolls the mouse around on the screen while little highlighting boxes appear around the nearest surveyed points. She positions the mouse so that the point that appears to be the summit of the peak is highlighted and clicks to describe it. In one of the auxiliary statistics windows, the point prints out its (x,y,z) value and provenance. Rebecca notes that the elevation of this point is implausible and was likely due to a data entry error; a flourish of the mouse suffices to delete it from *Surveyed Surface* and the Delaunay triangulation is recalculated.

Although the new contour map lacks the extraneous 500 meter peak, Rebecca is unhappy with the appearance of the contours. Planar triangular facets intersect with horizontal planes in line segments. Thus, contour lines are not smooth but rather piecewise-linear with jagged corners at triangle boundaries. Although these are not necessarily less accurate than smooth lines, Rebecca is a traditionalist and has trouble visualizing the site.

Rebecca's next mouse click asks SITE CONTROLLER for smooth contouring. Each triangle is now a nine-parameter polynomial surface, with first derivatives continuous across triangle borders. These polynomial surfaces intersect with horizontal planes in smooth curves and Rebecca is happy.

Wishing to obtain design assistance from SITE CONTROLLER in the future, Rebecca decides to add objects to the site model. Using a digitizing tablet to trace over some existing blueprints, she defines roads and enters data about their pavement width, right-of-way width, centerline path, and 3D profile. Rebecca traces over the property boundaries, buried gas and electric lines, and important site features such as buildings. SITE CONTROLLER represents these real-world

---

[1]Class names and other fragments of Lisp code such as procedure and message names are shown in upper case; instances of classes are shown in italics.

objects with corresponding objects in the site database. Site Controller can answer questions about gas lines because a gas line is represented as a full-fledged object known to be a hazard, not just as a few graphical entities on the screen with a human-readable label.

2.2. Studying Alternative Surfaces

Rebecca has been hired by Evergreen Development to figure out how to turn some nice rolling farmland (Bucolic Farm) into the maximum legal number of tract houses (Evergreen Estates). SITE CONTROLLER helps Rebecca to break up Bucolic Farm into lots that satisfy the zoning laws of all relevant political entities. In particular, SITE CONTROLLER is able to represent zoning criteria such as minimum frontage (along a road), minimum lot area, how far back septic drainage fields must be set from the property boundaries, and whether portions of the lot that are steeply sloped may be included in the minimum area. Once a general layout is complete, Rebecca turns her attention to sculpting the terrain.

Each structure must sit on a flat "building pad" and Rebecca clicks the mouse to define a new prospective surface for the first house lot (*Lot 1 Surface*). Stretching a boundary polygon around the area to be reshaped, she sketches in new contour lines and surveyed points. After the contours for the lot look good, she instructs SITE CONTROLLER to combine the real *Surveyed Surface* with her prospective *Lot 1 Surface*. SITE CONTROLLER does this by "sewing together" the two surfaces at the boundary polygon, preventing any elevation information from being interpolated using data from the other side of the boundary. All the original surveyed data within *Lot 1 Surface*'s boundary polygon is ignored. Rebecca asks for a contour map of the new combined surface and satisfies herself with the soundness of the result. Rebecca proceeds in this manner to sketch in surfaces for more houses, roads, and possibly common parks.

After Rebecca has finished a *Final Surface* consisting of the original *Surveyed Surface* combined with dozens of prospectives, Evergreen asks her to estimate the earthmoving cost. SITE CONTROLLER is able to make a map of the cut-and-fill required to realize *Final* when starting with *Surveyed*, i.e., each area that must be excavated is highlighted in one color while areas that must be filled in are shown in another. In addition, SITE CONTROLLER automatically calculates the amount of earth that must be trucked on or off the site to realize *Final*. Finding this number to be excessive, Rebecca raises the tennis courts to use up excess earth excavated from the house lots.

However, this causes Miriam, Evergreen's architect, to worry that the buyers of Lot 37 will have an obscured view of a nearby lake. Rebecca asks SITE CONTROLLER to calculate intervisibility between the lake and a hypothetical second floor window in Lot 37. Learning that the view is in fact obscured, Rebecca raises the building pad for Lot 37 and Miriam becomes concerned that the terrain looks unnatural when viewed from Lot 21 and that this won't appeal to their granola-eating customer base. Rebecca says "look at the contour map—you can see the natural beauty of the tract." Unfortunately, Miriam isn't used to reading contour maps and can't visualize the site. SITE CONTROLLER obliges by generating a perspective view from the viewpoint of a first floor window on Lot 21 facing Lot 37. Miriam is now satisfied.

Hydrology questions are important at this point; i.e., where are the ridges, streams, drainage basins, and pools. Rebecca wants to know what will happen if there is a "100-year storm" (the worst one might expect in 100 years on this site). She finds that a stream cuts right across one of the roads and concludes, by looking at some of the ridge lines, that the stream drains a large basin. A severe storm will drop a substantial quantity of water in that basin and the stream will therefore have enough power to wash out the road. Rebecca adds roadside drainage ditches and a culvert (a pipe underneath the road to let the stream flow through).[2]

2.3. Simulating Construction

Once the design objectives are clear and agreed upon, Rebecca's engineering task is complete and she hands off the design to Rachel, a local contractor. Rachel uses the same site database to plan the earthmoving job of turning Bucolic Farm into Evergreen Estates. She uses SITE CONTROLLER's simulator to specify the paths of two bulldozers moving over the site, cutting *Surveyed Surface* to the elevation of *Final Surface*. The simulator shows Rachel the bulldozers moving and the evolution of the surface as the bulldozer blades carve. Rachel can use the simulator to plan construction so that machines are utilized efficiently and the entire job is done at a reasonable cost on a schedule acceptable to Evergreen.[3]

2.4. Controlling Construction

Evergreen accepts Rachel's bid and work commences. Rachel installs SITE CONTROLLER on a computer inside the construction site's trailer and attaches radio modems to the computer's serial port. Rachel has fitted her earthmovers with Global Positioning System (GPS) navigation systems, implement position sensors and, in some cases, computer interfaces to the hydraulic systems that position the implement.

Because Rachel has a stationary GPS system in a fixed known location on the site, the mobile GPS receivers are correctable to an accuracy of a few millimeters. While the earthmovers are operating, they keep SITE CONTROLLER apprised of their position and status by sending packets back via the radio modems. SITE CONTROLLER in turn pulls relevant data from the site model and sends it back to the earthmover. For example, as a backhoe operator's blade gets close to a gas line, a warning is automatically sent out (if the vehicle were appropriately equipped, data from the site model would restrict the range of the backhoe's motion so that contact with the gas line was impossible). Bulldozers roughing out sections of the site are sent data about the design surface so the operator may be informed that "the blade is now 2 feet above the design surface." Motor graders finishing a layer of gravel on a roadbed are completely controlled by SITE CONTROLLER working with on-board computers—blade height control is taken out of the operator's hands, enabling the vehicle to be driven faster and by less skilled operators.

In fact, servo control of implements enables Rachel to perform precision work with cheap machines such as bulldozers when formerly expensive graders were required. A motor grader is

[2] This portion of Site Controller needs substantial additional development.

[3] This portion of Site Controller needs some rethinking before it will be useful.

not 35 meters long because that is the minimum size necessary to carry the blade, but because a human operator is incapable of working smoothly without a long baseline. However, a computer can rapidly compensate for a vehicle's roll or pitch and therefore a hyperstable platform may be superfluous.

2.5. Litigation (the Way of All Business)

A year after the yuppies move into Evergreen Estates, a portion of the main road crumbles. Evergreen sues Rachel, alleging that she inadequately compacted the roadbed before paving. At the trial, Rachel brings her laptop running SITE CONTROLLER into the courtroom. The jury sees that SITE CONTROLLER is able to reproduce from its database the state of the Bucolic/Evergreen project at any time during construction. Rachel moves the playback history date up to the day when the roadbed was compacted and the jury sees the compactor rolling back and forth the 10 times specified by Rebecca's engineering plans. Rachel is off the hook. Of course, Rebecca is now sued for specifying an inadequate degree of compaction... .

Twenty years after Evergreen Estates are finished, occupants of adjacent Hidden Valley Mansions sue Rebecca, Rachel, and Evergreen when their doctors find elevated levels of cadmium among Hidden Valley residents. The lawsuit alleges that 1) the farmer on Bucolic Farms was a gadget freak who tossed hundreds of dead NiCd batteries behind his barn, 2) Rachel used the earth containing the batteries to fill in the Evergreen Estates baseball diamond, and 3) Rebecca's plan allowed runoff from the baseball diamond to flow into Hidden Valley. Rachel uses SITE CONTROLLER's historical database to show that earth excavated from behind said barn was used for fill at the tennis courts and further uses SITE CONTROLLER's hydrology analysis tools to show that no runoff from the baseball diamond or the tennis courts could flow into Hidden Valley. Case dismissed.

3. Why Earthmoving?

Construction is the largest industry in most countries, even in highly developed regions such as Europe, North America, and Japan. Indeed, the products of the construction industry determine whether or not a country is considered “developed” or “developing.” Third World countries may contain regions of great productivity or great beauty, but without roads to transport goods and people those resources are useless. By contrast, countries not blessed by Nature may make up for deficiencies with construction. Portions of Europe blighted by bad weather and uninteresting topography have been rendered charming by structures built over the centuries. Similarly, regardless of what resources are inherent in the land, high productivity can be achieved with roads, power plants, high-speed rail links, and modern factories.

Although construction in the United States is not typical of the rest of the world, it is well characterized. New construction in the U.S. was worth $446 billion in 1990, or 8% of GNP. Adding the value of alterations, maintenance, and “unspecified” work, the total is nearer $700 billion, or 12% of GNP [U.S. Department of Commerce 1991; Bureau of the Census 1987]. U.S. construction directly employs over 5 million people, or about 5% of the labor force—without including people employed by material suppliers and equipment manufacturers [U.S. Department of Labor 1989]. This fraction of the labor force has been more or less constant since 1920, when statistics were first calculated.

Since the mid-1960’s, both industry productivity and size relative to the economy have been declining. The value of new construction fell from 11.9% of GNP in 1966 to 8.1% in 1990. Productivity has similarly declined by at least 33% over the same period [Demsetz 1989]. A variety of factors have contributed to the decline in importance and effectiveness of construction in the U.S.:

- Many of the things worth building have been built. The first bridge over a river is built in the easiest location and has the most value to people formerly forced to take ferries. Subsequent bridges provide some convenience and congestion relief, but on average cost more and are worth less. Similarly, most of the best sources of hydroelectric power in the U.S. have been tapped.
- People purchase less of goods and services that represent poor value. When car radios and mobile phones were expensive and highway construction cheap, people spent tax dollars on highways. Today, people may think it wiser to buy a Sony car stereo and cellular phone that will make traffic jams endurable than to pay an inefficient government/industry coalition to build a new highway.
- As an ever-greater portion of the nation is covered with buildings, roads, and factories, a growing percentage of construction is renovation and redevelopment, which may be more labor-intensive than new construction.
- Techniques have stagnated, yet wages rise. Since productivity is the amount of output per labor dollar, people who produce no more per hour but get paid more per hour will be called “less productive.” U.S. firms invest only about 0.4% of revenues in R&D and thus innovations must be imported from Europe and Japan, where 1% of revenues are so invested.

Construction in the rest of the world differs in several consistent respects. In the Third World, a greater proportion of GNP is spent on construction and much of that is spent on traditional earthmoving projects. Because Third World construction still uses a lot of manual labor, productivity is rising as more machines are introduced. Europe and Japan differ from the U.S. primarily in that their construction firms invest in R&D and are constantly developing new techniques that increase productivity.

When tackling the problem of improving construction productivity, earthmoving is a natural place to look for automation opportunities. All kinds of earthmoving jobs are essentially similar, whether hazardous waste is being excavated or a roadbed being laid. This contrasts with building construction where techniques used to build a steel-and-glass office tower may be of no use when constructing a brick apartment house. Earthmoving represents a sizable percentage of the entire construction industry, from about 10% in the U.S. to perhaps 30% in the Third World. Furthermore, all the building blocks that make computer-aided earthmoving possible have become available recently. These include the following:

- accurate rugged inexpensive location systems, notably the Global Positioning System of satellites
- computational geometry algorithms capable of efficiently handling problems as the amount of data gets large enough to accurately model real sites
- object data management systems that are orders of magnitude faster at handling the kind of data that arise in civil engineering than are traditional relational databases
- inexpensive powerful computers.

Before diving into the specifics of applying new technologies, let's step back for a minute and consider which earthmoving problems are considered inadequately addressed by current technology.

Hazardous waste site remediation is one of the few civil engineering problems that has attracted the attention of the American public. Respondents to a Louis Harris poll rated a clean environment more important than a satisfactory sex life [Stutz 1990].[4] In the United States alone, there are over 1,200 Superfund sites and cost estimates for remediating those sites range from 500 to 1,200 billion dollars. More sites are added to the Superfund list each year and yet, since the program's inception in 1980, only 33 sites have been fully cleaned up at a cost of over $10 billion. Superfund sites represent only a fraction of the contaminated earth in the U.S. and thus it should be clear that even a rich society such as the U.S. cannot afford to clean up all of its hazardous waste with existing techniques.

Cleaning up a hazardous waste site consists essentially of surveying the site, developing a plan for excavation, excavating the contaminated earth and moving it to a disposal or incineration site, and finally, somehow restoring the original site to a usable condition. Many of the steps involved in cleanup are the same as for a traditional earthmoving job, except that record-

---

[4] *I can remember when the air was clean and sex was dirty.* — George Burns

keeping requirements are more stringent and workers must be protected from hazardous materials.

Another glaring problem, albeit one that overlaps somewhat with hazardous waste remediation, is that of equipment utilization. Ask most people to visualize a construction site and their first response is "I see a bunch of enormous yellow machines sitting idle while a few workers stand around either making measurements or staring at blueprints." Given that each yellow machine can cost nearly one million dollars and is only used about one third of the time on a typical construction job, it is natural to think that if one could keep the equipment running constantly the job could be completed faster and cheaper. Machine operators often must wait for hours while surveyors study blueprints, make measurements and finally drive stakes into the ground to show the operators where and how much to cut. The key to moving earth productively is to use location and control systems that free equipment operators from reliance on surveyors and blueprints. This is well understood by mine operators, who strive for, and regularly achieve, over 90% equipment utilization, 24 hours a day.

That equipment productivity is the most important goal is thrown into sharp focus by Third World projects. Labor is nearly free when compared with a $500,000 bulldozer or $700,000 truck, so the cost of the project is entirely determined by the number of days on which machines are rented and the cost of maintaining those machines.

One of the biggest inadequately solved problems is one that we have created ourselves: compliance with legal and regulatory demands. If one wanted to start an automobile company in turn-of-the-century America, all one needed to do was design a saleable product. Today, one would first have to hire dozens of lawyers to work for a year just to read all the relevant laws and regulations. Cars are safer and cleaner today, but 19th century business practices no longer suffice. Litigation and regulation have similarly complicated earthmoving, yet business practices have not adapted nearly as well to the new environment. One result is that compliance with regulations is enormously burdensome and therefore expensive. Another is that nearly every large construction job gives birth to at least one lawsuit.

Inaccurate estimates often give rise to lawsuits. If the estimate is too low, the contractor will lose money unless he somehow manages to foist the blame for the cost overrun on someone else. In fact, some contractors underbid jobs on purpose with the expectation that a fair price will be recovered in a subsequent lawsuit over undisclosed or unforeseen conditions on the site. This practice dates back at least to the mid-1800's and the enlargement of the Erie Canal: "It was a notorious fact that contractors did not scan very closely the character of the work they were bidding for, the main thing being to get the work, which according to law, must go to the lowest bidder. Political, and possibly corrupt influence, was firmly relied upon to get 'relief' from the Canal Board at Albany." [Ainsworth 1901] Computer-aided estimation and simulation available to both the client and contractor might make for a better understanding of the task by both parties, more realistic estimates, and hence less litigation.

Feeding the government paperwork monster is a task to which computers have proved equal in other fields. However, without complete information about a construction project from start to finish, such as where each bucketful of contaminated earth came from and is going to, it is impossible for computer systems to be of much use in satisfying regulatory filing requirements.

To summarize, I chose to work in computer-aided earthmoving because it is an area where a small amount of engineering effort can yield enormous dividends due to the primitive nature of current techniques and the new challenges that face society.

4. Pioneering Civil Engineering Software

One machine can do the work of fifty ordinary men. No machine can do the work of one extraordinary man. — Elbert Hubbard

Civil engineers have been extremely poorly served compared with other kinds of engineers by the computer software community. Factors contributing to the dearth of useful software include the following:

- Computers can do almost nothing to help construction without knowing where machines and materials are physically located, yet accurate, inexpensive location systems are barely available even today.
- Civil engineering is unstructured and requires much more flexible databases and software than other disciplines. On a construction site, nothing can be predicted; in a factory, everything can be controlled.
- Design civil engineers get paid by the hour and are therefore not necessarily hungry for productivity aids.
- Design civil engineers have little money and are not used to making capital investments, yet civil engineering problems require massive computational resources that were prohibitively expensive until recently.
- Efficient computational geometry algorithms were not available until the 1980's so that, even given massive computers, civil engineering problems would have been intractable with the algorithms of the day
- Civil engineers generally lack formal computer science backgrounds and are unable to write complex tools themselves. By contrast, the most sophisticated CAE (computer-aided engineering) programs to date were written by electrical engineers to solve integrated-circuit design problems.

Characterizing the smallest civil engineering site to a useful level of detail requires at least several megabytes of memory. Manipulating and displaying that data in real time requires a computer capable of at least 10 MIPS. Until the late 1980's, the smallest computers capable of being useful assistants to the civil engineer cost over $100,000. Thus, building design software for civil engineers would have been pointless because, even if one had given away software free, few users would have been able to afford the requisite hardware. Construction firms accustomed to investing heavily in capital equipment would not have found such software useful because location systems were too expensive and inaccurate.

Even in the 1990's, building a comprehensive solution to the problems of civil engineering design, earthmoving, and site maintenance will require the most advanced tools, the best programmers, the latest, most efficient algorithms. Such a system would likely be larger and more complex than the most comprehensive electrical and mechanical CAE systems.

Undaunted by these formidable challenges, a group at MIT in the 1960's developed the Integrated Civil Engineering System (ICES), the progenitor of all computer-aided civil engineering programs [Roos 1967]. ICES was a programming environment for solving civil engineering problems. Starting from the primitive foundation of FORTRAN running on the IBM System/360, ICES provided many of the features of modern programming environments.

A cornerstone of ICES is metalinguistic abstraction (see §5.1.4), i.e., defining "problem-oriented languages." The most famous ICES languages are COGO (COordinate GeOmetry), carried over from a 1960 effort on the IBM 1620, and STRUDL, a system for structural analysis. All of the problem-oriented languages were implemented in ICETRAN, an extension of FORTRAN that was compiled into FORTRAN by a preprocessor.

Dynamic linking, compound data structures, dynamic memory allocation, and pointers for linking data structures made ICETRAN programmers substantially more productive than their counterparts using FORTRAN and standard tools.

COGO was widely adopted for processing survey data by civil engineers working with computers. Even today, dozens of COGO implementations are available for personal computers. A more important legacy of ICES was the blossoming of software development groups inside large civil engineering firms during the 1970's. In-house programs were developed to facilitate land and building design, generate three-dimensional graphics, and perform all kinds of analysis.

Digital Equipment's VAX minicomputer was the first machine capable of supporting draftsmen and engineers interactively at a reasonable price. By the early 1980's, CAD firms such as Intergraph were selling products for mapping and earthworks design to anyone willing to invest $1 million. Unfortunately, such systems seldom proved economical. Clients were unwilling to pay an hourly supplement for a designer using a computer, the management-of-complexity problems did not justify a $1 million system, and the available programs were simply not much more powerful than manual techniques.

The 1990's should be the decade in which civil engineers are finally able to get meaningful assistance from computers, and SITE CONTROLLER is an example of the kind of software that I envision becoming commonplace by the end of the decade.

5. Selecting Software Development Tools

In building SITE CONTROLLER, I hoped to substantially improve on the state of the art in computer-aided earthmoving with only my own efforts over a period of a few months. It was therefore important to choose a productive software development environment, consisting of a programming language and tools such as compilers, linkers, and debuggers.

5.1. Language Requirements

I settled upon the following language requirements:

- automatic storage management
- modern object system with method combination and a class hierarchy that can be extended at run-time
- ability to write higher-order procedures
- ability to perform metalinguistic abstraction
- powerful arithmetic incorporating both a formal model of absolutely machine-independent generic arithmetic and support for exact arithmetic
- compact (few lines of code)

The next sections describe in detail each requirement.

5.1.1. Automatic Storage Management

Computer-aided design and engineering programs all have the characteristic that objects of unpredictable size are frequently created in response to user actions. With an integrated-circuit design program, the user might add a transistor or connection. With a mechanical design program, the user might add a lightening hole to a suspension arm. With a site modelling program, the user might add a simple blueprint annotation or something as complex as a scanned photograph. In all of these cases, the object must be stored until it is explicitly deleted by the user. It is possible for the programmer to manage the storage of these objects explicitly, allocating storage from a heap when the user creates an object and calling a deallocation procedure when the user deletes the object.

Much more difficult is the management of objects created as intermediate steps in geometric computation. For example, a database may store certain data in a singly-linked list for storage efficiency. However, it may be impossible to perform a certain kind of analysis efficiently without a doubly-linked list. In response to a user query, data is extracted from a portion of the database, stored in a doubly-linked list and manipulated by numerous procedures to obtain a result. A typical bug arises when a procedure deallocates the temporary data structure, but another procedure, possibly programmed by a different individual, still holds a pointer to that data structure and doesn't realize that the pointer is no longer valid. Meanwhile, the heap manager has allocated the storage space to another part of the program and two procedures are hammering away at the same memory locations, each unaware that the other has overwritten its data.

It is theoretically possible for programs to track every object they use and explicitly identify objects that are no longer needed. In practice, especially in the CAE world where complex systems and multiple cooperating programmers are the rule, programmers make mistakes and storage allocation bugs bedevil the system. Typical large CAE programs are hobbled by *ad hoc* storage management schemes that are both slow and imperfect. Users come to accept the fact that the program will crash occasionally and guard against it by storing multiple versions of the database on disk.

Automatic storage management works by giving the programmer the illusion that the computer's memory is infinite. There are procedures that allocate storage, but no deallocation procedures. Every so often the *garbage collector* looks through the memory, finds objects that can provably no longer be accessed, and makes the memory consumed by those objects available for reuse. For example, a temporary data structure that was available only as a local variable of a procedure that has terminated would be garbage collected because there is no pointer anywhere in the memory that refers to it. In theory, garbage collection slows down execution because the garbage collector is dumb, i.e., it knows and assumes nothing about the data. A programmer should be able to do a better job because he knows which variables store what—this is in fact the case for very simple programs. However, garbage collectors not only eliminate storage allocation bugs, but also make programs run faster for anything as complex as even the simplest CAE systems. Some programmers will argue the point, but it is important to keep in mind that compilers were considered inefficient by many programmers for decades—despite evidence that they did a better job of register allocation and translation into machine language than humans once a certain complexity was exceeded.

No comprehensive civil engineering CAE system has a chance of working reliably and satisfying regulatory and legal requirements if its programmers cannot take advantage of automatic storage management. However, standard implementations of most older computer languages preclude garbage collection. The simple fact of permitting pointers to be stored as ordinary integers prevents a garbage collector from knowing whether or not something is unused. (If there is a variable somewhere with the value 10247, does that mean that the object stored at memory location 10247 is not garbage, or that a loop just finished executing 10247 times?). Standard implementations of popular languages such as C, C++, and PASCAL all have this shortcoming. Modifications to the runtime system and the way in which data is stored can allow older languages to be garbage-collected with certain limitations (see [Weiser 1989], [Yip 1991], [Detlefs 1990]). Modern languages such as Lisp, Smalltalk, and MODULA-3 incorporate automatic storage management as a standard feature.

5.1.2. Object-Oriented Programming

A modern object-oriented programming system allows efficient organization of software by automatically combining pieces of code from different locations. A description of all the objects in a system may be factored into *classes* and any given object is an *instance* of a particular class. A typical object will be an instance of a class that inherits from many superclasses.

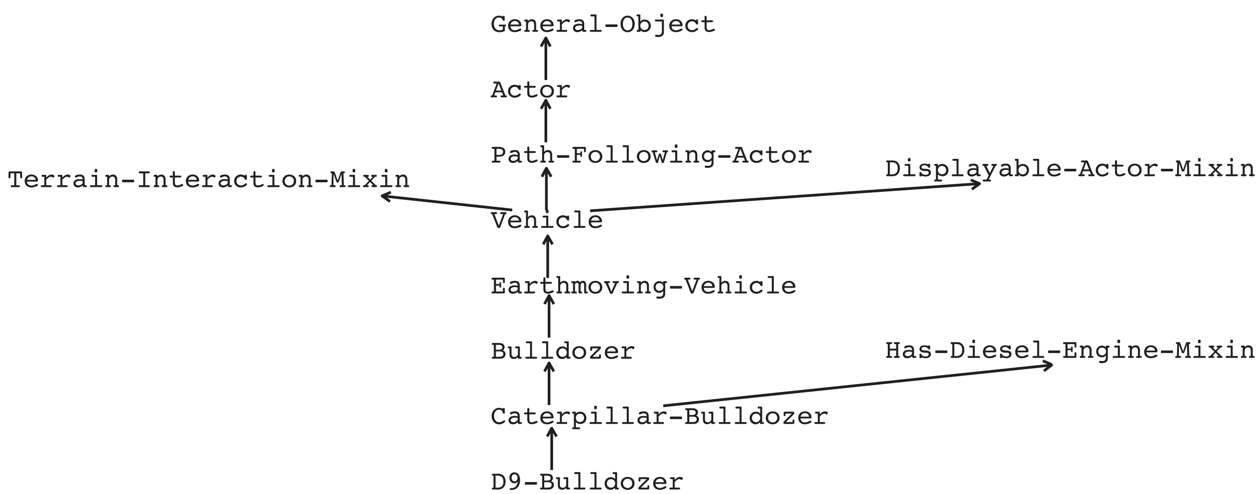

Figure 5.1 Sample class hierarchy for the representation of a Caterpillar D9 Bulldozer.

For example, a bulldozer might be represented as an instance of the class D9-BULLDOZER, which is a subclass of CATERPILLAR-BULLDOZER, which is itself a subclass of BULLDOZER.

Functions that are needed by several different kinds of objects need only be written, debugged, and modified in one place. For example, consider the object *Bulldozer W1FB*, an instance of the class D9-BULLDOZER. The message TOP-SPEED would be handled by a function associated with the class VEHICLE, using data supplied by the class D9-BULLDOZER.

Method combination is a feature that allows small pieces of code associated with different classes to be combined to form a single method of responding to a message. Sending a DRAW-YOURSELF message to the object *Bulldozer W1FB* might result in a piece of code associated with the class DISPLAYABLE-ACTOR-MIXIN clearing a region of the viewing window, a piece of code associated with the class CATERPILLAR-BULLDOZER drawing an icon in that newly-cleared space, then a piece of code associated with the class D9-BULLDOZER drawing a “D9” symbol in the middle of the icon. Code for clearing the appropriate portion of the screen is shared among all displayable actors; code for drawing the Caterpillar bulldozer icon is shared among all different types of Caterpillar bulldozers.

Multiple superclasses are illustrated in Figure 5.1 where BULLDOZER and HAS-DIESEL-ENGINE-MIXIN are both superclasses of CATERPILLAR-BULLDOZER. Description factorization can be much more complete when each independent behavior is represented in a separate class. Thus the essential elements of a bulldozer are captured in the BULLDOZER class but the complexities of diesel engines are separated in the HAS-DIESEL-ENGINE-MIXIN class and that class can be mixed into descriptions of trucks, backhoes, graders, and anything else that uses a diesel engine.

An aspect of object-oriented programming that is essential to CAE systems is the ability to extend the class hierarchy at runtime. All CAE systems share the property that a rich collection of building blocks is provided to the user and that this set provided falls just short of being comprehensive enough. Implementing these building blocks using a factored description in a class hierarchy is straightforward. However, if this class hierarchy cannot be extended at runtime, the program and the user are forced into contortions. For example, an architectural design system that provided 750 possible kinds of windows would surely leave out the one window that Joe User wanted—Joe wants one with four panes in the top sash and eight in the bottom. A few mouse motions and commands from Joe should be enough to describe the new window and a few hundred bytes of code should suffice to define a class in the window class hierarchy. However, if this hierarchy cannot be extended at runtime, there is no obvious way that Joe's custom window can be handled by the same code that handles the existing 750 types.

A final desirable feature of an object system is support for operations that depend on the class of two arguments, which are rather common. In a geometry system, the generic operation INTERSECT would do very different things if applied to a point and a line or if applied to a plane and a sphere. Different code would be required and results of different types would be returned. An object-oriented programming system that supports dispatching on the classes of multiple arguments can be very useful for organizing software to handle operations such as INTERSECT.

Smalltalk (see [Goldberg and Robson 1983]) and the Common Lisp Object System (CLOS, see [Gabriel 1991; Steele 1990; Keene 1989]) are the only two object-oriented programming environments that contain the preceding basic features essential for CAE programs.

5.1.3. Higher-Order Procedures

A procedure is a means of abstraction. Many parts of a CAE programs need to clear a window before displaying some information. Clearing a window is a compound operation that has many subparts, some of which will depend intimately upon the window system and/or computer being used. Rather than distribute copies of code to perform this compound operation throughout the CAE program, a programmer would typically define a CLEAR-WINDOW procedure. Defining procedures reduces code size, increases perspicuity, and facilitates maintenance and porting.

Procedures that manipulate procedures are called *higher-order procedures* [Abelson and Sussman 1985] and such procedures vastly increase the expressive power of a computer language. Suppose that you want to print out the value stored at every leaf in a tree data structure and also accumulate the sum. Without higher-order procedures, you'd have to write a procedure, PRINT-LEAVES, that included possibly complex tree-walking code as well as the simple value extraction, summing and printing code. Worse yet, if you ever decide to change the way trees are represented, you'll have to change PRINT-LEAVES.

Suppose that you create a higher-order procedure, APPLY-TO-LEAVES, that takes a tree, T1, and a procedure, LEAF-PROC, as arguments. APPLY-TO-LEAVES walks through the tree and calls the procedure LEAF-PROC on every leaf. Any part of the CAE program that needs to perform computation on the leaves of a tree can use APPLY-TO-LEAVES and if the tree data structure is modified, only this one procedure need be updated.

Higher-order procedures are mostly useful in languages, such as Lisp, where procedures are first-class objects and can be created at run-time. For example, in the case of using APPLY-TO-LEAVES to print leaf values, you won't know where the user wants the information printed until runtime and therefore it is convenient to wait until the user invokes a command before creating a LEAF-PROC that incorporates the window or other destination specified by the user.

5.1.4. Metalinguistic Abstraction

No general-purpose computer language is adequate to solve the most complex problems. To express ideas effectively and compactly, it is frequently necessary to develop new formal languages. This process is called *metalinguistic abstraction* [Abelson and Sussman 1985].

Describing municipal zoning laws is a typical problem that can be easily solved with metalinguistic abstraction. A simple formal declarative language would suffice to permit the expression of concepts such as "septic tanks must be set back at least 50 meters from any property boundary."

Developing new languages has traditionally been considered difficult because people become preoccupied with irrelevant matters such as syntax and parsing. Lisp programmers have taken advantage of this method of abstraction for decades. It is easy to develop a language embedded in Lisp for the following reasons:

- Lisp has uniform syntax: parentheses surrounded function names and arguments. This syntax can be used for any embedded language and the resulting language can be parsed by the same system procedures that read Lisp code.
- Lisp was designed to be introspective, i.e., Lisp programs are able to manipulate and construct other Lisp programs. All the native Lisp functions that are used to pull apart Lisp code can be used to analyze the embedded language.
- Lisp, although normally compiled, contains as part of its standard the EVAL procedure. EVAL evaluates Lisp code at runtime, functioning as an interpreter. Thus, embedded languages can be translated into Lisp code at runtime and the resulting Lisp evaluated directly.

Civil engineering encompasses such a diversity of problems that metalinguistic abstraction becomes an essential tool.

5.1.5. Arithmetic

Most people think of computers as exceptionally talented at arithmetic. Unfortunately, most computer languages are so primitive in this regard as to be almost useless for many of the surface modelling and other problems in computer-aided civil engineering. For example, a C program may produce different results when executed on computers with different word sizes or that store the bytes in multibyte integers in a different order. Neither execution will terminate in an error, and the user will end up with two different answers, both probably wrong. A FORTRAN program developed on a 32-bit machine and that has worked flawlessly for two years may break in an incomprehensible way when run on a 64-bit machine.

Floating-point arithmetic has been fraught with perils since the dawn of electronic computation. Consider the following equations:

$$2x = x + x, \quad x + y = y + x, \quad 1x = x, \quad (x + y) + z = x + (y + z).$$

What do they have in common? They are all false in some implementations of floating-point arithmetic. Yet an optimizing compiler may reorder operations based on those equations in order to make a program run faster. Recent standards [IEEE 1987] for floating-point arithmetic eliminate many pitfalls, both in day-to-day arithmetic and when moving a program to new computer systems. However, the IEEE standards do not specify how certain features are to be made available in programming languages. Thus a program that initializes a variable to "IEEE infinity" might not run on another computer, even if both claim to support the same language and IEEE floating-point arithmetic.

As inaccurate as they are, it is disturbing to discover that floating-point numbers are enshrined in virtually every numerical algorithm implementation available, a tribute to the high speed of floating-point hardware and the inflexibility of most computer languages. Algorithms are usually generic ideas that are independent of the type of numbers used. For example, a program that constructs a Delaunay triangulation of a set of points shouldn't care whether the points are specified by integers, floating-point numbers, extended precision floating-point numbers, rational numbers, etc. Yet almost all computer languages require every procedure that manipulates numbers with the language primitives to declare the type of number expected. Code is thus more difficult to reuse and becomes cluttered with irrelevant details.

Computational geometry algorithms published in academic journals are only guaranteed to work if arithmetic is precise. Unfortunately, few computer languages provide exact arithmetic except for integers within a limited range. Small errors in floating-point arithmetic can produce topological errors in data structures and geometric reasoning. For example, a point might be thought outside of a triangle when it is actually enclosed within the triangle. Traditionally, the programmer will add "epsilon" tolerances in every part of the program that fails during testing. A procedure that determines if a point is on a line by calculating an inner product and checking to see if the result is zero will instead check to see if the result is within a small, arbitrarily-determined "epsilon" of zero. Unfortunately, this kind of *ad hoc* kludging inevitably leads to the program failing when a user hands it an unanticipated case.

It is possible to represent and compute with integers of arbitrary size. Then, overflow results in longer execution time, not an erroneous result or aborted execution. With the provision of rational numbers, i.e., quotients of arbitrarily large integers, it is possible to perform any geometric computation exactly. Exact arithmetic allows programmers to construct simple and assuredly robust implementations of geometric algorithms.

Despite the luxuries it affords programmers, exact arithmetic has been "out of the mainstream" for commercial geometric programs. Once integers are allowed to grow to an arbitrary size, rather than being neatly restricted to a single machine word, implementation in a language without automatic storage management becomes extremely tedious. Integers must be explicitly allocated and deallocated constantly, which makes software more complex. A Delaunay triangulation algorithm using rational numbers implemented in C spent 70 percent of its time in MALLOC and FREE (storage management functions) [Karasick 1991], although this amount was later reduced by preallocating space for small integers.

Exact arithmetic is a standard feature of Lisp and has been for decades. Consequently, big integers (BIGNUMs) and rationals can be manipulated by exactly the same programs that manipulate floating-point numbers and integers that fit into machine words. Storage management for numbers is performed by the same system that manages storage for functions and data structures. Despite decades of improvements in low-level implementations of BIGNUMs, even Lisp programmers often resort to floating-point to achieve faster execution speed.

Despite defections to floating-point, the Lisp community has nurtured significant work on exact arithmetic using continued fractions to represent computable real numbers. Bill Gosper, the legendary pioneering Lisp hacker, opens his privately distributed work on continued fractions by admitting that "people who like continued fractions eat pickled okra and drive Citroens ... [I feel pity] for everyone who must crunch his numbers decimally or cast his points to float among electronic registers." As if to prove Gosper's point about Citroens, the most practical and complete continued fraction system has been implemented in LeLisp [Vuillemin 1988; Vuillemin 1987].

"A future so bright, you'll need to wear sunglasses" — this appears to be the case for exact arithmetic. The havoc wreaked by floating-point imprecision on geometric algorithms has attracted the attention of academics, with survey articles [Farouki 1989; Hoffmann 1989], papers focusing on using exact arithmetic in certain applications [Karasick 1991], and attempts to tame the floating-point monster [Guibas 1989; Milenkovic 1988]. Possibly more important for practical applications, computer architects are designing special hardware for rapidly performing exact or high-precision calculations. Modest hardware parallelism and modular arithmetic permit 900-bit operations to proceed at speeds comparable to standard workstations performing double-precision calculations [Wu 1989]. Gosper's work has inspired bit-serial processor designs for exact rational arithmetic [Kornerup and Matula 1987].

5.1.6. Compactness

Marcel Proust's *À la recherche du temps perdu* is supposedly one of the 20th century's greatest novels.[5] However, few will appreciate, much less fully understand, its 3300 pages filled with tangled sentences. A 200,000 line computer program is similarly inaccessible to a newcomer and modifications made by people without a comprehensive understanding of the program often have unintended effects. A programmer who doesn't understand Proust might be at a disadvantage in a Parisian *salon*; a programmer who doesn't understand the 200,000 line program he is modifying usually generates bugs and concomitant user suffering. Much effort has been devoted toward making programs harder to break and many computer languages, notably

[5]Proust (1871-1922) might have made a good hacker. Maintaining that "real books should be the offspring not of daylight and casual talk but of darkness and silence," he secluded himself in a cork-lined room where he wrote from 1905 until his death in 1922.

ADA, were designed specifically with this goal in mind. There is, however, no substitute for compactness.

A 100 line program is almost always easier to understand than a 10,000 line program. Smaller programs have fewer bugs and cost less to produce. Although native language power affects program size, abstraction is the primary means for reducing program size and perceived complexity. A bug in an algorithm is much easier to spot if the algorithm is not surrounded by irrelevant declarations, data type conversions, data structure manipulations, storage management function calls, etc. Factoring a system into procedures that are widely shared also facilitates compactness and debugging. Consider an algorithm implementation where 95% of the work is done by helper procedures that have been in use for years by other algorithms. A programmer can assume that the helper procedures are functioning correctly and focus his attention on the 5% of the code that is new. Such assumptions are sometimes wrong, but general-purpose procedures used in dozens of places are generally more reliable than freshly-constructed code.

5.2. Environment Requirements

Even with the choice of programming language held fixed, software development productivity is affected dramatically by the programming environment. I decided upon the following set of desirable features:

- incremental compilation and dynamic linking
- interactive debugging, including ability to restart computation
- algorithm animation tools, including steppers
- metering tools
- window system and user interface tools extendible via a class hierarchy

Unfortunately, when I started writing SITE CONTROLLER in 1986, no programming environment included all the features I wanted.

5.3. Choosing the Lisp Machine

Common Lisp and the Symbolics Lisp Machine fulfilled nearly all the requirements I set forth for both programming language and environment. The Lisp Machine's hardware and operating system support for clever garbage collection algorithms enabled me to work for weeks without rebooting the machine or thinking about storage allocation. Although the Common Lisp Object System was not standardized when SITE CONTROLLER was begun, the Lisp Machine Flavor system contained virtually everything I needed except the ability to dispatch on the type of two arguments (i.e., operations such as INTERSECT were difficult to express elegantly).

Common Lisp itself provides ample facilities for writing higher-order procedures and performing metalinguistic abstraction. Although Common Lisp's formal model of arithmetic has proven helpful when porting SITE CONTROLLER, I was forced to use floating-point arithmetic rather than Common Lisp's exact arithmetic due to the anemic performance of the 1 MIPS Symbolics processor. Finally, Common Lisp allowed me to express computation concisely and the core of SITE CONTROLLER is little more than 10,000 lines of code as a consequence.

The programming environment on the Symbolics machine that I used back in 1986, or for that matter on the MIT Lisp Machine I used back in 1979, is vastly superior to programming environments on modern computers (see [Greenblatt 1984; Walker 1987]). In addition to incremental compilation and dynamic linking, a flexible debugger, a class-based window system extendible at run-time, elaborate metering tools, and rudimentary algorithm animation, the Lisp Machine provided a seamless working environment where general rules and concepts made it possible to exploit the full power of even unfamiliar corners of the operating system and programming tools.

6. SITE CONTROLLER Description

6.0. System Overview

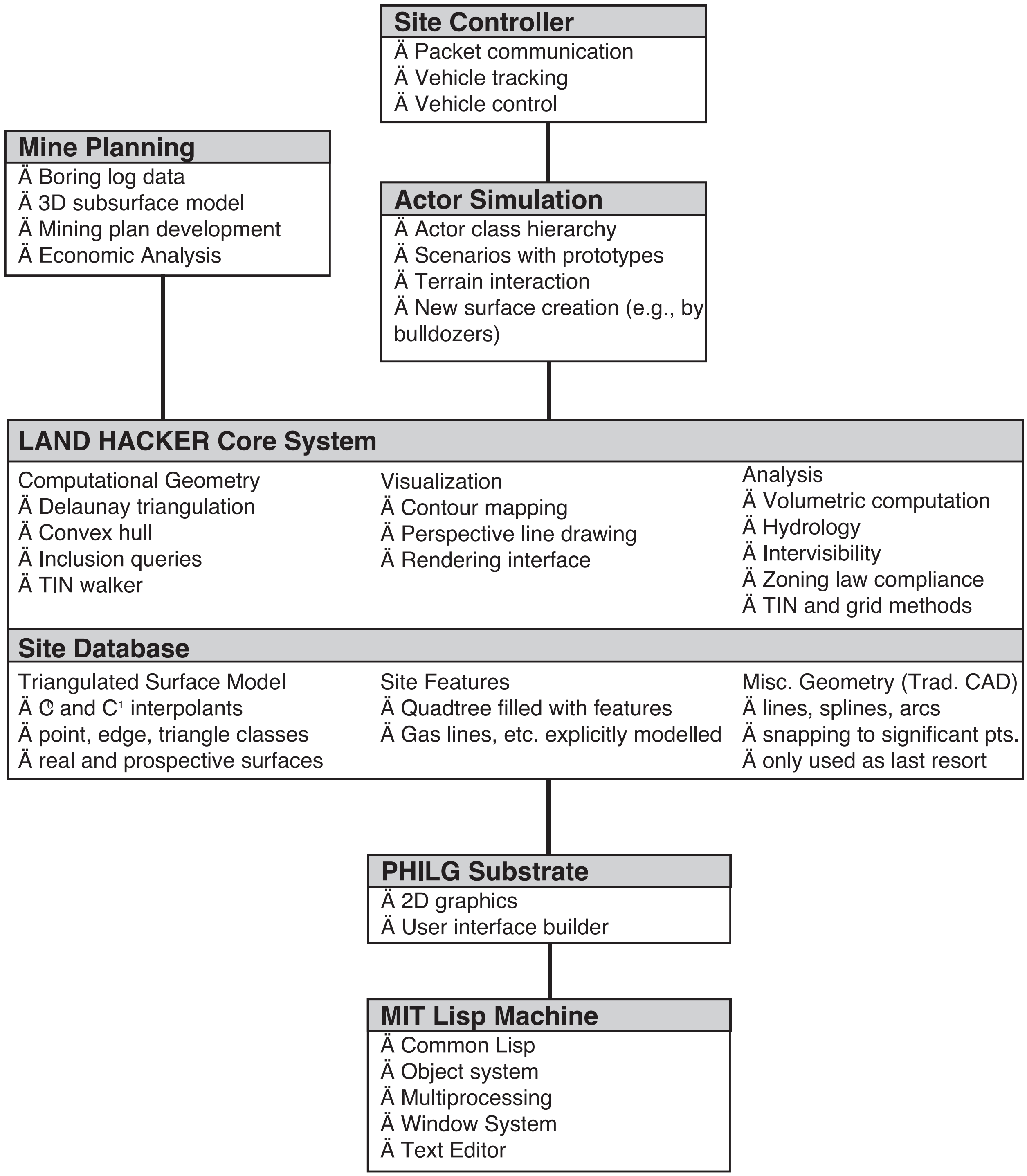

Figure 6.0 System Overview.

SITE CONTROLLER as described in this report may be broken up into the following components:

- base M.I.T. Lisp Machine system software
- PHILG substrate system, proving some simple graphics and user interface extensions
- LAND HACKER core system, where most of the site modelling and analysis is done, as well as much of the user interface
- Mine planning, an extension of LAND HACKER to model three-dimensional subsurface mineral seams
- Actor simulation, a framework for modelling vehicles and people
- Site Controller, *per se*, an extension of the actor simulator to communicate with, monitor, and control real construction vehicles

The overriding system design philosophy in SITE CONTROLLER is extension via subclasses. As much of the basic M.I.T. Lisp Machine operating and window system was retained as possible. Unless a native Lisp Machine facility was horribly ugly or slow, I extended it with a method or two and used rather than implement something from scratch. In practice, only the basic language, window, and multiprocessing systems were really useable. Many of my extensions were salvaged from previous applications and I collected them up into the arbitrarily-named "PHILG substrate system" (philg@mit.edu is my Internet address).

LAND HACKER is the core system that performs all of the functions necessary for site design and maintains the precious site database. LAND HACKER can stand by itself as a tool for a civil engineer and is complete with many user interface methods. Given all of the surface manipulation facilities built into LAND HACKER, the mine planning extension was surprisingly painless.

A much more significant effort was required to layer a general purpose actor simulator on top of LAND HACKER and define appropriate class behavior for earthmoving vehicles. When Caterpillar wanted to use the system for real world vehicle monitoring and control, I spent a few weeks extending the actor simulator and adding a packet communications module.

This section makes for rather heavy going at times and for that I apologize. However, much of the elegance of SITE CONTROLLER lies in the integration and the details; I could find no way to pick out the highlights and make them comprehensible on their own.

6.1. PHILG Substrate System

The PHILG substrate system was adapted from previous Lisp Machine efforts involving similar graphics display and user interface problems. Principal facilities of the PHILG system include user interface functions and window system extensions for graphics and plotting.

6.1.1. Graphics and Plotting

The VIEW-MIXIN class can be combined with any Lisp Machine window to allow the users of the window to perform operations in a projection plane; i.e., all the drawing methods and user interface methods scale automatically between a programmer-specified projection plane region and the actual pixels occupied on-screen by the window. In addition to scaling and obvious graphics drawing commands, the class contributes methods for centering or justifying text, a method for getting a mouse-specified rectangle from the user in projection coordinates, and methods for zooming and scrolling. Virtually every window in the SITE CONTROLLER system inherits from this class.

A simple 2D plotting package is rolled into yet another window class (PLOTTING-VIEW-MIXIN). This can be used to both present data to and capture data from the user, i.e., a program can take input from a user sketching a curve or touching points on a graph.

HARDCOPYING-VIEW-MIXIN is a class that, when mixed into any window inheriting from the VIEW-MIXIN class, sends output simultaneously to the screen and a printer. Mostly, this code consists of a set of :AFTER methods that will be combined with the primitive drawing commands of VIEW-MIXIN. Every time a drawing message is sent to a window inheriting from both VIEW-MIXIN and HARDCOPYING-VIEW-MIXIN, a primary method from VIEW-MIXIN executes first to draw on the screen then an :AFTER method from HARDCOPYING-VIEW-MIXIN is invoked to send appropriate commands to a printer. There are also methods here for turning the hardcopy shadow on and off.

6.1.2. User Interface Facilities

PHILG contains a system for automatically assembling a keyboard and mouse user interface, complete with on-line help, from distributed command definitions. By building the overall system “frame” to inherit from KEYBOARD-COMMANDS-MIXIN, the programmer would ensure that, at system startup, a keyboard command data structure would be created by methods of this class. By defining the method combination for the :KEYBOARD-COMMANDS operation to be :APPEND, any class with commands to contribute need only define a method for this operation that returns a list of those commands. At startup, the lists would be appended by the Lisp Machine object system and returned as the result of the operation :KEYBOARD-COMMANDS.

Here is an example of the code to define a single keyboard command:

```
(define-keyboard-command #\help :command-help "Display helpful
information")
```

This says that when the HELP key is pressed, send the frame the message :COMMAND-HELP and that, if asked for help about what this key does, print that it will "Display helpful information".

MOUSE-COMMANDS-MIXIN works similarly, except that each window in the frame has its own data structure for handling mouse clicks. On a Symbolics machine, there are three mouse buttons and five shift keys (SHIFT, CONTROL, META, SUPER, and HYPER), thus allowing 96 possible mouse clicks. MOUSE-COMMANDS-MIXIN constructs two interfaces. The first allows novices to choose commands from a menu by rolling the mouse over various command names while viewing short descriptions of each command in the Lisp Machines documentation line near the bottom of the screen. Experienced users invoke comands with mouse *chords*, i.e., by holding down a combination of shift keys with the left hand and pressing one of the three mouse buttons with the right.

The beauty of this implementation is that commands and on-line documentation cannot get out of sync. Whenever one defines a command, one must supply documentation that the system automatically feeds to the user at appropriate times. Commands and documentation are similarly removed atomically.

More traditional user interface support is provided by the class INTERACTIVE-VIEW-MIXIN. There are methods that "rubberband" lines, piecewise-linear curves, rectangles, circles and ellipses while the user manipulates the mouse. The user plays with a shape by moving the mouse, sees the shape at all times, and specifies the shape by touching a mouse button or the END key. The :GENERIC-MOUSE-HANDLER method nicely illustrates the power of higher-order procedures. This method takes a 2x2x2x2 matrix (corresponding to the states of the HYPER, SUPER, META and CONTROL shift keys) as its argument. Each entry is a list of four procedures and a documentation string describing the effect of each. The method tracks the user's mouse movements and applies the appropriate procedure after each mouse move (there are three buttons on a Lisp Machine mouse and hence one needs four procedures, one for each button plus one for "no buttons pressed.")

What is important about :GENERIC-MOUSE-HANDLER is that the user procedures are spared from so many messy details of the Lisp Machine window system: 1) understanding how to "grab" the mouse and ensure that all mouse action is reported to this window; 2) figuring out an efficient method of detecting mouse moves smoothly; and 3) sensing the state of four shift keys and updating the documentation line appropriately as the user presses shift keys. User procedures are thus much smaller and more portable across window systems.

For use interactions that are global in nature, i.e., not restricted to a single view, INTERACTIVE-FRAME-MIXIN is a class that contributes numerous user interface methods to any frame that inherits from it. Notably these include methods to prompt and query the user, to display output on a special roll-down "typeout" window, and similar methods that shield the rest of the program from Lisp Machine window and I/O system details.

6.2. LAND HACKER Core System

6.2.1. Geometry and Graphics Substrate

This 2D and 3D geometry substrate includes simple vector manipulation procedures such as addition, dot product, cross product, scalar multiplication, and normal vector construction. Slightly more complex are a full complement of transform operations such as "rotate about point." Oriented lines are modelled, and there are primitives for determining on which side of a line a point falls, the distance of a point from a line, and segment intersection. Polygon, parallelogram, and segment objects are defined and are sufficiently full-bodied to know how to draw themselves.

The 3D-VIEW-MIXIN class gives a window the capability of displaying perspective and parallel projections of 3D objects. This class supports the programmer by enabling a window to automatically scale itself to fit a 3D bounding box, i.e., ensure that everything inside that box will be visible in the window.

6.2.2. Surface Modelling Data Structures

Civil engineering surface modelling can be broken up into two problems:

- given a set of points of known elevation, interpolate a surface through those points that accurately and efficiently represents the site
- capture from the user and represent a model of what the site should look like

A good surface model for civil engineering must have the following characteristics:

- ability to model different areas of the site at different levels of detail, i.e., a uniform grid is inadequate
- ability to faithfully reflect site topology, i.e., features such as ridges and streams

Older surface modelling programs tend to use fixed regular rectangular grids to represent surfaces. Scattered survey data is used to calculate the elevations of every grid point. Elevations of query points are interpolated using a few nearby grid points. The principal shortcomings of this technique are that all areas of the surface are represented at the same level of detail, which means that a need for fine detail in even a small corner implies an enormously large grid, and that site topology cannot be reflected, i.e., ridge and stream lines may be blurred.

6.2.2.1. Triangulated Irregular Networks

SITE CONTROLLER, along with most modern surface modelling software, uses a *triangulated irregular network* (TIN) [Peucker 1978; Fowler and Little 1979]. Looking down at a planar map of a set of surveyed points, consider the smallest convex polygon that contains all the points, the *convex hull* of the points. The convex hull is shaped like a rubber band stretched around the points. Now connect the points so that the interior of the convex hull consists of triangles whose vertices are surveyed points (see Figure 6.1). This is a TIN model of the site.

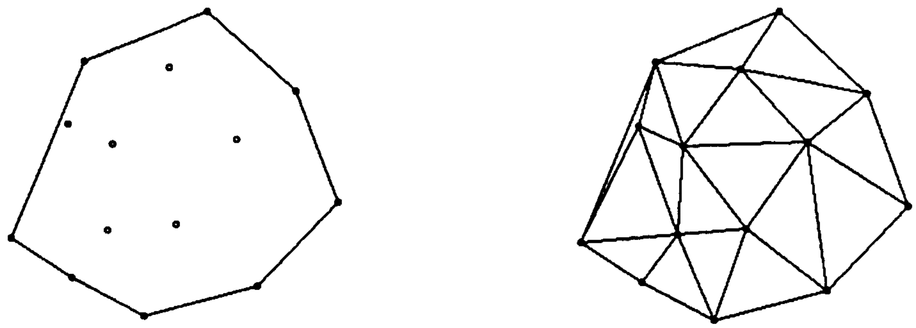

Figure 6.1 **(a)** Surveyed points and convex hull; **(b)** interior of convex hull triangulated.

6.2.2.2. Piecewise-Planar Model

Three (non-collinear) points determine a plane. Each triangle vertex is a *surveyed* point, i.e., a point whose elevation is known. Thus, we may obtain a piecewise-planar surface model of the site simply by regarding each triangle as a section of a plane. This looks something like the faceted surface of a cut diamond, although a gem would have much more relief than a typical civil engineering site. This is a C^0 model of the surface, i.e., there are no holes, but the first derivative is discontinuous at the triangle edges. Although crude, a linear fit has the advantages of computational speed and faithfulness to manually-surveyed data. Surveyors are instructed to record a point *at every slope change* and are thus already fitting a piecewise-planar surface to the site in their heads.

A SITE CONTROLLER surface model consists of surveyed points connected by edges in a triangulation.

6.2.2.3. Surveyed Points

Surfaces are abstract mathematical entities constrained to pass through a set of specified points. Each point is represented by an instance of the class SURVEYED-POINT that contains the following: *x,y,z* coordinates; a pointer to the surface of which it is a part; a list of other points to which the particular point is connected in a triangulation; a list of triangles of which this point is a vertex; a list of edges of which this point is an endpoint; and provision for annotations such as how the point was generated (e.g. designer input, surveyed data measured by Joe Surveyor, data automatically collected from a bulldozer, etc.).

In addition to straightforward methods for managing their data structures, surveyed points are responsible for a lot of the work in building the overall surface model. For example, it is the surveyed points that create edges. Surveyed points also have methods that aid hydrological analysis plus the usual :DRAW-SELF, :HIGHLIGHT-SELF, and :UNHIGHLIGHT-SELF methods that enable surveyed points to work smoothly with the SITE CONTROLLER user interface.

6.2.2.4. Triangles

Triangles are instances of the fairly simple class TRIANGLE. Each instance stores the three vertices and a specification for the type of terrain (e.g. silty sand), caches oriented 2D lines for each edge (useful for answering point inclusion queries) and also the direction of the gradient of the plane that passes through the three vertices.

A triangle is a remarkably capable object, able to say whether a query point is inside itself, which of its edges are cut by a ray (useful for problems such as intervisibility), what the elevation at any included query point is, the location of its centroid, and its 2D area. Triangles also provide the active methods for drawing and annotating contour lines, as well as standard user interface methods for drawing, highlight and unhighlighting.

6.2.2.5. Edges

Edges in the triangulation are full-fledged objects. Here we have traded space for speed in a particularly profligate manner. Only two pointers, one for each surveyed point, need be kept and yet an object of the class EDGE contains no fewer than 16 instance variables. There are representations of each point as a 2D and 3D vector, a normal vector in the direction of the edge, a full size vector pointing from POINT-1 to POINT-2, an oriented line through the two surveyed points, and a collection of information about elevation that is useful when contouring.

An edge is capable of interpolating along the line passing through its endpoints, intersecting itself with a line segment (both in space and in projection onto the x-y plane), saying whether a point is to the left or the right of the edge (in a 2D projection), helping the program walk through the data structure finding all edges cut by a ray, determining whether it is a stream or a ridge, and interacting with the user interface.

6.2.3. Triangulation

Accuracy in a triangulated surface model is intimately dependent upon the *triangulation* of the points, i.e., which points are connected to which other points. If triangles are long and skinny, the interpolated elevation of a query point will depend on far away surveyed points despite the existence of nearby surveyed points that are probably more indicative of the query point's true elevation. It is therefore desirable to choose a triangulation where most of the triangles are close to being equilateral.

If a triangle cuts through a ridge or wall, interpolated elevations on one side of the ridge are influenced by surveyed points on the other side and the ridge itself may be obscured. Choosing a triangulation where the edges coincide with the ridge ensures that accuracy is maintained and that the surface model remains faithful to the topology of the site.

6.2.3.1. Delaunay Triangulation

There are numerous ways to measure the "equilateralness" of a triangulation. Consider a triangulated quadrilateral, i.e., one broken up into two triangles by a diagonal line. If switching the triangulation to use the alternative diagonal would not increase the minimum of the six interior angles, then the triangulation is said to be *locally equiangular*. Note that small minimum angle is bad, corresponding as it does to a long, skinny triangle. Determining global equiangularity is more complicated. First of all, one must use Euler's relation to show that any triangulation of n non-collinear sites in the plane must contain exactly $m = 2n - 2 - k$ triangles, where k is the number of boundary edges, i.e., the edges of the convex hull. Thus, every triangulation of a set of points has a total of $3m$ interior angles and the sum of those angles is $m\pi$.

Next, consider the set of interior angles $(\alpha_1, \alpha_2, \ldots, \alpha_{3m})$ indexed so that $\alpha_i \leq \alpha_j$ if $i < j$. An alternate set of interior angles, i.e., from an alternate triangulation of the same points, $(\alpha'_1, \alpha'_2, \ldots, \alpha'_{3m})$ is said to be *less than* (i.e., worse than) the original set if there is an index $1 \leq j \leq 3m$ such that $\alpha'_j < \alpha_j$ and $\alpha'_i = \alpha_i$ for $1 \leq i < j$. A triangulation is said to be *globally equiangular* if all other possible triangulations are less than it, i.e., that its sorted list of angles gets "big" faster than any other triangulation's.

The *Delaunay triangulation* is locally equiangular, which has been shown to be equivalent to global equiangularity, and is the only such triangulation [Edelsbrunner 1987]. Although there are many ways of generating the Delaunay triangulation, an easy way to see it is as the dual of the *Voronoi Diagram*. Consider a set of points $p_1, p_2, \ldots, p_n$. The Voronoi polygon of p_3 is the region of query points closer to p_3 than to any other point p_i.

A naive method of generating the Voronoi polygon of a point p_i is to construct the perpendicular bisector of each pair of points $[(p_i, p_j) \;\; i \neq j]$ and intersect the half-planes containing p_i. The resulting polygon is the Voronoi polygon of p_i. Note that constructing a Voronoi Diagram that shows the Voronoi polygon for each of n points requires $O(n^2)$ time if carried out in this naive manner.

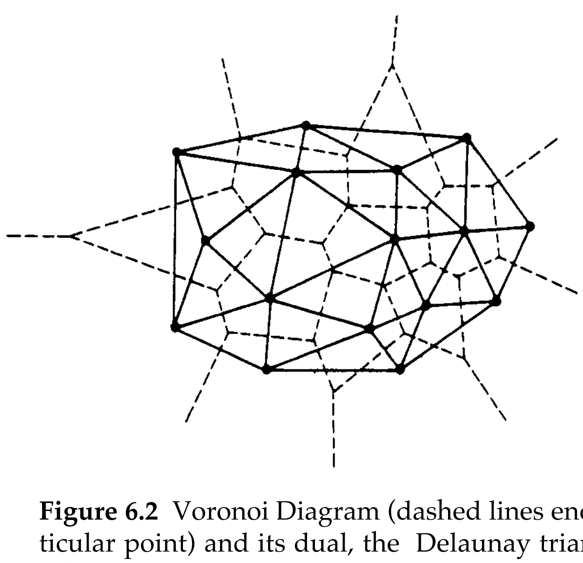

Figure 6.2 Voronoi Diagram (dashed lines enclosing regions of plane closest to a particular point) and its dual, the Delaunay triangulation (solid lines connecting points whose Voronoi polygons are adjacent).

Connecting pairs of points that share an edge in the Voronoi Diagram generates the Delaunay triangulation of the set of points in $O(n)$ time (see Figure 6.2). Considering the number of surveyed points in a construction site, and the number that are added in real time by the activity of construction vehicles, the Delaunay triangulation would not be very useful if it required $O(n^2)$ time (to construct the Voronoi diagram and then the triangulation). Fortunately, there are numerous $O(n \log n)$ algorithms to construct Voronoi diagrams, the most elegant of which is a plane sweep algorithm [Fortune 1987]. There are also efficient algorithms for constructing Delaunay triangulations directly [Aurenhammer 1991].

6.2.3.2. Constrained Delaunay Triangulation

Although using the Delaunay triangulation by and large solves the problem of inaccuracy due to skinny triangles, the problem of the triangulation not faithfully representing the topology remains. We would like to constrain the model so that certain edges are preserved, i.e., certain points are connected to certain other points. We would then like to generate the triangulation that is closest to the Delaunay triangulation subject to those constraints, i.e., the *Constrained Delaunay triangulation*. It is possible to construct a Constrained Delaunay triangulation or its dual, a constrained Voronoi diagram, in $O(n \log n)$ time and $O(n)$ space using a plane sweep algorithm [Aurenhammer 1991]. This is an important result, for if exponential time were required, the utility of triangulation-based construction automation software might be restricted to toy problems.

Given this rich body of literature, what path did I choose for SITE CONTROLLER? Well, Jim Little and I spent an afternoon translating one of his ancient 3000-line PL/I Delaunay triangulation programs into a 150-line Common Lisp program. This is not an efficient algorithm, taking $O(n^2)$ time to do an $O(n \log n)$ job, but since our initial data sets were small, it didn't seem worth spending a lot of time trying to implement an optimal algorithm from the literature.

Once the Delaunay triangulation is complete, it is corrupted by a brute force linking of points that are paired up in *forced connections* usually to respect the topology. The data structures are then fleshed out, with surveyed points being linked to newly created edge objects and vice versa. Finally, triangles are created and doubly linked to the points. Finding all the triangles is not as simple as it sounds, since most obvious algorithms will find too many triangles, i.e., ones that contain other triangles.

6.2.4. Surfaces

Surfaces are represented by instances of the class SURFACE. The class definition, initialization values, user interface for editing surface parameters, and information about which instance variables to save in a site database are all obtained from one list of lists, each sublist of which has the following form:

```
(name :name nil :set-name t t :string-or-nil t)
```

This list specifies the instance variable NAME, with get message :NAME, initial value NIL, and set message :SET-NAME. The last four values declare the variable to be both user-visible and user-settable, inform the user interface code to accept only strings or NIL as valid NAMEs, and ensure that this value will be saved along with a surface.

Once again, we see how defining something this way ensures that the data structures and user interface don't get out of sync.

Information carried around by surfaces includes the following:

- a list of surveyed points
- a flag indicating whether those points are connected in a valid triangulation (triangulation is expensive and hence only done when necessary)
- a list of triangles, a list of "interpolating triangles" capable of supporting a smooth surface model with continuous first derivative (these take up a lot of space so we don't compute them unless the user asks for a smooth contour map or something else for which a piecewise-planar surface model is inadequate)
- a parameter specifying a tradeoff between execution speed and contour smoothness
- a list of forced connections
- a boundary polygon that is the convex hull of the surveyed points
- the edges of the boundary polygon (useful for contour labelling)
- the default contour approximation spacing to be used when the user sketches in contour lines with the mouse (specifies how far apart surveyed points are to be laid down to approximate the continuous contour)
- some hydrology modelling parameters
- a tolerance for cut-and-fill display that specifies when a difference in height is too small to be worth considering.

Surfaces are fairly powerful objects. There are numerous methods for maintaining the pointers among point, edge, and triangle data structures. There is a method for forcibly connecting two points and fixing broken edges so that the points and connections still constitute a triangulation. Most queries about the surface-modelling data structures are handled as messages to an object of the class SURFACE. In particular, a surface can return a list of edges that intersect a query segment (useful for intervisibility calculations, for example), or the triangle that contains a query point, or whether the convex hull contains a query point at all.

Surfaces can respond to an :ELEVATION-AT-POINT message with the interpolated z at any (x,y). They can also contour themselves if supplied with a graphics window. For 3D line drawings, the surface is able to supply a profile corresponding to a line segment cutting the triangulation.

For debugging purposes, surfaces are able to animate some of their computations and are even able to give themselves a random shape.

6.3. Surface Visualization

Civil engineers must visualize surfaces in order to design a site plan. Architects must visualize surfaces when judging aesthetics of proposed earthmoving projects. Laymen, especially government officials, must visualize surfaces when deciding whether or not to approve a project.

Traditionally civil engineers have relied on contour maps to visualize surfaces. A contour map is created by drawing the intersection of a surface with a collection of evenly-spaced horizontal planes. Contour maps are easily generated by draftsmen and civil engineers quickly learn to get a feel for terrain based on a contour map. Unfortunately, laymen have great difficulty visualizing surfaces with contour maps and even experts often find more powerful visualization tools helpful.

Ever since the first pen plotters were introduced in the 1950's, surfaces have been visualized with line drawings. Laymen can acquire some insight from these black & white pictures. The most realistic images are those that have become common only in the 1980's with the advent of the 3D workstation. By shading polygons with different colors depending on the terrain, intensities depending on the angle relative to light sources, and patterns depending on the texture of the surface, it is possible to obtain images of great realism.

6.3.1. Contour Mapping

Contour mapping is easy on a piecewise-planar surface. For each triangle, one simply draws a line segment between the points where the horizontal contour plane cuts the edges. These points may be found by linear interpolation along the triangle edges. SITE CONTROLLER performs contour mapping locally, i.e., each triangle draws what contour line segments pass through itself. Difficult issues in contour mapping include labelling and drawing contour maps efficiently with pen plotters. These issues are covered by a comprehensive survey article [Sabin 1985] but I did not attack any of the tough problems in my implementation.

6.3.2. Perspective Line Drawings

SITE CONTROLLER contains no software to do traditional rendering of its polygonal surfaces. This choice was made because of the staggering expense and hence scarcity of the Symbolics 24-bit color display. Furthermore, since so many standard rendering facilities on graphics workstations were available, it seemed simpler to merely generate surface descriptions in a form readable by existing rendering programs rather than reinvent the wheel. However, SITE CONTROLLER does contain a traditional perspective line drawing facility for black & white displays.

Line drawings are projections of the intersection of a surface with a series of regularly spaced vertical planes. Iterative algorithms that generate such drawings are commonly implemented in FORTRAN, PL/1, or C, sometimes requiring thousands of lines of code and more than one man-year to debug. James L. Little and I wrote a 70-line Common Lisp program in two hours that accomplishes the same task. The program illustrates a number of important programming techniques and is therefore presented in detail.

Traditional programs such as multi-pass compilers serially convert data from one format to another until the desired final format is achieved. Using Lisp's recursion and dynamic storage allocation, a program can decompose a problem as it recurses, then incrementally construct the solution as it returns. The canonical example of this approach is a rudimentary compiler that associates code with each node in a parse tree and then collects that code as the tree is collapsed.

Our algorithm only works with surfaces defined by $z(x, y)$ so that the surface has a unique elevation for a given point in the x-y plane—overhangs and caves cannot be modelled. Furthermore, we take advantage of the fact that we are working with the surface of the earth and need only show one side of the surface. For example, if the surface is a small oil field in Texas with a 5,000-meter-deep hole in the ground, we cannot see the exterior of the hole poking out from underneath our surface if our view point is above the surface.

6.3.2.1. General Algorithm

First, we intersect the surface with a series of vertical planes orthogonal to the projection of the viewing direction onto the x-y plane. We then convert each intersection into a list of line segments (if the surface is not piecewise-planar we must approximate higher-order curves in a piecewise-linear fashion, something we would have to do anyway for most output devices) and call the list a *profile*. We make a list of the profiles by stepping in fixed increments along the viewing direction projection (if profile A is closer to the view point than profile B, then A precedes B in the list).

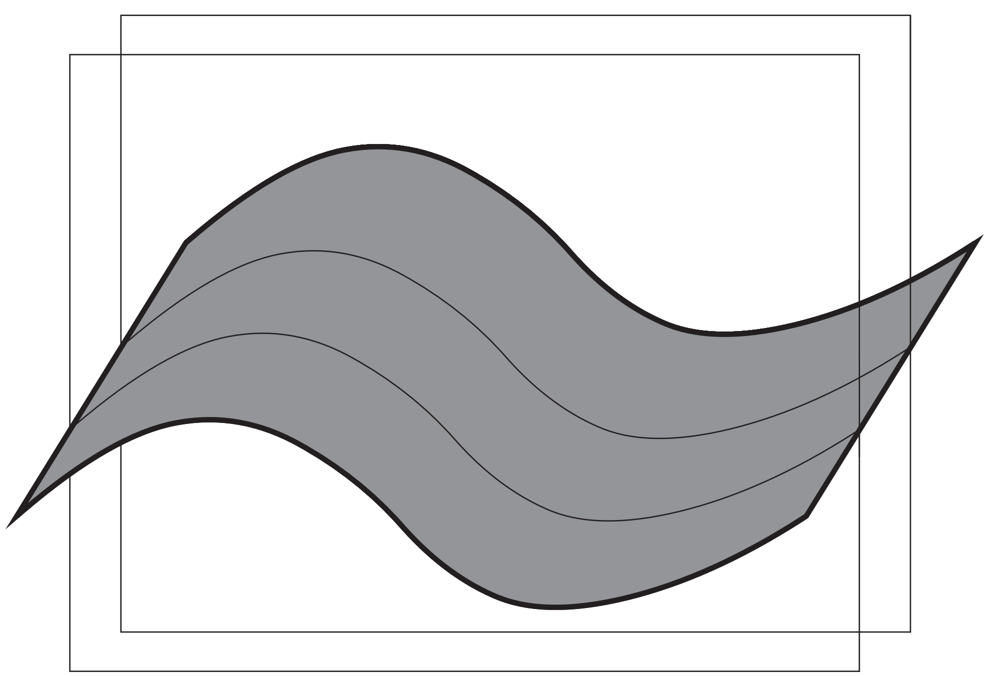

Figure 6.3a An example surface intersected by two vertical planes.

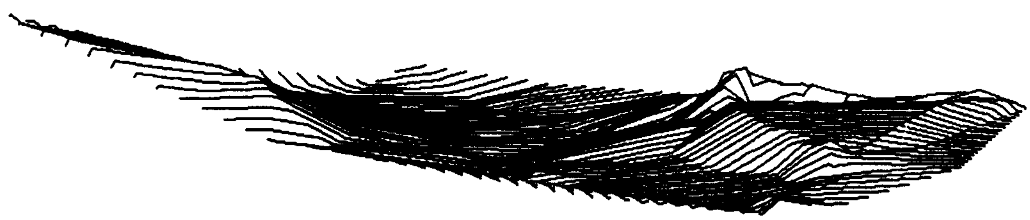

Figure 6.3b A real site in New Hampshire, rendered with 60 profiles.

We accomplish hidden-line elimination by two dimensional clipping against a horizon on the view plane. We display only those line segments (or portions of line segments) whose projections are above a horizon established by projecting and drawing previous profiles. Our initial horizon is a "segment" from $(-\infty, -\infty)$ to $(+\infty, -\infty)$ so that anything in the first profile is guaranteed to be drawn.

The core function, CLIP, takes a profile and existing horizon and returns that portion of the profile that was visible and a new horizon. OUTER-LOOP starts off the recursion with all the profiles and the initial horizon by calling OUTER-LOOP-1. OUTER-LOOP-1 checks to see whether there are any profiles left and, if so, calls CLIP to clip the first remaining profile against the horizon. OUTER-LOOP-1 then draws whatever visible segments CLIP returned and recurses with the remaining elements of the profile list (the CDR, or the rest of the list after the first element) and the new horizon returned by the call to CLIP.

6.3.2.2. Recursive clipping

CLIP takes two arguments. The first argument is a list of profile segments and the second is a list of horizon segments. CLIP uses the Common Lisp multiple value facility to return two values, a list of segments to draw (that part of the profile that was visible) and a new horizon (also a list of segments).

CLIP assumes that the segments within each list either do not overlap or that they overlap just at their endpoints. The segments are assumed to be arranged from left to right (in order of increasing *x*) and it is assumed that the horizon extends at least as far in both directions as the profile.

As in any recursion, we first check for termination conditions. If the list of horizon segments is empty, we signal an error (since the horizon should extend to -∞). If the list of profile segments is empty, we are done and return NIL (the empty list) as our first value (the list of segments to draw) and the remaining horizon list as the new horizon. These values will be added to by surrounding calls to CLIP.

Another easy case is when the first profile segment is completely to the right of the first horizon segment (Figure 6.4).

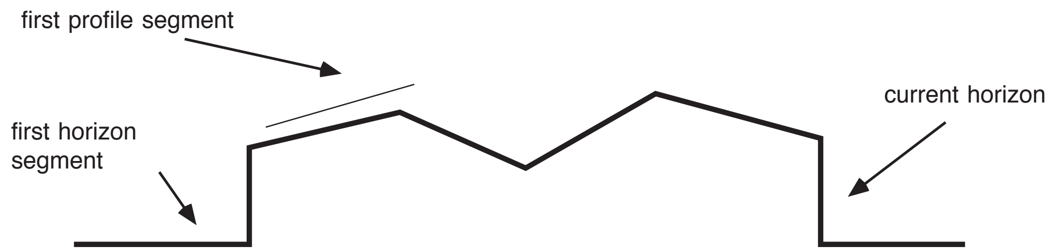

Figure 6.4 The first profile segment does not overlap the first horizon segment. In this case we recurse, passing down the complete profile segments list, but stripping off the first horizon segment. The value returned by this call to CLIP cannot simply be the value of the recursive call, however, since that would result in the loss of part of the horizon. The SEGMENTS-TO-DRAW value is simply whatever the recursion returned, but the new horizon is a list whose CAR is the horizon segment we stripped off and whose CDR contains the horizon segments returned by the recursive call.

6.3.2.3. Real work

We've now dealt with the easy cases and must consider the case where there is some overlap between the first horizon segment and the first profile segment (Figure 6.5 illustrates two representative cases).

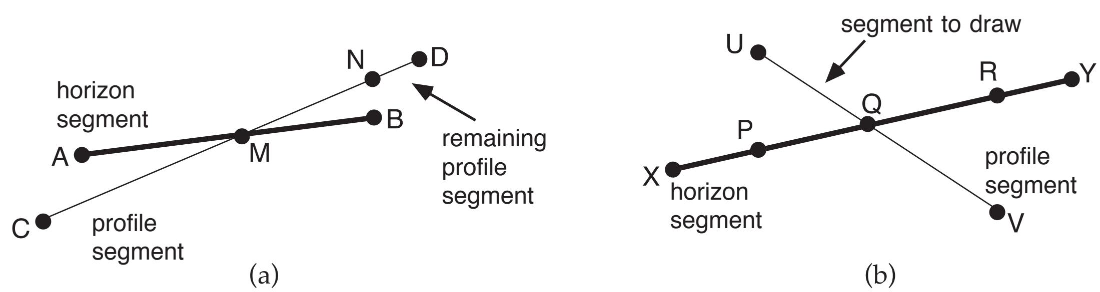

Figure 6.5 The operation of CLIP-SEGMENTS. **(a)** *AB* is the input horizon segment, *CD* is the input profile segment, *MN* is the SEGMENT-TO-DRAW, *AM* and *MN* are the NEW-HORIZON-SEGMENTS, and *ND* is the REMAINING-PROFILE-SEGMENT. The REMAINING-HORIZON-SEGMENT is NIL. **(b)** *XY* is the input horizon segment, *UV* is the input profile segment, *UQ* is the SEGMENT-TO-DRAW, *XP*, *UQ*, and *QR* are the NEW-HORIZON-SEGMENTS, and *RY* is the REMAINING-HORIZON-SEGMENT.

The key idea here is to deal with as small a part of the problem as possible and let the recursion do the rest of the work. CLIP calls CLIP-SEGMENTS to compare the two segments until it hits the right edge of either. CLIP-SEGMENTS returns four values. The first is a segment to draw (at most one can result from the intersection of a single profile segment and a single horizon segment). The second is a list of new horizon segments lying horizontally between the left edge of the original horizon segment and the leftmost right edge of either segment. This list will contain between one and three elements (one if the profile segment was entirely below the horizon, two if the left edges of both segments were aligned and they intersected somewhere in the middle, and three if the profile segment's left edge was to the right of the horizon segment's left edge and they also intersected in the middle).

CLIP-SEGMENTS returns as its third value that portion of the profile segment that extended beyond the right edge of the horizon segment. The fourth value is that portion of the horizon segment that extended beyond the right edge of the profile segment. At most one of these will be non-NIL (both will be NIL if the segments' right edges are aligned, otherwise one segment must extend beyond the other).

Most recursive algorithms in Lisp generally completely process one element of a list, strip that off and recurse with the rest of the list. CLIP is unusual in that it sometimes doesn't completely process the first element, so it has to recurse with the unhandled piece of the first element stuck CONSed onto the rest of the list. CLIP is also unusual in that it is recursing down two lists at once and returning two values.

So CLIP calls itself recursively, with the first argument (profile segments) being the result of a conditional. If the profile segment processed on this call extended beyond the horizon segment processed, we strip off the entire first profile segment (using CDR), but stick on the piece of the first profile segment that we didn't handle (the REMAINING-PROFILE-SEGMENT value returned by CLIP-SEGMENTS). Otherwise, we have completely finished with the first profile segment and can recurse after stripping it off. The horizon segments argument for the recursion is similarly calculated.

After receiving the values returned by the recursive call, we must add to them the pertinent information from the section this call handled. If CLIP-SEGMENTS found a visible segment, it is CONSed onto the list of segments to draw returned by the recursive call; otherwise, CLIP just returns the result of the recursion. Because every call to CLIP is guaranteed to make some progress to the right, some horizon is generated for each call. CLIP calculates the new horizon value by appending the NEW-HORIZON-SEGMENTS returned by CLIP-SEGMENTS and the new horizon returned by the recursive call.

One valid objection to this algorithm is that it tends to produce fragmented horizons. This is easily dealt with by making each subsegment point to the segment from which it was created. Subsegments that were created from the same segment are guaranteed to be collinear and hence can be collapsed by OUTER-LOOP-1. In one example, when displaying a triangulated digital terrain model surface, the horizon following the clipping and display of 60 profiles was a list of 750 segments. Collapsing segments after each profile was displayed resulted in a final horizon of 60 segments and dramatically reduced run time.

A nice feature of this algorithm is its ability to deal with different segment representations. Only the functions MAKE-SEGMENT and CLIP-SEGMENTS need to know how the data are structured and both are straightforward. Also, this algorithm can be generalized to finite surfaces by maintaining both top and bottom "horizons."

```
(defun outer-loop (profile-list)
  (when profile-list
    (let* ((initial-horizon-segment
             (make-segment (make-point (- *infinity*) (- *infinity*))
                           (make-point *infinity* (- *infinity*))))
           (initial-horizon (list initial-horizon-segment)))
      (outer-loop-1 profile-list
                    initial-horizon))))

(defun outer-loop-1 (profile-list horizon)
  (when profile-list                        ; This will terminate when we
                                            ;   run out of profiles.
    (multiple-value-bind (segments-to-draw new-horizon)
        ;; CLIP takes the next profile on the list and the existing
        ;; horizon and returns SEGMENTS-TO-DRAW and the new horizon.
        (clip (car profile-list)
              horizon)
      (draw-segments segments-to-draw)      ; Display the segments on some
                                            ;   output device.
      (outer-loop-1 (cdr profile-list)      ; Strip off the profile we've just
                    new-horizon))))         ;   handled

(defun clip (profile-segments horizon-segments)
  (declare (values segments-to-draw new-horizon-segments))
  (cond ((null horizon-segments)
         (error "Ran out of horizon"))
        ((null profile-segments)
          ;; We have run out of profile segments, so we return a NIL to
          ;; CONS onto as SEGMENTS-TO-DRAW and return the remaining
          ;; horizon.
          (values nil horizon-segments))
        ((not (overlap? (car profile-segments) (car horizon-segments)))
          ;; The first profile segment is completely to the right of the
          ;; first horizon segment
          (multiple-value-bind (rest-segments-to-draw
                                rest-new-horizon-segments)
              (clip profile-segments            ; We haven't processed any,
                                                ;   so we pass through.
                    (cdr horizon-segments))     ; Strip off the first one.
            (values rest-segments-to-draw       ; Nothing new to draw,
                    (cons (car horizon-segments) ;  but must return the
                                                ;  new horizon piece
                          rest-new-horizon-segments))))
        (:else
         ;; The left endpoint of the first profile segment falls within
         ;; the first horizon segment.
         (let ((profile-segment (car profile-segments))
               (horizon-segment (car horizon-segments)))
           (multiple-value-bind (segment-to-draw
                                 new-horizon-segments
                                 remaining-profile-segment
                                 remaining-horizon-segment)
               (clip-segments profile-segment horizon-segment)
             ;; We now have all the information we need for the recursive
             ;; call.
             (multiple-value-bind (rest-segments-to-draw
                                   rest-new-horizon-segments)
                 (clip
```

```
      ;; Calculate the PROFILE-SEGMENTS to send down.
      (if remaining-profile-segment ; If some of the profile
                                    ;   segment wasn't handled
          (cons remaining-profile-segment
                (cdr profile-segments))
        (cdr profile-segments))
      ;; Calculate the HORIZON-SEGMENTS to send down
      (if remaining-horizon-segment ; If some of the horizon
                                    ;   segment wasn't handled
          (cons remaining-horizon-segment
                (cdr horizon-segments))
        (cdr horizon-segments)))
 (values
   ;; Calculate the SEGMENTS-TO-DRAW to return.
   (if segment-to-draw
       (cons segment-to-draw        ; Our contribution.
             rest-segments-to-draw) ; Result of recursion.
     rest-segments-to-draw)         ; This call contributes
                                    ;   nothing.
   (append new-horizon-segments     ; We always contribute
                                    ;   something to the
                                    ;   to the horizon.
           rest-new-horizon-segments)))))))
```

6.3.3. Rendering

In computer graphics terminology, a *scene* is a mathematical representation of a picture. An *image* is the manifestation of a scene on a display device. *Rendering* is the processing of turning a scene into an image. In standard computer graphics implementations, a scene is a collection of 2D polygons oriented in a 3D space, with the visible portion of each polygon calculated and marked given a particular viewpoint, plus a scene illumination function. An image typically consists of a collection of pixel intensity and color values for a CRT display.

Rendering a scene already broken up into planar triangles or polygons is a standard feature of many workstations and window systems and is often assisted by special hardware. All a computer-aided engineering system need do is format a list of triangles appropriately and let the rendering program run (I hooked up SITE CONTROLLER with two different rendering programs in this fashion and never spent more than one hour writing the interface) . This approach has advantages: 1) the CAE system programmers are not burdened with the task of reviewing the computer graphics literature and implementing algorithms, and 2) with an abstraction barrier between the CAE system and the renderer, the CAE system safely benefits from whatever efficiencies the implementors of the renderer obtained by exploiting special hardware and/or clever or ugly software.

Comprehensive discussions of rendering can be found in numerous graphics textbooks [Foley and Van Dam 1982; Rogers 1985].

6.3.4. Prospective Surfaces

Surfaces that do not exist in the real world are referred to by the user interface as *prospective surfaces*. In fact, these are represented with instances of exactly the same class (SURFACE) as *surveyed surfaces*. Prospective surfaces are created by sending a message to an existing surveyed or prospective surface with a boundary polygon for the new surface. This polygon is projected onto the existing surface and its vertices, along with "enough" (see below) other points to make a good seam, are given elevations interpolated by the existing surface. These points become the first surveyed points of the new prospective surface.

Additional surveyed points will be added either by the user explicitly, by the user sketching contour lines, or by SITE CONTROLLER monitoring earthmoving, for example, in order to reflect the path of a bulldozer's blade. Calculating the "best" surface from contours is a research problem [Meyers 1992] and one that I made no attempt to solve. My code simply approximates contours by surveyed points at regularly-spaced intervals and hopes that the Delaunay triangulation will result in something reasonable. The user is free to add, delete or move points if he doesn't like what he sees.

One of the toughest challenges here is figuring out how many surveyed points to string along the boundary of the prospective surface, especially since the prospective surface is usually going to be sewn together with the original surface at some point. What would be particularly undesirable is for elevations within the new surface boundary to be interpolated based on surveyed points outside the boundary. This can be prevented by ensuring that no triangle cuts across the boundary. A brute force solution is just to add points every foot and expect that the standard Delaunay triangulation will not result in any cross-boundary triangles. However, adding this much data might make the system unacceptably slow, especially on 1 MIPS Symbolics machine. Thus, the method works by adding a surveyed point wherever the new surface boundary cuts a triangulation edge of the old surface. When surfaces are sewn together, the edges along the boundary are added to the list of forced connections. Thus, triangles do not cut across the boundaries and enough intermediate points have been added that we don't have a long skinny triangle along the side of the boundary polygon. Figure 6.6 illustrates the process of defining a prospective surface in detail.

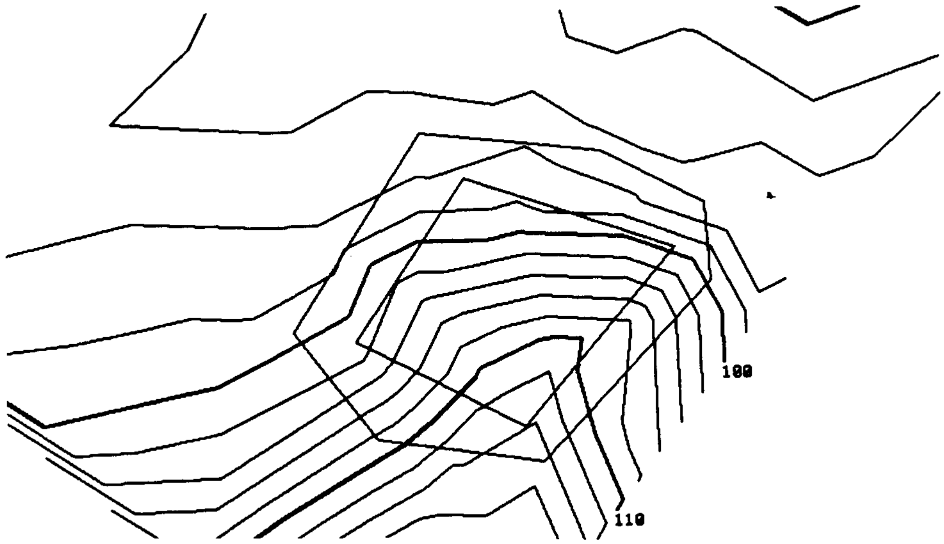

Figure 6.6a Designing a prospective surface begins with the specification of the boundary seam (outer polygon). Then the user sketches in new contours for the interior, which for this building pad consist only of a single contour at 105.7′.

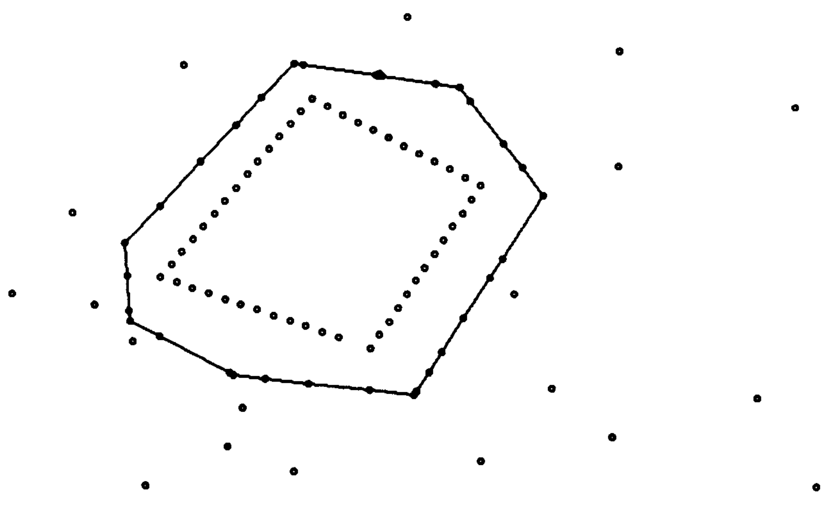

Figure 6.6b Sewing together the building pad with the original surveyed surface requires ensuring that triangulation never cuts across the boundary polygon. Thus, elevations inside the boundary are completely determined by surveyed points inside the boundary and elevations outside the boundary (i.e., on the original surface) are unaffected by the new contours. To accomplish this, the points along the boundary are connected with forced connections (the edges shown here).

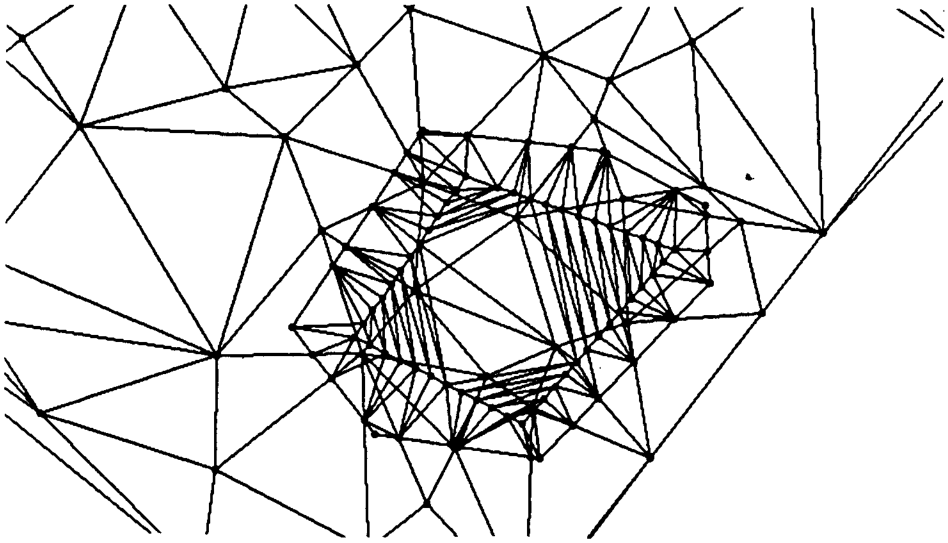

Figure 6.6c Graphically overlaying the new building pad surface with the original surveyed surface, we see that all the "extra points" along the boundary polygon are positioned at each intersection of the boundary polygon and the original surface's triangulation edges.

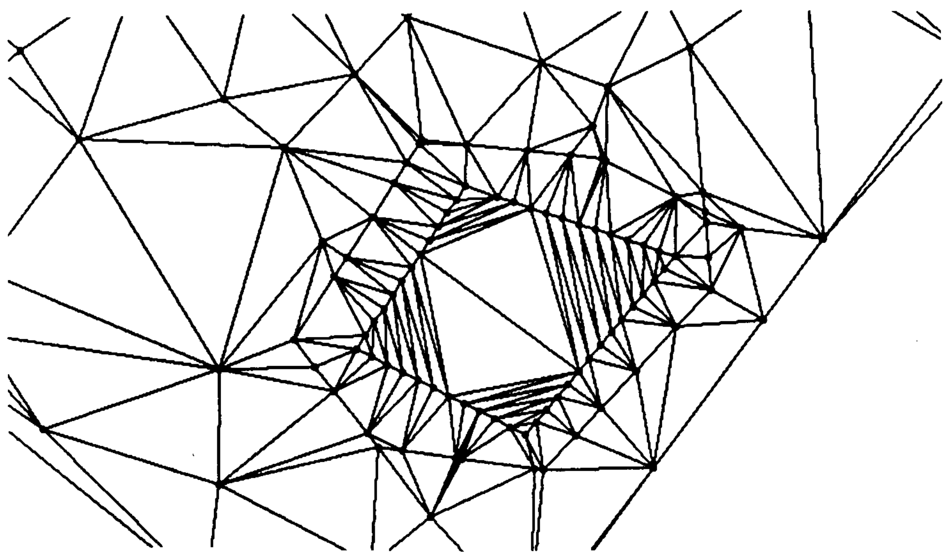

Figure 6.6d Here is the triangulation of the combined surface. Note that, as advertised, no triangle cuts through the boundary polygon and none of the triangles outside the boundary polygon are too long or skinny.

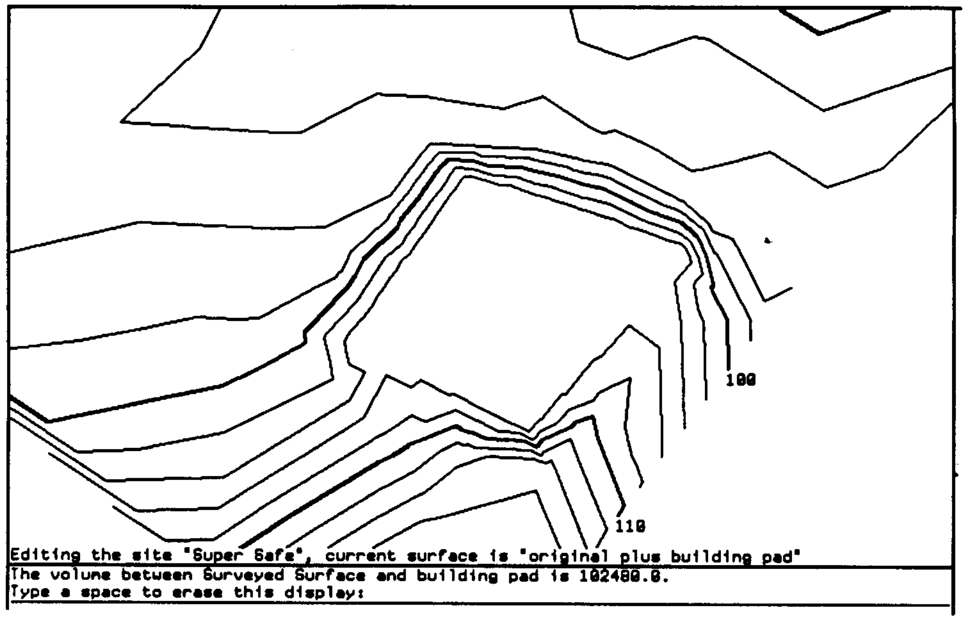

Figure 6.6e Contouring the finished surface reveals a large flat area suitable for building. Note that SITE CONTROLLER has calculated the overhaul, i.e., the net amount of earth that will have to be trucked onto the site to realize the building pad.

6.3.5. Volumetric Computation with Surfaces

Given an actual and a prospective surface, it is natural to ask questions about the cost of realizing the prospective surface. How much earth must be moved? What is the *overhaul*, i.e., the net amount of earth that will have to be brought onto or hauled away from the site? Can the overhaul be reduced to zero simply by moving a building pad up or down?

In computing the overhaul, one is asking the size of the faceted volume between two surfaces. This volume is probably not convex or even contiguous. However, one need not ever consider the volume per se, but only compute the height of the surface above sea level. This is done with five lines that loop through every triangle in the surface, multiplying the area of the triangle's projection onto the x-y plane by the elevation of the triangle's centroid and summing each product. If the two surfaces have the same boundary polygon, subtraction is all that remains to be done. If the two surfaces are not co-extensive in the plane, the smaller one is projected onto the larger one and just that piece of the larger one is carved out for use in the computation:

```
(defmethod (:volume-between-you-and-larger-surface surface) (larger-surface)
  (let* ((other-surface-piece (send self :piece-of-larger-surface-below-self
                                     larger-surface))
         (my-volume-above-sea-level (send self :volume-above-sea-level))
         (other-volume (send other-surface-piece :volume-above-sea-level))
         )
    (- my-volume-above-sea-level other-volume)))

(defmethod (:piece-of-larger-surface-below-self surface) (larger-surface)
  (let* ((my-boundary-points (send boundary-polygon :vertices))
         (projected-boundary-points
          (send larger-surface :project-surveyed-points my-boundary-points))
         (real-points-inside-projection
          (send larger-surface
                :surveyed-points-within-polygon boundary-polygon))
         (copied-real-points (loop for point in real-points-inside-projection
                                   collect (send point :copy-self)))
         (combined-points (append projected-boundary-points
                                  copied-real-points))
         (new-surface (make-instance 'surface :name
                                     (format nil "Piece on ~a below ~a"
                                             larger-surface self)))
         )
    (send new-surface :set-surveyed-points combined-points)
    new-surface))
```

When determining the difficulty of a project, a standard civil engineering drawing is the *cut and fill* diagram. This shows the areas of the site where earth must be removed (cut) shaded differently from those where earth must be added (filled). Again I wimped out, but this time by imposing a rectangular grid over the area in question, asking each surface for its elevation at each grid vertex and then declaring that grid vertex to be either cut, fill, or neutral as appropriate (see Figure 6.7).

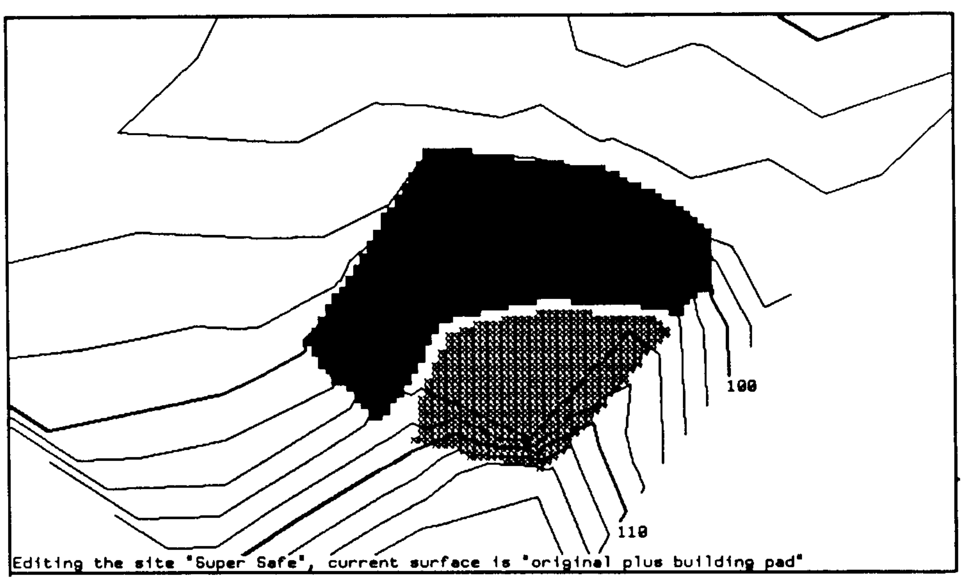

Figure 6.7 Cut and fill diagram for adding the prospective surface building pad to the original surveyed surface. Computation was performed on a 64x64 grid. Areas that must be filled are shown in black; those that are to be cut are shown in gray.

Representing terrain with a fixed rectangular grid calls to mind all the primitive limited FORTRAN surface modelling systems of the 1960's and 70's. A fixed grid does not satisfy any of the desiderata I've given for a good surface model. In particular, it cannot represent complex areas of the site at a finer level of detail and it cannot reflect the topology of natural terrain. However, grids enormously simplify many algorithms.

Attacking the cut and fill problem directly with the two triangulated surface models would involve many nasty triangle projections and intersections. A programmer might be kept busy for days working this out. However, SITE CONTROLLER surfaces already have a method for interpolating the elevation at arbitrary points. Using this method to calculate elevations at grid points for both the hypothetical and actual grids requires less than 50 lines of code. The resultant two grids can be compared trivially. If the hypothetical grid is higher than the actual grid at one point, that point is colored black for fill; if lower, gray for cut; if nearly equal, white for neither cut nor fill.

Although one might argue that this approach is inaccurate, consider the fact that output devices only have so much resolution. Once the grid is fine enough so that each pixel is covered by a single grid vertex, no better diagram can be produced. Producing this elaborate intermediate data structure, i.e., the grid, and then throwing it away seems computationally wasteful, but if this approach is fast enough to satisfy the user, spending extra programmer days or weeks on a more elegant algorithm is pointless.

In general, many problems lend themselves naturally to gridded solutions. Before tackling a difficult computational geometry problem with triangulated surfaces, it is often worth asking if a straightforward method using grids might not work just as well.

6.3.6. Intervisibility

A comprehensive intervisibility algorithm answers the following questions: (military) *Could an enemy on that hill see a tank dug in behind this ridge?* (urban planning) *Could the occupant of a third floor bedroom in this house see the city dump?* (architecture) *What kind of light would this basement room get?* (town council) *How many people in this condo development could see the red light district from their balconies?*

Point-to-point intervisibility questions are easily answered by intersecting a query line from *p1* to *p2* with the surface. Given a piecewise-planar surface model, an efficient algorithm computes the intersection of the planar projection of the query line with the model's triangle edges. If the elevation of any edge at the intersection point is higher than the line, then there is no intervisibility. If the query line is higher than each edge at its intersection points, then the view from *p1* to *p2* is unobstructed. (Note: we can safely ignore data within the triangles because we assume a piecewise-planar surface model and therefore the extremes must occur at edges.)

I implemented the preceding point-to-point intervisibility algorithm as a method of the class SURFACE, but could not think of an efficient solution to overall intervisibility problems. Recently, I uncovered a large body of work that has been done in the area of shadow generation for computer graphics [Woo 1990]. Shadow generation asks the question "Which parts of the surface are hidden from the light source?" If one positions the light source at a potential viewer's location, then the same algorithm answers the question "Which parts of the surface are invisible to the viewer?"

Shadow generation algorithms that are useful crutches for intervisibility calculations are primarily ones that postulate a static world and movable light sources. A good example of the genre is in [Chin and Feiner 1989] where a Shadow Volume Binary Space Partitioning tree is used to organize polygonal terrain elements so that shadows can be computed in $O(n)$ time. Each polygon must be examined once, but its interactions with all the other polygons need not be considered.

6.3.7. Convex Hull

An implementation of Graham's scan determines the convex hull of a set of points in the plane. (Remember that a convex hull is the smallest convex polygon that contains a set of points in the plane—shaped like a rubber band stretched around a bunch of pins, as in Figure 6.1.) In particular, this algorithm works in $O(n \log n)$ time on any set of points that have methods to handle messages such as :2D-DISTANCE and :2D-ANGLE to other points. The algorithm runs as follows:

1. Find an internal location q (this can be done by taking the centroid of any three points in the set S)
2. Sort the points of S by polar angle and distance from q and arrange them into a circular doubly-linked list, with START equal to the rightmost, smallest y coordinate point (certainly a hull vertex).
3. Repeatedly examine triples of consecutive points in counter-clockwise order. If $p_1 p_2 p_3$ forms an angle of greater than 180°, call it a "right turn" and throw out the middle point from consideration because it must be an interior point; now back up and consider the angle formed by $p_0 p_1 p_3$. If $p_1 p_2 p_3$ forms a "left turn," declare the first point to be part of the hull and advance the scan to consider the next three points ($p_2 p_3 p_4$).
4. Stop when we get back to the START.

The implementation of Graham's Scan is listed below in order to illustrate the compactness of Lisp in specifying computational geometry algorithms.

```
(defun graham-hull (point-set-1)
  (let* ((point-set (copy-list point-set-1))
         (internal-point (find-internal-point point-set))
         (sorted-points
          (sort-about-point internal-point point-set))
         (circular-list (make-circular-list sorted-points))
         (starting-point (find-starting-point (aref circular-list 0))))
    (loop with vertex = starting-point
          ;; these will be updated with every iteration
          for next = (send vertex :next-element)
          for next-next = (send next :next-element)
          with have-we-ever-advanced-beyond-start? = nil
          ;; we stop when we've once gotten beyond the start
          ;; but we are now back there
          until (and have-we-ever-advanced-beyond-start?
                     (eq next starting-point))
          do
       (cond ((left-turn? vertex next next-next)
              (when (and (not have-we-ever-advanced-beyond-start?)
                 (eq vertex starting-point))
              ;; we're at the starting point and about to advance
              (setq have-we-ever-advanced-beyond-start? t))
              (setq vertex next))
             (:else
              (send next :delete-self)
              (setq vertex (send vertex :previous-element))
             )))
    (send starting-point :collect-all-values)
    ))
```

Note that the code is greatly simplified by the ability of the points to take care of themselves. For example, this program only has to refer to the circular list once when getting the starting point. After that, navigation through and modification of the data structure is performed by sending messages to points in the data structure. For a fuller explanation of the algorithm, albeit with a serious error (corrected in the preceding code fragment), see [Preparata and Shamos 1985]. The original algorithm description may be found in [Graham 1972].

6.3.8. Hydrology

The study of water flow on and below the surface of a site is called *hydrology*. Surface water can do tremendous damage to earthworks and structures; groundwater can carry toxins to remote locations in unexpected ways. Thus, hydrology is one of the most important kinds of analysis to support. I implemented some very primitive hydrology code that fails miserably due to a combination of naiveté and the piecewise-planar nature of the default surface model. Points are declared to be peaks if they are higher than all the points to which they are connected; points are pits if all their neighbors are higher. For each edge in the triangulation, the code calculates the gradient, i.e., direction of fastest ascent, for both adjacent triangles. Next, the cross product of the projection of the edge in the x-y plane is computed with both gradients. If the cross products have z components with different signs, the gradients either both point in and the edge is part of a ridge, or both point out and the edge is part of a stream.

Intuitively, these local methods should produce a nice global map, with peaks, passes and pits marked amidst a collection of streams and ridges that would naturally lead a human to see the drainage basins. What happens in practice is that water flows in sheets across these planar triangles and the ridges and streams are discontinuous, as shown in Figure 6.8.

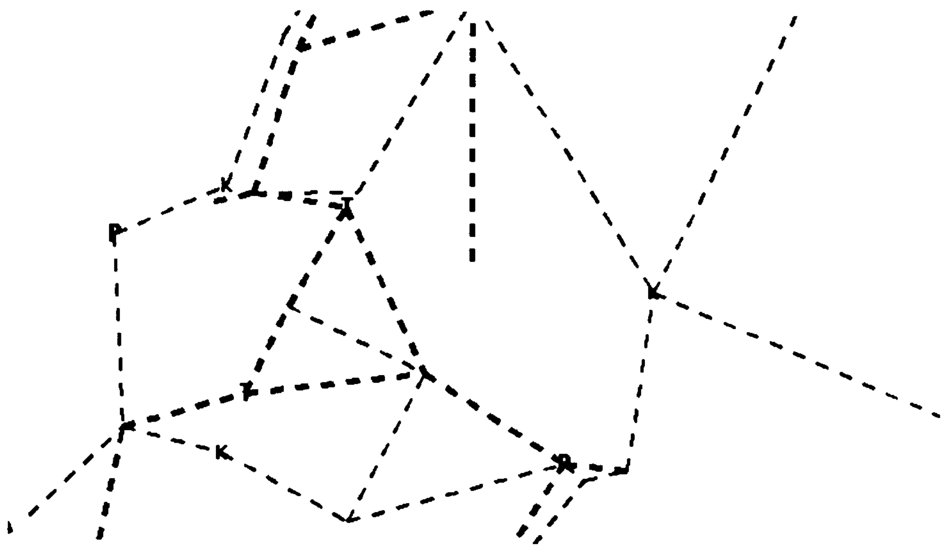

Figure 6.8 Hydrology map computed simplistically on a C^0 piecewise-planar surface. Note that the ridges (thin dashed lines) and streams (thick dashed lines) do not form coherent networks. This is a consequence of only triangulation edges being ridge or stream candidates. Points are classified as peaks (K), pits (T), or passes (P).

I am currently considering ways to compute better hydrology maps either from the C^1 surface model or by using more clever algorithms on the piecewise-planar surfaces.

6.3.9. Traditional CAD subsystem

Although a computer-aided *engineering* system is superior to traditional computer-aided *design* (CAD) systems in its knowledge of what is being modelled, and hence in its ability to support analysis, sometimes one simply wants to sketch a picture or add a piece of text to a drawing. SITE CONTROLLER supports this via *geometry bags*, which are simply collections of graphical entities about which the system knows nothing. This is just what a traditional CAD system gives the user, although SITE CONTROLLER provides a tree-structured hierarchy of named geometry bags where CAD systems normally offer only a flat choice of numbered layers.

Currently, the only graphical entities implemented are text, segments, splines and circular arcs. Text can be in any size and orientation from any vector font (see §6.3.20). The CAD system snaps most mouse clicks to nearby "significant points" of existing graphical entities (e.g., a user who starts a new line segment 3 pixels from an endpoint of a circular arc will find that the segment starting point has been snapped to the circular arc endpoint—if he doesn't like this, he can always zoom in and work at a finer resolution).

6.3.10. FRAME code

The Lisp Machine window system thinks of the world as consisting of one or more screens. On each screen is a single *frame*. A frame is a large window that breaks up the screen for its *panes*, which are daughter windows. None of these panes overlap and the amount of space each occupies is determined by applying a series of constraints to the screen size. Thus, the frame is usually referred to as a constraint frame.

SITE CONTROLLER's primary constraint frame is the center for user interface and, after establishing itself on the main screen, grabs control of any other screens (typically color). Keyboard commands all go to this frame and include commands for printing help, undoing recent actions, saving data structures out to files, loading files, accessing utility functions, changing the configuration of the frame, and resetting the system.

All the panes (subwindows) in this frame inherit from the class LAND-HACKER-PANE, which has methods to forward user interface messages, process mouse clicks in a uniform way, and pick an appropriate sibling pane for interaction with the user. The most important panes are instances of the class SITE-VIEW-PANE, which combines basic surface and site viewing classes with specialized capabilities from such classes as MINE-VIEW-MIXIN. As a diehard believer in the modeless religion of EMACS and the Macintosh, one of my saddest days implementing SITE CONTROLLER was the day I added a command to this class that lets the user select among six modes:

- standard CAE (surface definition and volumetric calculation)
- actor simulation (planning excavation—see §6.5)
- logistics (an extension of SITE CONTROLLER to help the U.S. Army lay out field ammunition dumps—not described in this report)
- location system simulation (another undescribed extension built to aid development of a radio frequency location system)
- real-time vehicle control (monitoring and assisting earthmoving—see §6.6)
- mine planning (figuring out where to send a dragline to expose coal seams—see §6.4).

Despite my aversion to modes, I concluded that there was simply no way to fit all the possible commands into the mouse command datastructure without both running out of shift keys and hopelessly cluttering the user interface.

Statistics panes are available to show results of computations or dynamic information as the user moves the mouse. Packet display panes graphically display communications with vehicles. Perspective views show views from arbitrary positions of site surfaces or a display, updated in real time, of the view from a vehicle. Bar graph and semicircular gauges are available to display vehicle performance and loads on implements. A special zoning law database editor window is available as well as a vector font editing utility pane.

Overall, 17 panes appear in different combinations and relative positions on the main screen in a total of 10 separate configurations. Color screens may have either one large or four small site view panes visible at once.

6.3.11. Site Data Structure

As with surfaces, the class definition, database storage and user interface code for site data structures are all automatically generated from one big list. Site instances contain information on the name of the site, the customer who is interested in the site, the size and political location of the site, parameters for surface visualization, data structures for managing surfaces, data structures for managing lot boundaries, a list of geometry bags for miscellaneous annotations, a quad tree organizing site features that are not properly associated with any surface, a graph of roads on the site, and a list of utility lines running through the site.

Methods of the class SITE mostly have to do with maintaining a site's data structures and providing a user interface for both data structure inspection and manipulation. Very little analysis is done by methods of SITE.

6.3.12. Site Feature Quadtree

Many features in a site may be modelled formally and geometrically but do not properly belong to any single surface. Examples of such features include bodies of water, barrels of hazardous waste, groves of trees, and buildings. These must be modelled formally to permit analysis, i.e., a grove of trees should be represented as an instance of TREE-GROVE rather than as a bunch of lines in a geometry bag. On a large site that is modelled in detail with many features it would be nice to know which features we must consider when working in a particular location.

The Site Feature Quadtree organizes the site features so as to permit relevant features to be located in $O(\log n)$ time, where n is the number of features. A quadtree is a tree where every node that is not a leaf has four children. Each node claims a rectangle of space in the plane and it divides that space among its children by splitting the rectangle with two lines, usually equally so that each child gets one fourth of the space (see Figure 6.9). A rectangle of space encompassing the entire site is allocated to the root node. When trying to find the leaf that surrounds a query point, one need only start at the root and ask two questions: Is the query point to the left of the vertical splitting line? Is the query point below the horizontal splitting line? The answers to those two questions direct the search to one of the four children and each $O(1)$ comparison results in three fourths of the search space being eliminated, thus resulting in $O(\log n)$ for the complete search.

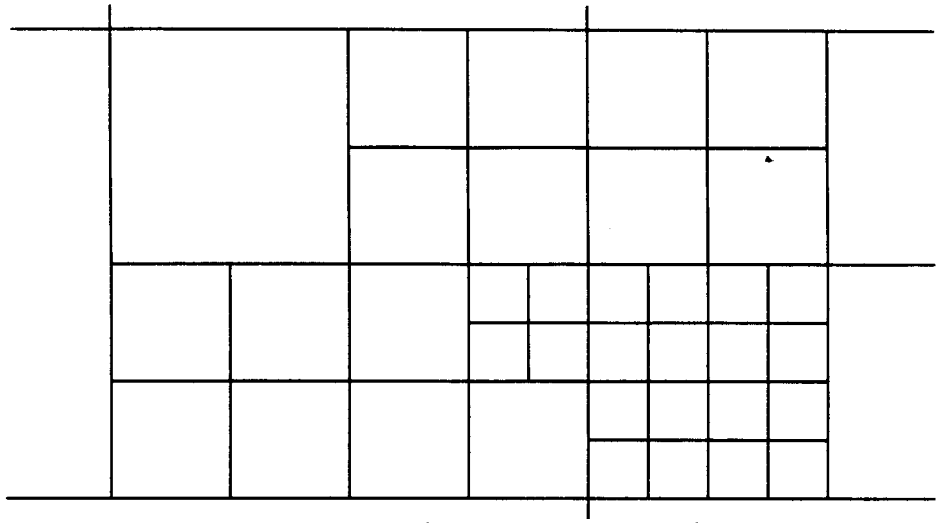

Figure 6.9 Sample quadtree. Note that different areas of the site are represented at different levels of detail.

Every site feature contains an *area of interaction*, represented either as a circle or a polygon. Any query point within the area of interaction may theoretically by affected in some way by the site feature. For example, the area of interaction for a pond might be the area flooded by the pond during a 100 year storm. A site feature is stored at the smallest quadtree node that completely contains its area of interaction. Thus, a very large feature might be stored at the root node (or even a small feature that happened to overlap a splitting line).

Finding all the site features that might affect a query point is as simple as traversing the quadtree to the smallest containing leaf node, collecting all the site features on the way down. If too many site features congregate at nodes too high up in the tree, there are standard algorithms for choosing splitting lines so that the data are distributed better [Samet 1984]. However, I never implemented any optimizations because I never encountered a site where even linear search would have been a problem.

6.3.13. C^1 Surface Model

SITE CONTROLLER's standard triangulation of points of known elevation may be employed to generated C^1 and C^2 models, i.e., surface models continuous through the first and second derivatives. Imagine that each triangle is a patch of a surface whose elevation is determined by a polynomial with nine or more terms. Different coefficients are selected for each triangle and thus each triangle is a patch of a different continuous surface. It is possible to set the coefficients such that first and even second derivatives are continuous across the triangle edges, i.e., the boundaries between the patches. The resulting surface is much smoother than a piecewise-planar surface, *but not necessarily more accurate* (this is why SITE CONTROLLER does not waste precious Lisp Machine cycles computing everything with smooth surfaces).

Continuity issues are of tremendous importance to airplane and automobile designers. Whether a surface is C^1 or C^2 may affect the aerodynamics, ease of fabrication, and visual appeal. A C^2 model, where the second derivative is everywhere continuous, might at first be thought superior in every case. However, curve fitting polynomials tend to behave perversely. Restrictions in one region result in absurd oscillations and bulges in other areas. Further complicating matters is the current vogue in G^2 surfaces, where the direction of the gradient remains constant across patch boundaries, but the magnitude need not. By relaxing the restriction that continuity be maintained for magnitude, the surface oscillates less while appearing just as smooth to the human eye.

Fortunately for computer-aided civil engineering programmers, a bulldozer is not exactly a precision instrument. Continuous surface models are useful mainly for drawing smooth contour maps that please the user's eye and when performing certain kinds of computation that degenerate on a C^0 surface.

There is a tremendous body of literature on surface modelling, reflecting the fact that an infinite variety of polynomials that will fit any given data set. Most of the methods aim to minimize a metric of "peculiarity," such as oscillation, torsion or tension. For example, consider ten points in a horizonal line in the plane, e.g. (0,1), (1,1), (2,1), etc. A 15th-order polynomial that went through all ten points, but did so by oscillating between -100 and +100 would not be considered by most people to be as good a fit as a 15th-order polynomial that looked more or less like a horizontal line.

An interesting illustration of the dramatic changes that can be wrought in an interpolated C^1 surface merely by changing a tension parameter can be found in [Montefusco and Casciola 1989]. A conference symposium [Rice 1977] contains a wealth of classic papers, including a formal justification for choosing particular triangulations by C.L. Lawson, a survey of surface modelling with an emphasis on triangle-based interpolations of irregularly spaced data by Robert E. Barnhill, and an attempt to automatically compensate for incompatibilities among floating-point implementations by W.S. Brown.
Smooth contouring in SITE CONTROLLER is performed using a good basic C^1 algorithm described in [Nielson 1980]. Each triangle has a different nine-parameter polynomial. First derivative continuity is accomplished by ensuring that interpolated elevations very close to a triangle edge are almost entirely determined by the elevations of the two endpoints of that edge, i.e., elevation data from the "third" point of each triangle does not contribute to interpolations along the opposite edge.

A new data structure is created to support Nielson's interpolant, an INTERPOLATING-TRIANGLE, which inherits from the basic TRIANGLE class but adds a cache for 15 interpolant parameter values and a list of child triangles. The child triangles are used for contouring and rendering. Rather than write software to calculate the intersection of a contouring plane and a nine parameter polynomial surface, then scan convert that intersection curve into pixels, a grid of subtriangles is imposed on each interpolating triangle. The interpolant is used to calculate the elevation of each subtriangle vertex and piecewise-planar interpolation is used from that point on to generate piecewise-linear contour lines. By making the subtriangle grid fine enough, any desired degree of smoothness may be achieved up to the maximum resolution of the graphics output device. There are formal methods of determining the appropriate grid density [Cheng 1992], but I did not implement any of them.

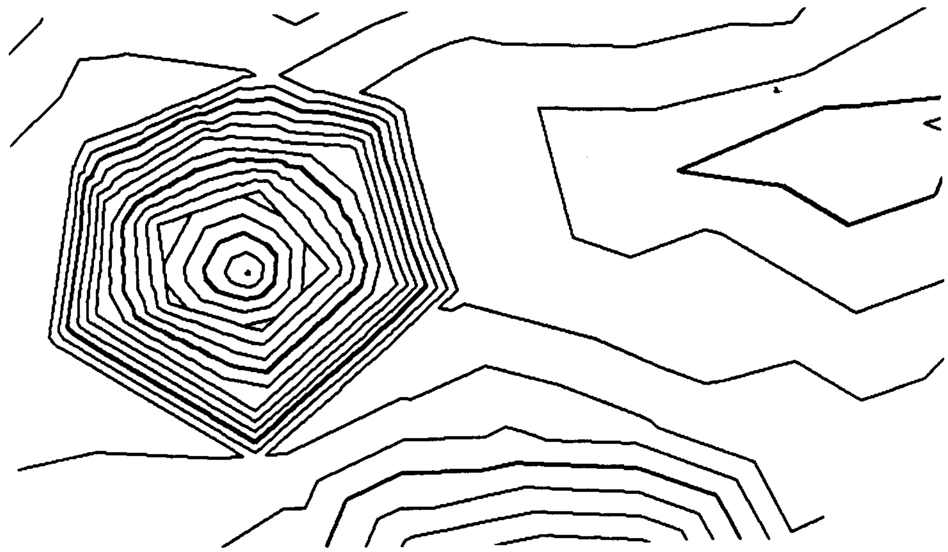

Figure 6.10 Contours from the standard (and efficient) piecewise-planar surface model.

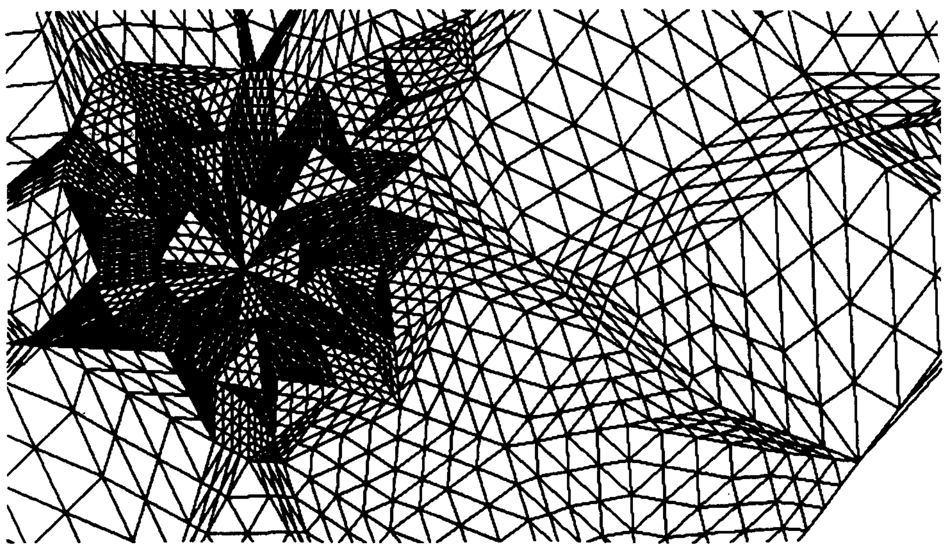

Figure 6.11 Each triangle in the original piecewise-planar triangulation becomes an INTERPOLATING TRIANGLE and calculates a polynomial surface patch. Each large triangle creates a regular grid of subtriangles and determines the elevation of each vertex in the fine grid with the polynomial interpolant. Contouring is done on the piecewise-planar model defined by the subtriangle grid.

Figure 6.12 Contours of the C^1 surface with two levels of subtriangles.

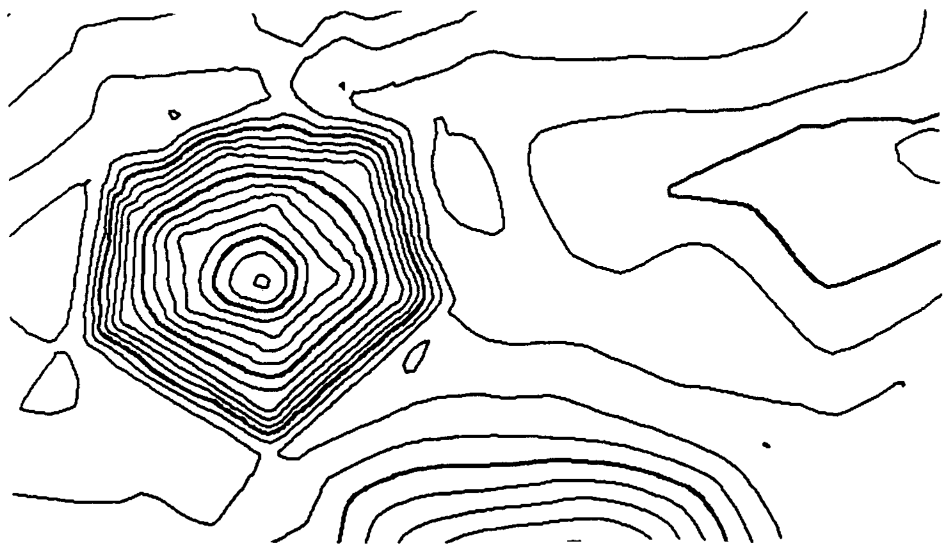

Figure 6.13 Contours of the C^1 surface with three levels of subtriangles.

6.3.14. Drawing a Scale

As is so often the case in software development, problems that one thinks will be very challenging turn out to have simple solutions. Conversely, problems that should be trivial require 100 lines of ugly code that takes days to write. A good example of this situation is the software that draws a scale on a map of the site, i.e., one of those blocks of alternating black and white rectangles with a "1:1000" label on top. Figure 6.14 shows a typical SITE CONTROLLER view with a scale.

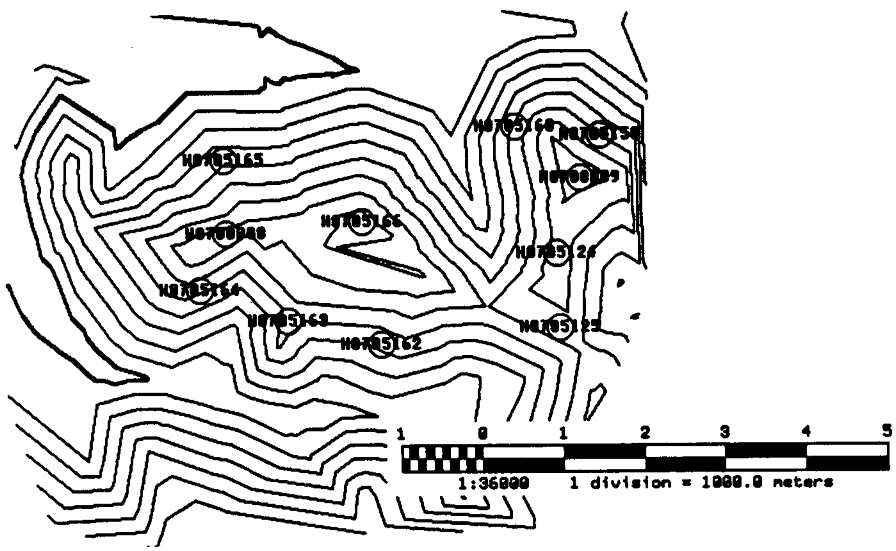

Figure 6.14 View containing a scale—the contours are the topography of a coal mine plus markers for locations where boring samples have been taken.

Probably the best way to see just how ugly this problem was is to examine a piece of the code:

```
(defmethod (:draw-self map-scale) (center-x center-y)
  (multiple-value-bind (extent number-gross-divisions
                               extent-per-division five-or-ten)
                       (send self :calculate-scale-parameters)
    (let* ((text-height-above-scale-in-pixels
             (+ (send self :scale-rectangle-height) 10))
            (text-height-below-scale-in-pixels
             (+ (send self :scale-rectangle-height) 10))
            (text-height-above-scale (send map :scale-dimensions-inverse
                                               text-height-above-scale-in-pixels))
            (text-height-below-scale (send map :scale-dimensions-inverse
                                               text-height-below-scale-in-pixels))
            (scale-height (send map :scale-dimensions-inverse
                                (* 2 (send self :scale-rectangle-height))))
            (height-to-be-erased (+ text-height-above-scale
                                    text-height-below-scale scale-height))
            (bloat-width-pixels 12)
            (bloat-width (send map :scale-dimensions-inverse
                                   bloat-width-pixels))
            (width-to-be-erased (+ (* 2 bloat-width) extent
                                   extent-per-division))
            (text-center-y (+ center-y text-height-above-scale))
            )
      (send map :show-rectangle width-to-be-erased height-to-be-erased
            (- center-x extent-per-division bloat-width)
            (+ center-y (* 1.5 text-height-above-scale )) :erase)
      (loop for i from -1 to 5
            for draw-x first (- center-x extent-per-division)
            then (+ draw-x extent-per-division)
            do
            (send map :show-centered-string
                  (format nil "~d" (if (eq five-or-ten :five)
                                       (abs i)
                                       (* 2 (abs i)))) draw-x text-center-y))
      (send map :show-centered-string
            (format nil "~a    1 division = ~6f meters"
                    (send self :string-description-of-scale)
                    extent-per-division)
            (+ center-x (* 2 extent-per-division))
            (- center-y text-height-below-scale))
      (multiple-value-bind (wcenter-x wcenter-y)
                           (send map :p-to-w center-x center-y)
        (send self :draw-rectangles wcenter-x wcenter-y extent
              number-gross-divisions extent-per-division)))))
```

It is well to keep in mind that not only does the code included here require numerous helper procedures, but that it does not even do any of the rectangle drawing.

6.3.15. Ruler

If the user wants to measure something on the site, a scale can be helpful, but a mouse-manipulable *ruler* lets him see exactly how far apart two points are and the angle between them. When ruler measurement is selected, SITE CONTROLLER goes into a modal state where moving the mouse moves the ruler and moving the mouse with a button held extends or rotates the ruler. This code defines four unnamed procedures, each containing a lexical reference to the recently created ruler object . These procedures are organized into a matrix and passed down to the generic mouse handler defined in the PHILG system. Here then is an example of higher-order procedures at their best when constructing user interfaces:

```
(defmethod (:generate-ruler ruler-mixin)
           (&optional ruler-start ruler-end window-to-display)
  (unless (and ruler-start ruler-end)
    (multiple-value (ruler-start ruler-end)
      (send self :initial-ruler-position)))
  (let* ((ruler (make-ruler ruler-start ruler-end
                            self window-to-display))
         (move (lambda (wdx wdy)
                 (multiple-value-bind (pdx pdy)
                     (send self :scale-dimensions-inverse wdx wdy)
                   (send ruler :translate pdx (- pdy)))))
         (rotate (lambda (wdx ignore)
                   (send ruler :rotate (/ wdx 80.0))))
         (extend-left (lambda (wdx ignore)
                        (let ((pdx (send self :scale-dimensions-inverse wdx)))
                          (send ruler :extend-end (- pdx)))))
         (extend-right (lambda (wdx ignore)
                         (let ((pdx (send self :scale-dimensions-inverse
                                          wdx)))
                           (send ruler :extend-start pdx))))
         (operation-matrix (make-array '(2 2 2 2)))
         (first-entry
          (list move extend-left rotate extend-right
                "No buttons: drag L: extend left M: rotate R: extend right")))
    (setf (aref operation-matrix 0 0 0 0) first-entry)
    (send self :generic-mouse-handler operation-matrix)
    (send ruler :kill)
    ))
```

6.3.16. Undo System

Every time an undoable operation is performed, a object of the class UNDO-SPECIFICATION is pushed onto a list by the user interface. The facility is very simple because all the information about how to do something is encapsulated in a procedure stored in an instance variable of the undo specification object. The object also contains a description of the operation to fit into the sentences "Undo ____?" and "Redo ____?" and a toggle flag that indicates whether the operation is to be undone or redone. Multiple undo is supported by moving down the list.

6.3.17. Lot Boundaries

When a civil engineer is subdividing a large tract of land into numerous lots for houses, the customer's main goal is to squeeze as many houses into the tract as is legally possible. Furthermore, road design within the subdivision is intimately intertwined with the lot layout. SITE CONTROLLER provides a primitive facility for modelling and changing lot layouts.

The only classes are LOT-BOUNDARY-END-POINT and LOT-BOUNDARY, and instances of the two are connected together via instance variables to form a doubly-linked list. Provision is made for lot boundaries that are splines or circular arcs, but only line segments have been implemented. SITE-VIEW-MIXIN contains some rudimentary methods that allow the user to sketch in a lot layout, move lot boundary end points, and calculate lot areas.

I have not implemented the linkage among the zoning law database, road modelling datastructures, and lot boundary data structures that would have made zoning law compliance analysis possible.

6.3.18. Zoning Law Tree

> *The young man knows the rules, but the old man knows the exceptions.* — Oliver Wendell Holmes

A tree editor substrate allows a window to graphically display tree data structures, automatically makes nodes mouse-sensitive, lets the user navigate the tree, and provides mechanisms for user editing of both tree structure and data associated with nodes. The tree editor is capable of manipulating all trees whose nodes are able to respond to a small set of messages, e.g. :PARENT, :CHILDREN, :DRAW-SELF, :HIGHLIGHT-SELF. In practice, all the trees in SITE CONTROLLER use nodes that inherit from the classes BASIC-NODE, which contributes instance variables and methods for the basic data structure, and DISPLAY-NODE-MIXIN, which contributes instance variables and methods for positioning and displaying nodes recursively on the screen.

Using the basic classes described above, zoning laws are represented by a tree of political entities with authority over particular tracts of land, associating with each node rules such as minimum lot area, minimum septic setback, minimum frontage, and whether frontages may be summed on a corner lot (Figure 6.15).

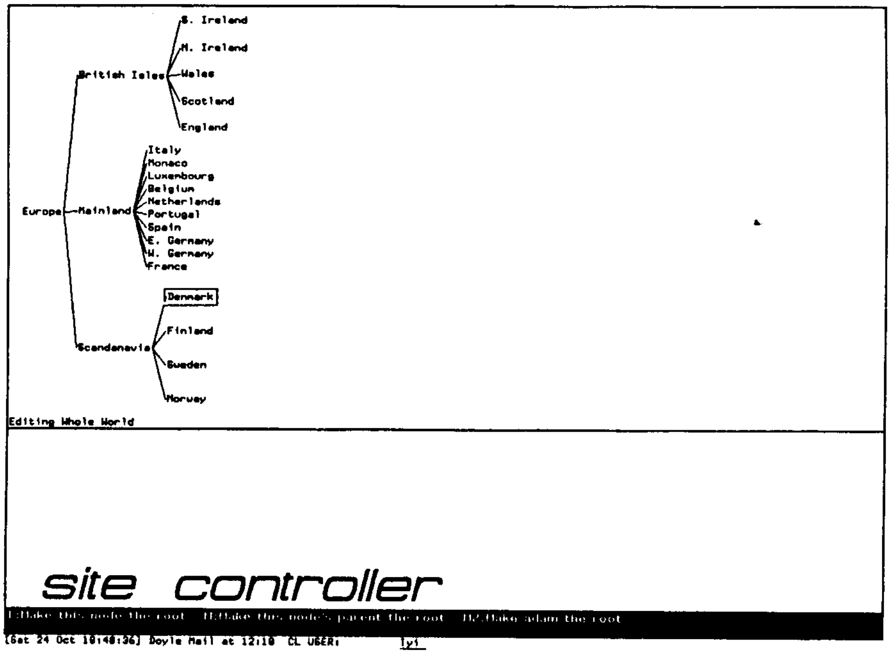

Figure 6.15 SITE CONTROLLER configured for editing the zoning law database for Europe.

Each rule in the zoning law database is specified by a macro invocation of the following form:

```
(make-lot-parameters-mixin-spec minimum-lot-area -infinity t :number)
```

This says that each node shall hold a statutory minimum lot area, that the default value shall be minus infinity, that the value is user-settable, and that the user interface software should ensure that any new value is a number. This specification relies on the default method for combining values from different nodes, which is to collect all the values from a node and its ancestors into a list, then choose the maximum from the list. What allegedly distinguishes an expert system from a mere program is that the expert system can generate a human-readable justification for its conclusions. Thus, the zoning law node class incorporates a mechanism to explain which political entity is the limiting one when responding to a query about a requirement.

6.3.19. Utility Lines

Utility lines are represent as objects with a description of the contents of the line, the diameter of the pipe, and a 3D path of line segments and circular arcs. Utility lines can draw themselves in site views. I never implemented the real-time interference checking software that would be nice for controlling earthmoving equipment (especially in light of the number of backhoe operators who are killed each year when they accidentally strike gas lines).

6.3.20. Vector Fonts

Because the Lisp Machine window system only supported bit-mapped fonts, which are not drawable in arbitrary orientations, I implemented code for defining and displaying vector font characters. A vector font object contains the following: the font name, a hash table mapping *strings* to vector font characters, and a default bounding box that guides the user in defining new characters that are about the same size as all the others in a font. Note that strings, not ASCII characters, are mapped to vector font characters. Thus, descriptive names for characters are possible such as “bulldozer” or “brick depot”.

The vector font editor is a mode of the SITE CONTROLLER system wherein the user is able to add new characters to any font by either sketching in an entirely new set of segments or by copying an existing character from any defined font.

6.3.21. Site View User Interface

The class SITE-VIEW-MIXIN is the heart of the SITE CONTROLLER user interface. Although methods of this class only implement mouse commands, 99% of the user’s interaction with SITE CONTROLLER is through methods of SITE-VIEW-MIXIN.

6.3.21.1. Substrate of Higher-Order Procedures

In order to reduce development effort, ease maintenance, and decrease window system dependence, the top-level user interface procedures rest on a tower of higher-order procedures. At the base is one of the most powerful—the method :GET-FROB-FROM-USER. This takes as arguments a CLOSEST-PROCEDURE, which will cough up the closest relevant object to any point in the projection plane, a documentation string that will tell the user what is expected of him, and information about when to abort and what to return after an abort. The only restriction on CLOSEST-PROCEDURE is that it return an object capable of responding to the :HIGHLIGHT-SELF and :UNHIGHLIGHT-SELF messages.

The basic user interface of SITE CONTROLLER is operator prefix, the opposite of the Macintosh interface. For example, on the Macintosh one selects an object with the mouse and then pulls down an operation from a menu. In SITE CONTROLLER, one specifies the operation to be performed with a mouse chord and then chooses one or more objects from those that are legal. A perennial problem on the Macintosh is *selection*, i.e., in a complex drawing the user’s intent in clicking the mouse on a particular pixel is unclear. Given the complexity of comprehensive site models, this problem would be 100 times worse in SITE CONTROLLER if not for the fact that the search space is restricted as soon as the operation is known.

The following illustrates the utility of this way of constructing user interface methods.

```
(defmethod (:get-triangle-from-user site-view-mixin)
           (&optional (doc "Click inside the triangle you want to select"))
  (check-site-and-surface)
  (send self :get-frob-from-user
        (lambda (x y) (send (send site :current-surface)
                            :triangle-containing x y))
        doc))
```

GET-TRIANGLE-FROM-USER takes a single optional argument, a documentation string to be displayed while the user is rolling the mouse around the site. The CHECK-SITE-AND-

SURFACE procedure displays a standard error message if this method is invoked on a window that is not currently viewing a particular site with a defined "current surface" (i.e., the one that is selected for editing). The meat of this method is in the CLOSEST-PROCEDURE, an unnamed lambda that takes arguments X and Y and asks the site's current surface to return the triangle containing the point specified by X and Y. Not only is this 7 line (actually only 5 on the wide Lisp Machine screen) procedure refreshingly simple for the programmer, but the selection problem is neatly side-stepped. We need not consider whether the mouse was clicked closer to a surveyed point or a triangulation edge and therefore if that might be what the user intended.

In a similar vein, there are procedures that query the user for surveyed points, triangulation edges, polygons, lot boundaries, lot boundary end points, site features, road segments, road intersections, road graph arcs, pieces of text, etc.

6.3.21.2. Mouse Commands

All the mouse commands defined by SITE-VIEW-MIXIN may be split up into the following categories: screen scaling, terrain model editing, lot-boundary related, utility, surface-visualization, site features, geometry bag, and miscellaneous.

The screen scaling category consists of an unremarkable collection of zoom, unzoom, zoom to fit, scroll, and refresh. The only mildly innovative idea here is the following: for scrolling, the user is presented with two pop-up menus; the first requests a direction (left, right, up or down), the second offers a predefined list of amounts; if the user waves the mouse out of the second menu, he is prompted in a typeout window to type in an arbitrary scrolling amount.

Terrain model editing commands constitute the largest category. There are simple commands for adding surveyed points and contours, more powerful commands that define prospective surfaces and calculate relationships among surfaces, a command to actually combine a real surface with one or more prospective surfaces, a management command to select the current surface on which other commands operate, and finally commands that allow sophisticated users to manipulate the data structures such as the list of forced connections.

Surface visualization commands allow the user to request a single, mouse-specified profile cut through the current surface or a pop-up 3D view from a mouse and keyboard-specified viewpoint. This category also contains commands to select which site is to be edited in this view, from that site which surfaces are to be displayed, and which site features and information are to be marked as well.

Utility commands provide a user interface to the ruler, scale and area-calculation facilities. However, the most interesting command in this category is the *dynamic mouse* facility. When selected, this lets the user roam over the site with the mouse while information about the terrain at the mouse position is dynamically updated in a statistics window. Information available depends on the mouse buttons and shift keys depressed, but includes 2D position, elevation on the current surface (either crudely or smoothly interpolated), and gradient on both the big triangles and the triangular mesh of the smooth interpolation data structures. This might be useful to a civil engineer quickly checking the position of site objects and proved very useful while developing software. The gradient display facility was implemented while debugging the smooth interpolation code—it was possible to actually see the gradient discontinuity across big triangle boundaries when I had misinterpreted Nielson's notation for naming edges (edge 1 is *opposite* vertex 1). Ten minutes of graphics programming sufficed to save several hours of head scratching and staring at code.

Site feature and geometry bag commands all serve to add to those parallel data structures. Miscellaneous commands include those having to do with roads and utility lines.

6.4. Mine Planning Extension

The extensibility of LAND HACKER and SITE CONTROLLER was really put to the test for two weeks when I wanted to extend the software to represent and perform analysis with 3D information about an Appalachian coal mine. Seams of coal were deposited in the Appalachias millions of years ago when the area was a lake bed. Thus, any individual coal seam is at about the same elevation above sea level for thousands of square kilometers. Where the mountains are high, the seam will run to the edge of the mountain and not exist again until one is across the valley on another ridge. Surface mining, a.k.a. *strip mining*, in the Appalachias involves ripping the tops off the mountains and hurling them into the valleys below until the coal seam is exposed, then moving in with shovels and trucks to cart away the meter-thick coal itself before ripping down another dozen meters to the next seam.

A *dragline* is the most important tool in this kind of mining. This machine throws a bucket one hundred meters out in front of itself and then drags it back with steel cables, filling the bucket in the process. A $20 million machine can move 3000 cubic meters of earth per hour. (These machines are custom built and assembled on-site as they are too large to be moved by truck or rail.) A large mine would likely only have one dragline and the entire mine's productivity is determined by this one machine's productivity. However, if the excavation plan is not optimal, a lot of time will be spent moving the dragline from one place to another ("deadheading"), building a "foot" on which the dragline can sit and operate, or moving the two-meter-thick power cord (there are no transmissions strong enough to handle the power of a dragline; hence all the dragline's motors are electric).

Extending the LAND HACKER system proved remarkably easy. The class MINE adds only five instance variables to those it inherits from SITE. These variables contain the boring data, coal seam data structures, and mining plan. Instances of the class SEAM contain a top bounding surface, a bottom bounding surface, a name (Appalachian coal seams are all well-known to those in the industry and have colorful names such as "Stockton A Upper"), and a *lithology* that describes the composition of the seam (e.g. coal, shale, boney coal, siltstone). All the surveyed points in the seam bounding surfaces are instances of the class SEAM-SURVEYED-POINT, which inherits from SURVEYED-POINT and adds no methods but contains a single important instance variable: the boring layer, which is a data structure describing measurements made of the material found at this location, such as the percentage of sulphur or the BTUs/pound obtainable by burning the coal.

6.4.1. Boring Data

All the information about a site's stratigraphy, i.e., what lies under the surface, is obtained from *borings*. A boring is obtained by extracting a narrow cylinder from the ground, perhaps as narrow as 10 cm in diameter and as deep as the volume being considered for mining. An object of the class BORING stores the mine with which the boring is associated, the 3D position of the top of the boring, the maximum boring depth, the date on which the boring was taken, the name of the person who supervised the measurements, an identification number for the boring corresponding to paper logbooks, a string of optional comments, and, most importantly, a list of boring layers.

Each object of the class BORING-LAYER maintains the following data: the boring of which it is a part, the name of the seam from which this cylindrical piece came, a symbol giving the lithology of the piece, the elevation above sea level of the top of the sample, the elevation above sea level of the bottom of the sample, and some coal quality parameters that are used only if it is a sample of a coal seam.

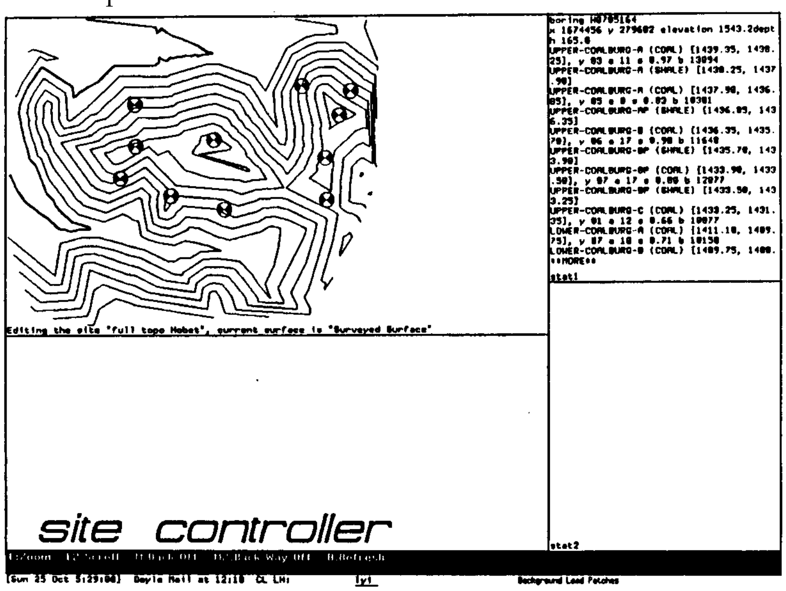

Figure 6.16 Topography of the Hobet mine in West Virginia, with boring locations superimposed on the contour map. Note the detailed description of a single boring in the upper right window.

6.4.2. Building Seams from Borings

Once the boring data structures have been constructed from raw textual tables, a method of the MINE class generates seam objects. The first step is to look through all the boring layers in all the borings and build a set of all coal seam names (this version of the software only worries about coal seams). For each name, a helper method loops through each boring again and finds the sample that corresponds to that seam (if any). Each boring that contains information about a seam contributes two objects of the class SEAM-SURVEYED-POINT, one to the top surface and one to the bottom. When all the borings have been examined, the top and bottom surfaces are triangulated and the finished seam added to the mine object's list of seams.

This technique is really half-baked because it assumes that a coal seam covers the entire convex hull of the borings that contain samples of that seam. In reality, the fact that a boring contained no sample of a seam should tell us that the seam disappears in that location.

6.4.3. Representing a Mining Plan

Excavation is conducted in parallel strips, with the dragline moving back and forth on a yet-to-be-excavated strip while ripping away at the current one. This software models a stripping plan as a collection of parallelograms, each one divided into a number of strips of reasonable width. The stripping software is built on a substrate of generic parallelogram modelling and user interface code. Objects of the class SIMPLE-PARALLELOGRAM contain a base point and two vectors that define the geometrical object as well a cache of four oriented lines that will be useful for quickly answering point inclusion queries. The most important user interface method is :GET-PARALLELOGRAM-FROM-USER, which makes use of the :GENERIC-MOUSE-HANDLER higher-order procedure to let the user manipulate the arms, orientation, and global position of the parallelogram.

Objects of the classes STRIP and STRIP-COLLECTION both inherit from the class SIMPLE-PARALLELOGRAM, but strip collections contain daughter strips that store a sequence number (in the mine's overall excavation plan). Methods of the class STRIP-COLLECTION are primarily concerned with dividing the parallelogram into fixed-width strips. Methods of the class STRIP are primarily analysis-oriented.

The following method shows some of the power that is inherent in LAND HACKER:

```
(defmethod (:analyze-case strip) (top-seam bottom-seam)
  (declare (values yds-of-overburden effective-strip-ratio tons-of-coal))
  (let* ((top-bottom-sfc (send top-seam :bottom-surface))
         (bottom-top-sfc (send bottom-seam :top-surface))
         (bottom-bottom-sfc (send bottom-seam :bottom-surface)))
    (when (and (send self :are-you-contained-within? top-bottom-sfc)
               (send self :are-you-contained-within? bottom-top-sfc)
               (send self :are-you-contained-within? bottom-bottom-sfc))
      (let* ((strip-sfc (send self :generate-surface top-bottom-sfc))
             (overburden (send strip-sfc
                               :volume-between-you-and-larger-surface
                               bottom-top-sfc))
             (overburden-plus-coal
              (send strip-sfc :volume-between-you-and-larger-surface
                    bottom-bottom-sfc))
             (just-the-coal (- overburden-plus-coal overburden))
             (yds-of-coal (cubic-feet-to-cubic-yards just-the-coal))
             (tons-of-coal (* yds-of-coal *tons-per-cubic-yd-of-coal*))
             (yds-of-overburden (cubic-feet-to-cubic-yards overburden))
             (effective-strip-ratio (/ yds-of-overburden tons-of-coal)))
        (values yds-of-overburden effective-strip-ratio tons-of-coal)))))
```

This procedure calculates how much work will have to be done to get at the coal under this strip. Its arguments are two seams and the assumption is that excavation is from the bottom of the top seam to the top of the bottom seam. The method first checks to make sure that the strip's planar projection is contained within the planar projections of the seams. It then projects itself onto the bottom surface of the top seam to create a surface for use in volumetric calculations. At this point, calculating the *overburden*, i.e., amount of worthless rock that must be excavated to get at what is presumably valuable coal, is as simple as sending a message to the strip surface to ask the volume between it and the top surface of the bottom seam. Another similar call serves to get the volume including the coal under the strip. A simple subtraction yields the volume of coal and a division determines the all-important *strip ratio* — the amount of rock that must be removed per ton of coal. If this ratio is excessive, then mining an area may be uneconomical.

6.4.4. User Interface

All the user interface methods for mining are associated with the class MINE-VIEW-MIXIN, which must be combined with SITE-VIEW-MIXIN to do anything really useful. The additional display capabilities contributed by this class are the ability to superimpose boring locations and/or detailed boring information on the site, an excavation plan display capability, and a method for showing a simultaneous vertical cross section through all the subsurfaces.

The user interaction methods can be divided into the following categories: stripping plan manipulation, boring input and query, point and strip productivity analysis, hypothetical appearance, and spreadsheet interface.

Stripping plan manipulation methods allow the use to specify parallelograms to be excavated, divide them up into strips, and number the strips automatically or manually. Boring query methods are a kind of hypertext where clicking on a boring reveals detailed information about it in another window. Productivity analysis uses either the method described in the preceding section for strip analysis or another method based on point interpolation to answer questions about the amount of work required to extract coal and the expected quality and value of that coal. Hypothetical appearance methods show the topography following excavation, i.e., what would the site look like if we ripped down to the Stockton A seam? (Unfortunately, this software does not model the effect of hurling the spoil down into the valley, but assumes that the excavated material magically disappears.) Finally, the spreadsheet interface cranks out excavation data to a file readable by *1-2-3* or *Excel* so that spreadsheet models may be used to determine excavation costs. Unfortunately, as there was no popular commercial spreadsheet program for the Lisp Machine, this interface is static rather than dynamic, i.e., one cannot change the stripping plan and see the spreadsheet numbers update in real time.

6.5. Actor Simulator

SITE CONTROLLER fundamentally consists of a mixing of LAND HACKER and an actor-based simulation system.

6.5.1. Simulation Control

Standard sites in SITE CONTROLLER inherit from the classes SIMULATING-SITE-MIXIN and SIMULATOR-MIXIN. SIMULATOR-MIXIN contributes instance variables that keep track of which actors are being simulated, simulation parameters such as how much time to advance with each tick, and simulation state variables such as the current time. Methods of the simulator use *actor prototypes* from a *scenario* to initialize a simulation. An actor prototype is actually represented with the same class hierarchy as a real actor and it contains all the information that can be specified before simulation about what its corresponding actors should do. Because simulation is often probabilistic, it is useful to distinguish between an *actor prototype*, which reflects the user's intent in general for a simulation, and an *actor*, whose state variables reflect probabilistic outcomes in a particular simulation. Scenarios are nothing more than named collections of actor prototypes.

6.5.2. Actor Class Structure

The base class ACTOR keeps track of the following information: the site where simulation is taking place, model space position, time of last update tick from simulator (i.e., the site), creation time (at which the actor begins to exist), prototype object that created the actor object (or the symbol :PROTOTYPE if the actor is itself a prototype), a statistics window in which dynamic information is to be displayed (NIL if no display is enabled), and a perspective view window in which to display the view from an object's "front window."

User interface menus are automatically generated from an actor object's response to the :SETTABLE-PARAMETERS and :RUN-TIME-SETTABLE-PARAMETERS messages, corresponding to those state variables that may be changed in an actor prototype and in an actively simulated actor, respectively. Responses to these messages are the result of appending lists contributed by methods of all of the superclasses from which a particular actor inherits. When the user wants to modify a parameter, he is first presented with an automatically-generated menu of parameter names. After he has chosen a parameter, depending on the specification list from the :SETTABLE-PARAMETERS methods, he is prompted for a path, polygon, a choice from a constrained list of symbols, another actor prototype, one of the standard Lisp Machine data types, etc. as appropriate. Once again, the principle of automating the generation of user interface keeps the class hierarchy extensible, for it is just as easy to add to the user interface as it is to add instance variables and methods.

Although virtually all actors are in fact displayable, the state and behaviors necessary to survive on the screen are expressed in a separate class, DISPLAYABLE-ACTOR-MIXIN. The instance variables contain the following state: whether the actor should currently be visible, exactly how the character was drawn on the screen last time (so that it can be erased if need be), information for displaying either a bit-map or vector-font character at the appropriate scale and rotation.

All drawing is done by sending messages to the site. The site is capable of handling these messages because it inherits from SIMULATING-SITE-MIXIN and hence keeps track of which views on which screens are currently active for simulation. SITE CONTROLLER is capable of showing a simulation at different scales and in different locations on multiple windows.

PATH-FOLLOWING-ACTOR is a class fundamental to all simulated construction vehicles. Actors inheriting from this class can follow a segmented path at predetermined velocities. Furthermore, this class defines the base method for the :TICK message from the simulator.

Actually, the ACTOR class defines a *whopper* for :TICK, which executes before any of the regular methods, both base, before and after, and immediately returns if the current simulation time is not later than the actor's creation time. If the simulation time is after the actor's creation time, i.e., the actor "exists," PATH-FOLLOWING-ACTOR's :TICK method first calculates the time elapsed since the last tick, uses that number to generate a new position (determining in the process whether a new path segment is needed), and then sends a :DRAW-SELF message so that DISPLAYABLE-ACTOR-MIXIN will erase the old image if necessary and draw a new one at the updated position.

Actors that inherit from TERRAIN-INTERACTION-MIXIN calculate, after every :TICK message, the surface model triangle they are on, what kind of terrain that is, the gradient, and their elevation. Reasonable efficiency is achieved by caching the current triangle and only looking for another when the current triangle reports that its projection in the x-y plane does not contain the actor's current projection. When the actor has run off the current triangle, the code tries to walk through the triangulation from the current triangle rather than start a global search, which is used only as a last resort.

ACTOR-TERRAIN-SCROLLING-MIXIN allows a user to monitor an overhead view of a vehicle's surround even as the vehicle moves. After each :TICK, the window (if any) is scrolled so that the vehicle remains in the window's center as the terrain slides underneath.

VEHICLEs inherit from ACTOR-TERRAIN-SCROLLING-MIXIN and PATH-FOLLOWING-ACTOR, then add standard characteristics: maximum velocity, mass, engine operating measurements, maintenance history, and the operator identity. Methods of the class VEHICLE can display additional statistics in a window and update an entire monitoring frame if the user wants to monitor a vehicle in detail, i.e., by looking at simulated engine gauges and the operator's view. LAND-VEHICLEs add the abilities to declare their propulsion system to be either wheels or tracks and to predict the likely direction of the operator's gaze.

Bulldozers are modelled by the class KILLDOZER (after the 1970's horror film of the same title wherein an alien-possessed, operator-less bulldozers murders workers in the middle of the jungle until a final epic battle with a human-operated shovel). A bulldozer's fundamental behavior is controlled by its MODE instance variable. Possible modes include :IDLE, where the bulldozer is just driving around, :CUT-TO-HEIGHT where the blade will remain at a constant elevation above sea level, :TAKE-OFF where the blade will slice a fixed amount off the top of a surface, and :FOLLOW-DESIGN where the blade will remain on an arbitrary design surface.

In addition to the mode, various state and characteristic variables are maintained, such as the elevation above sea level of the bottom of the blade, the force on the blade, the accumulated soil in front of the blade, and the blade width.

The most interesting capability of a KILLDOZER object is that of creating a new surface that shows what the current surface would look like after excavation. To do this, the object incrementally builds up two sets of points, one to form the convex hull of the new prospective surface patch and one to define the actual path of the blade. Forced connections are inserted so that the original surface outside the hull is unaffected by the blade path elevations and also so that the area underneath the blade is unaffected by elevations outside the blade path. The area in between the hull and the blade path is allowed to take on any shape (presumably it will be a slope from the original surface down to the blade path). Without the forced connections, the resulting surface might look like almost anything and would certainly not have a neatly-carved trench in it.

SURVEYING-VEHICLEs inherit from TERRAIN-INTERACTION-MIXIN and TRUCK. These actors maintain an *augmentation surface*, which will be filled with surveyed points at periodic intervals as the vehicle drives over a piece of terrain equipped with some kind of location system. These actors are useful for simulating ultimate real-world practice where surveys, particularly of hazardous waste sites, will be conducted from within enclosed vehicles taking advantage of precise location systems such as GPS. There are also more specialized surveying vehicle classes that sample radiation or hazardous waste barrels in addition to standard topographic data.

6.5.3. Personnel Database

For numerous operational, legal, and historical reconstruction reasons, it is worthwhile to keep accurate records of which operators performed which operations on a site. An object of class OPERATOR contains the following data: name, years of employment, start date, accident and injury history, license information, machine qualifications, hourly cost, previous sites, and previous employer. There are the beginnings of a hypertext system in SITE CONTROLLER where clicking on displayed attributes of objects such as operators would hop to descriptions of the attribute value.

6.5.4. Terrain Objects

Any particular kind of terrain is represented as an object of a class that inherits from BASIC-TERRAIN. Any kind of terrain can give information about the coefficients of friction and drag for wheels, tracks or feet (human). In addition, terrain objects know how to draw and describe themselves.

6.6. Real-World Site Control

Caterpillar used a copy of SITE CONTROLLER to monitor and oversee the operation of an autonomous mining truck (AMT). The AMT is built on top of a $600,000, 1200 horsepower dump truck. The standard vehicle is fitted with GPS and inertial navigation systems and an on-board computer capable of learning several routes. I spent a few weeks extending SITE CONTROLLER to work in real time with the AMT and then travelled to the Arizona desert to install the software and watch it work. The programmers at Caterpillar made life very easy for SITE CONTROLLER by providing a clean high-level interface to the truck via an RS-232 connection to a radio modem.

6.6.1. Basic System Structure

Each real-world vehicle is shadowed by an *actor slave* in SITE CONTROLLER. The slave is supposed to embody all the most recent knowledge about the real vehicle's position, speed, and any other available information. Communication with a real vehicle is accomplished by sending messages to the corresponding slave.

Communication to the vehicle occurs only as a consequence of some user action. Consequently, sending packets is done by the normal SITE CONTROLLER process—although in practice the simulation is not held up waiting for the serial line because the Lisp Machine buffers the output bytes and transmits them at the selected baud rate from a special “serial output” process. Input, on the other hand, is received asynchronously by a special communications monitor process that periodically updates a shared data structure with newly-received packets. All communications in and out may be archived in a file that is replayable to show the activity at a site on a particular day, with the user able to step forward in time at his own pace.

All the software is written to cope with an arbitrary number of vehicles, although in practice only one truck was ever available with the appropriate on-board hardware.

6.6.2. Actor Class Extensions

In order to ensure that SITE CONTROLLER would work when plugged in, I added the class SIMULATE-CAT-TRUCK-MIXIN that drives around a site in simulation, periodically sending out status packets to the serial port. During testing, I connected the output and input pins of the serial port so that signals looped back into the computer. A packet-reading process would eventually notify an object inheriting from the class CAT-SLAVE-MIXIN that packets had been received. Each time a packet came in with an updated position, one could see the CAT-SLAVE actor jump to coincide with the SIMULATE-CAT-TRUCK actor. This method of testing resulted in only one line of code having to be changed when the real truck was hooked up.

6.6.3. User Interface

All user interaction with the AMT is supported by mouse-accessible commands added by the class CAT-SITE-VIEW-MIXIN. There are commands to request a vehicle’s status, change a vehicle’s velocity, and select a vehicle’s route. Utility commands include such items as resetting the communications history and saving the current history to a file. Finally, there are commands to control replay of communications from a file.

In addition to mouse commands supported by the site view window, there is a hypertext facility built into the communications display window. Each packet occupies a line in a tall, narrow window that scrolls up as new packets are received. There are scroll bars that enable the user to select any portion of the history and hypertext hooks that pop up a full description of any packet that is mouse-clicked.

6.6.4. Communications

Communication was accomplished with 1200 baud radio modems. All communication must take the form of a packet, which starts with an STX character and ends with an ETX. Each packet contains vehicle identification as well as a CRCC error detection word. All packets inherit from the class DATA-PACKET. Packets going to a vehicle inherit from the class PACKET-TO-VEHICLE and incoming packets inherit from PACKET-FROM-VEHICLE; methods of these two classes are where the meat of transmission and self-description occur. There is a leaf class corresponding to each distinct packet type. Typical packet types include the following: SELECT-ROUTE, REQUEST-STATUS, STATUS-FROM-VEHICLE (containing position), HEALTH-FROM-VEHICLE (engine and load statistics), MESSAGE (containing arbitrary ASCII text to be displayed to the user of SITE CONTROLLER or a human in the vehicle, if any).

Packets are first collected in raw form by a very simple parser that just looks for STX, ETX and escape characters used to precede STX or ETX bytes that must occur in the middle of a packet. All the bytes between STX and ETX are handled over to a finite-state machine parser that picks out the vehicle-id, message type, the type-dependent data bytes, and the CRCC word. The parser creates either a GARBLED-PACKET object or a packet object of the appropriate class. If the packet is otherwise OK, but the CRCC word did not check then a real packet is created anyway, but its CRCC-OK? instance variable is set to NIL. This might be useful, for example, if one received a human-readable ASCII message from a vehicle in a hostile environment and getting the message quickly with one character wrong might be better than not seeing it at all until a perfect message could be transmitted.

Breaking apart the data bytes into appropriate fields is left to methods of the specialized packet classes. After this is complete, the completed packet is pushed onto the front of a list containing all packets received. This list is the value of a global variable and, since all Lisp Machine processes run in a single virtual address space, is accessible to the SITE CONTROLLER user process.

A note on error control is in order here. A standard quick-and-dirty error detection method is the one-byte checksum, i.e., transmitting the result of XOR'ing all the bytes in the packet together. This is better than nothing, but two single-bit errors can cancel each other in the XOR, thus resulting in a corrupted transmission being accepted. CRCC, or *cyclic redundancy code check*, error detection works by regarding the entire message as a single bit string, say N bits long. These N bits are taken to be the coefficients of an Nth order polynomial over the integers mod 2, i.e., a polynomial that consists only of monomials such as x^{15}. The coefficients are restricted to take on values 1 or 0 and if we divide this polynomial by a *generating polynomial* of degree M, the remainder will have M coefficients that are either 1 or 0.

SITE CONTROLLER uses the CCITT standard generating polynomial $x^{16} + x^{12} + x^5 + 1$ and hence the remainder will be 16 bits. It turns out that a very simple shift register plus XOR gate architecture can be used to accomplish the polynomial division, which is why this technique became so popular. The benefit of using CRCC error detection is that two single bit errors do not cancel each other as with XOR, but merely further corrupt the remainder. The only problem with raw CRCC is that the message "all 0's" will be accepted as uncorrupted since the remainder of 0 when divided by anything is 0. I fixed this by using a standard technique of preloading the CRCC register with 1's, essentially prefixing every polynomial to be divided by a standard header. Thus, the message "all 0's" will not be accepted because even a block of 0 data would have a non-zero CRCC remainder due to the prefixing.

7. Lessons Learned

7.1. Comprehensive Site Databases are Good

Although I would like to say that SITE CONTROLLER provides a compelling example of the benefits of comprehensive site databases, it seems clear that the best example in recent memory was provided by the Great Chicago Flood of April, 1992 [McManamy 1992]. Various miscommunications among engineers and a lack of information exchange resulted in piles being driven through the bottom of the Chicago River into freight tunnels underneath. Imagine if the pile-driver operators were continually presented with a computer-generated cross section of the area in which they were working, complete with relevant site features such as the freight tunnels. It seems likely that they would then have questioned the pile locations. As the flooding of downtown Chicago's basement cost society over $1 billion, it is probably the case that averting this one disaster would have paid much of the cost of automating America's entire construction industry.

Bringing together information that is today in separate programs or on separate pieces of paper is fundamentally powerful by itself. A planning program with hydrology data can trivially warn the user that a proposed excavation will become flooded if there is any rain. Keeping track of utility lines, especially gas lines, during both design and construction can save lives as well as millions of dollars.

It is important to distinguish between raw geometry, the kind of data in traditional CAD systems, and full-fledged objects. Raw geometry consists of lines, text, polygons, circles, etc. that together communicate something meaningful to a human user. Full-fledged objects represent their real-world counterparts in a sufficiently rich manner to permit computation about real-world situations. For example, a raw geometry line/text combination that communicates "gas line" is nice to have on a blueprint, but only an object of the class GAS-LINE can conceivably cause a backhoe-operator assistant program to declare a full red alert when there is a risk of explosion.

Once comprehensive site databases are available, analysis algorithms will grow to take advantage of the data. Without the data, nobody will bother to write software.

7.2. Real Database Management would be Nice

The simplest computer-aided civil engineering programs do not need any kind of database. If a program's only ability is reading a file of (x, y, z) points and producing a contour map, no intermediate results need be stored. SITE CONTROLLER builds complex data structures in the computer's memory that might be worth saving. Lisp systems are able to store on disk an image of the memory, or "core dump," that can later be restored. Users can back up results or store intermediate results before trying radical ideas. Further, by storing snapshots of the program at regular intervals, it is possible to later see how the design progressed.

Because a core dump saves everything that has been added to or changed in the memory since Lisp was started, this approach is extremely wasteful of disk space. Cooperation is difficult as well. If Rebecca is working on a design on one workstation, Rachel is unable to add anything to Rebecca's design from another workstation. Rachel would have to come in at night and work on Rebecca's memory image. Finally, a core dump made on a computer using, for example, a 68000 microprocessor might become unusable if users switched to computers using a RISC processor with a different format for numbers.

Writing everything into a "moby file" (from *Moby Dick*) is a somewhat better approach and this is what SITE CONTROLLER presently does. Users can recover the exact state of the system when the moby file was written, just as they did by restarting a core dump. Only essential data need be stored and therefore a moby file may be kept to a reasonable size. Furthermore, different sites may be stored in different moby files so that comparing projects is as simple as reading in multiple files. Machine independence is also achievable.

Despite these advantages, moby files retain many of the core dump's drawbacks. First, Rebecca and Rachel still can't cooperate effectively. Rebecca must write a file then wait for Rachel to read it, work on the design, and write it back to disk before she can work again. Second, all the data concerning a project must be read even if only a tiny amount is needed. As site models grow to tens of megabytes, this may dramatically affect performance. Third, no consistent journal is kept of how the design evolved. Except by laborious byte-by-byte comparison of moby files, there is no way to determine who changed what, or when and why.

7.2.1. Standard Database Management Systems

A *database* is structured information that can be accessed or updated in pieces by multiple users. Commercial database management systems (DBMS) are highly evolved collections of programs for solving standard business database problems. Their facilities are so powerful that many have been tempted to shoehorn CAE data into standard DBMS models.

Concurrency control prevents simultaneous updates by multiple users from interfering with one another. Ensuring that updates occur in a controlled manner and that the data remain consistent have been primary DBMS concerns since the late 1960's, when computerized airline reservation systems were developed. Reading and writing to the database are restricted to take place during atomic *transactions*. The DBMS ensures that if a transaction cannot be completed for any reason, the database remains unchanged from the way it was before the transaction commenced. When a record is to be updated, the transaction first gets a *lock*, changes the record, then releases the lock. While Rebecca's transaction holds a lock on a record, Rachel's attempt to update is blocked. It is possible for Rebecca's transaction to be blocked waiting for a lock held by Rachel's transaction that is blocked waiting for a lock held by Rebecca's transaction. This results in a *deadlock*, and deadlock detection facilities generally abort one or both transactions involved.

Skillful exploitation of standard DBMS concurrency-control facilities and techniques by a CAE system enables users to cooperate on projects. Rachel can "check out" a portion of the design and work on it while Rebecca works on some other aspect of the problem. If Dina tries to make changes to Rachel's portion, the database tells her to wait until Rachel has "checked in" her changes. When Rachel checks in her solution, Rebecca and Dina's working copies are updated to reflect Rachel's changes.

Because the first DBMS applications were financial, keeping a transactions log has always been a built-in defense against hardware and software unreliability. A *journal* of transactions is kept and therefore the state of the database at any time is recoverable so long as an initial state and the transactions journal are retained.

Commercial DBMS devote much attention to query mechanisms. Business problems require frequent extraction of tiny bits of information from vast databases. The quest for greater ease of specifying which data are required has spawned hundreds of query languages. Query optimization algorithms and even special hardware have been designed to improve database performance. CAE data stored in a commercial DBMS may be easily and efficiently accessed even in ways not foreseen by the original system designers.

In the mid-1960's, IBM developed the *hierarchical* data model where everything is organized into trees. A family's genealogy would fit nicely into the hierarchical model. However, if one wanted to also keep track of all family picnics and which of the family members had attended each one, one would be forced to construct additional trees to represent that information. To avoid duplicating records in these multiple trees and to facilitate the representation of complex relationships, hierarchical database systems incorporate a variety of kludges, notably virtual records.

GE developed what became the *network* data model at roughly the same time. Data is represented in a directed graph, with each one-way link signifying a relationship between two pieces of data. The family picnic example above can be represented more or less directly, although one additional record must be introduced for each link.

Both the network and the hierarchical models require the programmer to have knowledge of the graph or tree structure. Queries are procedural, instructing the DBMS to traverse one link after another. This makes programs hard to write and obscures their meaning. The *relational* data model [Codd 1970] lends itself to declarative query languages. Instead of specifying how to do something, the user need only specify what is wanted. *Relational algebra* is a consistent set of operations on relations whose results are themselves relations. Because these operations are closed under the set of relations they may be freely composed.

Relational algebra has been the basis of many query languages for relational DBMS. Of the relational languages, SQL, developed in the late 1970's by IBM, has become the most popular. SQL was intended to meet the needs of both interactive non-programmers users and experienced programmers and hence does not fully satisfy either type of user: SQL cannot be the world's best interactive query language and also an elegant data manipulation sublanguage for conventional programming languages. Furthermore, when SQL is grafted into host languages, its syntax and model of computation often strike a discordant note. SQL itself is more limited in power than standard programming languages. One could not, for example, write a CAE system in SQL.

Instead, a CAE system written in Lisp would include some embedded SQL commands to be handed off to the DBMS.

Declarative queries, ease of use, and flexibility in relational databases were initially achieved at the price of sluggish performance. However, advances in query optimization and computer hardware have made relational databases the most practical solutions for most business problems in the early 1990's. Unfortunately, the systems remain orders of magnitude too slow to be used by CAE programs.

Many excellent textbooks serve to educate the future archivists of corporate America in the art of database management [Elmasri and Navathe 1989; Ullman 1988]. Examples of ways in which geometric models can be stored in different types of databases are contained within [Kemper and Wallrath 1987].

7.2.2. Object-Oriented Databases

Popular database models such as relational, hierarchical and network were developed for business data-processing problems and are well suited to the simple objects and straightforward processing required in the world of business administration. Transactions generally occur quickly and involve small amounts of data, e.g. "reserve seat 12A on flight 53 on September 28". Schema modification is rare and tends to affect a large number of entries, e.g. a personnel database might be modified to store each employee's race to comply with government regulations. Version control is a non-issue—it is not generally valuable to remember that Employee 674 used to be single and is now married, or that the Supervisor of Paper Shuffling is Melissa Hegel but used to be Donald Frobenius.

Computer-aided engineering gives rise to complex objects and relationships that must be described obliquely in standard database systems. Information associated with an object is often spread around in the database and thus reading an object from disk entails multiple seeks. Since a disk seek takes as long as one million processor operations, minimizing disk seeks is the central goal when striving for high-performance database systems.

Transactions in CAE systems can take hours and may involve megabytes of data. Consider the following example: supply a surface to an analysis algorithm, wait for the water flow over the surface to be calculated, present that analysis to an engineer for annotation, and finally write everything back into the database so that other engineers can use the results to keep structures out of the way of floods. Conversely, transactions in CAE systems might take microseconds. For example, dragging a mouse across a screen might result in changes to dozens of portions of the site model and it might be nice to handle those changes as database transactions in order to capture the benefits of concurrency and version control. If Rachel and Rebecca were working closely together, it might be desirable to synchronize their activities through the database so that every time Rebecca dragged something around with the mouse, Rachel would see the effects on her screen.

Schema modification is a routine occurrence in a CAE system. An engineer may decide that a new type of building block is required, or that objects never before linked should be associated now. The diversity of the construction industry ensures that computer-aided earthmoving will require flexibility in database schema. If a job demands working with asbestos, a foreman might need to store information about which workers smoke cigarettes so that the computer would be able to automatically keep them away from risky positions around the site.

Numerous object-oriented database management systems (ODMS) have been developed recently [Cattell 1991; Bertino and Martino 1991] that are much more suitable than traditional business-oriented database systems. At a minimum, an ODMS provides for *object grouping*, storing in one place all the data that pertain to one real-world or abstract entity, and *object identity*, unique ID pointers for objects that are independent of the values contained within those objects. Most systems are attempts to make persistent the object-oriented features available ephemerally in C++, Common Lisp Object System, and Smalltalk. Consequently, these object-oriented databases offer the same class definition, inheritance, method definition, and instance variable storage features of the object-oriented programming language. Ancillary query languages and database administration tools may be included, but fundamentally object-oriented databases do not require programmers to learn new data-manipulation languages.

Typifying the best of the new systems is ObjectStore [Lamb 1991]. ObjectStore exploits existing virtual memory hardware and operating system support to gain high performance and transparency to existing software. Persistent objects on disk are represented bit-for-bit identically to transient C++ objects in memory. This kind of technique has been used by at least four commercial products and is called the *disk-image* method. The entire relevant portion of the database in mapped into the user process's virtual memory space. Accessing an object for the first time takes as much time as handling a page fault, i.e., mostly however long it takes to load the page from the local disk or from across the network. Subsequent reads or writes to a persistent object occur at normal memory speed. Pointers to objects are ordinary address space pointers. ObjectStore is able to handle databases substantially larger than standard virtual memory address spaces, but the extra long pointers are not visible to user programs. Note that the authors of ObjectStore were able to save themselves the trouble of writing a data caching system—they get it for free from the operating system's virtual memory code, which will strive to keep the data that is most likely to be needed in memory at all times.

Programs save time and programmers save effort with this architecture for several reasons. First, no translation need be performed between the database representation and the in-memory representation.[6] Second, standard library procedures may be used without even recompilation to manipulate database objects. As far as the library procedure is concerned, it is just reading to or writing from data structures in memory. Third, pointers to objects are dereferenced by virtual memory hardware in nanoseconds, rather than being translated by software in microseconds. Finally, the virtual memory system already contains hardware and software support for determining which pages have been modified and need to be written back to disk.

---

[6] Commercial object database systems often incorporate ambitious schemes to enable processors with different number representations to automatically and transparently share a database. Thus, if the data were stored in Big Endian fashion and were to be used by a Little Endian processor, conversion would be performed by the database at some point.

Rather than locking individual objects, ObjectStore locks virtual memory pages. Thus, hundreds of objects may be locked with a single swift operation. Although one might conjecture that this would lead to pointless concurrency conflicts since some unneeded objects are locked along with the ones that will actually be updated, in reality the system is speeded up so much by not having to keep track of thousands of separate locks that transactions execute faster and hence are less likely to conflict.

Ultimately, any choice among databases must take performance into account. After all, any database management system is capable of supporting any task given a fast enough computer and enough extra work by the programmer user. However, in the real world, substantially more efficient execution makes the difference between a practical solution and an interesting bit of research. Benchmarks on the kinds of pointer-chasing done by object-oriented programs show that ODMSs in general are about 100 times faster than relational databases and that there are significant performance differences among ODMS products [Cattell and Skeen 1992].

Although ObjectStore solves many problems faced by CAE programmers, it also serves to illustrate some of the shortcomings of ODMSs. Using the disk-image approach yields dramatic performance improvements but exacts a penalty in data integrity. A program that can inadvertently modify any memory location is now capable of inadvertently corrupting the database. Anyone who has ever used a C program on a personal computer that mistakenly wrote over the operating system and crashed the machine would probably have reservations about a C++ program managing critical financial data in ObjectStore. (Note: there are disk-image object databases, such as O_2, that use the disk-image approach but achieve high data integrity by supporting inherently safer languages, such as Lisp and Smalltalk.)

Just as C++ is a throwback to the computer languages of the 1960's and 1970's, ObjectStore in some ways dredges back up limitations of hierarchical and network databases: in order to achieve high performance, ObjectStore does not by default keep enough information about objects to support fully general declarative queries. The programmer must understand the structure of the database in order to query it.

A final problem with object database management systems is that no standards have evolved in this world, either in data models, query languages, or programming languages. A user of ObjectStore, for example, is shackled to that product in a way that an SQL programmer will never be limited to a particular relational database. This is particularly distressing since no existing product or even research prototype provides a really complete set of capabilities.

While the preceding problems with ODMS are likely to keep traditional relational products selling for many years to come, it is hard to imagine a commercially-viable system along the lines of SITE CONTROLLER that is not based on some kind of ODMS.

7.2.3. Version Management and Coordinating Multiple Designers

> *Committee: a group of the unfit appointed by the unwilling to do the unnecessary.*
> — Stewart Harrol

Version management in CAE systems is a research problem. In supporting civil engineers, one must keep track of who modified what and when in the design, ensure that derived data is marked invalid when the generating data is modified, and try to ensure that designers are kept out of each other's way. When the task evolves into supporting construction, the goal is that the state of the site at all times should be recoverable from a database.

The common substrates that are required here are the ability to track object evolution and to aggregate objects into consistent versions. Version management has been tackled by the design automation community since the early 1980's, for electrical engineering [Bhateja and Katz 1987]. The general approach is to add records to the database that describe aggregations of objects, or IS-A-PART-OF relationships, and the evolutionary history of objects, or IS-A-KIND-OF relationships. In addition, these systems support automatically supplying the latest stable version of an object to most of the designers, while allowing one or two designers to continue working on a new version of that object. Facilities for checking-out and then checking-in objects are common, as are change propagation mechanisms.

Finally, it is often necessary to collect mutually consistent versions of objects in a database into a *configuration*. In a software development system, this would correspond to a release of a new version of the product. In a civil engineering system, a configuration might be one possible way to lay out a site given a radically changed road network.

No standard version model or management system has emerged, but some unifying work has been done [Katz 1990] and version management is a standard feature of at least six object-oriented database products [Kim and Chou 1988; Object Design 1992]. Given the practical realities of the large size of civil engineering projects and the litigious environment, it seems clear that a successful civil engineering design system must have substantial version management support.

7.3. Floating-point Arithmetic is Bad

Due to the lackluster speed of the Lisp Machine, I made a Faustian bargain with the Goddess of Floating Point and eschewed exact arithmetic for most geometric computation. However, I paid for the speed She gave me with terrible hours of debugging and digging for numerically-stable algorithms. Consider the following problem: is a point inside a triangle?

It is worth remembering that the standard equation of a line, $l \equiv ax + by + c = 0$, has an orientation. If we plug an arbitrary point (x_1, y_1) into the left side of this equation, a value of zero indicates the point is on l, a negative value means (x_1, y_1) is to the right of l, and a positive value means (x_1, y_1) is to the left of l. We can get the same line oriented in the opposite direction simply by multiplying the equation through by a negative constant.

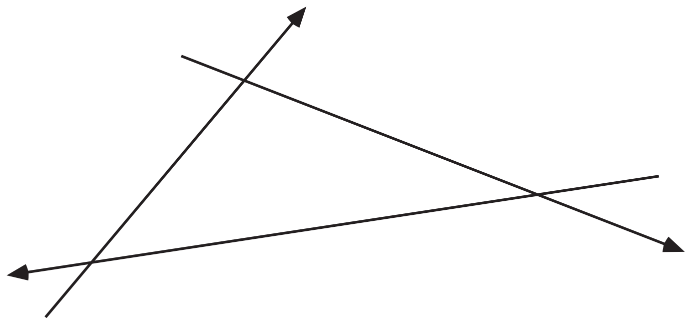

Figure 7.1 Oriented lines arranged around the edges of a triangle. Points to the right of all three lines are inside the triangle.

If we build appropriately oriented lines collinear with the edges of a triangle, we can determine query point (p_q) inclusion by plugging the point's coordinates, (x_q, y_q), into all three line equations. If the results all have the same sign, then p_q is in the triangle (see Figure 7.1). Consider the computation being performed here: $ax_q + by_q + c$. If, for example, the summands ax_q and by_q are large, then the contribution of c may be nil if it is combined with by_q first. I struggled with what I thought was a subtle bug in a geometry algorithm for several hours. I finally zeroed in on a triangle that was reporting p_q as outside of the triangle even though zooming in revealed p_q to be just barely inside. When Jim Little heard about this, he sagely noted that "Inner product accumulation frequently requires double precision arithmetic."

About 10% of my time working on the core of SITE CONTROLLER was devoted to hunting down bugs created by my unjustified faith in floating point numbers and then searching for alternative algorithms. Faster machines and floating point implementations with more precision simply postpone the day of reckoning. Exact arithmetic, i.e., rational numbers, is the only long-term solution that will give anyone confidence in complex geometric computation systems.

7.4. Programming Environments Matter

In the "good old days" (for programmers, that is), computers were expensive and so were programmers. Thus, in the early 1970's, a corporation might spent $500,000 or more on a mainframe and $50/hour or more for really skilled programmers. *The Mythical Man-Month* convinced software development managers that, even with all the money and programmers in the world, the growing complexity of ambitious computer systems made success doubtful [Brooks 1975]. When it was observed that computers were getting cheaper and programmers more expensive, the industry scrambled to make precious programmers more productive. The apotheosis of the programmer was best evidenced by the MIT Lisp Machine, a $100,000 (in 1980!) personal computer made by and for some of the world's best programmers. The reasoning behind the Lisp Machine was this: rather than hire ten mediocre programmers who will spend most of their time fighting low-level languages and stepping on each other's toes, hire one good programmer with a Lisp Machine who can solve the whole problem by himself in a fraction of the time.

However, the 1980's brought two simultaneous revolutions: 1) the dumping on the market of thousands of people with bachelor's degrees in computer science, and 2) a collapse in the price of computer power. By 1990, the world looked very different from what anyone imagined in 1970. A corporation might still spend $500,000 for a mainframe but the programmers only cost $15/hour in 1970 dollars (i.e., after adjusting for inflation). Mostly, however, companies buy inexpensive prepackaged software on inexpensive personal computers and don't hire programmers at all.

Occasionally programming is unavoidable, either for a custom solution or because a company wishes to sell prepackaged software. Conventional wisdom today calls for a large group of low-paid programmers using 1960's-style tools such as UNIX, MS/DOS, and C. Any competent computer scientist could prove that this approach simply cannot work, yet at first glance it appears to and has produced billions of dollars in profits for many corporations.

There are a variety of reasons for the success of the "horde of C hackers" approach. One of the most important is that programmers today are solving problems that were first solved many years ago. For example, UNIX was developed in the 1970's with a subset of features of 1960's operating systems, most notably MULTICS; Microsoft *Word* was an incremental improvement of *MacWrite*, which was in turn inspired by work done at Xerox PARC in the 1970's; PC networks are reimplementations of systems that were developed on mainframes in the 1960's and 1970's; most database systems are rehashed versions of old IBM products.[7] When one has a concrete idea of what the completed system should do, one is seldom forced to risk introducing bugs by modifying structures.

Another reason the "horde of C hackers" approach has worked remarkably well is that, although the resultant software is rife with bugs, most users have no experience with anything better. When an MBA's Macintosh Quadra crashes due to a C programmer's error in Microsoft *Excel*, he doesn't say "I remember back in 1978 that my VAX 11/780, with only one tenth the processing power of this machine, had memory protection between processes so that a bug in one program couldn't corrupt the operating system or other applications." Rather, he is likely to say, "Well, it is still easier than using pencil and paper."

Finally, big C programming projects succeed because they simply aren't that big. The biggest, most ambitious programs of the 1970's would still be big, ambitious programs today. For example, *MACSYMA*, a circa 1970 Lisp program, is still the world's most powerful computer algebra system, despite the hundreds of man-years and massively powerful new computers that have gone into making competitive C programs. Significant computer power was not widely available in 1970, so only a tiny number of people remember using really big programs two decades ago and therefore users believe today's programs to be remarkably big and ambitious.

---

[7] *The copy of a beautiful thing is always an ugly thing. It is an act of cowardice in admiration of an act of energy.* — Rémy de Gourmont. (This is my favorite description of C++.)

Computer-aided civil engineering is different. Because the Global Positioning System is new and because sufficiently powerful computers are only now cheap enough for civil engineers, there are no mainframe or minicomputer systems that can serve as models for what PC software should do. For example, nobody knows a good way to capture a construction plan from a user. Any program that is thrown out into the real world is likely to be thrown right back with hundreds of deficiencies noted. If adding features to address those deficiencies introduces bugs, another important difference of civil engineering is going to make itself apparent.

Users may not be so forgiving of bugs in systems whose errors can cause the loss of millions of dollars or even human lives. Rebooting a PC because the word processor or spreadsheet crashed the machine is an annoyance. Bugs in complex computer-aided VLSI design systems may delay a integrated circuit's introduction to market, but would be unlikely to cost as much as the $1 billion lost in the Chicago flood. Anyone roasted in a Ford *Pinto* would probably argue that mechanical engineering errors can be just as deadly as civil engineering errors, but mechanical systems may be tested exhaustively before an investment is made in mass-production and dramatic failures are consequently rare.[8] By contrast, civil works are normally one-of-a-kind and realistic tests are often impossible. Thus, the Tacoma Narrows suspension bridge blows down, costing tens of millions of dollars, and the Kansas City Hyatt bridge collapses, killing scores of people.

Finally, computer-aided civil engineering is going to require enormous systems, much larger than the "mostly working" C programs currently being developed, with the added kicker that they must operate in real time. A system incapable of real-time response and error-free operation will be relegated to supporting civil engineering design only, a tiny portion of the industry. Construction is where the dollars are; models must be comprehensive and queried and updated in real time.

The problems seen by developers of the 1960's and 1970's most ambitious software systems have never been solved and they will come back to haunt developers of computer-aided civil engineering systems in the 1990's. Obvious problems include the following: 1) inherent complexity, 2) inherent dynamism of data structures, 3) maintaining extensibility so that unenvisioned capabilities may be added, and 4) maintaining high performance despite massive amounts of data and complex geometric problems.

An army of C hackers will not be able to surmount these problems. They will spend most of their time stepping on each other's toes, fighting their storage allocation bugs, ripping apart the system to add new features, and digging out from under the sluggishness of their *ad hoc* memory management algorithms. SITE CONTROLLER shows that a small group of programmers, working with good tools such as automatic storage management, a modern object system, higher-order procedures, metalinguistic abstraction, exact arithmetic, and a concise language, has a chance of overcoming the management-of-complexity and extensibility problems inherent in this area.

---

[8] In the case of the *Pinto*, the problem had been noted and debated within the company well before anyone was incinerated—that the company continued to use the design for reasons of cost was a failure of management, not of engineering.

7.5. Algorithm Animation Aids Debugging

Everyone knows that damage is done to the soul by bad motion pictures. — Pope Pius XI

Bug-ridden programs make their mistakes instantly, quietly, and secretly. Tools that allow programmers to watch execution proceed at a human pace *animate algorithms*. The crudest form of algorithm animation is the *stepper*. Stepping through a program, a programmer gets to see each piece of code before it executes, complete with the values of local variables and arguments. Only after the programmer takes a positive action, such as typing a character, does the code actually execute. Since a typical computer can execute 10 million instructions per second and a typical human can type no more than 500 characters per second, this process slows down the program by a factor of at least 20,000 and is therefore impractical unless the programmer already has the problem narrowed down to a fragment of a big system. Thus, it was only on occasion that I was only able to gain insight into bug-plagued portions of SITE CONTROLLER by using the Lisp Machine stepper.

Algorithm animation is an active research area, spurred to some extent by frustration with the fact that the programmer has gained almost nothing from the hardware revolution that gave every computer user a big screen filled with windows and graphics. Even powerful, multiwindow programming environments such as the MIT Lisp Machine, Smalltalk [Goldberg and Robson 1983], and Cedar [Teitelman 1984], basically use graphics workstations as multiple ASCII terminals. If only those windows could be filled with different simultaneously-updated graphical views of an algorithm in action, the programmer could debug, the student might learn, and the customer would be impressed.

Deciding what information to present and at what stage in an algorithm's execution to update that information requires positive action on the part of the programmer. The programmer normally must also decide how information is to be presented, since standard programming environments provide no support for the graphical display of even fairly basic data structures, such as arrays, linked lists, and hash tables.

Consider a program that stores data in a hash table. An animation of this program might include the following:

- a window showing the source code with the currently executing line highlighted
- a window showing all the buckets of the hash table and their contents. If there are too many buckets, a view of the entire table shows a simple schematic drawing, perhaps with one pixel turned on for each entry in a bucket. However, zooming in would reveal detail and the bucket contents would be more fully described.
- windows showing input and output data marching along with program execution
- meters showing information derived from the program's data structures and execution, such as hashing efficiency and average access time

By watching this animation, a programmer would instantly be alerted to a bad hashing algorithm (clumps of data would collect in just a few buckets). If the clustering problem were related to the input data, that would be visible as well.

Geometric algorithms lend themselves to animation in more obvious ways. A plane-sweep Delaunay triangulation algorithm comes to life as the edges between points are drawn as the algorithm builds the final data structure [Gillett 1988]. Any algorithm that walks through a geometric data structure should be able to successively highlight each edge and vertex considered and give the programmer a printed report on what conclusions are being made. All the geometric algorithms in SITE CONTROLLER are implemented with some self-animation capability. This is because I discovered quite early that such programs invariably either work perfectly the first time or are impossible to debug without periodic graphical snapshots of their state. If a program works, the animation capability will be useful in training programmers new to the project. If a program fails, the animation capability will make debugging painless. Depending on the programming language, there are numerous convenient ways of ensuring that the animation hooks do not degrade performance or increase program size when not in use.

The 1990's will be a period of evolution for algorithm animation programming tools. Toolkits for animating standard data structures and interacting with arbitrary animations should evolve from academic experiments into standard commercial tools. A lucid overview of the topic and a description of Balsa-II, a pioneering system, are contained in [Brown 1988]. Tango, a spiritual descendant of Balsa that is less restrictive, is described concisely in [Stasko 1990].

The bottom line is that *ad hoc* algorithm animation saved me hundreds of hours in the development of SITE CONTROLLER; better and more complete algorithm animation should greatly facilitate the development of more complex successor systems.

8. Conclusion

The time is ripe for computer-aided civil engineering and especially computer-aided earthmoving. SITE CONTROLLER demonstrates that, with so much infrastructure in place, a comparatively tiny amount of work suffices to yield enormous productivity dividends.

Let's look back at the three components mentioned in Section 1: a central computer system such as SITE CONTROLLER; a location system; and on-vehicle computers. Recent advances in differential Global Positioning System receivers have made it abundantly clear that the precision required for earthmoving will be available soon in rugged, inexpensive packages. Research and development activities at machine vendors like Caterpillar indicate that the on-board computer and control systems required by SITE CONTROLLER can be designed and manufactured internally by any of the major American, European or Japanese manufacturers. My work on SITE CONTROLLER, study of the literature, and years of software development experience lead me to believe that a few dozen good programmers, supported by the best current tools, would be able to deliver practical, reliable software for every part of an integrated computer-aided earthmoving system.

Cutting the cost of earthworks by half is an entirely reasonable goal. Cutting the cost of hazardous waste site remediation by a factor of ten is probably an essential goal. Such dramatic cost reductions would change the shape of the planet and have an immediate effect on the lives of billions. All for the price of a few lines of code.

Glossary

CAD computer-aided drafting or computer-aided design, programs primarily intended to support draftsmen.

CAE computer-aided engineering, a newer genre of programs with more powerful models of the underlying problems, primarily intended to support engineers.

CAR built-in Lisp procedure that returns the first element of a list. This acronym is from a 1950's M.I.T. implementation of Lisp on the IBM 704: "contents of address register."

CDR built-in Lisp procedure that returns the portion of a list that follows its first element. CDR, pronounced "could-er" is the complement of CAR, and was originally "contents of decrement register."

class definition for a type of object in an object-oriented programming system. Thus, HUMAN is a class, and the individual "Naomi Rosenblum" is an **instance** of that class. Actually, Naomi would probably be represented as an instance of the **subclass** WOMAN,which would inherit from HUMAN, which would in turn inherit from the **superclass** ANIMAL.

CONS built-in Lisp procedure that glues two Lisp objects together, short for "construct." With this fundamental mechanism, it is possible to represent lists, trees, and other more complex data structures.

convex hull the smallest convex polygon that contains a set of points (in the plane for Site Controller, although this concept generalizes to higher dimensions). Imagine stretching a rubber band around a set of pins stuck into a board.

Delaunay triangulation (of a set of points S) a connection of the points in S such that the convex hull of S is broken up into triangles and those triangles are as close to equilateral as possible. The dual of the **Voronoi diagram**.

dragline machine that throws a bucket in front of itself and then drags it back via steel cables, thus filling the bucket with earth and rock. Generally used at surface mines to expose mineral seams.

GPS global position system of satellites launched by the U.S. military that enable one to fix one's position in three dimensions anywhere in the world to high accuracy.

hypertext is a directed graph, where each node contains some amount of text or other information. Users are able to get to a new node by selecting a word or phrase at the current node.

instance is a particular example of a class in an object-oriented program. Thus, "Pinto with vehicle identification number 74I23BURN97FAST" is an instance of the class FORD-PINTO.

kludge an inelegant and/or expedient solution to a problem, analogous to patching a leaky radiator with duct tape.

leaves terminal elements of a **tree** data structure, i.e., ones that have no children.

Lisp Machine powerful personal computer developed at M.I.T. in the late 1970's to facilitate rapid software development, especially in the Lisp programming language.

metalinguistic abstraction defining a new formal language to facilitate the solution of programming problems in a particular domain.

message and **method** primitive concepts in the Flavors object-oriented programming system. A *message* is the name of a generic operation and it is *sent* to an object. When an object receives a message, it handles it with a collection of appropriate *methods* that have been defined by its class or its superclasses.

MIPS millions of instructions per second, a rough measure of a computer's computational capability.

mixin a class whose characteristics do not neatly divide a space and in fact are orthogonal to those of other mixins. Thus, SOCIAL-ANIMAL would properly be a mixin since it would be a useful component of classes for monkeys, dogs, wolves, whales, lions. By contrast MAMMAL, PRIMATE, CAT, etc. would not be appropriate mixins.

nodes intermediate elements in a **tree** or network data structure.

orders of growth, e.g. $O(n^2), O(n \log n), O(n)$, etc. Formally defined, $O(g(n))$ is the set of functions $f(n)$ such that there exist positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$. As a pracical matter, what people mean when they say a function is $O(n^2)$ is that its running time (or sometimes space), as the amount of data gets large, grows no faster than the function n^2.

quadtree data structure for organizing two-dimensional space so that searches can be performed in $O(\log_4 n)$ time. Every node in the tree has four children, hence "quad," and each child is allocated one quarter of the space owned by its parent.

subclass more specific type in an object-oriented programming system. Thus, IVY-LEAGUE-SCHOOL is a subclass of UNIVERSITY.

superclass less specific type in an object-oriented programming system. Thus, DOG is a superclass from which the class HUNTING-DOG might inherit.

surveyed point SITE CONTROLLER data structure representing a point whose elevation is known. This is the fundamental input to SITE CONTROLLER surface models.

tree data structure shaped like the plant of the same name, with a root at the bottom, branches that divide at **nodes** and terminate in **leaves**. Trees are useful for organizing data so that searching can be performed in $O(\log n)$ time.

Voronoi polygon (of a point p_i) the locus of points closer to p_i than to any other point in the set $p_1, p_2, \ldots, p_N$. In the plane, this locus ends up being a polygon around p_i.

Voronoi diagram (of a set of points S) a partition of the plane into polygons, each one surrounding one of the points in S. The dual of the **Delaunay triangulation**.

References

Abelson, Harold and Sussman, Gerald J. *Structure and Interpretation of Computer Programs.* McGraw-Hill, New York, 1985

Ainsworth, D.H. *Recollections of a Civil Engineer.* self-published Newton, Iowa 1901

Aurenhammer, Franz. Voronoi Diagrams—A Survey of a Fundamental Geometric Data Structure. *ACM Computing Surveys 23*, 3 (September 1991)

Bhateja, B. and Katz, R.N. A Validation Subsystem of a Version Server for Computer-aided Design Data. *Proceedings of the 24th ACM/IEEE Design Automation Conference* (Miami, FL, June 1987)

Bertino, Elisa and Martino, Lorenzo. Object-Oriented Database Management Systems: Concepts and Issues. *IEEE Computer 24*, 4 (April 1991)

Brooks Jr., Frederick P. The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley, 1975

Brown, Marc H. *Algorithm Animation.* MIT Press, 1988

Bureau of the Census, *Census of Construction*, U.S. Department of Commerce, 1990

Cattell, R.G.G. *Object Data Management.* Addison-Wesley 1991

Cattell, R.G.G. and Skeen, J. Object Operations Benchmark. *ACM Transactions on Database Systems 17*, 1 (March 1992)

Cheng, Fuhua. Estimating Subdivision Depths for Rational Curves and Surfaces. *ACM Transactions on Graphics 11*, 2 (April 1992)

Chin, Norman and Feiner, Steven. Near Real-Time Shadow Generation Using BSP Trees. *ACM SIGGRAPH 1989 Conference Proceedings*

Codd, E.F. A Relational Model for Large Shared Data Banks. *Communications of the ACM 13*, 6 (June 1970)

Demsetz, L.A. Task Identification for Construction Automation, *Proceedings of the 6th International Symposium on Automation and Robotics in Construction* (San Francisco), June 1989, a more complete version is available as *Task Identification and Machine Design for Construction Automation*, Ph.D. thesis, Department of Civil Engineering, M.I.T.

Detlefs, David. *Concurrent Garbage Collection for C++*. Technical Report CMU-CS-90-119, Carnegie Mellon University, May 1990.

Edelsbrunner, Herbert. *Algorithms in Combinatorial Geometry.* Springer-Verlag, 1987

Elmasri, Ramez and Navathe, Shamkant B. *Fundamentals of Database Systems.* Benjamin/Cummings Publishing, Redwood City, California, 1989

Farouki, R.T. Numerical Stability in geometric algorithms and representations, *Proceedings Mathematics of Surfaces III*, D.C. Hanscomb, editor. Oxford University Press, New York 1989

Foley, J.D. and Van Dam, A. *Fundamentals of Interactive Computer Graphics.* Addison-Wesley, 1982

Fortune, S. A Sweepline Algorithm for Voronoi Diagrams. *Algorithmica 2*, 153-157 (1987)

Fowler, R.J. and Little, J.J. Automatic Extraction of Irregular Network Digital Terrain Models. *ACM SIGGRAPH 1979 Conference Proceedings*

Gabriel, Richard P., et. al. CLOS: Integrating Object-Oriented and Functional Programming, *Communications of the ACM 34*, 9, (September 1991)

Gillett, Walter. Animated Plane-Sweep $O(n \log n)$ Delaunay Triangulation. Part of *SitePlanner*, a program sold by ConSolve Incorporated, Lexington, Massachusetts, 1988

Goldberg, A. and Robson, D. Smalltalk-80: *The Language and its Implementation.* Addison-Wesley, Reading, Massachusetts, 1983

Graham, R.L. An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set. *Info. Proc. Lett. 1*, 237-267 (1972)

Greenblatt, R. et al. The Lisp Machine in *Interactive Programming Environments*, eds Barstow, D.R. et al., McGraw-Hill 1984

Guibas, Leonidas et al. Epsilon geometry: Building robust algorithms from imprecise computations, *Proceedings fo the Fifth Annual ACM Symposium on Computational Geometry* (June 1989)

Hoffmann, Christoph M. The problem of accuracy and robustness in geometric computation, *IEEE Computer 22*, 3 (March 1989)

IEEE. IEEE Standard 754-1985 for Binary Floating-Point Arithmetic, IEEE. Reprinted in *ACM SIGPLAN 22*, 2 (1987)

Karasick, Michael, et. al. Efficient Delaunay Triangulation Using Rational Arithmetic, *ACM Transactions on Graphics 10*, 1 (January 1991)

Katz, Randy H. Toward a Unified Framework for Version Modeling in Engineering Databases. *ACM Computing Surveys 22*, 4 (December 1990)

Keene, Sonya E. *Object-Oriented Programming in Common Lisp—A Programmer's Guide to CLOS.* Addison-Wesley, 1989

Kornerup, Peter and Matula, David W. A Bit-Serial Arithmetic Unit for Rational Arithmetic. *Proceedings of the Eighth Symposium on Computer Arithmetic* (Como, Italy), 1987

Kemper, Alfons and Wallrath, Mechtild. An Analysis of Geometric Modelling in Database Systems. *ACM Computing Surveys 19*, 1 (March 1987).

Kim, W. and Chou, H.T. Versions of Schema for Object-Oriented Database Systems. *Proceedings of the International Conference on Very Large Databases*, September 1988

Lamb, Charles et al. The ObjectStore Database System. *Communications of the ACM 34*, 10, (October 1991)

McManamy, Rob. Chicago Shifts to Finger-pointing. *Engineering News-Record 228*, 18 (May 4, 1992)

Meyers, David, et. al. Surfaces from Contours. *ACM Transactions on Graphics 11*, 3 (July 1992)

Milenkovic, V.J. Verifiable implementations of geometric algorithms using finite precision arithmetic. *Artificial Intelligence 37* (December 1988) a more complete exposition of the same ideas is found under the same title in Carnegie Mellon Computer Science Department report CMU-CS-88-168

Montefusco, Laura Bacchelli and Casciola, Guilio. ALGORITHM 677: C^1 Surface Interpolation. *ACM Transactions on Mathematical Software 15*, 4 (December 1989)

Nielson, Gregory M. Minimum norm interpolation in triangles. *SIAM J. Numerical Analysis 17*, 1 (1980)

Object Design. *ObjectStore Technical Overview.* Object Design, Inc., Burlington, Massachusetts, 1992

Peucker, T.K. et al. The Triangulated Irregular Network. *Proceedings of the American Society of Photogrammetry Digital Terrain Model Symposium* (St. Louis, Missouri), May, 1978

Preparata, Franco P. and Shamos, Michael Ian. *Computational Geometry—An Introduction.* Springer-Verlag, 1985

Rice, John R. editor. Mathematical Software III. Academic Press, Boston, Massachusetts, 1977.

Rogers, David F. *Procedural Elements for Computer Graphics.* McGraw-Hill, 1985

Roos, Daniel. *ICES System Design.* MIT Press, 1967

Sabin, M.A. Contouring—The State of the Art. in *Fundamental Algorithms for Computer Graphics*, R.A. Earhshaw editor. Springer-Verlag, 1985

Samet, Hanan. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys 16*, 2 (June 1984)

Stasko, John T. Tango: A Framework and System for Algorithm Animation. *IEEE Computer 23*, 9 (September 1990)

Steele Jr., Guy L. *Common Lisp the Language, Second Ed.* Digital Press, 1990

Stutz, Bruce. Cleaning up: hazardous waste cleanup has become big business, and more-restrictive laws mean more profits for a growing number of consultants. *The Atlantic 266*, 4 (October 1990)

Teitelman, Warren. A Tour Through Cedar. *IEEE Software 1*, 2 (April 1984)

Ullman, Jeffrey D. *Database and Knowledge-Base Systems.* Computer Science Press, Rockville, Maryland, 1988 (in two volumes)

United States Department of Commerce. Construction Outlook for 1992. *Construction Review 37*, 6 (Nov/Dec 1991)

United States Department of Labor. *Handbook of Labor Statistics 1989.*

Vuillemin, Jean E. *Exact Real Computer Arithmetic with Continued Fractions.* Report 760. INRIA, 78150 Rocquencourt, France (November 1987)

Vuillemin, Jean E. Exact Real Computer Arithmetic with Continued Fractions. *1988 ACM Conference on Lisp and Functional Programming* (Snowbird, Utah)

Walker, J. et al. Symbolics Genera Programming Environment. *IEEE Software 4*, 6 (November 1987)

Weiser, Mark. The Portable Common Runtime Approach to Interoperability. *ACM SIGNPLAN Notices*, December 1989

Woo, Andrew et. al. A Survey of Shadow Algorithms. *IEEE Computer Graphics and Applications 10*, 6 (November 1990)

Wu, Henry. A Multiprocessor Architecture Using Modular Arithmetic For Very High Precision Computation. *1989 International Symposium on Computer Architecture and Digital Signal Processing* (Hong Kong) conference proceedings

Yip, G. May. *Incremental, Generational Mostly-Copying Garbage Collection in Uncooperative Environments.* Massachusetts Institute of Technology, Master's Thesis in Electrical Engineering and Computer Science, May 1991

Index