A Foundation for
Intelligent Multimodal
Drawing and Sketching Programs

by

Luke Weisman

M.S. Computer Science/Artificial Intelligence (1999)
Submitted to the Department of Electrical Engineering
and Computer Science in Partial Fulfillment
of the Requirements for the Degree of
Master of Science
at the
Massachusetts Institute of Technology
May, 1999

© 1999 Massachusetts Institute of Technology
All Rights Reserved

Signature of Author.................................................................

Luke Weisman
May, 1999

Certified by.................................................................

Randall Davis
Professor of Computer Science and Engineering

Accepted by........................................

Arthur C. Smith
Chair, Committee on Graduate Students
A Foundation for
Intelligent Multimodal
Drawing and Sketching Programs

by

Luke Weisman

Submitted to the Department of Electrical Engineering
and Computer Science in Partial Fulfillment of the Requirements for the
Degree of Master of Science at the
Massachusetts Institute of Technology

Abstract
Computers should be able to assist an engineer or designer at all stages of design without chasing the engineer away or frustrating her. In order for this to occur, the computer needs to be able to understand what the engineer is doing and provide unobtrusive help. Furthermore the engineer needs to be able to interact with the computer in a natural manner. To support this design methodology I implement a foundation on which a programmer can make such applications. I provide two illustrative applications which demonstrate the foundation's power.

Thesis Supervisor: Randy Davis
Professor of Computer Science and Engineering
# Table of Contents

1 Table of Contents ........................................................................................................... 3  
2 Introduction ..................................................................................................................... 5  
   2.1 Illustrating the platform ............................................................................................ 7  
      2.1.1 The Finite State Machine Application .......................................................... 7  
      2.1.2 The House Application ................................................................................ 14  
3 Motivation & Vision ......................................................................................................... 22  
   3.1 A Brief Overview of the Platform's Behavior ......................................................... 26  
4 Implementation of the Platform ....................................................................................... 29  
   4.1 The Base Components ............................................................................................ 30  
      4.1.1 Surface ......................................................................................................... 30  
      4.1.2 Widgets ........................................................................................................ 31  
      4.1.3 Recognizers .................................................................................................. 31  
      4.1.4 Modules ........................................................................................................ 33  
   4.2 The Maintenance Parts and Systems ....................................................................... 36  
      4.2.1 Undo and Redo ............................................................................................... 36  
      4.2.2 The Recognition Queue ................................................................................ 38  
      4.2.3 Registration ................................................................................................... 39  
      4.2.4 Fast Recognition ............................................................................................ 40  
      4.2.5 The Conversation System ............................................................................. 41  
5 Making an Application ..................................................................................................... 45  
   5.1 Defining the Widgets ............................................................................................... 45  
   5.2 Writing the Recognizers .......................................................................................... 49  
      5.2.1 Start Building From the Bottom Up ............................................................... 51  
   5.3 Build the Data Structure Recognizers .................................................................... 52  
   5.4 Implement the "Changing" Recognizers ................................................................. 55  
   5.5 Assemble the Recognizers ..................................................................................... 57  
   5.6 Overlay the Talking ............................................................................................... 58  
   5.7 Enhance .................................................................................................................. 61  
6 Pitfalls in Making the System ......................................................................................... 62  
   6.1 Undo and Redo ....................................................................................................... 62  
   6.2 Who Does the Work ............................................................................................... 62  
   6.3 Granularity .............................................................................................................. 63  
   6.4 Modes ...................................................................................................................... 64  
7 Related Work .................................................................................................................. 65  
8 Where to Go ................................................................................................................... 68  
   8.1 The Intelligent Room .............................................................................................. 70  
   8.2 Physical Devices ..................................................................................................... 70  
   8.3 Dirty Sketches ........................................................................................................ 71  
   8.4 Simultaneous Being of Widgets ............................................................................. 72  
   8.5 Learning .................................................................................................................. 72  
9 Conclusion ....................................................................................................................... 74  
10 References ..................................................................................................................... 76
Acknowledgements

I would like to thank the people of the Intelligent Room Research Project, who have made my experience an enjoyable one, and Randy Davis, who has provided excellent support, critique, and encouragement.

This material is based upon work supported by the Advanced Research Projects Agency of the Department of Defense under contract number F30602-94-C-0204, monitored through Rome Laboratory. Additional support from Ford Motor Company,
2 Introduction

There is a need for intelligent computer drawing environments where engineers can easily express their ideas, yet be understood by a computer. We are unwilling to trade understanding for engineer's drawing convenience; hence we need an unobtrusive interface. A good and time-honored engineering sketching method is paper and pen; this is the model we start from.

A pleasing and plausible extension of the pen and paper would be an environment where a computer watches the engineer draw and discusses with the engineer what was going on. The engineer, through voice interaction, could easily tell the computer what mistakes the computer made, what is being drawn, and answer any questions the computer might have. By using speech synthesis the computer in turn could interact with the engineer in a way that did not require that the engineer move his eyes from the paper or his hand from the pen.

More importantly, the computer could manipulate the drawing itself. At a minimum the computer could straighten lines, make circles circular, and squares square. A step up from this would be a system that had limited comprehension of the engineers' activities, allowing it to notice mistakes. For example, if the engineer was designing a house and failed to provide a bearing wall of adequate thickness, the computer might point this out, making it a partner rather than a passive tool. By being active, the computer can be much more helpful. Because this interaction would be through voice, the engineer would not even have to lift his pen to respond. In addition, the engineer would not have to deliberately check for help or suggestions, but could instead respond to them as they occurred. This is unlike such "helpers" as Microsoft's paper clip where, when it has an "idea," you have to stop what you are doing to indicate that you are not interested in what it has to say. A good assistant should be able to be ignored.
With sketching, and other free-form creative processes, any kind of structural limitation is irritating and possibly detrimental. These programs should be intuitive; they should not require the user maintain a vast body of knowledge, meaning the user should be able to easily express knowledge of the engineering domain without having to know a lot about how to use the application.

My thesis is a base platform that supports the building of applications that adhere to the principles outlined above. This base platform encourages the programmer to design applications in keeping with these principles, and allows for highly flexible interaction. This base platform is illustrated with two example applications.

The first example application is a finite state machine drawing program. A program that helps draw such machines is one example of how this model of interaction assists an engineer in his task. The second, more complicated, example is an application for sketching floor plans of houses. In this application, I begin to explore how a CAD system might be implemented through a sketching interface with straightforward interactions yet the power and expressiveness of current CAD systems.

My system can be viewed in terms of three fundamental levels of behavior. The base level is getting users to feel they are drawing rather than "using a computer" and having the computer see every stroke made. The next level is having the computer adjust drawings as necessary, e.g. straightening lines, smoothing curves, etc. The final level is having the computer understand what users are doing and unobtrusively assisting the users' endeavors.

The system is made accessible to the programmer so the knowledge base of a particular engineering task can be changed while still taking advantage of fundamental features of drawing. The system allows a programmer to easily layer knowledge by using recognition algorithms in a mix-and-match
fashion. The system encourages programmers to describe recognition in a bottom up approach; as the
computer recognizes simple things, more complicated things can be built from them.

2.1 Illustrating the platform
To illustrate the base platform and the general concepts I am trying to explore I wrote two sample
applications: a finite state machine sketching program, and a floor plan sketching program. The first
of these exercises some very basic sketching capabilities, and second represents a more useful appli-
cation.

2.1.1 The Finite State Machine Application
The finite state machine sketching program sprang from a conversation I had with my peers when I
was first writing the base platform. When my associate Brenton Phillips saw my program recognize
circles and lines, the first thing he said was, "I need that, I need that to draw finite state machines for
my class." This response is a perfect illustration of how this kind of technology is needed for daily
tasks. It is difficult to get a diagram of a finite state machine on a computer. People are often frus-
trated in their attempts to draw flow charts or finite state machines in the various CAD or drawing
programs currently available. The problem is frequently either excessive complexity or lack of con-
textual support and expressiveness; the overhead required to do simple things with the complicated
applications is prohibitive, and simple applications end up demanding large amounts of work. As a
result, today's drawing programs are of little assistance in sketching and design.

In the end people avoid using computers for initial sketching tasks as much as possible. The only
way users would use a computer to make such a diagram as part of a creative process would be if the
computer did not get in their way.
Because my program understands what a finite state machine is, activities such as moving nodes around is trivial. The ease of use stems from understanding the drawing beyond simple geometric knowledge. By knowing that the user is drawing a finite state machine, the application can automatically do low-level tedious tasks (such as aligning nodes and edges appropriately).

_The Interaction_
The user sits in front of a computer with a headset microphone and a stylus. The screen in front of her is an empty surface with no menus or toolbars at all. A cursor on the screen shows where the stylus is on the tablet. When one presses the stylus to the tablet, one sees a corresponding mark on the screen. When one releases the pressure on the stylus, the program processes the line and makes any changes necessary. Sometimes the program dynamically changes the screen while the user is in the middle of a stroke (this is called fast recognition). The user can talk to the computer and give it a variety of commands. The computer can, through speech synthesis, respond through a set of speakers to either side of the screen.

In the following scenario, Sally, a user, uses this system to sketch out a finite state machine. She knows the problem requires five nodes, so she quickly sketches them out, making one of them bigger to indicate to herself that it is the center node.

![Sketch of a finite state machine](image)

_Figure 1 – This is what Sally actually drew._
As she sketches each node out, they are automatically labeled A through E. The last one, being the center node, she wants to really label, so she tells the computer, "name that node." The computer responds, "what you want to name it?" And Sally replies, "start." The computer then re-names the center node to "start".

Figure 2—The nodes sketched out and ready for further work. Sally has labeled the big node "start" through a voice command.

Above is the result of this work. However, what she actually drew looks like the following diagram. The computer has done simple cleaning up of the lines, which it can do only because it knows what in particular is being designed.

Now Sally sketches edges, labeling them as she goes. In this case, since the regular language in the assigned problem consists only of 0s and 1s she can name edges by saying, "name that 0" as she draws them.
Sally draws edges as anyone would. The computer replaces them with straight double lines and aligns them to any nodes they are connected to. Arrowheads are recognized and attached to the nearest endpoints, and things in general are kept clean and legible. The computer also gives the edges names as best it can, knowing that sometimes it will guess right, and that it is just as easy to change a name as to add one.

As Sally is sketching the edges, things seem to be going well. However she manages to work herself into a corner where she can no longer clearly see how the machine works. To aid herself she draws a few more nodes so that there are fewer transitions going from node to node because that helps her separate out the problem. In the process of adding all of these intermediate nodes, her drawing sur-
face is getting crowded. Luckily, she can simply grab nodes and drag them around and the edges follow them.

![Figure 5](image1.png)

**Figure 5**—A node being dragged to a new location with the edge automatically following.

Furthermore, if she decides she needs a node between two nodes that are connected by some edge, she can simply draw on top of the edge, and the program interprets this as a desire to insert the node and does so. This illustrates exploiting the knowledge that the user is drawing a graph to give a natural interface. In other drawing programs, Sally would have to draw the node, delete the old edge (or at least adjust it so it points to the new node), and then draw the other half of the edge going from the new node to the second old node.

![Figure 6](image2.png)

**Figure 6**—Adding a node between two other nodes merely by drawing over the edge.

When she finally gets her now quite large finite state machine in order, she decides she wants to simplify it, so she deletes some bridge nodes, and the program automatically collapses the edges in the correct fashion such as to preserve the language the finite state machine recognizes. This is an-
other example of exploiting contextual knowledge of finite state machines to assist the user. This is the kind of thing that should be prevalent throughout all of the sketching programs. With this latter step, Sally can either delete nodes by name by saying, "delete node D," or she can select the node with her stylus and click on it again to delete it.

![Diagram](image)

Figure 7—Deleting a node by clicking on it preserves the language defined by the finite state machine. Note how the 1 and 0 edges are merged into a "10" edge, and the 1 and 1 edges merged into a "11" edge.

Sally can make a node an accept node by drawing a circle around it. In essence, Sally draws what she wants—in standard mathematical texts an accept node is two concentric circles, and this symbolism is exploited by the application. The user draws what they know—there is no indirection needed to express oneself. Other applications might demand the user know a set of steps to make a node an accept node, thus requiring additional knowledge unrelated to the task at hand. This drawing recognition style avoids this.

Sally can also make self-pointing edges, edges which go from a node to itself, by drawing a loop on top of the node. In this case the application uses another form of context—the relationship between nearby objects—to know that it is an edge and not another node, for these two things look very similar to each other.
Figure 8—Drawing a loop over a node could be another node or an edge. Only context separates the two possibilities.

Sally, getting carried away with the concept of simplifying her finite state machine, deletes one node too many and can no longer understand her machine. However, she can say "undo that" and the computer will do so. It she wants to restore the state she can say "redo that" to get back to where she was.

Figure 9—The final finite state machine

At one point, Sally decides that an entire branch of the finite state machine is not used or needed. To delete it you draw a circle around it and then draw a line through it. She can also ask simple questions such as "how many nodes are there?"
The program also arranges edges, bending them if necessary, to make the drawing clear. Because the application has a built-in conception of what a clean and clear diagram is, and what the meaningful information in the diagram is, it can do useful things when displaying it.

Throughout this interaction the application is exploiting context to assist the user, allowing the user to sketch a finite state machine in a far more natural way than any current CAD program would permit. Even dragging a node and having the edges follow is simpler than in alternative programs where to get this behavior, you have to inform the application that the edge and node are connected. With this application relationships are implicit; the program assumes that if the user has drawn an edge and node overlapping, they are connected. This type of knowledge is obtained through knowing the context; task-independent drawing programs cannot be as smooth and intuitive at this task.

2.1.2 The House Application
The second application I designed using my base platform is a utility to sketch the floor plans of houses. The application allows the user to rough out a general plan for a house, including the walls, doors, and shapes of the various rooms. This application is not supposed to provide finished and polished floor plans that one would hand to an architect; it is rather a vehicle to explore the very early stages of design. This means it must allow complete freedom in drawing, and also means that the level of understanding and built-in knowledge representation does not need to be as complex as a fully featured sketching recognition program for architectural blueprints would require.
This application also illustrates the full range of features of the platform we created; each of the various abilities of the platform are given a brief tour.

In many architectural sketches of floor plans—such as back of the envelope sketches generated during preliminary design conversations—the architect draws out the floor plan, and then if he or she needs to change it, simply draws heavier lines over the old ones. At the end of the day, while the people watching the architect know the development process and can parse the resulting sketch, others will see only an incomprehensible scribble.

One of the main purposes of my drawing program is to enable the user to manipulate the drawing very quickly—as quickly as simply drawing over the old drawing—yet keep the result cleaner and more comprehensible. This is achieved by enabling the architect to easily select and move pieces of the drawing, and by exploiting knowledge of the domain, merging the pieces back together when the architect is done modifying them.

Making it easy to manipulate drawings allows users to easily wander in design space, exploring various options. In the pen and paper world manipulating drawings of any significant size involves extensive amounts of copying which acts as a deterrent for exploring alternatives. However, if it is truly easy to manipulate the drawing, these paths can be explored with little effort.

Tools like this are particularly useful for novice architects, or to people who are not architects at all, because they can try alternatives, yet things are easily undone. Applications of the sort will help to remove the fear of heading down the wrong path, enabling designers to make creative leaps and get out of the established routes of generic design.
The impact of design tools in the design process is well-known. This style of interface software may open up new directions in design that people did not bother with due to the difficulty. Modern-day CAD programs coupled with 3-D modeling software allowed architects to design three-dimensional buildings—buildings almost impossible to visualize through sketches and two-dimensional drawings. It is not that it was impossible to do this kind of design without these modelers, it was simply that to explore the space efficiently would have required too much time. Tools which are used to express inherently limit the designer's imagination; new tools lead to new design possibilities.

The Interaction
The user wishes to sketch a floor plan of his new house. He begins by roughing out the outline of the building, and then adding some interior walls. As soon as the program sees closed shapes, it assumes the user has drawn a region, replaces the surrounding lines with walls, and labels the interior as a region.

![Figure 11—Drawing the connecting line on the right causes the system to recognize a region and give it a unique name, "region 0"](image)

If a user draws walls over a region, the application assumes that the region is meant to be divided into two smaller regions and does so. This allows the user to rough out the outline and then subdivide the main region into the interior pieces quickly and efficiently. However the user is not bound to this particular design style. He could alternatively build up the entire diagram by drawing the
interior pieces one at a time. This might occur, for example, if the user drew the central hallway, and then drew all the rooms coming off of it.

The application attempts to guess what a region is by looking at features such as relative size of the region. If the application cannot guess what the regions represent it labels them with unique names such as "region 1" or "region 2".

![Diagram](image)

**Figure 12—The user divides the region into 2 parts with a line.**

The user can define any of the regions by saying, "name region one" and the computer responds, "what you want to name it?" The computer also knows many standard names for regions, allowing to user to say things such as, "region two is a kitchen."
Regions are more than polygons with names. The computer can look at them and understand that they represent particular types of living space based on what they are called. This gives the computer some basic knowledge about different standard regions, allowing it to, for example, reason about the relationships between various regions, or the sizes of various regions. For example, if the user had drawn the kitchen and the bathroom the same size, the computer points that out as being an unusual choice.

Doors are drawn by drawing a box over the wall. The application then puts a specific icon down where desired. The application uses a fixed size for the doors, using built in conceptions of dimension. This means the user does not have to tweak and measure lines carefully to make sure the sketch is relatively to scale. However, on the down side this means the plans always have doors which are effectively the standard 36" wide.

Moving along, the user divides the large region into two parts and then labels the parts as a bed room and a dining room. Now the user has the following:
Figure 14—The diagram after being divided and labeled.

Now the user begins to explore the shape of the house. He moves the bedroom to the other side of the dining room by dragging it. After the user releases the mouse, the system merges the moved region in with the other regions so the user can see the final result.

Figure 15—Moving the bedroom by dragging and letting go. Note how the dining room is deformed and readjusted to connect smoothly to the newly moved bedroom.

Note that through all this the walls bordering the regions have been "following" the regions around. When the user moves the bedroom, for example, any bordering walls divide into two new walls. Again, this kind of adjustment of the drawing can only spring from a built-in understanding of what is being drawn.
Similarly, when the user places the bedroom near the other side of the main diagram, the two overlapping walls are turned into one and the system notes that the bedroom is now connected in its new place. The system maintains a representation of region connectivity, so it is possible to query it with questions such as, "what is the kitchen connected to?"

Now the user decides that he wants a smaller house, so he deletes the kitchen entirely by circling it then drawing a line through it. (The user could have also have said, "delete the kitchen.") If the user then changes his mind he can undo the delete by saying, "undo that."

![Diagram](image)

**Figure 16—The kitchen is deleted by drawing a line through it.**

The system promotes ease-of-use by providing natural and intuitive ways of doing straightforward tasks, and by enforcing a small useful set of semantic relationships in the diagram through all of the manipulations the user does to it. By reacting appropriately when the user does natural things, the system maintains a sensible diagram regardless of the user’s understanding. This allows a novice to sketch out diagrams without undue concern as to the details. This is part of what sketching is—expressing ideas without being tripped up by little things.
The base system we created supported the building of the above two applications. These scenarios offer a starting point. Our attempts at making a base platform have also served to uncover some of the issues of complexity and stumbling blocks for the programmer attempting to program in this new paradigm.
3 Motivation & Vision

My thesis offers a base platform that supports a class of applications that I have dubbed "interactive sketch recognition". These applications provide an environment where one can sketch. In the following section, I describe what interactive sketch recognition is and tie this view to the system we have designed to motivate the design decisions underlying the system.

When a user uses an application, they are existing in a realm that obeys certain laws. The user affects this realm through a variety of input devices—in most current applications these devices are a mouse and keyboard. The user then observes the environment through another set of devices. Currently, output devices are usually limited to speakers and a monitor. Through these tools the user is controlling a virtual presence in an environment that reacts according to the laws of the program running.

Viewing an application in this fashion makes it clear that the environment should have certain essential features; in particular it should be natural and intuitive. "Natural" means the user should feel at home interacting with it. This covers a variety of issues such as physical comfort and speed of data entry. "Intuitive" means the environment should react to the user in ways the user would expect, and the user should understand how to affect her surroundings. An environment that does not present a consistent worldview is difficult to grasp and may require memorized lists of interactions. Often this can frustrate the user, especially a novice, and destroy any natural feeling the environment might have had.

Most applications attempt to deal with these issues by tying the environment metaphorically to some known environment, allowing the user to feel more comfortable because it is familiar. The user can
also employ her own intuition about how the known environment works to affect this new similar environment. The best known example of this is the GUI desktop metaphor.

The environment I envision for my interactive sketch recognition is the environment of pen and paper. Ideally, interacting with my application would be holding an electronic pen and drawing on a large display which has the feel of paper. The user would see the electronic pen draw exactly where the tip of the pen is, thus tying the cause and the effect closely together. For all purposes the user would be drawing as she would with non-enhanced drawing equipment.

However, this additional hardware is a branch of research in its own right, so instead we have the environment of a mouse or tablet and a monitor. Nevertheless, we use the mouse and monitor in our application in exactly the same way as a pen and paper, except that the input and output are separated into two devices. Unfortunately the feel of the mouse is (as most people know) not very similar to a pen due to its awkwardness, different grip, and lack of precision. This problem is somewhat alleviated by using a tablet and monitor, but still having the input and output separated greatly reduces the pen and paper feel of the interaction. We hope that despite the unnatural aspects of this arrangement, the overall idea will still be apparent. Regardless, the software layer is the same.

The envisioned sketching environment also includes a virtual presence that "looks over the shoulder" of the individual sketching. This presence can manipulate the drawing in an intelligent fashion, and discuss it—through voice recognition and synthesis—with the designer the task at hand. This aspect of the platform is conversational, thus extending the metaphor of designing with an assistant at one's shoulder, rather than using myriad tools on one's own.

The environment of sketching on a piece of paper with an assistant offering assistance and changing lines as needed is intended to be easily understood and comfortable. Providing an assistant is in-
tended to reduce the chance that the user will feel lost, fail to understand what is going on, or fail to understand how to do something. If nothing else, the user should feel safe because she can always ask the assistant for help. In addition, by using the model of sketching on a piece of paper, the knowledge the user has accumulated over the years about manipulating pen and paper can be readily applied.

Other application domains are heading down this general route of environments with assistants; for example, Microsoft products now have an assistant in the form of a paper clip. The difference between the paper clip and the type of assistant described above is that our assistant can be told to do things to make ones tasks simpler and more efficient, as well as be queried for information. Furthermore, our assistant is active and does not need to be directly queried but instead provides information unobtrusively without the user's initiative.

The system must not limit the user. Sketching is very different from drafting, copying other diagrams, or drawing finalized plans. Sketching is completely free—there are no rules to sketching, the user can draw whatever she wishes. Sketching must not be inhibited; it is like brainstorming, an activity defeated as soon as it is channeled or structured. Mark Gross [Gross 95] has made many arguments along these lines about the importance of interacting with the computer through sketching. He provides very good arguments such as freeing the creative mind and not chasing users away.

The user should be able to do whatever she wishes immediately. Some systems have so many features that it seems that there is no limitation—everything is possible. However, precisely because of the large number of features, the user often cannot find the appropriate tool needed to achieve a given task. Even if the user knows exactly what to do, activating the tool often requires several steps
that breaks the flow of design. With the sketching program the user is always holding the only tool needed for any task—the pen.

Sketching is a simple process. The results can be highly varied and expressive, but the input—the mode of interaction—is very straightforward. By providing a familiar system such as a pen and paper there is no knowledge required to begin to express ones ideas. The learning curve, in this sense, is very shallow. The amount of knowledge needed to accomplish a task is small, save for the knowledge about the actual task itself. The application itself has only a small body of lore. There is no point in making someone learn how to use a tool, the person should instead learn how to express themselves with the tool.

![Diagram](image)

**Figure 17**—The knowledge needed to complete a task with a non-intuitive application is far greater than using a simple one, such as a pen.

In the figure above, I loosely sketch out the fields of knowledge needed to create a blueprint on a computer. When creating that same blueprint on paper, an entire area of knowledge—using the ap-
plication itself—is replaced with knowledge of how to use a pen, which is much smaller. Up to now, skill in manipulation of the pen and paper has been replaced with needing to know a large amount about operating a particular program; there is no need for this and I believe it can be avoided. An incompetent illustrator can still communicate their ideas with a pen and paper, while a novice application user would be completely stuck. Even further, with an assistant that can lookup information by itself, much of the rote knowledge of symbols and codes might not be required of the user, thus reducing the need to maintain vast quantities of trivia in the mind.

3.1 A Brief Overview of the Platform's Behavior
We have created a base platform in which a developer can easily write applications that fit the general features described above. We took a distributed approach; we designed a system where the programmer, in a bottom-up manner, writes pieces of code for straightforward recognition of specific objects or gestures, and then combines these pieces to create the complex whole. In general, primitive lines made by the user are recognized as simple components, and then in turn clusters of simple components are recognized as more complicated components, and so on until the overall high-level parts are recognized. Alternatively, a stroke can be recognized as a "gesture command", for example a line going from an already existing object to another area on the screen could be interpreted as a command to move the object.

We separated the recognition machinery into individual components, called modules, so that it is easy to set the context by choosing which recognition modules are active at any given time. For example, the module that recognizes nodes in a finite state machine is turned on when the user is designing a FSM.
The main form of recognition is to have the program analyze what the user has done after each pen stroke. A stroke is begun when the pen hits the paper, is defined by where the pen goes, and ends when the pen lifts from the paper. At the end of the stroke, the recognizers, in a predetermined order, look at the stroke in the context of the previously recognized components. If the stroke matches any of the criteria they are looking for, they replace it with the recognized object, or carry out the recognized gesture command.

This is usually sufficient. However, sometimes an immediate response to a user's stroke is needed, so there are also "fast recognizers" that look at the stroke point by point as it is being created, and can effect change immediately. This is used, for example, in implementing a "drag and drop" interface. Due to the computational load of this, these recognizers are required to be kept simple. Furthermore, it is very difficult to determine what the user is doing mid-stroke, so only rarely and in very clearly set circumstances is this recognition technique appropriate.

The speech system is overlaid on top of the general recognition system. It uses the Metaglue platform designed by the Intelligent Room project that allows ViaVoice—a speech recognition program made by IBM—to be tied to a host of Java agents.

The sketching program is encapsulated in an agent in the Metaglue agent system. Other agents can then reach into the sketching program agent and tweak it or query it for information. This in turn allows for future extensions where external programs can gain access to information in the sketching program, and also tweak the drawings there based on such things as simulation data or external commands from other sources.

This endeavor primarily is designing a base platform that other people can use in the future to design sketching programs. We have created a base layer that both supports a certain feel to sketch recog-
nition, and which makes it possible to create domain specific applications easily. The two applications that I discuss as examples throughout this document serve as examples of this platform rather than finished applications.

The goal of the base platform is to make as easy as possible the implementation of whatever decisions the designer selects. I hope that by providing this level of help, the fairly daunting task of creating a truly understanding sketching environment is more accessible. Because the entire platform is component based, it is easily configurable and extensible. The two examples I have implemented suggest that the base platform can indeed be applied to different contexts.

From this starting point I hope it is possible to head in the many exciting directions I have outlined above. Happily, some of this is happening now; one of my associates, for example, is using our base platform as a front end to sketch the interacting components of mechanical parts. These diagrams will then be fed into a simulator, and subsequently redisplayed on the drawing surface, thus creating a sketching interface to a physical modeler.
4 Implementation of the Platform

The sketch recognition system base platform consists of a few basic elements with well-defined interactions between them. An application programmer extends these elements in a standard object-oriented programming fashion to create specialized objects that capture the context and parts of a particular application domain. In the following chapter, I describe these fundamental elements and how they interact. I illustrate these features by explaining how I implemented the two example applications. The discussion is fairly high-level, but should serve to illustrate how to, in general, design and build the base platform itself.

![Diagram](image-url)

**Figure 18**—The recognition involved in a line making a region. (1) First there were 3 lines. (2) Then the user adds a stroke. (3) The stroke is recognized as a line. (4) The line in conjunction with the three other lines is recognized as a region. (5) The region is given a name for reference.

Recognition is event driven. As illustrated in the figure above, the user first draws a stroke, causing activity to commence. First the system recognizes the stroke as a line, then recognizes the line in conjunction with the other 3 already existing lines as a region, and then labels the new region "region 0." Each step of this process was carried out by a component called a recognizer with no knowledge of the other components carrying out the other steps. When no activity is caused by any of the components changes, the system has finished recognition of the original stroke and waits for further input.

This process is broken down in more detail below; the individual parts are described along with the roles they play in this process. First is a description of the foundation of the base system, and fol-
ollowing this is a discussion of a collection of components which add a lot of additional power and expressiveness to the base platform and allow for complex interactions.

4.1 The Base Components
The system architecture comes in two basic levels: one deals with core recognition and the other provides extras which give added power to the overall system.

4.1.1 Surface
All sketching, recognition, and interaction happen on a single plane called the Surface. The Surface is the piece of paper in the pen and paper model. Perhaps in the future there will be more than one Surface—the code is structured to allow for this—but so far we have only needed a single surface at a time. The surface defines a draw method that allows it to be painted on a display. It also has an internal coordinate system which all the objects it holds use.

The surface provides control of the entire program. It runs the recognition process and contains all objects related to the recognition and processing of events. The surface is event driven, so when the user draws a stroke, the Surface object is activated and processes the user's activity.

The surface maintains a list of modules that in turn hold the recognizers that do all the activity. The surface keeps the list of modules in order of priority, and alerts them when a new stroke is entered by the user. The surface also controls when it should be drawn to the display, and, when invoked, draws all visible widgets as well.

The Surface also provides exterior hooks so remote programs can reach in and affect change or run queries. An example of this—the voice interaction—is discussed later.
4.1.2 Widgets
On the surface are Widgets. A Widget is a basic drawing element. Widgets range from lines to gears. Depending on the nature of the application, a Widget can be arbitrarily complicated. The most basic Widget is the Squiggle, a single stroke from the user's pen. Widgets are changed through a process called recognition; the most general form of recognition is a simple widget being recognized as a more complicated one and replaced with that higher level object. For example, a Squiggle might be recognized as a Circle that in turn might be recognized as a Node. A slightly more complex example, illustrated above, is a region being recognized. Here the user drew a stroke which was recognized as a line widget. Then that new line widget along with the three already existing line widgets are recognized as a single region widget, removed from the surface, and replaced by the region widget.

Widgets can also be invisible things. For example there are click widgets which are clicks from the user's stylus, and selection widgets which are selections of other widgets. They are used as event placeholders to aid the recognition process. If the user clicks the stylus, this is recognized as a click widget, then the click can be recognized as something else, depending on context.

4.1.3 Recognizers
The objects that do the actual work of recognition are called Recognizers. Each recognizer is responsible for some small aspect of recognition. For example, there is a recognizer that recognizes circles from squiggles, one that recognizes lines from squiggles, one that recognizes nodes from circles, etc.. The recognizers register interest in different types of widgets. When a widget of that type appears on the Surface, all the recognizers that have registered interest in that type are sequentially alerted to its existence. This is done by the Surface making a call to each recognizer object in turn, passing along a pointer to the Surface and the widget being recognized. Each recognizer gets an
opportunity to modify the Surface and the widget. The order in which recognizers get alerted is managed by the Surface and by objects called Modules, which are described below. When a recognizer gets control, it can return a flag informing the Surface whether recognizers further down the queue should also get a chance to see the Widget. If it returns true, this means the widget has been consumed and should not be shown to any other recognizers. If it returns false, then the widget has not been used up, and other recognizers should get a chance to recognize and change it.

For example, two of the recognizers in the finite state machine program handle circles (which are turned into nodes) and self-loop edges (edges which go from a node to itself). Both these things look the same (i.e. a rough circle) and so the order of the two is important. If the user draws a loop over a node for a self-pointing edge, and the circle recognizer sees it first, it will blindly turn it into a circle, stopping the recognition process for the stroke, and the stroke will not get turned into an edge as desired. This means the self-looping edge recognizer has to look at the stroke before the circle recognizer, because it is more choosy as to what strokes to take: it recognizes only those strokes which are on top of nodes and look like circles, rather than strokes which just look like a circle.

To summarize, the system's basic recognition cycle has five steps. (1) User draws something on the surface. (2) This thing, which is made into a Squiggle Widget object, is passed sequentially to all recognizers interested in Squiggle widgets. (3) One at a time, these recognizers can examine the widget, perhaps in the context of other objects on the surface, and then modify it and/or the surface. (4) Any new widgets created are in turn recognized by the relevant recognizers. (5) Eventually there are no widgets waiting to be recognized, and activity stops until the user draws something else. At any given time during the recognition process, there is a single widget that is the source of the current activity. We call this widget the spark widget.
Recognizers do not have to affect the surface. Some recognizers serve only as monitors of data. A very primitive example of this is a recognizer that counts the number of nodes on the surface. If this kind of information were important to keep around, and were too expensive to regenerate every time it was needed, such a recognizer could prove quite useful. These recognizers still register interest in particular types of widgets, but instead of making changes, they only adjust their data structure and then return false so that recognition continues uninterrupted.

A third style of recognizer is the type that tweaks the surface but does not stop the recognition process. For example, if a Region is drawn and recognized, the AutoNamer recognizer gives it a unique name so the user can talk about it easily. However, the AutoNamer does not stop recognition, but instead tells the system to continue to recognize the new Region and continue to make changes. This style of recognizer can be very dangerous however; by making a change and then allowing recognition to move on it becomes hard to trace who did what, and it becomes difficult to think about what activity will be caused by the recognition of any particular widget. This issue is essentially the same as side effects in function calls—it becomes very hard to trace what did what. For example, if the AutoNamer is forgotten, it is very difficult to determine when and how the Region was named, and this could lead to errors if changes are made without activating it. However, this power has proved to be crucial in building a more expressive recognition system.

4.1.4 Modules
Recognizers are collected into modules. Each module represents a different context, sub-context, or type of activity to recognize. The modules are kept in an ordered list and when a widget needs to be recognized, each module, starting with the one at the front of the list, get a chance to process the widget. Just as with recognizers, the module can return true or false indicating whether modules later on down the list should also get a chance to see the spark widget. The most basic form of the
module, called the BasicModule, holds a collection of recognizers in an ordered list; when invoked, this module gives each recognizer a chance to recognize the spark widget in turn. If any of the recognizers return true to the recognition process, the entire module returns true as well, halting the recognition process for that particular spark widget.

A module divides all its recognizers into sets based upon what the recognizers are interested in. So the Geometric Shape recognizer module has, for Stroke widgets, the line and circle recognizers, and for line widgets the polygon recognizers, and for polygons, the triangle, rectangle, and square recognizers. When the surface gives a Widget to a module to recognize, the Module hands it off to the subset of recognizers interested in widgets of that type.

An application can have many modules. For example, the simple application of the finite state machine recognizer has a simple geometric shapes module, a node and edge module, an interactive gesture module, and a selection module. Modules can be shared across different applications, for example both the finite state machine recognizer and the house sketching program use the selection module and the simple geometric shapes module.

![Figure 19—An example of selection. (1) The user clicks on the region, making a small mark. (2) This is recognized as a click and then as a selection of region 0 (3) a modify mode module is placed on the Surface's registration stack and the widget is turned red to indicate it has been selected.](image)

Modules can be added and removed easily. For example, in the house sketching program, if the user clicks on a region, she starts a recognition process, illustrated above, that goes as follows:
The surface creates a squiggle (a very short one, perhaps even as small as a single point).

All recognizers interested in the squiggle widget are given a chance to recognize it. One of the modules, the "selection" module, has a click recognizer that recognizes the squiggle as a click widget.

The surface runs the recognition process on the newly created click widget.

In the "selection" module, a selection recognizer notices that the click widget is lying on top of a previously recognized region widget on the surface. It removes the click widget from the surface and then installs a "modify mode" module. This module has a collection of recognizers that interpret strokes as commands to move the selected region widget around the screen. One of these recognizers interprets another click as a deactivation of the module, and removes the "modify mode" module from the Surface so that recognition can return to normal.

Modules can also be added through voice, or through other ways of recognizing a change in context. For example, if the user says, "I am drawing a finite state machine," the corresponding agent installs a finite state machine recognizer that would be able to process the drawing and all future strokes in terms of the new context.

In our current application model, when one loads up the system, an initialization file called "recog.ini" is read in. This file lists the names of modules to load into the system. The modules are loaded in the order found in the file. Each module is defined by another file that lists the recognizers that should be included in the module, again in order of priority. Other modules have to be loaded in explicitly through other Java source code, for example by remote programs that are interacting with the sketching environment, or recognizers that load in modules to deal with minor shifts in context such as the modify mode module.
4.2 The Maintenance Parts and Systems
The system has additional capabilities not needed for base recognition, but crucial to making a flexible system that can adapt dynamically to the user and new circumstances. These are described below.

4.2.1 Undo and Redo
The Surface is responsible for storing a record of all activity so that the user can undo and redo whenever she desires. The Surface does this through a TransitionRecorder object that stores a collection of transitions that in turn represent incremental change to the current state. By running through these transitions sequentially one can go back or forward in time. The TransitionRecorder looks at the universe as if it were a piece of film. At any given point in time there is a static image of what the world looks like. A single transition carries the world from one frame to the next frame. All transitions have an inverse so that it is possible to go in either direction. The TransitionRecorder maintains a pointer to the current frame that represents the current state of the system. The TransitionRecorder also allows marking a frame in the film so that you can rewind or fast forward to those particular frames. This device is used, for example, to mark the beginning of a stroke so that the user can easily undo all the effects caused by that stroke.

Figure 20—The undo and redo stack as a piece of film. Each frame of film is a static state, and transitions carry the state across the edges to the next frame. So if A was "Square" and B was "Square and Circle" then the transition between the two would be "Add Circle." In the above, the current state is State A, and State A and C are marked.

This undo and redo mechanism provides considerable power, as illustrated by looking at what happens when a region is being moved around the surface in the house sketching program.
Imagine in that case that the user has selected the region, and has moved it on top of another region. When the user does this, a collection of recognizers subtracts out the underlying region to make room for the newly moved region, and then connects the adjacent regions together. Now say the user wants to make a minor adjustment and move the region up slightly. When the user drags the region, the modify mode module rewinds the TransitionRecorder to the state, or frame, just after the previous move, thus undoing the merging of the regions, and the breaking up of the underlying region. After having restored the state to a good starting point, normal recognition takes place, allowing the two regions to be merged together correctly. Without an undo and redo system this would be considerably more complex because of the work of undoing the consequences of releasing the moved region on top of the underlying region at the wrong place. A large amount of work was done to break apart the walls in the correct fashion, to throw away the common walls, to reshape the new regions, etc., and without a good record all the work would be lost.

The Transitions are stored with the TransitionRecorder. Whenever a widget is changed, for example, the surface is informed and is passed a Transition object that can be activated to undo the change. This object is then passed to the TransitionRecorder and effectively "saved to tape". Most of this recording is done at as low level as possible and is automatically done by the system as much as possible, to make it less likely that a programmer forgets to save some state, whose absence would then throw the system off when it tried to undo and redo.
Figure 21—Saving undo information. The first picture we are at A, and B and C have been undone. In the second, when we go to D, the film is copied and attached backwards to keep the undo information.

Even the act of undoing and redoing is saved. If the user draws line A, undoes it, and then draws line B, the TransitionRecorder has a record of all of these events. If the user undoes one step, line B will disappear, if the user undoes another step, line A will appear, and then if the user undoes another step, line A will also disappear, putting the user back in the original state. This is done by, when the user has undone some sequence of events and then embarked upon a new path, copying all of the old "tape" following the current frame the user is on, copying the inverse of the old "tape", and then adding the new activity. See above figure for an illustration of this.

4.2.2 The Recognition Queue

It is possible for more than one widget to be waiting to be recognized. For example, in the house program, if the user draws one line crossing another, all lines are broken into segments. When this happens, new widgets are created and scheduled for recognition. These widgets are kept in a queue and recognized in a first-filed, first-served basis. As this object is a queue, the order of recognition is defined by which widget is schedules first; in the case of the line recognizer the pieces are re-recognized in the order defined by the recognizer which broke up the original line.
This queue allows for the re-recognition of widgets. If other widgets are altered in the process of recognizing a widget, for example if they are moved aside on the surface to make space for the new widget, they can easily be rescheduled so that any recognizers can note their change and perhaps make further modifications.

4.2.3 Registration
Any object can register interest in any number of specific widgets. Whenever a watched widget is deleted from the Surface, replaced by a collection of widgets, or altered in any way, the Surface alerts all objects that were interested in that particular widget. This process is managed by the Surface through the WidgetRegistry object.

The uses of registration are varied. Registering interest in widgets allows the programmer to maintain data structures that span multiple widgets more easily. For example, in the house sketcher it is important to know which regions border other regions, so there is a recognizer whose sole purpose is to maintain this information. Whenever a region is deleted, this recognizer needs to know so it can update its region connectivity information. Similarly, if a region is replaced with another collection of regions (i.e. subdivided) the recognizer should hear about it so can make the necessary adjustments.
Widgets can also register interest another widgets. For example in the finite state machine sketching program, edges register interest in the nodes that they are connected to, if any, so if the node is moved, the edge can adjust its end-points accordingly.

4.2.4 Fast Recognition
Sometimes it is necessary to respond to a user’s stroke before he has even finished it. Implementing a drag interface (where the user can move widgets around on the surface by clicking and dragging) would be impossible given the system described so far. To allow for a click and drag interface, a programmer can write fast recognizers that get each point as it is added to the current squiggle being drawn.

![Diagram of fast recognition process](image)

Figure 23—An example of fast recognition. As the stroke progresses, the selected widget moves along. When the pen is released, the stroke vanishes, leaving the widget in the new place.

As each point is added, all fast recognizers are given a chance either to (a) claim the stroke, (b) reject the stroke, or (c) wait for more information. If any fast recognizer claims a stroke it then is the only recognizer to hear about further points being added, and is the only recognizer that can make any changes. If a fast recognizer should later determine that the stroke is not in fact what it thought it was, it can reject the stroke, and all the other fast recognizers will get another chance at the stroke starting with all the information up to the current point.
To illustrate this, say there are two fast recognizers, one that allowed for dragging, and one that allowed for resizing. If the user began a drag interaction, the widget clicked on would dynamically move under the stylus. But then say the user doubled back and the drag recognizer decided that the user was not actually attempting to drag the object. In this case, the drag recognizer would reject the stroke being created. All the changes the drag recognizer made would be undone and the widget would be restored to its original state. Then the resizing recognizer would get a chance to recognize the entire sequence of points in the new stroke. If it decided that the sequence matched what it was looking for, it would resize the object and then continue to dynamically resize the object as user continued drawing the stroke.

As this example makes clear, it is hard to distinguish the intent of a stroke while it is being drawn. As a result the fast recognition process is used rarely, and is limited to simple and straightforward interactions, so that user will not be surprised by its behavior.

4.2.5 The Conversation System
The conversation system is implemented within the Metaglue system, a system designed by and for the Intelligent Room project. Metaglue is an agent support layer that allows multiple agents to interact easily through remote procedure calls. In this system, the sketch recognition system is encapsulated as a single agent that provides hooks to the outside world, so that other agents can query the surface, change it, and install recognition modules. For more information about Metaglue itself, see [Phillips].

The conversation system consists of a collection of agents that handle speech input and output. There is a general speech output agent so that other agents can send sentences to the user. Speech recognition is handled via ViaVoice, an off-the-shelf commercial speech recognition package by
IBM, which is used by a collection of grammar agents. A central agent sends any recognized utterances the user makes to the appropriate registered grammar agents. Each grammar agent captures a single type of conversational interaction with the user. They are each responsible for activating or deactivate themselves with the central agent based on the context, but can be activated or deactivated through other agents as well. The central agent in turn limits possible recognized utterances to those expected by the currently active grammars. This standard form of interaction within the Intelligent Room is further outlined in [Coen].

The conversation agents, or grammar agents, can be quite small and simple. For example, a single agent handles erasing objects. Erasing objects by name is handled by yet another agent. Each agent maps a collection of possible utterances to specific actions. They are like recognizers: small parts that are combined to create a complex whole.

A grammar agent can also be interactive, using other grammar agents to ask questions and for clarifications before electing to take action. For example, if the user says "erase everything," the Erase agent is activated. Before erasing everything, it uses the Asker agent to ask, "are you sure you want to erase everything?" In this case the Erase agent has used another grammar agent to provide a more complex and safe interaction.

Some grammar agents do not affect the surface, but query the surface for information that they then provide to the user. For example, when asked, the NodeCounter agent asks the Surface for how many nodes exist, and then tells the user this. Similarly, the Connector agent tells the user whether two regions are connected.

Remote agents can install modules into the surface to simplify monitoring activity there. They can install normal modules (described above) or a special kind of module called a "remote module".
Remote Modules
Remote modules can send messages out of the sketch program back to the installing parent agents. These modules have all the functionality of the standard modules described above, and are treated exactly as such by the surface. The surface in fact has no knowledge of whether a module is standard or remote. However, remote modules can "ship out information" to the parent program when they are activated by recognition activity by the Surface.

For example, one of the remote agents for the finite state machine system installs a module that tracks all of the nodes being recognized. The module registers interest in all nodes, tracks the nodes' names, and tells the parent agent whenever a new name appears. The parent agent then updates its grammar with all of the names of the nodes so the user can say things such as "delete node A."

Another example of remote module use is the layout evaluator agent in the house recognition program that installs a remote module that monitors the relationships of regions and how they are connected. When this evaluator module notices something possibly erroneous in the current floor plan, such as a kitchen and bathroom having a connecting door, it sends out a message to the remote agent that translates the message into an English comment, which is then in turn passed along to the speech output agent that then informs the user of it.

The basic remote module has a single query method. This method accepts a single Object as a parameter and returns a single Object as a return type. This method should be overridden by whatever behavior is desired. However, the programmer can also implement more complicated interfaces for the remote modules allowing for any number of different methods to be called by the parent with any types of parameters desired. As everything is written in Java, the remote modules are simple extensions of remote objects and so can be extended just as easily and quickly as normal Java remote objects.
The parent agent which installed the module on the Surface has to provide callback procedures so the module can communicate with it. This method is usually called ring, and is just like the query method on the remote module side. Again, a programmer can add other methods that the remote module could call, giving a more complicated interaction between the two.

**Gremlins**

Another way for agents to affect the Surface is through Gremlins. Gremlins are capsules of code (basically a menu of commands to carry out) which are given to the Surface by a remote agent to be run. The Surface runs a Gremlin as soon as it gets it. For example, a standard Gremlin is one that deletes all edges which do not have two nodes as end points on the Surface. A Gremlin is passed a reference to the Surface that it is run on so it can have access to the widgets on it. When it is finished the Gremlin can return a single object to the installing agent. Gremlins are used from very simple things to very complex.

Using the two tools outlined above—remote modules and Gremlins—it is possible to layer a conversational system on top of the sketch recognition system while keeping these two halves distinct. In fact, it is possible to run the sketch system entirely independent of the conversation system.
5 Making an Application
This chapter briefly describes the steps needed to create an application using the base platform outlined above. I use the floor plan sketching system as an example throughout. I do not describe the first major step in designing an application, which is defining the drawing semantics of application in question. However, designers should dedicate a large amount of time to this task in order to determine what the proper widgets and recognizers are.

5.1 Defining the Widgets
Write the actual objects that you want understood by the system. For example, in the house sketching program, there are walls, doors, and regions. Make these objects as complete as possible, and make sure that any methods that modify the objects' state have associated Transition objects that get built and given to the Surface.

For example, the house sketching program has a Wall Widget. This widget is inherited from the line widget class. The wall widget maintains a set of the regions which it borders. It has methods to add and remove regions from this set, to query what regions it is touching, and also has the universal methods to move it around the surface. The methods which add and remove regions store Transitions in the Surface so the set of bordering regions is always correct. The wall also implements draw, a method that describes how it should be drawn on a display.

The following is the code for the Wall widget. Note the use of Transitions to track what Regions the Wall borders in addRegion. Also note the major method, draw(), which controls how the Wall is drawn to the screen. The Wall has a special draw() method so a Region can draw it after it draws itself so the overlapping looks correct; this points to a lack of layering in the drawing program. Layering would be a good addition to the base platform to simplify code such as this.
package rec.house;

import java.awt.*;
import java.util.*;
import rec.util.*;
import rec.sys.*;
import rec.geo.*;
import rec.core.*;

public class Wall extends WLine
{
    Surface surface;

    // the neighboring regions
    HashSet regions;

    public Wall( Surface s, GPoint p1, GPoint p2 )
    {
        super( p1, p2 );
        regions = new HashSet();

        surface = s;
    }

    public Wall( Surface s, Squiggle st )
    {
        super( st );
        regions = new HashSet();

        surface = s;
    }

    public Object clone()
    {
        Wall w = new Wall( surface, getHead(), getTail() );

        w.regions = (HashSet)regions.clone();

        return w;
    }

    /**
     * Make a Wall, add it to the surface, add the transition.
     */
    static public Wall changeToWall( Surface s, WLine wl )
    {
        Wall w = new Wall( s, wl.getHead(), wl.getTail() );
        s.replaceWidget( wl, w );

        return w;
    }

    public void addRegion( Region r )
    {
        replaceRegion( null, r );
    }

    public void removeRegion( Region r )
    {
        replaceRegion( r, null );
    }
void replaceRegion( Region old, Region new_r )
{
    if ( old != null )
    {
        Assert.check( bordering( old ),
            "Region " + old + " not bordering " + this );
    }

    Transition t = makeReplaceRegion( old, new_r );

    // get the reverse and do the deed
    t = t.undo();

    surface.addTransition( t );
}

/**
 * Make transition which sets position p to new_r, and the reverse of
 * which sets position p to the old value.
 */
protected Transition makeReplaceRegion( Region old_r, Region new_r )
{
    final Region n = new_r;
    final Region o = old_r;
    final Wall the_wall = this;

    // make the transition effect here
    return new Transition( surface ) {
        public void doUndo()
        {
            if ( o != null )
                regions.remove( o );

            if ( n != null )
                regions.add( n );
        }

        public Transition makeRedoTransition( )
        {
            return makeReplaceRegion( n, o );
        }

        public String name()
        {
            return "repRegion[" + o + "->" + n + "]";
        }
    };
}

public boolean bordering( Region r )
{
    return regions.contains( r );
}

public boolean isBorder( )
{
    return regions.size() > 0;
}

public boolean full()
{
    return regions.size() >= 2;
public Iterator regions(){
    return ((HashSet)regions.clone()).iterator();
}

draw(Graphics g){
    draw(g, false);
}

draw(Graphics g, boolean parent){
    // WARNING--Make sure regions know us if we know them!
    // if we are border, then region will draw us.
    if ( 'parent && isBorder() )
        return;
    Color bk = g.getColor();
    GPoint h = getHead();
    GPoint t = getTail();
    if ( getColor() == null )
        g.setColor( HouseColor.WALL );
    // get perp to the segment.
    double dy = t.x - h.x;
    double dx = -1 * ( t.y - h.y );
    double len = getLength();
    dx = ( 5 * dx / len );
    dy = ( 5 * dy / len );

    g.drawLine( (int)( h.x + dx), (int)(h.y + dy),
                (int)(t.x + dx), (int)(t.y + dy) );
    g.drawLine( (int)( h.x - dx), (int)(h.y - dy),
                (int)(t.x - dx), (int)(t.y - dy) );

    GPoint cent = getLocation().getCenter();
    g.setColor( HouseColor.TEXT );
    g.setColor( bk );
}

The wall widget inherits from the line widget the methods which control how it is moved around the screen, methods that allow objects to query where its end points are, and methods that maintain what color the wall is. It also inherits methods that allow objects to query whether a given point lies inside or outside it.
package rec.fsm;

import java.util.*;
import java.awt.*;
import rec.util.*;
import rec.geo.*;
import rec.sys.*;
import rec.core.*;

public class LooseCircleRecognizer extends Recognizer {
    public static boolean isCircle( rec.sys.Squiggle st )
    {
        int num = st.getNumberOfPoints();

        double dist = 0.0;
        int cntr = 0;
        int jump = num / 20;
        if ( jump == 0 ) jump = 1;
        num -= jump;
        for ( int i = 0; i < num; i++ )
        {
            dist += Geometry.distance( st.getPointAt( cntr ),
                                st.getPointAt( cntr + jump ) );
        }

        if ( !( Geometry.distance( st.getFirstPoint(),
                                st.getLastPoint() ) < dist / 5 ) )
            return( false );

        if ( st.getLength() < 50 )
            return false;

        return( true );
    }

    public LooseCircleRecognizer()
    {
    }

    public boolean recognize( Surface s, Widget w )
    {
        rec.sys.Squiggle st = (rec.sys.Squiggle)w;

        if ( st.getLength() < 40 )
            return( false );

        if ( isCircle( st ) )
        {
            Console.writeln( "circle online" );

            WCircle c = new WCircle( st.getBounds() );
            s.replaceWidget( st, c );

            return( true );
        }

        return( false );
    }

    public Enumeration getInputTypes()
    {
    }
5.2 Writing the Recognizers
An individual recognizer has two major aspects. The first is what types of widgets it is interested in.

A recognizer is usually interested in Widgets which it can turn into other Widgets. For example, a circle recognizer would be interested in the user's unmodified strokes, since it can turn these into circles. To define this list, the programmer overrides the getInputTypes() method which returns a list of all the widgets which spark recognition with the given recognizer. The Surface, upon initialization, calls this method and installs the recognizer into the proper lists so that when any of the specified widgets are due for recognition, the recognizer is invoked.

The second, more relevant, aspect of a recognizer is the recognition procedure itself. The Surface calls the recognize() method of the recognizer and passes a pointer to the surface and the widget which is currently being recognized. The programmer needs to override this method to perform any recognition activity, make any changes due to recognition, and return true if recognition should cease on the given widget or false otherwise.

A very simple recognizer is the circle recognizer used by the finite state machine drawing program. Its getInputTypes method returns only Squiggle, which are the raw strokes a user enters before they are recognized as anything in particular. Its recognize method looks at the bounding box of the stroke, and makes sure that the width and height are within a factor of two of each other. If they are, it checks to make sure the end points of the stroke are within one-fifth of the stroke's length of each other. This "definition" of a circle sketch is very primitive, but it is sufficient in its context because there are not any other things that meet these very rough measurements. If it became necessary to have a more robust circle recognizer, this one could be replaced and rest of the system would still work exactly as it did before (as long as the new recognizer did not anything the previous recognizer did not recognize). The code for this recognizer is below.
Vector v = new Vector();
  v.addElement( "rec.sys.Squiggle" );
  return( v.elements() );
}

public Enumeration getOutputTypes()
{
  Vector v = new Vector();
  v.addElement( "rec.core.WCircle" );
  return( v.elements() );
}

Note that the recognition procedure is in a static method; this allows other recognizers to use the Circle Recognizer's code if desired for their own tests. If a Circle is recognized, it is defined to be the size of the bounding box of the spark Squiggle (Squiggles are the user's unmodified input.) The method getOutputTypes is for possible future use; currently it is not used at all.

5.2.1 Start Building From the Bottom Up
A well-designed system consists of a fundamental collection of widgets. These widgets should have straightforward recognizers that can be written in a context-free manner. Among these basic recognizers in the house sketching program are the one which turns strokes into lines, the one which turns collections of lines into regions, and the recognizer which automatically names regions.

Recognizers are normally very simple, because context plays such a large role in deciding what a stroke or collection of widgets is, and so it is how the Recognizers are combined which give the expressive power. An example of this is the simplicity of the circle recognizer used in the finite state machine drawing program described above.
5.3 Build the Data Structure Recognizers

If there are any data structures that span all of the widgets, add in the recognizers that maintain them.

This is much easier once the overall physical objects are being recognized (this having been done in the previous section). For example, in the house sketching program there is a recognizer which maintains regions border information. This recognizer maintains a graph of which regions are connected to which. It registers interest in all of the regions so that if any regions are moved or changed it can update its graph accordingly. Here is a sample data structure recognizer which keeps track of the connections between Walls and Regions.

```java
package rec.house;

import java.awt.*;
import java.util.*;
import rec.core.*;
import rec.util.*;
import rec.sys.*;

public class DataFixer extends Tracker
{
    public DataFixer()
    {
    }

    public Enumeration getInputTypes()
    {
        Vector v = new Vector();

        v.addElement("rec.house.Region");
        v.addElement("rec.house.Wall");

        return v.elements();
    }

    public Enumeration getOutputTypes()
    {
        return getInputTypes();
    }

    public void removed( Widget widget )
    {
        if ( widget instanceof Region )
        {
            // fix bordering walls.
            Region r = (Region)widget;
            Iterator e = r.walls();

            while( e.hasNext() )
            {
                Wall w = (Wall)e.next();
            }
        }
    }
}
w.removeRegion( r );

if ( !w.isBorder() )
{
    surface.unregister( this, w );
    surface.removeWidget( w );
}
}

public void replaced( Widget ow, HashSet wns )
{
    if ( ow instanceof Region )
    {
        Console.indent();
        // region has been split.
        fixWalls( (Region)ow, wns );
        Console.outdent();
    }
    else if ( ow instanceof Wall )
    {
        // wall was split
        Console.writeln( "DataFixer fixing split wall." );
        Wall w = (Wall)ow;

        Iterator e = w.regions();
        while( e.hasNext() )
        {
            Region r = (Region) e.next();

            // remove from old wall.
            r.removeWall( w );
            w.removeRegion( r );

            Iterator e2 = wns.iterator();
            while( e2.hasNext() )
            {
                Wall w2 = (Wall)e2.next();
                r.addWall( w2 );
                w2.addRegion( r );
            }
        }
    }
    else
        Assert.note( "unknown replace of " + ow );
}

public void replaced( Widget ow, Widget nw )
{
    Console.writeln( "DF: single replace of " + ow + " -> " + nw );
    if ( ow instanceof Wall )
    {
        // wall was split
        Console.writeln( "DataFixer fixing (merged?) wall." );
        Wall w = (Wall)ow;

        Iterator e = w.regions();
        while( e.hasNext() )
        {
            Region r = (Region) e.next();
// remove from old wall.
// note--this changes structure of widget so it could
// be doing nothing if this is the second round of the
// merge's events
r.removeWall( w );
w.removeRegion( r );

Wall w2 = (Wall)nw;
r.addWall( w2 );
w2.addRegion( r );
}
}

public void altered( Widget widget, Transition t )
{
}

private void fixWalls( Region r, HashSet w )
{
    Iterator e = r.walls();

    while( e.hasNext() )
    {
        Wall w = (Wall)e.next();
        Console.writeln( "looking at " + w );

        // remove the old region
        w.removeRegion( r );
        r.removeWall( w );

        // find new region and add it
        Iterator e2 = w.iterator();
        while( e2.hasNext() )
        {
            Region new_r = (Region)e2.next();

            if ( new_r.includes( w, RegionRecognizer.BREAK ) )
            {
                Console.writeln( w + " is in " + new_r );
                Console.indent();
                w.addRegion( new_r );
                new_r.addWall( w );
                Console.outdent();
            }
            else
                Console.writeln( w + " is not in " + new_r );
        }
    }
}
} // end DataFixer

The data itself is stored in the Region and Wall objects; the DataFixer is only responsible for making sure the data in these elements is correct. Note its use of registration of widgets to track when they
are removed or changed. The DataFixer also splits walls when a Region is moved. This is a dual purpose Recognizer, which might be argued as bad design.

5.4 Implement the "Changing" Recognizers
At this point you have a system which will passively recognize the user drawings. At this point the developer should add the recognizers that allow for the manipulation of the drawing. These recognizers need to be carefully constructed so they leave any data structures maintained in a correct state. For example, the recognizers which allow the user to drag regions around are carefully constructed so they update all of the walls’ pointers to bordering regions correctly. This was done by removing temporarily the region which was being moved and then placing it in its new location, thus causing all of the basic recognizers to connect it up to other regions accordingly.

The following is the base MoveRecognizer which allows for dragging of Widgets. It is sometimes extended to ensure data structures are left intact, but can be used as-is for most Widgets. It is a FastRecognizer, which means it modifies the Surface while the user is drawing a stroke. Note how it selects a stroke only if the stroke begins inside some target Widget.

```java
package rec.core;
import java.util.*;
import java.awt.*;
import rec.util.*;
import rec.geo.*;
import rec.sys.*;

public class MoveRecognizer extends FastRecognizer {
    // the thing being moved (null if none)
    protected VisibleWidget movie;

    public MoveRecognizer()
    {
    }

    /**
    * This is called with every point added to the stroke. If the
    * rec returns WANT it gets full control of all added points,
    * UNSURE makes no comment, and DONT_WANT stops all calls.
    */
```
* DONT_WANT is not absolutely needed. UNSURE works as well. */
public int want( Surface s, rec.sys.Squiggle st, boolean ending )
{
    if( movie == null ) {

        GPoint p = st.getFirstPoint();

        GeometryCalculator calc = new GeometryCalculator( s );
        VisibleWidget vw = calc.findSelectedVisibleWidget( p );

        movie = vw;
        if( vw == null ) {
            Console.writeln( "nothing selected." );
            return( DONT_WANT );
        }
        else {
            Console.writeln( vw + " selected-- going to take it" );
        }
    }

    if( st.getNumberOfPoints() > 5 )
        return( WANT );
    else
        return( UNSURE );
}

public void reset()
{
    movie = null;
}

public boolean recognize( Surface s, rec.sys.Squiggle st, boolean ending )
{
    GPoint prevLocation = movie.getLocation().getCenter();

    movie.move( st.getLastPoint() );

    Transition t = new MoveTransition( s,
            movie,
            prevLocation,
            movie.getLocation().getCenter() );

    s.alterWidget( movie, (AlterTransition)t, ending );

    s.addTransition( t );

    return( true );
}

public Enumeration getOutputTypes()
{
    return( null );
}
5.5 Assemble the Recognizers

Now the Recognizers need to be collected in modules. The easiest way to do this is use the BasicModule class which reads in plain-text files describing the order of recognizers in the module. An example of such a file follows:

```plaintext
# selector.mod
# this does fast drag recognition.
# rec.core.MoveRecognizer

# recognize modify mode
rec.core.ModifyModeRecognizer

# recognize strokes->clicks
rec.core.ClickRecognizer

# recognize clicks->selections
rec.core.SelectionRecognizer

# remove unused clicks
rec.core.ClickConsumer
```

As can be seen, these files are just a simple list of the recognizers to load into the system. The recognizers are kept in order with the top of the file first, and the bottom last. A master file, called "recog.ini", determines what modules to load in. Module files can also specify additional modules as well. A recog.ini file for finite state machine recognition looks like:

```plaintext
# recog.ini
# RecSystem initialization file

Surface rec.core.Surface

# Initialization commands
Console EnableOutput
Console EnablePrettyPrinting

# Basic recognizers
Module basic

Module fsm
```
Here the Surface can be specified (so if the programmer wants to load an extended Surface she can), and Modules are listed in order of priority with the last Module's mentioned being on top of the Module stack. There are additional commands which can be used for debugging or parameters of any sort, such as setting how the debugging Console prints out information.

5.6 Overlay the Talking
Now that there is a basic drawing platform, it is straightforward to overlay the conversational aspects on top of it. The easiest way to do this is to copy one of the basic remote agents (for example the EraseAgent) and modify it.

In general, the conversational aspects should be done with a bottom-up approach. First implement the agents that allow for the simple questions, such as "what is the area of that region." When this is complete, move on to the agents that are more sophisticated or highly interactive.

Here is a sample voice agent which loads or unloads the Finite State Machine recognition module. This allows the user to say, for example, "I want to draw a finite state machine," to inform the system of the necessary context.

```java
package rec.fsm.speech;

import java.rmi.*;
import java.util.*;
import speech.*;
import metaglue.*;
import rec.agent.*;
import rec.ui.*;
import rec.util.*;
import rec.sys.*;

public class RunFSMAgent extends TalkAgent
    implements RunFSM
{
    boolean loadedFSM = false;
    Id fsm_Id = null;

    AppGrammar stopgrammar = null;
```
public RunFSMAgent() throws RemoteException
{
    stopgrammar = loadGrammar("rec.fsm.speech.StopFSM");
    makeAlwaysActive(stopgrammar);
    stopgrammar.deactivate();
}

/**
 * Incoming speech goes through here.
 */
public void acceptedTagsResult(Hashtable tags) throws RemoteException
{
    if (tags.containsKey("runFSM"))
    {
        alert("okay");
        runFSM();
        alert("ready to draw");
    }

    else if (tags.containsKey("stopFSM"))
    {
        removeFSM();
        alert("f s m recognition removed.");
    }
    else
        error("do not recognize utterance.");
}

public void runFSM() throws RemoteException
{
    if (loadedFSM)
        return;

    loadedFSM = true;

    // install the FSM module
    System.err.println("Installing FSM Module.");
    fsm_Id = sketch.installModule(new ModuleMaker() {
        public Module make(Object parent)
        {
            try
            {
                RecIniFile f = new RecIniFile("fsm.mod");
                return f.getModule();
            }
            catch (Exception e)
            {
                e.printStackTrace();
                System.err.println("....making FSM module failed....");
            }

            return null;
        }
    });

    // now fix grammars
    grammar.deactivate();
    stopgrammar.deactivate();
    stopgrammar.activate();
}

public void removeFSM() throws RemoteException
Note the acceptedTagsResult() method through which all speech comes. The agent uses two grammars, one for activating the FSM recognition, and one for deactivating it. The agent inherits from Talk which is a base agent that automatically connects to the Sketch agent for the programmer's convenience. The Talk agent provides a single grammar (loading it automatically), but since the RunFSM Agent needs two grammars, it has to define an additional one for stopping.

The grammar looks as follows:

```
grammar RunFSM;

<fsm> = (finite state machine) | (FSM) | graph;

<need> = (I want to) | (I need to) | (I am about to);

public <go> = (<need> draw a <fsm> [now])<runFSM>;
public <go2> = (I am drawing a <fsm> [now])<runFSM>;
```

This is a very simplified version; grammars can get as large as desired. The larger the grammar, the more likely the user will utter a match, so in general grammars should be very complete.
5.7 Enhance
At this point there should be a complete system. Now, due to the component nature of the base platform, it is possible to extend it to create a more complex application which recognizes more of the user's actions.

The expressive power of the system springs from the careful assembly of the multiple simple pieces. For example, in the house recognizer when a user draws a line closing off a square, the system recognizes, labels, and connects up a new region. When one looks at the overall effect, it is fairly massive, but each step is quite simple and easy to comprehend: the squiggle is recognized as a line, the line with the other lines is recognized as a polygon, the polygon as a Region, the region is named, the region is connected, the various data structures are independently updated. The pieces themselves remain simple, and are easy to write and alter. They have no connection or built in understanding of what the other pieces are doing. Of course, each piece is also not so useful by itself.
6 Pitfalls in Making the System

In this chapter I discuss the major hurdles overcome in making the base platform. This chapter also asks some unresolved questions regarding the base platform and how to use it effectively.

6.1 Undo and Redo

Implementing an effective undo and redo regime proved to be difficult. The metaphor of film made the problem much more tractable, but programmers using the base platform are still faced with significant work writing transitions for every state change. Perhaps a check-pointing schema, or some way to give before and after Objects rather than Transitions which modify objects, would be better.

6.2 Who Does the Work

When implementing applications it is unclear what should be responsible for modifying state. There are two major camps: recognizers should modify widgets states, or widgets should modify their own state. For example, when a finite state machine node is moved one of two things could happen. The first possibility is a recognizer can be alerted to the moving node and could then adjust all affected edges. The second possibility is that each of the edges is alerted and adjusts itself to follow the node.

Both sides have their elegance. Widgets modifying themselves is nicely distributed, while the recognizer doing the work allows for complex manipulation of state in a fairly straightforward fashion. We experimented with having the widgets modify their own state, but this approach proved to increase in complexity very quickly. In the house sketching program it was hard to see what was going on in the relationship between regions and walls. The self-modifying widget technique also suffers from the problems of any distributed system: they are very hard to think about and get very complicated very fast.
6.3 Granularity

Another open question is how much work an individual recognizer should do and how complicated a widget should be. It is also unclear whether state should be stored in a widget or externally. Both these issues are issues of granularity. If an individual recognizer does not do enough, then there needs to be too many recognizers and the relationships between these myriad tiny recognizers becomes too complicated. If the individual recognizers are too large and do too much, it becomes hard to reuse code, and a system begins to suffer from the standard problems of overly centralized programming.

An example of all this is the WallCutter recognizer. This recognizer was responsible for dividing crossing lines into pieces. One approach to implementation would be to take a line and do all of the cutting at once and then schedule all of the resulting pieces for re-recognition. Another tack would be to have the recognizer look for a cut, make that cut, and reschedule the pieces for re-recognition, relying on future execution of the WallCutter to do any other cutting necessary. The latter approach allowed for a much smaller amount of code. However, there were some dangers inherent in it. It was possible for the first cut to be made and then have a piece get scheduled for re-recognition which was not crossing any other pieces. This piece, since it was scheduled for recognition before the other pieces which still needed to be cut, would then get passed to other recognizers before all of the wall cutting has been completed. In the case of the house recognition it did not prove to be a problem, as it did not matter whether the wall cutting happen before or after the other recognizers, but clearly one can imagine circumstances where this could be a problem.

One way around this problem was allowing for recognizers in the middle of recognition to call the Surface and force it to do more recognition recursively. Now one could easily write a recognizer that uses the WallCutter recognizer internally and forces the surface to loop over all the wall pieces
until WallCutter has nothing left to cut. Again, however, this recursion probably has many problems associated with it, not the least of which is clarity.

6.4 Modes
Modules make it easy to have many different contexts. However, because of this it is tempting to use modules as a way to change modes. This can leave the user in strange states where their actions have unexpected results or different results from moment to moment. A bad programmer can easily misuse the modules to create a highly modal system (a system where a user’s behavior has different effects depending on the system’s state) rather than a system that dynamically adapts to context. These two things are very similar, and perhaps the line dividing them is more a matter of taste and ease-of-use than anything else. Nevertheless, it seemed a design decision worth mentioning as a potential major pitfall.

This danger is discussed further in the Apple Computer interface guidelines. Here they described the dangers of having modes rather than a single environment. It could be argued that context is a form of mode, and if this is the case then perhaps fast context switching would not be as advantageous as hoped. It could be that the best possible interface would be all contexts being active all the time, but that would be demanding a lot from the recognition processes.

In building the base platform many very large questions were brought up which were not completely resolved. This chapter summarizes the main ones. It is hoped that further research will make clear which approaches should be taken to create a more easily extensible base platform, and to create a more intuitive collection of applications.
7 Related Work
One project that has captured a lot of what I am trying to do is the Electronic Cocktail Napkin [Gross 96]. However they are more interested in querying a database for design images for architectural planning based on the raw sketches than in interactive drawing environments. They have, however, built a system that extracts knowledge about a sketch. In the end we used their type of system for sketch recognition to recognize complex elements from simple ones. They differ in that they analyze a sketch to find relationships between components in it, and query a database based on that information, where we identify and use the relationships to replace a collection of components with a single more complicated component. However, in both cases the information extraction itself is the same.

The Electronic Cocktail Napkin was a main source of inspiration for how we did recognition. Their approach is to do recognition by the bottom-up component model. Their overall take, however, is slightly different. They have a database of things they call icons with definitions of icons in terms of geometric relationships between simpler icons. They then look at the relationships between a collection of simple icons and match them with their templates to see if they get a hit.

We, on the other hand, have individual recognizers that are implemented in any fashion the programmer likes. In fact, because of this flexibility, it is possible to implement the style of recognition employed by the Electronic Cocktail Napkin as a recognition process within our base system.

Another significant difference between our project and the Electronic Cocktail Napkin is the interactivity in our program. We are primarily interested in real-time drawing recognition, whereas they are primarily interested in processing static images.
Gesture recognition in a drawing context has been explored by Eric Saund [Saund 95, Saund 98] in Xerox Parc who has made a whiteboard, ZombieBoard that uses vision technology to observe the user. The main thrust of the board is building high-resolution images from multiple camera shots, but they are also exploring gesture recognition to perform functions such as printing and saving. His project is only an input device, however. The computer has no way of providing feedback on the board itself.

Dean Rubine [Rub 91] implemented a gesture recognition program that uses feature-based learning to match gestures against a database of possible icons. He claims to have used it successfully for sets of symbols such as the English alphabet. However, his work is solely interested in identifying individual icons outside of any context, and not with interactive context dependent applications.

The Media Lab at MIT has several projects somewhat related to mine. [Ishii98a, Ishii98b] The project that bears the most similarity to my work is the luminous room [Ishii98b]. This room features a so-called "I/O bulb" that projects images onto a wall and watches a user's interactions. The designer user interacts in a natural fashion using physical models and an active screen to put together or study aspects of architectural plans. This group is primarily concerned with mapping electronic information to physical objects, however, but a lot of the concerns with interface and intuition are the same.

CAD systems are of course very similar to what we are attempting to do. However, as discussed previously, their model of providing many tools and menus is fundamentally different from our model. Furthermore, the idea of a conversational assistant at ones side is alien to current generation the CAD programs. In addition, CAD programs recently have become more interested in issues of three-dimensional modeling which I have not addressed at all.
I believe that CAD programs, while quite useful, have also been trapped in a certain worldview of Windows, mice, and menus—namely the classical GUI. One of the main tenants of my research is to break away from this model, and approach sketching in an entirely new fashion. Hopefully this new fashion is more than a novel application showing off new technology, but is indeed fundamentally more accessible, and in the long run more useful even to an expert user, than the classical style of application.
8 Where to Go

There are three fundamental uses we can envision for this technology. The first is as suggested throughout the thesis—aiding the design process. There is no reason to require users to maintain a large body of knowledge about an application in order to use it. It is easy to train someone to use a pen and paper, and they can quickly (if sometimes inelegantly) express their knowledge through this medium. Computers should be no different.

A sketching program could be equipped with knowledge of building codes and standard architectural symbols such that a designer does not have to memorize them. The user could simply point at objects and inform the system as to what they were and the system would use the correct symbols. Similarly, if the user was perusing someone else's design, she could ask about any unfamiliar symbols and the system could explain them. This type of interaction would allow a programmer to embed reference manuals into the application itself, reducing the need for users to memorize large amounts of information.

The second use is design rational capture. Design rational capture is a sticky problem in that it is very useful for everyone but the original designer. Due to this, designers are placed in a circumstance where they are expected to expend effort to record their mental processes for no reward. Frequently the extra burden of this task discourages people from doing it. There has been a great deal of research into how to aid the designer in capturing design rationale, or in extracting the rationale post design [Moran]. Most of this research, unfortunately, still requires the designer to invest a certain amount of labor, so these techniques are not as useful as they might be.

Potentially, this base platform could record all of the designer's activity without any effort on the part of the designer. By later perusing this record it may become possible to find discarded alternative
designs, areas that were tricky and required a redesign, or even the arc that the designer took from the beginning of the project to the end.

Ideally the system could ask questions about areas of the design that are obscure. One can envision a scenario where the computer asks clarifying questions about areas of the design that seem tricky or possibly erroneous, and the user, while sketching in a natural fashion, explains to the computer what he is doing at the time. Occasionally the user will realize, due to the computer's inquiries, that he made a mistake and thus benefit from the interaction. Throughout all this the computer will collect the user's explanations for future reference. Such a system would be no more painful for users than having an intelligent assistant watching over their shoulder.

A third used for this technology is as a front end for simulators and other types of design tools. For example, the system will be used as a front end to a physical modeler so the user can sketch out a simulation and then simulate it, rather than using the simulators primitive design package. Instead of specifying a gear train through a menu-driven program one could, for example, sketch it out and then inform the system through voice, "this gear has 13 teeth", or, "this cam is 2 cm wide," etc..

Other more sophisticated systems that use knowledge of a particular domain to suggest alternative designs can also be accessed through this system. For example, Stahovich, Davis, and Shrobe have designed SketchIt, a system that gives alternatives for mechanical systems [Sta97a]. Our base platform is perfect for SketchIt, as its desired form of input is indeed a sketch. Their professed ideal is a designer sketching out the rough concept, telling the system to come up with alternative designs, and then perusing the wide selection of possibilities.

Another application that could benefit from such an interface as this is the architectural design modifier being designed by [Koile]. This system looks at floor plans of houses and proposes
modifications based on general concepts such as openness or connectivity. The user enters a design into the system and informs the system of something that they want changed; for example, she could ask for more openness between the living room and the dining room. The system would then search variations of the floor plan based on a knowledge base that allows it to evaluate the effects of various modifications and display the most promising results. This is a dialog where the user could sketch in floor plans and converse with the system, sketching and modifying features as desired, and allowing the system to modify the design until the desired result is obtained.

8.1 The Intelligent Room
A better environment could add immensely to the power of sketch recognition. I would like to more fully incorporate this base application into the Intelligent Room [Coen]. For example, using a sketch interface tied to the laser pointer system could unveil many more natural ways of interacting with wall displays. By adding pointing recognition and other forms of watching, the user could increase the power of the metaphor of the invisible assistant. Humans, in most interactions, rely on their physical body—using body language, gesture, direction of gaze—to aid their verbal communication. If the invisible assistant could have access to this kind of information, the ability to understand the user would be greatly enhanced.

8.2 Physical Devices
Intelligent Room aside, there are many other physical devices far superior to the tablet and monitor. Originally, I wanted to embed the system in an interactive whiteboard to allow the user to draw on the display surface itself, to better capture the feeling of drawing.

Making a completely natural system that did this is difficult, but I think it would add significantly to the feel and hence the effectiveness of the overall system. Miniaturization of this onto paper-sized displays similar in structure to the handheld Pilot computers could also be quite useful. Having a
lightweight sketch pad, weighing no more than a standard pad of paper, would be ideal because then the engineer, released from the tethers of the classical monitor and mouse, would be able to use the computer system as a sketching environment and would not be tempted to revert to scrap paper.

Most sketching is not done in front of a computer, it is done any place and every place. The only way engineers are going to use a system for their preliminary design work is if the system is wherever they are when they begin to do this work. It does not matter how natural the system is if the person cannot access it unless they are in front of a computer. A good physical embodiment of the system is crucial.

This is especially true for design rational capture where it is very important to capture the first few ideas and initial brainstorming. The system needs to be too convenient not to use rather than a thing that should be used. The system needs to be scrap paper.

8.3 Dirty Sketches
The computer should not be a completely clean and precise drafting machine. We need to maintain the quality of the original sketch along with the exact and shiny version. A rough sketch and a clean diagram can differ only in the wiggle in the lines, the lack of erase marks, and other such features. Exactitude has its uses, but can be limiting to the creative mind. In fact, there are programs that take architectural plans and distort them so they look more like a human sketch because precise blueprints assume such ponderous import to the viewer that the viewer is unable to easily question them.

Through our system we can do better than automated distortion; by storing and scaling the original drawings we can provide the flexibility of a CAD diagram while still keeping the ink-on-paper look. Since the original sketch is in the sketching system, one could, through standard morphing technologies, slowly fade from the original sketch to clean and precise CAD illustrations.
I did attempt to keep a "sketching" feel with the house program by intentionally not squaring up the walls, corners, and regions. I believe this makes the results much more expressive and aesthetically pleasing. They very much look like informal sketches, which are easily manipulated, rather than formal designs, which are to be viewed and not touched.

8.4 Simultaneous Being of Widgets
When watching someone drawing it is often unclear what they are drawing at the beginning. As the person continues to draw the possibilities dwindle and the drawing's meaning becomes more and more certain. In our sketching program, however, recognition occurs immediately. Each stroke is either recognized as a specific thing or not recognized at all. There are no provisions for reducing the range of possibilities.

A next, and very large, step would be to have a system that could simultaneously recognize objects as being more than one thing. A line could be tentatively labeled as both a wall and a wire until it was made more clear due to its context and surrounding drawing. Having a gradual recognition process rather than an edge driven one would make it easier to identify objects which consist of multiple strokes.

The current system will eventually hit an effective limit in the complexity of drawing it can support. To be able to go beyond this limit will require "tentative" recognition, where possibilities are identified but the original strokes are kept to influence further recognition.

8.5 Learning
Each user has an individual style of drawing. Each user would like to be able to inform the system of new objects and their meanings. For example, a user might want to create shortcuts so that if they
made some gesture, an entire room is outfitted in some standard fashion. They should be able to do this simply by telling the system, "this gesture means make this room look like that."

It should also be possible to describe various kinds of checks the system should make on a drawing. The user should be able to tell the system, "this is an electric socket" when the system has never heard of an electric socket before. The system could then find and label all of the electric sockets, and then the user would then be able to say, "make sure all the rooms have at least two electric sockets on the walls."

In general, the sketching system needs to have a general understanding of what objects are, and how to think about them. Given this kind of understanding, the user would be free to build context and context understanding as they sketch merely by informing the system what the system should do.
9 Conclusion

We have designed a system that is a base platform for generalized sketch recognition. We hope that with this system it is possible to develop fairly sophisticated sketch recognition programs for a variety of purposes. To illustrate the system we designed two simple sketching applications—one for drawing finite state machine, and one for designing house floor plans. Both of these applications illustrate the features of the system, and the two of them together suggest that the system is extendible.

The house-sketching program reveals some potential limitations with dynamically manipulating large amounts of data.

There has been extensive research on drawing recognition, voice interaction, and computer-aided design. There has even been some research into applying knowledge systems to drawing. These separate branches have not, however, been combined into a single whole. Due to this, products from these research endeavors have failed to produce something that is easily accessible and useful. By creating a system where a user can—through input familiar to them, namely voice and sketching—creatively design on a computer without its characteristic constraints, we can begin to address the question of how a computer might help the design process.

Today we have some dim visions as to what a computer could do for a user. Sadly, the frustration of trying to get the computer to do these things makes any user turn away from the keyboard and pick up their pen. By changing the way we interact with the computer we can really begin to exploit it. In the future I see architects hunched over a large screen resembling a drafting board happily drawing with an electronic pen. I see them sketching the floor plans of a house without using straight edges, rulers, or templates. They no longer need piles of books on building codes, symbols, and
standard measurements. I see the architect sketching out the house and pointing to things and telling
the computer what they are made of so the computer can properly shade them, scale them, or anno-
tate them. I see the architect getting help checking wiring for code. I see them easily manipulating
the design once it has been sketched out. I believe the such a system will do what it has only been
hoped that CAD programs would do: it will provide an environment as easy to use as pen and paper,
but which produces diagrams of the caliber you can only get of a CAD program. It will provide an
environment that frees up the architect so she can express her creative thoughts without hindrance,
without even the hindrance of the old tools of the architecture craft.

Word processing revolutionized how people thought about writing, and likewise having interactive
sketching such as this could revolutionize how people think about technical drawing. Designing on a
computer should be more than designing on paper rather than other than designing on paper. [Dix]
10 References


[Saund 89] Saund, Eric (1989) Adding Scale to the Primal Sketch. IEEE


