HUMAN COMPUTER INTERACTION

Direct Manipulation Principles

Dr. Debasis Samanta

INDIAN INSTITUTE OF TECHNOLOGY, KHARAGPUR

1.0 Introduction

The term ***direct manipulation*** was coined by Shneiderman (1974, 1982, 1983) to refer to systems having the following properties:

1. Continuous representation of the object of interest.

2. Physical actions or labeled button presses instead of complex syntax.

3. Rapid incremental reversible operations whose impact on the object of interest is immediately visible. (Shneiderman, 1982, p. 251)

1.2 Virtues of Direct Manipulation Systems

Direct manipulation interfaces seem remarkably powerful. Shneiderman (1982) has suggested that direct manipulation systems have the following virtues:

1. Novices can learn basic functionality quickly, usually through a demonstration by a more experienced user.

2. Experts can work extremely rapidly to carry out a wide range of tasks, even defining new functions and features.

3. Knowledgeable intermittent users can retain operational concepts.

4. Error messages are rarely needed.

5. Users can see immediately if their actions are furthering their goals, and if not, they can simply change the direction of their activity.

6. Users have reduced anxiety because the system is comprehensible and because actions are so easily reversible. (Shneiderman, 1982, p. 251)

Certainly there must be problems as well as benefits. It turns out that the concept of direct manipulation is complex. Moreover, although there are important benefits there are also costs. Like everything else, direct manipulation systems trade off one set of virtues and vices against another. It is important that we understand these trade-offs. A checklist of surface features is unlikely to capture the real sources of power in direct manipulation interfaces.

2. History of Direct Manipulation

Hints of direct manipulation programming environments have been around for quite some time. The first major landmark is Sutherland's *Sketchpad,* a graphical design program (Sutherland, 1963). Sutherland's goal was to devise a program that would make it possible for a person and a computer "to converse rapidly through the medium of line drawings." Sutherland's work is a landmark not only because of historical priority but because of the ideas that he helped develop: He was one of the first to discuss the power of graphical interfaces, the conception of a display as "sheets of paper," the use of pointing devices, the virtues of constraint representations, and the importance of depicting abstractions graphically.

Sutherland's ideas took 20 years to have widespread impact. The lag is perhaps due more to hardware limitations than anything else. Highly interactive, graphical programming requires the ready availability of considerable computational power, and it is only recently that machines capable of supporting this type of computational environment have become inexpensive enough to be generally available. Now we see these ideas in many of the computer-aided design and manufacturing systems, many of which can trace their heritage directly to Sutherland's work. Borning's *ThingLab* program (1979) explored a general programming environment, building upon many of Sutherland's ideas within the Smalltalk programming environment. More recently direct manipulation systems have been appearing with reasonable frequency. For example, Bill Budge's Pinball Construction Set (Budge, 1983) permits a user to construct an infinite variety of electronic pinball games by directly manipulating graphical objects that represent the components of the game surface. Other examples exist in the area of intelligent training systems (e.g., the Steamer system of Hollan, Hutchins, & Weitzman, 1984; Hollan, Stevens, & Williams, 1980). Steamer makes use of similar techniques and also provides tools for the construction of interactive graphical interfaces. Finally, spreadsheet programs incorporate many of the essential features of direct manipulation. In the lead article of *Scientific American's* special issue on computer software, Kay (1984) claims that the development of dynamic spreadsheet systems gives strong hints that programming styles are in the offing that will make programming as it has been done for the past 40 years - that is, by composing text that represents instructions - obsolete.

3. Two Aspects of Directness: Distance and Engagement

There are two distinct aspects of the feeling of directness. One involves a notion of the distance between one's thoughts and the physical requirements of the system under use. A short distance means that the translation is simple and straightforward, that thoughts are readily translated into the physical actions required by the system and that the system output is in a form readily interpreted in terms of the goals of interest to the user. We will use the term ***directness*** to refer to the feeling that results from interaction with an interface. The term *distance* will be used to describe factors which underlie the generation of the feeling of directness.

The second aspect of directness concerns the qualitative feeling of engagement, the feeling that one is directly manipulating the objects of interest. There are two major metaphors for the nature of human-computer interaction, a conversation metaphor and a model-world metaphor. In a system built on the conversation metaphor, the interface is a language medium in which the user and system have a conversation about an assumed, but not explicitly represented world. In this case, the interface is an implied intermediary between the user and the world about which things are said. In a system built on the model-world metaphor, the interface is itself a world where the user can act, and which changes state in response to user actions. The world of interest is explicitly represented and there is no intermediary between user and world. Appropriate use of the model-world metaphor can create the sensation in the user of acting upon the objects of the task domain themselves. We call this aspect of directness direct engagement.

3.1 Distance

We call one underlying aspect of directness distance to emphasize the fact that directness is never a property of the interface alone, but involves a relationship between the task the user has in mind and the way that task can be accomplished via the interface. Here the critical issues involve minimizing the effort required to bridge the gulf between the user's goals and the way they must be specified to the system.

An interface introduces distance to the extent there are gulfs between a person's goals and knowledge and the level of description provided by the systems with which the person must deal. These are referred to as the ***gulf of execution*** and the ***gulf of*** **evaluation** shown in the figure below. The gulf of execution is bridged by making the commands and mechanisms of the system match the thoughts and goals of the user. The gulf of evaluation is bridged by making the output displays present a good conceptual model of the system that is readily perceived, interpreted, and evaluated. The goal in both cases is to minimize cognitive effort.

We suggest that the feeling of directness is inversely proportional to the amount of cognitive effort it takes to manipulate and evaluate a system and, moreover, that cognitive effort is a direct result of the gulfs of execution and evaluation. The better the interface to a system helps bridge the gulfs, the less cognitive effort needed and the more direct the resulting feeling of interaction.

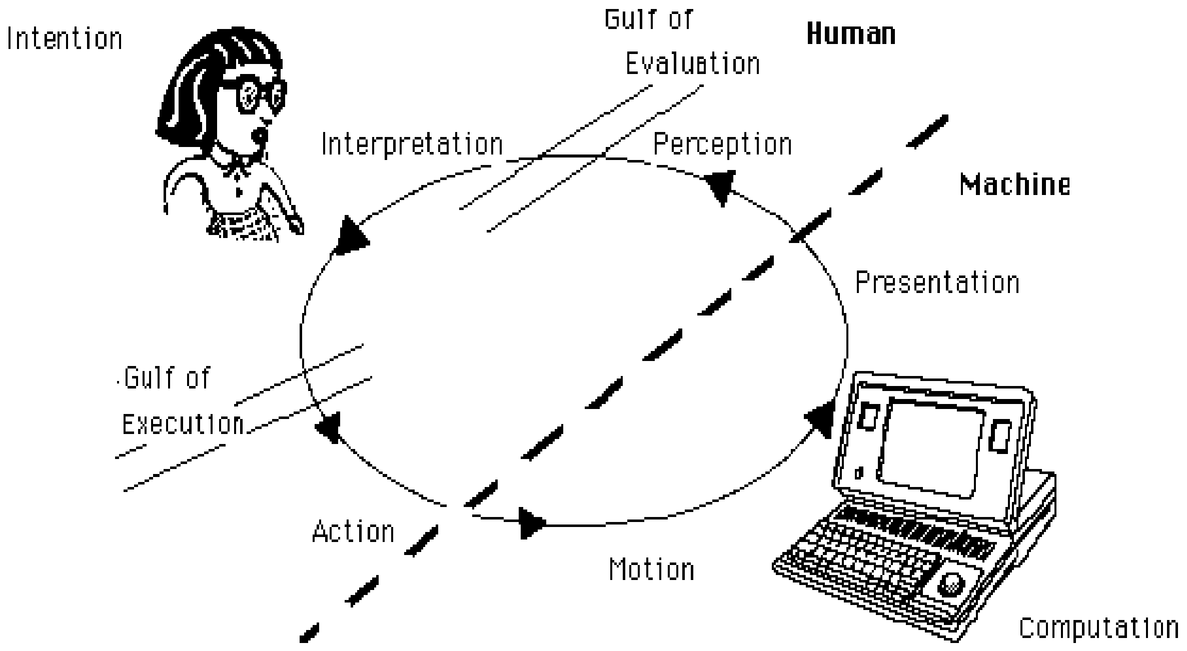


Fig. The gulfs of execution and evaluation. Each gulf is unidirectional: The gulf of execution goes from goals to system state; the gulf of evaluation goes from system state to goals.

3.2 Direct Engagement

The description of the nature of interaction to this point begins to suggest how to make a system less difficult to use, but it misses an important point, a point that is the essence of direct manipulation. The analysis of the execution and evaluation process explains why there is difficulty in using a system, and it says something about what must be done to minimize the mental effort required to use a system. But there is more to it than that. The systems that best exemplify direct manipulation all give the qualitative feeling that one is directly engaged with control of the objects- not with the programs, not with the computer, but with the semantic objects of our goals and intentions. This is the feeling that Laurel (1986) discusses: a feeling of first-personness, of direct engagement with the objects that concern us. Are we analyzing data? Then we should be manipulating the data themselves; or if we are designing an analysis of data, we should be manipulating the analytic structures themselves. Are we playing a game? Then we should be manipulating directly the game world, touching and controlling the objects in that world, with the output of the system responding directly to our actions, and in a form compatible with them.

Historically, most interfaces have been built on the conversation metaphor. There is power in the abstractions that language provides (we discuss some of this later), but the implicit role of interface as an intermediary to a hidden world denies the user direct engagement with the objects of interest. Instead, the user is in direct contact with linguistic structures, structures that can be interpreted as referring to the objects of interest, but that are not those objects themselves. Making the central metaphor of the interface that of the model world supports the feeling of directness. Instead of describing the actions of interest, the user performs those actions. In a conventional interface, the system describes the results of the actions. In a model world the system directly presents the actions taken upon the objects. This change in central metaphor is made possible by relatively recent advances in technology. One of the exciting prospects for the study of direct manipulation is the exploration of the properties of systems that provide for direct engagement.

Building interfaces based on the model-world metaphor requires a special sort of relationship between the input interface language and the output interface language. In particular, the output language must represent its subject of discourse in a way that natural language does not normally do. The expressions of a direct manipulation output language must behave in such a way that the user can assume that they, in some sense, are the things they refer to. DiSessa (1985) calls this "naive realism." Furthermore, the nature of the relationship between input and output language must be such that an output expression can serve as a component of an input expression. Draper (1986) has coined the term *inter-referential* 1/0 to refer to relationships between input and output in which an expression in one can refer to an expression in the other. When these conditions are met, it is as if we are directly manipulating the things that the system represents.

Thus, if we consider a system in which a file is represented by an image on the screen and actions are done by pointing to and manipulating the image. In this case, if we can specify a file by pointing at the screen representation, we have met the goal that an expression in the output language (in this case, an image) is allowed as a component of the input expression (in this case, by pointing at the screen representation). If we ask for a listing of files, we would want the result to be a representation that can, in turn, be used directly to specify the further operations to be done. Notice that this is not how a conversation works. In conversation, one may refer to what has been said previously, but one cannot operate upon what has been said. This requirement does not necessarily imply an interface of pictures, diagrams, or icons. It can be done with words and descriptions. The key properties are that the objects, whatever their form, have behaviors and can be referred to by other objects, and that referring to an object causes it to behave. In the file-listing example, we must be able to use the output expression that represents the file in question as a part of the input expression calling for whatever operation we desire upon that file, and the output expression that represents the file must change as a result of being referred to in this way. The goal is to permit the user to act as if the representation is the thing itself.

These conditions are met in many screen editors when the task is the arrangement of strings of characters. The characters appear as they are typed. They are then available for further operations. We treat them as though they are the things we are manipulating. These conditions are also met in the statistics example with which we opened this article (Figure l), and in Steamer. The special conditions are not met in file-listing commands on most systems, the commands that allow one to display the names and attributes of file structure.

The issue is that the outputs of these commands are simply “names” of the objects, and operating on the names does nothing to the objects to which the names refer. In a direct manipulation situation, we would feel that we had the files in front of us, that the program that “listed” the files actually placed the files before us. Any further operation on the files would take place upon the very objects delivered by the directory-listing command. This would provide the feeling of directly manipulating the objects that were returned.

The point is that when an interface presents a world of behaving objects rather than a language of description, manipulating a representation can have the same effects and the same feel as manipulating the thing being represented. The members of the audience of a well-staged play willfully suspend their beliefs that the players are actors and become directly engaged in the content of the drama. In a similar way, the user of a well-designed model-world interface can willfully suspend belief that the objects depicted are artifacts of some program and can thereby directly engage the world of the objects. This is the essence of the “first-personness” feeling of direct engagement. Let us now return to the issue of distance and explore the ways that an interface can be direct or indirect with respect to a particular task.

3.3 Two forms of Distance: Semantic and Articulatory

Whenever we interact with a device, we are using an interface language. That is, we must use a language to describe to the device the nature of the actions we wish to have performed. This is true regardless of whether we are dealing with an interface based on the conversation metaphor or on the model world metaphor, although the properties of the language in the two cases are different. A description of desired actions is an expression in the interface language.

The notion of an interface language is not confined to the everyday meaning of language. Setting a switch or turning a steering wheel can be expressions in an interface language if switch setting or wheels turning are how one specifies the operations that are to be done. After an action has been performed, evaluation of the outcome requires that the device make available some indication of what has happened: that output is an expression in the output interface language. Output interface languages are often impoverished. Frequently the output interface language does not share vocabulary with the input interface language. Two forms of interface language- two dialects exist to span the gulfs between user and device: the input interface language and the output interface language.

Both the languages people speak and computer programming languages are almost entirely symbolic in the sense that there is an arbitrary relationship between the form of a vocabulary item and its meaning. The reference relationship is established by convention and must be learned. There is no way to infer meaning from form for most vocabulary items. Because of the relative independence of meaning and form we describe separately two properties of interface languages: semantic distance and articulatory distance. The figure below summarizes the relationship between semantic and articulatory distance. In the following sections we treat each of these distances separately and discuss them in relation to the gulfs of execution and evaluation.

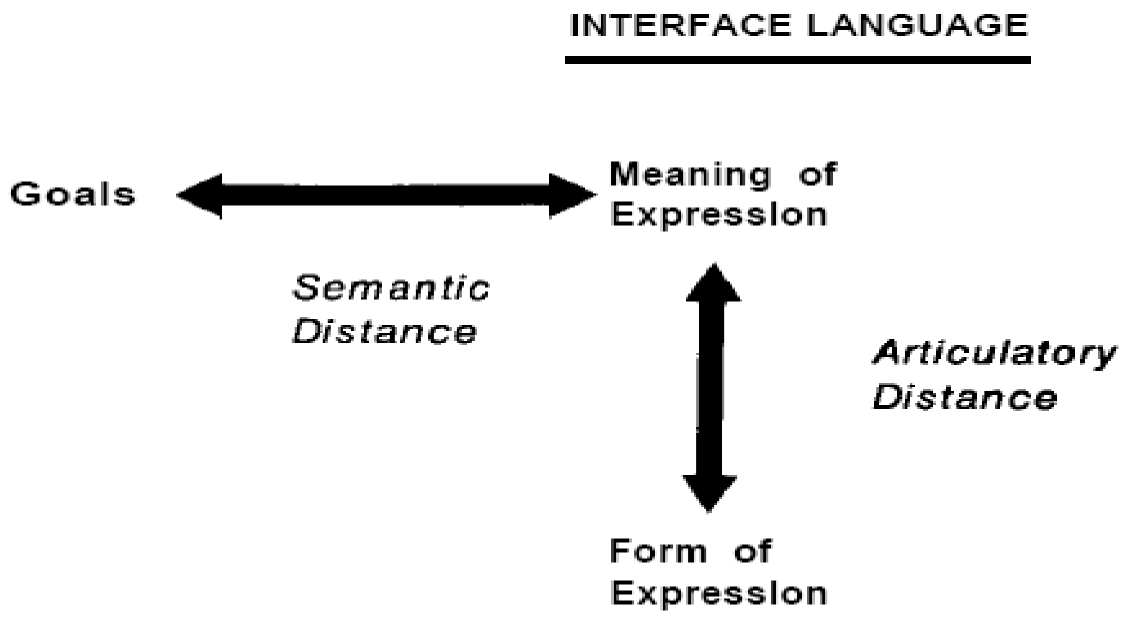


Fig. Every expression in the interface language has a meaning and a form. Semantic distance reflects the relationship between the user intentions and the meaning of expressions in the interface languages both for input and output. Articulatory distance reflects the relationship between the physical form of an expression in the interaction language and its meaning, again, both for input and output. The easier it is to go from the form or

appearance of the input or output to meaning, the smaller the articulatory distance.

3.4 Semantic Distance

Semantic distance concerns the relation of the meaning of an expression in the interface language to what the user wants to say. Two important questions about semantic distance are :

1. *Is it possible to say what one wants to say in this language?* That is, does the language support the user's conception of the task domain? Does it encode the concepts and distinctions in the domain in the same way that the user thinks about them?
2. *Can the thing1 of interest be said concisely?* Can the user say what is wanted in a straightforward fashion, or must the user construct a complicated expression to do what appears in the user's thoughts as a conceptually simple piece of work?

Semantic distance is an issue with all languages. Natural languages generally evolve such that they have rich vocabularies for domains that are of importance to their speakers. When a person learns a new language- especially when the language is from a different culture - the new language may seem indirect, requiring complicated constructs to describe things the learner thinks should be easy to say. But the differences in apparent directness reflect differences in what things are thought important in the two cultures. Natural languages can and do change as the need arises. This occurs through the introduction of new vocabulary or by changing the meaning of existing terms. The result is to make the language semantically more direct with respect to the topic of interest.

3.5 Semantic Distance in the Gulfs of Execution and Evaluation

The Gulf of Execution

At the highest level of description, a task may be described by the user's intention: "compose this piece" or "format this paper." At the lowest level of description, the performance of the task consists of the shuffling of bits inside the machine. Between the interface and the low-level operations of the machine is the system-provided task-support structure that implements the expressions in the interface language. The situation that Perlis (1982) called the "Turing tarpit" is one in which the interface language lies near or at the level of bit shuffling of a very simple abstract machine. In this case, the entire burden of spanning the gulf from user intention to bit manipulation is carried by the user. The relationship between the user's intention and the organization of the instructions given to the machine is distant, complicated, and hard to follow.

Where the machine is of minimal complexity, as is the case with the Turing machine example, the wide gulf between user intention and machine instructions must be filled by the user's extensive planning and translation activities. These activities are difficult and rife with opportunities for error.

Semantic directness requires matching the level of description required by the interface language to the level at which the person thinks of the task. It is always the case that the user must generate some information-processing structure to span the gulf. Semantic distance in the gulf of execution reflects how much of the required structure is provided by the system and how much by the user. The more that the user must provide, the greater the distance to be bridged.

The Gulf of Evaluation

On the evaluation side, semantic distance refers to the amount of processing structure that is required for the user to determine whether the goal has been achieved. If the terms of the output are not those of the user's intention, the user will be required to translate the output into terms that are compatible with the intention in order to make the evaluation. For example, suppose a user's intent is to control how fast the water level in a tank rises. The user does some controlling action and observes the output. But if the output only shows the current value, the user has to observe the value over time and mentally compare the values at different times to see what the rate of change is. The information needed for the evaluation is in the output, but it is not there in a form that directly fits the terms of the evaluation. The burden is on the user to perform the required transformations, and that requires effort. Suppose the rate of change were directly displayed. This indication reduces the mental workload, making the semantic distance between intentions and output language much shorter.

3.6 Reducing the Semantic Distance That Must Be Spanned

Higher-Level Languages

One way to bridge the gulf between the intentions of the user and the specifications required by the computer is well known: Provide the user with a higher-level language, one that directly expresses frequently encountered structures of problem decomposition. Instead of requiring the complete decomposition of the task to low-level operations, let the task be described in the same language used within the task domain itself. Although the computer still requires low-level specification, the job of translating from the domain language to the programming language can be taken over by the machine itself. This implies that designers of higher-level languages should consider how to develop interface languages for which it will be easy for the user to create the mediating structure between intentions and expressions in the language. One way to facilitate this process is to provide consistency across the interface surface. That is, if the user builds a structure to make contact with some part of the interface surface, a savings in effort can be realized if it is possible to use all or part of that same structure to make contact with other areas.

The result of matching a language to the task domain brings both good news and bad news. The good news is that tasks are easier to specify. Even if considerable planning is still required to express a task in a high-level language, the amount of planning and translation that can be avoided by the user and passed off to the machine can be enormous. The bad news is that the language has lost generality. Tasks that do not easily decompose into the terms of the language may be difficult or impossible to represent. In the extreme case, what can be done is easy to do, but outside that specialized domain, nothing can be done. The power of a specialized language system derives from carefully specified primitive operations, selected to match the predicted needs of the user, thus capturing frequently occurring structures of problem decomposition. The trouble is that there is a conflict between generality and matching to any specific problem domain. Some high-level languages and operating systems have attempted to close the gap between user intention and the interaction language while preserving freedom and ease of general expression by allowing for extensibility of the language or operating system. Such systems allow the users to move the interface closer to their conception of the task.

The Lisp language and the UNIX operating system serve as examples of this phenomenon. Lisp is a general-purpose language, but one that has extended itself to match a number of special high-level domains. As a result, Lisp can be thought of as having numerous levels on top of the underlying language kernel. There is a cost to this method. As more and more specialized domain levels get added, the language system gets larger and larger, becoming more clumsy to use, more expensive to support, and more difficult to learn. Just look at any of the manuals for the large Lisp systems (Interlisp, Zetalisp) to get a feel for the complexity involved. The same is true for the UNIX operating system, which started out with a number of low-level, general primitive operations. Users were allowed (and encouraged) to add their own, more specialized operations, or to package the primitives into higher-level operations. The results in all these cases are massive systems that are hard to learn and that require a large amount of support facilities. The documentation becomes huge, and not even system experts know all that is present. Moreover, the difficulty of maintaining such a large system increases the burden on everyone, and the possibility of having standard interfaces to each specialized function has long been given up.

The point is that as the interface approaches the user's intention end of the gulf, functions become more complicated and more specialized in purpose. Because of the incredible variety of human intentions, the lexicon of a language that aspires to both generality of coverage and domain-specific functions can grow very large. In any of the modern dialects of Lisp one sees a microcosm of the argument about high-level languages in general. The fundamentals of the language are simple, but a great deal of effort is required to do anything useful at the low level of the language itself. Higher-level functions written in terms of lower-level ones make the system easier to use when the functions match intentions, but in doing so they may restrict possibilities, proliferate vocabulary, and require that a user know an increasing amount about the language of interaction rather than the domain of action.

Make the Output Show Semantic Concepts Directly

An example of reducing semantic distance on the output side is provided by the scenario of controlling the rate of filling a water tank, described above. In that situation, the output display was modified to show rate of flow directly, something normally not displayed but instead left to the user to compute mentally. In similar fashion, the change from line-oriented text editors to screen oriented text editors, where the effects of editing commands can be seen instantly, is another example of matching the display to the user's semantics. In general, the development of WYSIWYG ("What You See Is What You Get") systems provides other examples. And finally, spreadsheet programs have been valuable, in part because their output format continually shows the state of the system as values are changed. The attempt to develop good semantic matches with the system output confronts the same conflict between generality and power faced in the design of input languages. If the system is too specific and specialized, the output displays lack generality. If the system is too rich, the user has trouble learning and selecting among the possibilities. One solution for both the output and input problem is to abandon hope of maintaining general computing and output ability and to develop special-purpose systems for particular domains or tasks. In such a world, the location of the interface in semantic space is pushed closer to the domain language description. Here, things of interest are made simple because the lexicon of the interface language maps well into the lexicon of domain description. Considerable planning may still go on in the conception of the domain itself, but little or no planning or translation is required to get from the language of domain description to the language of the interface. The price paid for these advantages is a loss of generality: Many things are unnatural or even impossible.

Automated Behavior Does Not Reduce Semantic Distance

Cognitive effort is required to plan a sequence of actions to satisfy some intent. Generally, the more structure required of the user, the more effort use of the system will entail. However, this gap can be overcome if the users become familiar enough with the system. Structures that are used frequently need not be rebuilt every time they are needed if they have been remembered. Thus, a user may remember how to do something rather than having to re-derive how to do it. It is well known that when tasks are practiced sufficiently often, they become automated, requiring little or no conscious attention. As a result, over time the use of an interface to solve a particular set of problems will feel less difficult and more direct. Experienced users will sometimes argue that the interface they use directly satisfies their intentions, even when less skilled users complain of the complexity of the structures. To skilled users, the interface feels direct because the invocation of mediating structure has been automated. They have learned how to transform frequently arising intentions into action specifications. The result is a feeling of directness as compelling as that which results from semantic directness. As far as such users are concerned, the intention comes to mind and the action gets executed. There are no conscious intervening stages. (For example, a user of the vi text editor expressed this as follows: "I am an expert user of vi, and when I wish to delete a word, all I do is think 'delete that word,' my fingers automatically type 'dw,' and the word disappears from the screen. How could anything be more direct?" The frequent use of even a poorly designed interface can sometimes result in a feeling of directness like that produced by a semantically direct interface. A user can compensate for the deficiencies of the interface through continual use and practice so that the ability to use it becomes automatic, requiring little conscious activity. While automatism is one factor which can contribute to a feeling of directness, it is essential for an interface designer to distinguish it from semantic distance. Automatization does not reduce the semantic distance that must be spanned; the gulfs between a user's intentions and the interface must still be bridged by the user. Although practice and the resulting expertise can make the crossing less difficult, it does not reduce the magnitude of the gulfs. Planning activity may be replaced by single memory retrieval so that instead of figuring out what to do, the user remembers what to do. Automatization may feel like direct control, but it comes about for completely different reasons than semantic directness. Automatization is useful, for it improves the interaction of the user with the system, but the feeling of directness it produces depends only on how much practice a particular user has with the system and thus gives the system credit for the work the user has done. Although we need to remember that this happens, that users may adjust themselves to the interface and, with sufficient practice, may view it as directly supporting their intentions, we need to distinguish between the cases in which the feeling of directness originates from a close semantic coupling between intentions and the interface language and that which originates from practice. The resultant feeling of directness might be the same in the two cases, but there are crucial differences between how the feeling is acquired and what one needs to do as an interface designer to generate it.

The User Can Adapt to the System Representation

Another way to span the gulf is for the users to change their own conceptualization of the problem so that they come to think of it in the same terms as the system. In some sense, this means that the gulf is bridged by moving the user closer to the system. Because of their experience with the system, the users change both their understanding of the task and the language with which they think about issues. This is related to the notion of linguistic determinism. If it is true that the way we think about something is shaped by the vocabulary we have for talking about it, then it is important for the designer of a system to provide the user with a good representation of the task domain in question. The interface language should provide a powerful, productive way of thinking about the domain.

This form of the users adapting to the system representation takes place at a more fundamental level than the other ways of reducing semantic distance. While moving the interface closer to the users' intentions may make it difficult to realize some intentions, changing the users' conception of the domain may prevent some intentions from arising at all. So while a well-designed special purpose language may give the users a powerful way of thinking about the domain, it may also restrict the users' flexibility to think about the domain in different ways. The assumption that a user may change conceptual structure to match the interface language follows from the notion that every interface language implies a representation of the tasks it is applied to. The representation implied by an interface is not always a coherent one. Some interfaces provide a collection of partially overlapping views of a task domain. If a user is to move toward the model implied by the interface, and thus reduce the semantic distance, that model should be coherent and consistent over some conception of the domain. There is, of course, a trade-off here between the costs to the user of learning a new way to think about a domain and the potential added power of thinking about it in the new way.

Virtuosity and Semantic Distance

Sometimes users have a conception of a task and of a system that is broader and more powerful than that provided by an interface. The structures they build to make contact with the interface go beyond it. This is how we characterize virtuoso performances in which the user may "misuse" limited interface tools to satisfy intentions that even the system designer never anticipated. In such cases of virtuosity the notion of semantic distance becomes more complicated and we need to look very carefully at the task that is being accomplished. Semantic directness always involves the relationship between the task one wishes to accomplish and the ways the interface provides for accomplishing it. If the task changes, then the semantic directness of the interface may also change. Consider a musical example: Take the task of producing a middle-C note on two musical instruments, a piano and a violin. For this simple task, the piano provides the more direct interface because all one need do is find the key for middle-C and depress it, whereas on the violin, one must place the bow on the G string, place a choice of fingers in precisely the right location on that string, and draw the bow. A piano's keyboard is more semantically direct than the violin's strings and bow for the simple task of producing notes. The piano has a single well-defined vocabulary item for each of the notes within its range, while the violin has an infinity of vocabulary items, many of which do not produce proper notes at all. However, when the task is playing a musical piece well rather than simply producing notes, the directness of the interfaces can change. In this case, one might complain that a piano has a very indirect interface because it is a machine with which the performer "throws hammers at strings." The performer has no direct contact with the components that actually produce the sound, and so the production of desired nuances in sound is more difficult. Here, as musical virtuosity develops, the task that is to be accomplished also changes from just the production of notes to concern for how to control more subtle characteristics of the sounds like vibrato, the slight changes in pitch used to add expressiveness. For this task the violin provides a semantically more direct interface than the piano. Thus, as we have argued earlier, an analysis of the nature of the task being performed is essential in determining the semantic directness of an interface.

3.7 Articulatory Distance

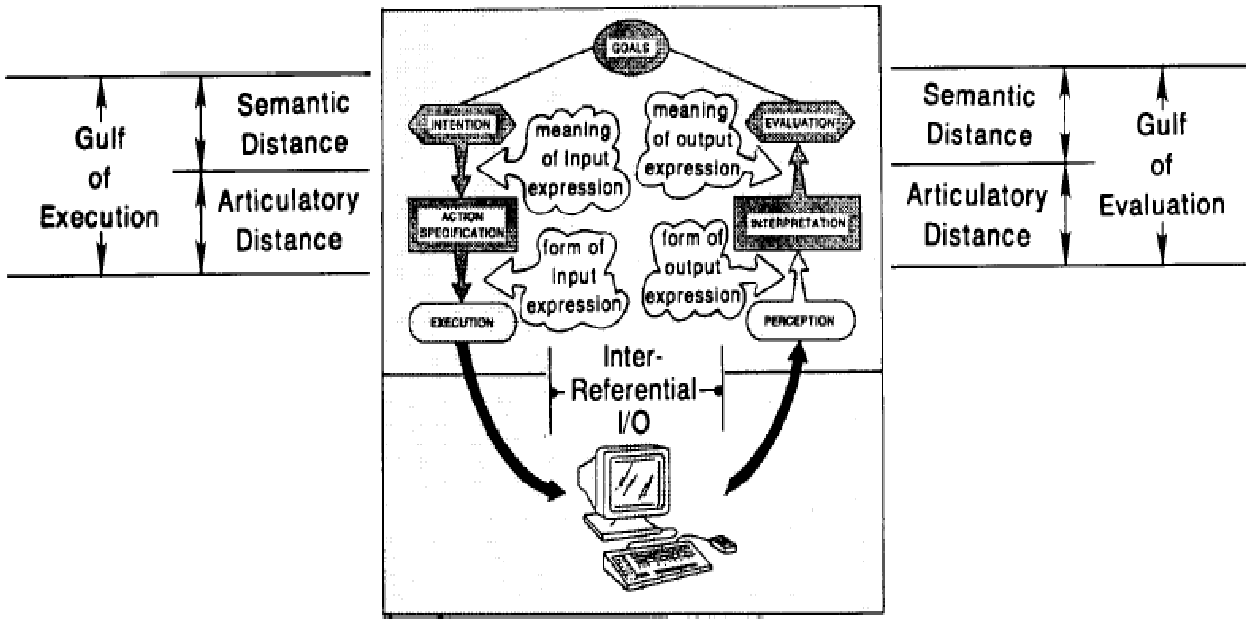
In addition to its meaning, every vocabulary item in every language has a physical form and that form has an internal structure. Words in natural languages, for example, have phonetic structure when spoken and typographic structure when printed. Similarly, the vocabulary items that constitute an interface language have a physical structure. Where *semantic distance* has to do with the relationship between user's intentions and meanings of expressions, *articulatory distance* has to do with the relationship between the meanings of expressions and their physical form. On the input side, the form may be a sequence of character-selecting key presses for a command language interface, the movement of a mouse and the associated "mouse clicks" in a pointing device interface, or a phonetic string in a speech interface. On the output side, the form might be a string of characters, a change in an iconic representation, or variation in an auditory signal. There are ways to design languages such that the relationships between the forms of the vocabulary items and their meanings are not arbitrary. One technique is to make the physical form of the vocabulary items structurally similar to their meanings. In spoken language this relationship is called onomatopoeia. Onomatopoetic words in spoken language refer to their meanings by imitating the sound they refer to. Thus we talk about the "boom" of explosions or the "cock-a-doodle-doo" of roosters. There is an economy here in that the user's knowledge of the structure of the surface acoustical form has a non arbitrary relation to meaning. There is a directness of reference in this imitation; an intervening level of arbitrary symbolic relations is eliminated. Other uses of language exploit this effect partially. Thus, although the word "long is arbitrarily associated with its meaning, sentences like "She stayed a looooooooooong time" exploit a structural similarity between the surface form of "long" (whether written or spoken) and the intended meaning. The same sorts of things can be done in the design of interface languages.

In many ways, the interface languages should have an easier time of exploiting articulatory similarity than do natural languages because of the rich technological base available to them. Thus, if the intent is to draw a diagram, the interface might accept as input drawing motions. In turn, it could present as output diagrams, graphs, and images. If one is talking about sound patterns in the input interface language, the output could be the sounds themselves. The computer has the potential to exploit articulatory similarities through technological innovation in the varieties of dimensions upon which it can operate. This potential has not been exploited, in part because of economic constraints. The restriction to simple keyboard input limits the form and structure of the input languages and the restriction to simple, alphanumeric terminals with small, low-resolution screens, limits the form and structure of the output languages.

3.8 Articulatory Distance in the Gulfs of Execution and Evaluation

The relationships among semantic distance, articulatory distance, and the gulfs of execution and evaluation are illustrated in the figure below. Take the simple, commonplace activity of moving a cursor on the screen. If we do this by moving a mouse, pointing with a finger or a light pen at the screen, or otherwise mimicking the desired motion, then at the level of action execution, these interactions all exhibit articulatory directness. The meaning of the intention is cursor movement and the action is specified by means of a similar movement. One way to achieve articulatory directness at the input side is to provide an interface that permits specification of an action by mimicking it, thus supporting an articulatory similarity between the vocabulary item and its meaning. Any nonarbitrary relationship between the form of an item and its meaning can be a basis for articulatory directness. While structural relationships of form to meaning may be desirable, it is sometimes necessary to resort to an arbitrary relationship of form to meaning. Still, some arbitrary relationships are easier to learn than others. It may be possible to exploit previous user knowledge in creating this relationship. Much of the work on command names in command language interfaces is an instance of trying to develop memorable and discriminable relationships between the forms and the meanings of command names.

Articulatory directness on the output side is similar. If the user is following the changes in some variable, a moving graphical display can provide articulatory directness. A table of numbers, although containing the same semantic information, does not provide articulatory directness. Thus, the graphical display and the table of numbers might be equal in semantic directness, but unequal in articulatory directness. The goal of designing for articulatory directness is to couple the perceived form of action and meaning so naturally that the relationships between intentions and actions and between actions and output seem straight forward and obvious. In general, articulatory directness is highly dependent upon I/O technology. Increasing the articulatory directness of actions and displays requires a much richer set of input/output devices than most systems currently have. In addition to keyboards and bit-mapped screens, we see the need for various forms of pointing devices. Such pointing devices have important *spatio-mimetic* properties and thus support the articulatory directness of input for tasks that can be represented spatially. The mouse is useful for a wide variety of tasks not because of any properties inherent in itself, but because we map so many kinds of relationships (even ones that are not intrinsically spatial) on to spatial metaphors. In addition, there are often needs for sound and speech, certainly as outputs, and possibly as inputs. Precise control of timing will be necessary for those applications where the domain of interest is time sensitive.



Forming an intention is the activity that spans semantic distance in the gulf of execution. The intention specifies the meaning of the input expression that is to satisfy the user's goal. Forming an action specification is the activity that spans articulatory distance in the gulf of execution. The action specification prescribes the form of an input expression having the desired meaning. The form of the input expression is executed by the user on the machine interface and the form of the output expression appears on the machine interface, to be perceived by the user. When some part of the form of a previous output expression is incorporated in the form of a new input expression, the input and output are said to be inter-referential. Interpretation is the activity that spans aritculatory distance in the gulf of evaluation. Interpretation determines the meaning of the output expression from the form of the output expression. Evaluation is the activity that spans semantic distance in the gulf of evaluation. Evaluation

assesses the relationship between the meaning of the output expression and the user's goal.

Perhaps it is stretching the imagination beyond its willing limits, but Galton (1894) suggested and carried out a set of experiments on doing arithmetic by sense of smell. Less fancifully conceived, input might be sensitive not only to touch, place, and timing, but also to pressure or to torque (see Buxton, 1986; Minsky, 1984).

Direct engagement occurs when a user experiences direct interaction with the objects in a domain. Here there is a feeling of involvement directly with a world of objects rather than of communication with an intermediary. The interactions are much like interacting with objects in the physical world. Actions apply to the objects, observations are made directly upon those objects, and the interface and the computer become invisible. Although we believe this feeling of direct engagement to be of critical importance, in fact, we know little about the actual requirements for producing it. Laurel (1986) discusses some of the requirements. At a minimum, to allow a feeling of direct engagement the system requires the following:

1. Execution and evaluation should exhibit both semantic and articulatory directness.

2. Input and output languages of the interface should be inter-referential, allowing an input expression to incorporate or make use of a previous output expression. This is crucial for creating the illusion that one is directly manipulating the objects of concern.

3. The system should be responsive with no delays between execution and the results, except where those delays are appropriate for the knowledge domain itself.

4. The interface should be unobtrusive, not interfering or intruding. If the interface itself is noticed, then it stands in a third-person relationship to the objects of interest, and detracts from the directness of the engagement.

5. In order to have a feeling of direct engagement, the interface must provide the user with a world in which to interact. The objects of that world must feel like they are the objects of interest that one is doing things with them and watching how they react. For this to be the case, the output language must present representations of objects in forms that behave in the way that the user thinks of the objects behaving. Whatever changes are caused in the objects by the set of operations must be depicted in the representation of the objects. This use of the same object as both an input and output entity is essential to providing objects that behave as if they are the real thing. It is because an input expression can

contain a previous output expression that the user feels the output expression is the thing itself and that the operation is applied directly to the thing itself.

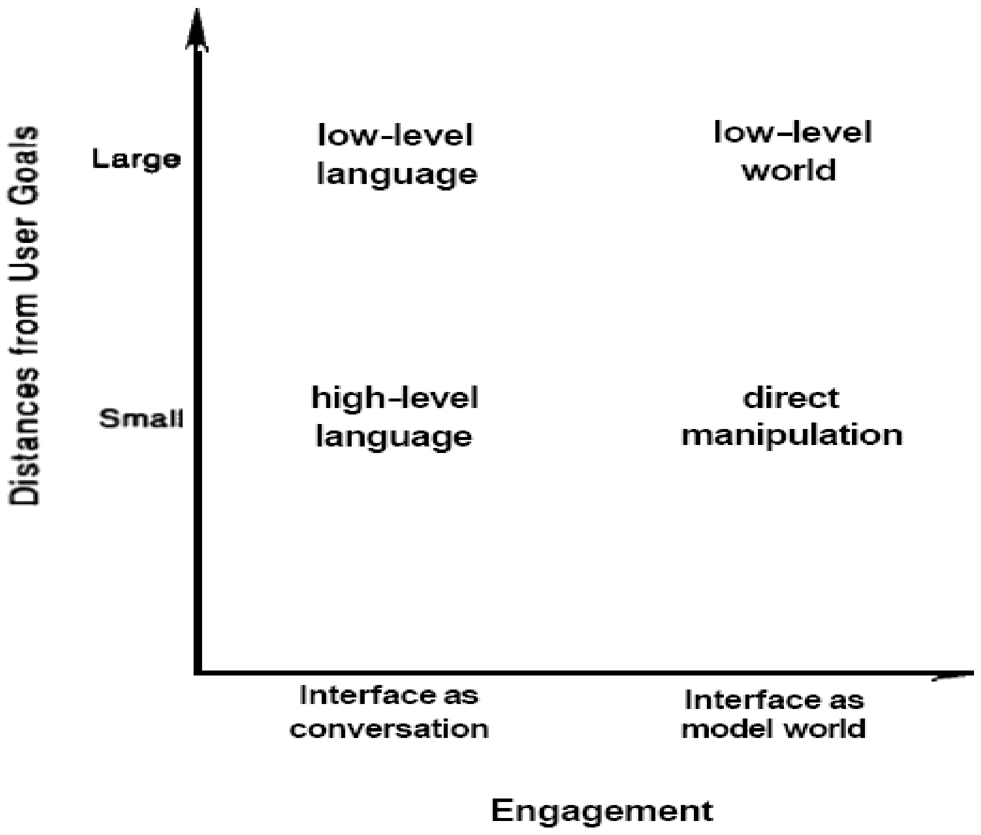
6. In addition, all of the discussions of semantic and articulatory directness apply here too because the designer of the interface must be concerned with what is to be done and how one articulates that in the languages of interaction. But the designer must also be concerned with creating and supporting an illusion. The specification of what needs to be done and evidence that it has been done must not violate the illusion, else the feeling of direct engagement will be lost.

7. One factor that seems especially relevant to maintaining this illusion is the form and speed of feedback. Rapid feedback in terms of changes in the behavior of objects not only allows for the modification of actions even as they are being executed, but also supports the feeling of acting directly on the objects themselves. It removes the perception of the computer as an intermediary by providing continual representation of system state. In addition, rapidity of feedback and continual representation of state allows one to make use of perceptual faculties in evaluating the outcome of actions. We can watch the actions take place, monitoring them much like we monitor our interactions with the physical world. The reduction in the cognitive load of mentally maintaining relevant information and the form of the interaction contribute to the feeling of engagement.

3.9 A space of Interfaces

Distance and engagement are depicted in the figure below as two major dimensions in a space of interface designs. The dimension of engagement has two landmark values: One is the metaphor of interface as conversation; the other is the metaphor of interface as model world. The dimension of distance actually contains two distances to be spanned: semantic and articulatory distances, the two kinds of gulfs that lie between the user's conception of the task and the interface language. The least direct interface is often one that provides a low-level language interface, for this is apt to provide the weakest semantic match between intentions and the language of the interface. In this case, the interface is an intermediary between the user and the task. Even worse, it is an intermediary that does not understand actions at the level of description in which the user likes to think of them. Here the user must translate intentions into complex or lengthy expressions in the language that the interface intermediary can understand.

A more direct situation arises when the central metaphor of the interface is a world. Then the user can be directly engaged with the objects in a world; but still, if the actions in that world do not match those that the user wishes to perform within the task domain, getting the task done may be a difficult process. The user may believe that things are getting done and may even experience a sense of engagement with the world, yet still be doing things at too low a level. This is the state of some of the recently introduced direct manipulation systems: They produce an immediate sense of engagement, but as the user develops experience with the system, the interface appears clumsy, to interfere too much, and to demand too many actions and decisions at the wrong level of specification. These interfaces appear on the surface to be direct manipulation interfaces, but they fail to produce the proper feelings of direct engagement with the task world. Closing the distance between the user's intentions and the level of specification of the interface language allows the user to make efficient specifications of intentions. Where this is done with a high-level language, quite efficient interfaces can be designed. This is the situation in most modern integrated programming environments. For some classes of tasks, such interfaces may be superior to direct manipulation interfaces.



A space of interfaces. The dimensions of distance from user goals and degree of engagement form a space of interfaces within which we can locate some familiar types of interfaces. Direct manipulation interfaces are those that minimize the distances and maximize engagement. As always, the distance between user intentions and the interface language depends on the nature of the task the user is performing.

Finally, the most direct of the interfaces will lie where engagement is maximized, where just the right semantic and articulatory matches are provided, and where all distances are minimized.

3.10 Adaptive agents and user models versus control Panels

Some designers promote the notion of adaptive and/or anthropomorphic agents that would carry out the users' intents and anticipate needs. Their scenarios often show a responsive, butler-like human being to represent the agent (a bow-tied, helpful young man in Apple Computer's 1987 video on the Knowledge Navigator), or refer to the agent on a first-name basis (such as Sue or Bill in Hewlett-Packard's 1990 video on future computing). Microsoft's unsuccessful BOB program used cartoon characters to create onscreen partners. Others have described "knowbots," agents that traverse the World Wide Web in search of interesting information or a low price on a trip to Hawaii.

Many people are attracted to the idea of a powerful functionary carrying out their tasks and watching out for their needs. The wish to create an autonomous agent that knows people's likes and dislikes, makes proper inferences, re-spends to novel situations, and performs competently with little guidance is strong for some designers. They believe that human–human interaction is a good model for human–computer interaction and seek to create computerized partners, assistants, or agents. They promote their designs as intelligent and adaptive, and often, they pursue anthropomorphic representations of the computer to the point of having artificial faces talking to users. Anthropomorphic representations of computers have been unsuccessful in bank terminals, computer assisted instruction, talking cars, or postal service stations, but some designers believe that they can find a way to attract users. A variant of the agent scenario, which does not include an anthropomorphic realization, is that the computer employs a "user model" to guide an adaptive system. The system keeps track of user performance and adapts its behavior to suit the users' needs. For example, several proposals suggest that, as users make menu selections more rapidly, indicating proficiency, advanced menu items or a command-line interface appears. Automatic adaptations have been proposed for response time, length of messages, density of feedback, content of menus, order of menu items, type of feedback (graphic or tabular), and content of help screens. Advocates point to video games that increase the speed or number of dangers as user's progress though stages of the game. However, games are quite different from most work situations, where users have external goals and motivations to accomplish their tasks. There is much discussion of user models, but little empirical evidence of their efficacy. There are some opportunities for adaptive user models to tailor system responses, but even occasional unexpected behavior has serious negative side effects that discourage use. If adaptive systems make surprising changes, users must pause to see what has happened. Then, users may become anxious because they may not be able to predict the next change, interpret what has happened, or restore the system to the previous state. Suggestions that users could be consulted before a change is made are helpful, but such intrusions may still disrupt problem-solving processes and annoy users.

The agent metaphor is based on the design philosophy that assumes users would be attracted to "autonomous, adaptive, intelligent" systems. Designers believe that they are creating something lifelike and smart, however users may feel anxious and unable to control these systems. Success stories for advocates of adaptive systems include a few training and help systems that have been extensively studied and carefully refined to give users appropriate feedback for the errors that they make. Generalizing from these systems has proven to be more difficult than advocates hoped. The philosophical contrast is with "user-control, responsibility, and accomplishment" Designers who emphasize a direct manipulation style believe that users have a strong desire to be in control and to gain mastery over the system. Then users can accept responsibility for their actions and derive feelings of accomplishment. Historical evidence suggests that users seek comprehensible and predictable systems and shy away from complex unpredictable behavior, such as the pilots who disengage automatic piloting devices or VCR users who don't believe that they can properly program it to record a future show.

Comprehensible and predictable user interfaces should mask the underlying computational complexity, in the same way that turning on an automobile ignition is comprehensible to the user but invokes complex algorithms in the engine- control computer. These algorithms may adapt to varying engine temperatures or air pressures, but the action at the user-interface level remains unchanged. A critical issue for designers is the clear placement of responsibility for failures. Agent advocates usually avoid discussing responsibility. Their designs rarely allow for monitoring the agent's performance, and feedback to users about the current user model is often given little attention. However, most human operators recognize and accept their responsibility for the operation of the computer, and therefore designers of financial, medical, or military systems ensure that detailed feedback is provided. An alternative to agents and user models may be to expand the control-panel metaphor. Current control panels are used to set physical parameters, such as the speed of cursor blinking, rate of mouse tracking, or loudness of a speaker, and to establish personal preferences such as time, date formats, placement and format of menus, or color schemes. Some software packages allow users to set parameters such as the speed in games or the usage level as in HyperCard (from browsing to editing buttons to writing scripts and creating graphics). Users start at level 1, and can then choose when to progress to higher levels. Often, users are content remaining experts at level 1 of a complex system, rather than dealing with the uncertainties of higher levels. More elaborate control panels exist in style sheets of word processors, specification boxes of query facilities, and scheduling software that carries out processes at regular intervals or when triggered by other processes. Computer control panels, like cruise-control in automobiles and remote controllers for televisions, are designed to convey the sense of control that users seem to expect. Increasingly, complex processes are specified by direct-manipulation programming or by graphical specifications of scheduled procedures, style sheets, and templates.

4. A Specification Language for Direct-Manipulation User Interfaces

A direct-manipulation user interface presents its user with a set of visual representations of objects on a display and a repertoire of generic manipulations that can be performed on any of them. Some of these techniques were first seen in interactive graphics systems; they are now proving effective in user interfaces for applications that are not inherently graphical. With a direct manipulation interface, the user seems to operate directly on the objects in the computer instead of carrying on a dialogue about them. Instead of using a command language to describe operations on objects that are frequently invisible, the user “manipulates” objects visible on a graphic display. This ability to manipulate displayed objects has been identified as direct engagement. The displayed objects are active in the sense that they are affected by each command issued; they are not the fixed outputs of one execution of a command, frozen in time. They are also usable as inputs to subsequent commands. The ultimate success of a direct-manipulation interface also requires directness in the form of low cognitive distance, the mental effort needed to translate from the input actions and output representations to the operations and objects of the problem domain itself. The visual metaphor chosen to depict the problem domain should thus be easy for the user to translate to and from that domain, and the actions required to effect a command should be closely related to the meaning of the command in the problem domain.

4.1 Specifying a Direct Manipulation User Interface

It is useful to be able to write a specification of the user interface of a computer system before building it, because the interface designer can thereby describe and study a variety of possible user interfaces without having to code them. Such a specification should describe precisely the user-visible behavior of the interface, but should not constrain its implementation. Specification techniques for describing the user-visible behavior of conventional user interfaces without reference to implementation details are gaining currency;

most have been based on state transition diagrams or BNF (and a few on other models listed below); there are some reasons to prefer the state diagrams. If the specification language itself can be executed or compiled, it can also serve as the basis for a user-interface management system (UIMS). To be useful, a UIMS needs a convenient and understandable way for the user-interface designer to describe the desired interface. The choice of specification language is thus at the heart of the design of a UIMS. UIMSs have been built using BNF or other grammar-based specifications, state- transition-diagram-based specifications, programming-language-based specifications, frames, flow diagrams, and other models. More recently, several investigators have used an object-oriented approach. Research is also under way in describing user interfaces by example, where the interface designer is not concerned with a programming or specification language.

Although direct manipulation can make systems easy to learn and use, such user interfaces have proved more difficult to construct and specify. Direct manipulation interfaces have some important differences from other styles of interfaces and these must be understood in order to develop an appropriate specification technique for them. Although state-transition-diagram-based notations have proved effective and powerful for specifying conventional user interfaces, they must be modified to handle direct-manipulation interfaces. State diagrams tend to emphasize the modes or states of a system and the sequence of transitions from one state to another. Although direct-manipulation user interfaces initially appear to be modeless and thus unsuited to this approach, they will be shown below to have a particular, highly regular moded structure, which can be exploited in devising a specification technique for them.

4.2 Structure of a Direct Manipulation dialogue

In order to develop an appropriate specification language for direct-manipulation interfaces, it is necessary to identify the basic structure of such an interface as the user sees it. The goal of this specification method is not strictly compactness or ease of programming, but rather capturing the way the end user sees the dialogue. Many existing specification techniques could be extended in various ways to describe the unusual aspects of direct-manipulation dialogues. However, the real problem is not just to find some way to describe the user interface (since, after all, assembly language can do that job), but to find a language that captures the user's view of a direct-manipulation interface as perspicuously as possible and with as few ad hoc features and extensions to the specification technique as possible. The object is to describe the interface or dialogue between the system and its end user, as seen by that user, rather than to describe the structure of the system or its components at some other level.

First, consider what a dialogue specification should describe. Trying to capture the layout and precise appearance of the display of a direct-manipulation interface at every turn would make the top level of the dialogue specification excessively detailed and complex. Instead, the initial specification should be centered around the sequence of abstract input and output events that comprise the dialogue. The syntax of an interactive user interface-whether conventional or direct manipulation is effectively described by such a sequence of input and output events, with the specification of the meanings of the events in terms of specific input actions or display images deferred. The abstract input or output events themselves are called tokens and are then described individually in separate specifications. Information about display representation and layout is isolated there, rather than as part of the description of the syntax of the dialogue. This decomposition of direct-manipulation dialogues follows the model of general user-computer dialogues introduced by Foley and Wallace. The sequence of input and output tokens comprises the syntactic level, while the individual token descriptions comprise the lexical level. The semantic level is defined by a collection of procedures that implement the functional requirements of the system; they are invoked from the syntactic-level specification. This three level separation has been used to good effect in user-interface management systems. Separating the abstract dialogue sequence and overall display organization (syntactic) description from the precise input and output format (lexical) description is of particular importance for direct-manipulation interfaces, because such interfaces typically provide rapid and rich graphical feedback and may vary the appearance of the display considerably during a dialogue. Users may also be permitted to rearrange windows and other images arbitrarily to suit their preference.

Despite such variations, there are some more fundamental characterization of the dialogue than moment-to-moment display appearance should thus be identified and used as the foundation for a clear specification; the sequence of abstract events or tokens is proposed to provide this foundation. The issue did not arise with early user interfaces based on tele-printers or scrolling display terminals. The sequence of specific input and output events precisely determined the appearance of the display in a simple and straightforward way. Later display terminals added some special commands, such as clear screen, vertical tab, or cursor motions, which disrupted the relationship between sequence of inputs and outputs and display appearance. These have required some extensions to conventional specification techniques. With a full graphic display, however, much more complex user interfaces have been built. It is still true in principle that the sequence of input and output events completely determines the final appearance of the display, but in a far less straightforward way-a way that the user-interface specifier should not have to understand. The specification writer needs to be able to speak about the display appearance at a higher level: the sequence of input and output events. Details about graphical representations, sizes, windows, particular input/output devices, and the like can then be abstracted out of the dialogue specification. Even the choice of particular modes of user-computer communication can be isolated, since an output token can be any discrete, meaningful event in the dialogue, including, for example, an audible or tactile output. Note that building the syntax specification around the sequences of tokens does not preclude semantic-level feedback. For example, as a file icon is dragged over various directory icons, those directories (and only those) into which the user is currently permitted to move that file might be highlighted. The specification technique permits such an operation, but it divides the description of the feedback into its three appropriate aspects. The decision as to which directories should be highlighted is given in the semantic-level specification; the specification of when in the dialogue such highlighting will occur is given in the syntactic-level specification (as transitions that test the condition and call a highlight token); and the description of the highlighting operation itself is given in the lexical level specification (as the definition of the highlight token).

Consider next the basic sequence of events in a direct-manipulation dialogue. A direct-manipulation user interface resembles an interacting collection of active and/or responsive objects more than it does a single command language dialogue with the user. The display typically presents a variety of graphical objects. Users can select any of them (most often by moving a cursor). Once selected, the user can begin a dialogue about that object-adjusting a parameter, deleting or moving an object, etc. Each object thus has its own particular dialogue, which the user may activate or deactivate at any time. Further, some object dialogues remember their state between activations. For example, if the user moves the cursor to a type-in field and types a few characters, moves it somewhere else and performs other operations, and then returns to the type-in field, the dialogue within that field would be resumed with the previously entered characters and insertion point intact. As a better example, if the user had begun an operation that prompted for and required him or her to enter some additional arguments, the user could move to another screen area and do something else before returning to the first area and resuming entry of the arguments where he or she had left them. Given this structure, it is unnatural, though possible, to describe the user interface of a direct-manipulation system as a conventional dialogue by means of a syntax diagram or other such notation. Instead the user sees a multitude of small dialogues, each of which may be interrupted or resumed under the control of a simple master dialogue. Each of the individual objects on the screen thus has a particular syntax or dialogue associated with it. Each such dialogue can be suspended (typically if the user moves the cursor away) and later resumed at the point from which it was suspended. The relationship between the individual dialogues or branches of the top-level diagram is that of co-routines. So, the basic structure of a direct-manipulation interface is seen to be a collection of individual dialogues connected by an executive that activates and suspends them as co-routines. The specification technique for direct-manipulation interfaces will thus allow the individual dialogues to be specified individually and to exchange control with each other through a co-routine call mechanism.

4.3 Modes in the user interface

Many traditional user interfaces are highly moded, and this has made it convenient to specify them using state transition diagrams. Modes or states refer to the varying interpretation of a user's input. In each different mode, a user interface may give different meanings to the same input operations. Some use of modes is necessary in most user interfaces, since there are generally not enough distinct brief input operations (e.g., single keystrokes) to map into all the commands of a system. A moded user interface requires that users remember (or the system remind them) of which mode it is in at any time and which different commands or syntax rules apply to each mode. Modeless systems do not require this; the system is always in the same mode, and inputs always have the same interpretation. Direct-manipulation user interfaces appear to be modeless. Many objects are visible on the screen; and at any time the user can apply any of a standard set of commands to any object. The system is thus nearly always in the same "universal" or "top-level" mode. This is approximately true of some screen editors, but for most other direct-manipulation systems, where the visual representation contains more than one type of component, this is a misleading view. It ignores the input operation of moving the cursor to the object of interest. A clearer view suggests that such a system has many distinct modes. Moving the cursor to point to a different object is the command to cause a mode change, because once it is moved, the range of acceptable inputs i.e. reduced and the meaning of each of those inputs is determined.

This is precisely the definition of a mode change. For example, moving the cursor to a screen button, such as the "Display" buttons in the message system, should be viewed as putting the system into a mode where the meaning of the next mouse button click is determined (it displays that message) and the set of permissible inputs is circumscribed (e.g., keyboard input could be illegal or ignored). Moving the cursor somewhere else would change that mode. As shown in following 1, the top level of a typical direct-manipulation interface such as the message-system example could thus be described by a large state diagram with one top-level state and a branch (containing a cursor motion input) leading from it to each mode (marked with a "+"). Each such branch continues through one or more additional states before returning to the top-level state. There is typically no crossover between these branches. If direct-manipulation user interfaces are not really modeless, why do they appear to have the psychological advantages over moded interfaces that are usually ascribed to modeless ones? The reason is that they make the mode so apparent and easy to change that it ceases to be a stumbling block.

The mode is always clearly visible (as the location of a cursor), and it has an obvious representation (simply the echo of the same cursor location just used to enter the mode change command), in contrast to some special flag or prompt. Thus the input mode is always visible to the user. The direct-manipulation approach makes the output display (cursor location to indicate mode) and the related input command (move cursor to change mode) operate through the same visual representation (cursor location). At all times the user knows exactly how to change modes; he or she can never get stuck. It appears, then, that direct-manipulation user interfaces are highly moded, but they are much easier to use than traditional moded interfaces because of the direct way in which the modes are displayed and manipulated.

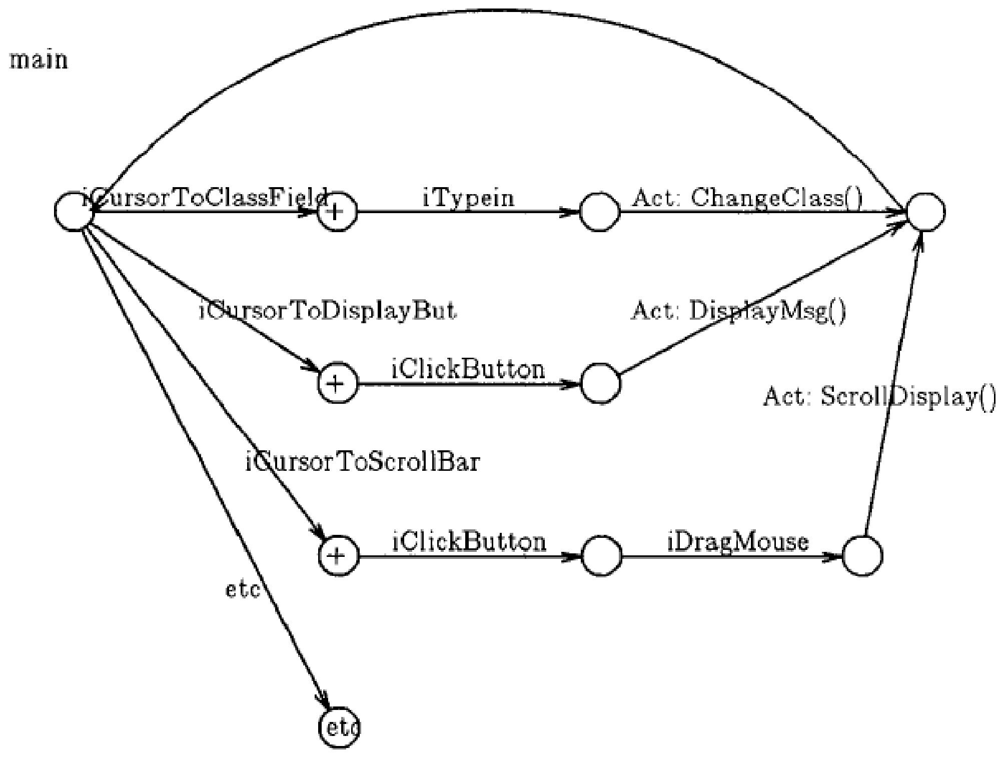


Fig. State-diagram specification of the top level of a simple direct-manipulation user interface.

4.4 A SPECIFICATION LANGUAGE

Figure above shows a typical direct-manipulation user interface represented as a state transition diagram. Although a simple direct-manipulation interface could be specified in this fashion, it has some shortcomings. The top-level state diagram for each new direct-manipulation interface will be a large, regular, and relatively uninformative diagram with one start state and a self-contained (i.e., no crossover) path to each mode and thence back to start state. It is essentially the same for any direct-manipulation system and need not be specified anew for each system. Moreover, since the individual paths are usually self-contained and interact with each other in very limited ways, it would be clearer to separate their specifications. A more serious problem with this approach is that there is often a remembered state within some of the paths (partial type-in on a field, an item awaiting confirmation, etc.), which are suspended when the cursor leaves the field and resumed when it reenters. This requires that the paths of the diagram be handled separately. Each path will thus now be specified separately (as a co-routine), and an executive will be given for the outer dialogue loop. A specification language based on the characteristics found in the foregoing examination of direct-manipulation interfaces can now be described:-A direct-manipulation interface was found to comprise a collection of many relatively simple individual dialogues. Thus the specification will be centered around a collection of individual objects, called interaction objects, each of which will have a separate specification. Each of the dialogues of the direct-manipulation interface will be specified as a separate interaction object with an independent dialogue description. The individual dialogues of a direct-manipulation interface were found to be related to each other as a set of co-routines. Thus the specification language will permit the dialogue associated with each interaction object to be suspended and resumed, with retained state, like a co-routine. A simple executive will be defined to manage the overall flow of control. It specifies the interconnection of the interaction object dialogues, allocates input events, and suspends the individual dialogues to relinquish control to others as needed. Because of the complexity and variability in the layout of the display of a direct-manipulation interface, it was found that the dialogue should be specified as a sequence of abstract input and output events, with layout and graphic details given separately. Thus the dialogue specification for each interaction object will be written using input and output tokens, which represent input or output events. The dialogue specification will define the possible sequences of input and output tokens. The internals of the tokens themselves will then be specified separately from the dialogue. These token definitions will contain details of layout, graphical representation, and device handling.

Direct-manipulation interfaces were seen to have definite modes or states, despite their surface appearance. This applied both to the overall structure and to the retained state within each co-routine. Thus state transition diagrams are a suitable notation for describing the individual interaction-object dialogues. The state diagrams will assume co-routine calling between them. Given this structure, a direct-manipulation user interface will be specified as a collection of individual, possibly mutually interacting interaction objects, organized around the manipulable objects and the loci of remembered state in the dialogue. These objects will often coincide with screen regions or windows, but need not. A typical object might be a screen button, individual type-in field, scroll bar, or the like. ach such object will be specified separately, and then a standard executive will be defined for the outer dialogue loop. Thus, to describe a direct-manipulation user interface, it will be necessary to

(1) Define a collection of interaction objects,
(2) Specify their internal behaviors, and
(3) Provide a mechanism for combining them into a coordinated user interface.

As noted, a goal of this notation is to capture the way the end user sees the interface. The underlying claim is thus that the user indeed sees the direct manipulation dialogue as a collection of small, individual objects or dialogues, each suspendable and resumable like a co-routine, joined by a straightforward executive. The specification language is defined by devising a mechanism for each of the three tasks in the preceding paragraph:

1. How should the user interface be divided into individual objects? An interaction object will be the smallest unit with which the user conducts a meaningful, step-by-step dialogue, that is, one that has continuity or syntax. It can be viewed as the smallest unit in the user interface that has a state that is remembered when the dialogue associated with it is interrupted and resumed. In that respect, it is like a window, but in a direct-manipulation user interface, it is generally smaller-a screen button, a single type-in field on a form, or a command line area. It can also be viewed as the largest unit of the user interface over which disparate input events should be serialized and combined into a single stream, rather than divided up and distributed to separate objects. Thus an interaction object is a locus both of maintained state and of input serialization.

2. How should an input handler for each interaction object be specified? Observe that, at the level of individual objects, each such object conducts only a singlethread dialogue, with all inputs serialized and with a remembered state whenever the individual dialogue is interrupted by that of another interaction object. Thus a conventional single-thread state diagram is the appropriate representation for the dialogue associated with an individual interaction object. The input handler for each interaction object is specified as a simple state transition diagram

3. How should the specifications of the individual objects be combined into an “outer loop” or overall direct-manipulation user interface? As noted, a direct manipulation interface could be described with a single, large state diagram, but since the user sees the structure of the user interface as a collection of many semi-independent objects, that is not a particularly perspicuous description. Instead, a standard executive will be defined that embodies the basic structure of a direct-manipulation dialogue and includes the ability to make co-routine calls between individual state diagrams. This executive operates by collecting all of the state diagrams of the individual interaction objects and executing them as a collection of co-routines, assigning input

events to them and arbitrating among them as they proceed. To do this, a co-routine call mechanism for activating state diagrams must be defined. This means that whenever a diagram is suspended by a co-routine call to another diagram, the state in the suspended diagram is remembered. Whenever a diagram is resumed by a co-routine call, it will begin executing at the state from which it was last suspended. The executive causes the state diagram of exactly one of the interaction objects to be active at any one time. As the active diagram proceeds, it reaches each state, examines the next input event, and takes the appropriate transition from that state. It continues in this way until it reaches a state from which no outgoing transition matches the current input. Then, the executive takes over, suspending the current diagram, but remembering its state for later resumption. (It follows that a diagram can only be suspended from a state in which it seeks an input token.) The executive examines the diagrams associated with all the other interaction objects, looking at their current (i.e., last suspended from) states to see which of them can accept the current input. It then resumes (with a co-routine call) whichever diagram has a transition to accept the input. If there is more than one such diagram, one is chosen arbitrarily. In typical designs, however, there will be only one diagram that can accept the input. Since entering and exiting disjoint screen regions will be important input tokens in a typical direct-manipulation interface, this is straightforward to arrange when the interaction objects correspond to screen regions. (In some situations, such conflicts can also be detected by static analysis of the interface specification.) Depending on the overall system design, an input token acceptable to no diagrams could be discarded or treated as a user error. While the language assumes a single top-level executive, the use of component objects and synthetic tokens described below allows the specification to use a deeper hierarchy in describing systems.

The initial design for the executive called for a list of acceptable input events or classes to be associated with each state in each diagram. This list would act like a guard in a guarded command or a when clause in a select/accept statement in Ada. By associating different guards with different states, a diagram could dynamically adjust the range of inputs that it will accept. The executive for such a system would examine the guard associated with the current state of every diagram in execution to decide which diagram should be called to accept each new input. The current design should be viewed as achieving the same result, even though it does not identify the guards explicitly. What would have been given as the guard for each state is now derived implicitly from the range of inputs on the transitions emanating from that state. This requires somewhat more care in specifying “catchall” transitions, but greatly reduces the redundancy and bulk of the specification. The new specification language also makes heavy use of techniques of object oriented programming. The interaction objects themselves are specified and implemented as objects, in the sense of Smalltalk or Flavors, and diagram activations and tokens are implemented as messages. The notion of co-routines, however, is superimposed upon the objects as the means for describing how the individual interaction objects are bound together into the top-level dialogue that the user ultimately sees. Other recent work on specifying and building graphical user interfaces has also used an object-oriented approach. Typically, they model the dialogue by a collection of separate objects, each with an input handler. However, they have not proposed that the input handlers explicitly specify their state-dependent responses by means of state transition diagrams or that they retain their states during execution by co-routine activation. Cardelli and Pike achieved a similar result using communicating finite-state machines with actual concurrency. The use of co-routines in the present language, combined with the synthetic tokens described below, can also be mapped into the abstract device model introduced by Anson, but that, too, does not use state diagrams to describe the state and behavior of the abstract devices. Anson points out the weakness of a single-thread state diagram for describing direct-manipulation interfaces: “It cannot simulate a device . . . which retains its value between uses and which can be changed by the user at any time”. The present technique attempts to remedy this problem without giving up the benefits of state diagrams for depicting device state and state-dependent behavior.

5. A Direct Manipulation Interface for 3D Computer Animation

Computer animation is a painstaking process requiring hand adjustment of hundreds of key positions for every object in an animated scene. Most animation systems provide precise control of motion using two-dimensional graphs of individual parameters (e.g. x translation vs. time). Animators must mentally integrate this 2D information with static 3D views and occasional motion previews to maintain a clear sense of the motion which they are creating. The principles of direct manipulation are used to achieve the goal of fluid and natural interaction. The solution uses existing key frame and parametric techniques in combination with *displacement functions* inspired by digital signal processing for real-time direct manipulation of spatial and temporal changes

5.1 Problems in Existing Animation Systems

Several problems found in a majority of commercial and research animation systems. Not all of these problems are present in all systems, but these are current trends in a large class of existing systems.

a] Animators can completely visualize and edit motion only in separate 2D graphs. The only means to edit an object’s time-varying properties and visualize the value of these properties over time is through 2D graph editors. The 3D scene view is used primarily for viewing and editing an object at a single point in time.

b] Editing of motion curves is limited to single channels of motion.
Motion curves are normally limited to representing a one-dimensional parameter vs. time (e.g. x translation vs. time, y rotation vs. time, red color component vs. time). Animators must mentally integrate all of these channels to visualize the animation which they are creating.

c] The natural parameterization of splines does not advance uniformly with respect to distance. Many systems allow the animator to specify the path of an object through space with a two- or three-dimensional spline curve. Motion along this curve is then described by a single function of u vs. time, where u is the parameter of the spline curve. However, equal steps in u result in unequal distances traveled along the curve. In these systems,
a graph that appears to indicate constant velocity will actually result in a velocity that varies based on the shape of the curve and the spacing of its control points. The animator is forced to cancel out the timing induced by the spline before creating the desired motion.

d] The shape of a motion curve is altered to achieve timing goals.
Some systems alter the shape of a motion path when users edit the timing of an animation. This problem is also a result of tying motion to the u-parameter of a spline. The actual shape of the curve must be changed in order to alter the distance travelled over equal time steps.

e] Direct manipulation of the animated object is allowed only at control points.
When a spline curve is used as the underlying representation of spatial change,most systems only allow the animator to change the object at the spline control points [2][14]. If the animator wants to alter a position between control points, shemust either work indirectly, altering surrounding control points and tangents, or shemust add a new control
point. Adding control points can introduce undesired complexity to the animation and reduces the range over which changes have effect.

f] Animations with densely spaced keyframes are difficult to modify.
Most production quality animations end up being specifiedby very densely pacedkeyframes (10-15 keyframes/second is normal). If an animator decides that part of the motion should be changed, she must individually change a wide range of control points surrounding the specific change in order to blend it with the surrounding motion—there are no tools for modifying multiple keyframes simultaneously. Many animators find it faster to re-do the animation from scratch in this situation

5.2 Goals for Animation Control

The following set of goals is an attempt to describe an animation system which addresses the above set of problems:

1. Create an system which allows visualization and editing of temporal and spatial information in a single 3D view.

2. Express motion goals in terms of distance or velocity vs. time.

3. Maintain temporal and spatial continuity while editing animations.

4. Allow an arbitrary range over which editing tools are applied.

5. Develop motion control techniques which are natually extensible to orientation, scale and any other animated parameters.

6. Provide real-time performance for complex scenes.As an interface to the above goals, we require direct-manipulation tools which correspond to the high-level goals of an animator.

7 *Temporal translation*
Satisfies the goal "Reach this point at this time" while maintaining the shape of the motion path, but changing the speed at which the object travels along the given path.

8. *Spatial translation*
Satisfies the above goal by modifying the spatial curve while maintaining either the duration or velocity of the given segment.

9. Temporal scale
Changes the duration of segment of animation. Satisfies the goal "Make this segment of animation longer, shorter, or a specific duration"

10. *Velocity modification*
Satisfies the goals "Go faster", "Go slower", or "Reach a specific velocity" at a given point, while maintaining the shape of the spatial curve and the duration of the temporal segment.

6. Pick-and-Drop: A Direct Manipulation Technique for Multiple Computer Environments

In a ubiquitous computing (UbiComp) environment, we no longer use a single computer to perform tasks. Instead, many of our daily activities including discussion, documentation, and meetings will be supported by the combination of many (and often different kinds of) computers. Combinations of computers will be quite dynamic and heterogeneous; one may use a personal

digital assistant (PDA) as a remote commander for a wall-sized computer in a presentation room, others might want to use two computers on the same desktop for development tasks, or two people in a meeting room might want to exchange information on their PDAs. Other than the UbiComp vision, we often use multiple computers for more practical reasons; PCs, UNIXs, and Macs have their own advantages and disadvantages, and users have to switch between these computers to take full advantage of each (e.g., writing a program on a UNIX while editing a diagram on a Mac).
However, using multiple computers without considering the user-interface introduces several problems. The first problem resides in a restriction of today's input devices. Almost all keyboards and pointing devices are tethered to a single computer; we cannot share a mouse between two computers. Therefore, using multiple computers on the same desk top often results in a "mouse (or keyboard) jungle", as shown in the figure below. It is very confusing to distinguish which input device belongs to which computer.

The other problem is the fact that today's user interface techniques are not designed for multiple-computer environments. Oddly enough, as compared with remote file transmission, it is rather cumbersome to transfer information from one computer to another on the same desk, even though they are connected by a network. A cut-and-paste on a single computer is easy, but the system often forces users to transfer information between computers in a very different way. A quick survey reveals that people transfer information from display to display quite regularly. Interestingly, quite a few people even prefer to transfer data *by hand* (e.g., read a text string on one display and type it on another computer), especially for short text segments such as an e-mail address or a universal resource locator (URL) for the World Wide Web. These tendencies are caused by a lack of easy direct data transfer user interfaces (e.g., copy and- paste or drag-and-drop) between different but nearby computers.

The first problem is partially solved by using more sophisticated input devices such as a stylus. Today's stylus input devices such as WACOM's, provide untethered operation and thus can be shared among many pen sensitive displays. This situation is more natural than that of a mouse, because in the physical world, we do not have to select a specific pencil for each paper. With the second problem, however, we have much room for improvement from the viewpoint of user interfaces. Although some systems use multi-display configurations, direct manipulation techniques for multi-display environments have not been well explored to date. Multi-display direct manipulation offers many new design challenges to the field of human-computer interfaces.

A new pen based interaction technique called "Pick-and-Drop" lets a user exchange information from one display to another in the manner of manipulating a physical object. This technique is a natural extension to the drag-and-drop technique, which is popular in today's many GUI applications.

The figure below shows the conceptual difference between the traditional data transfer method and Pick-and-Drop.

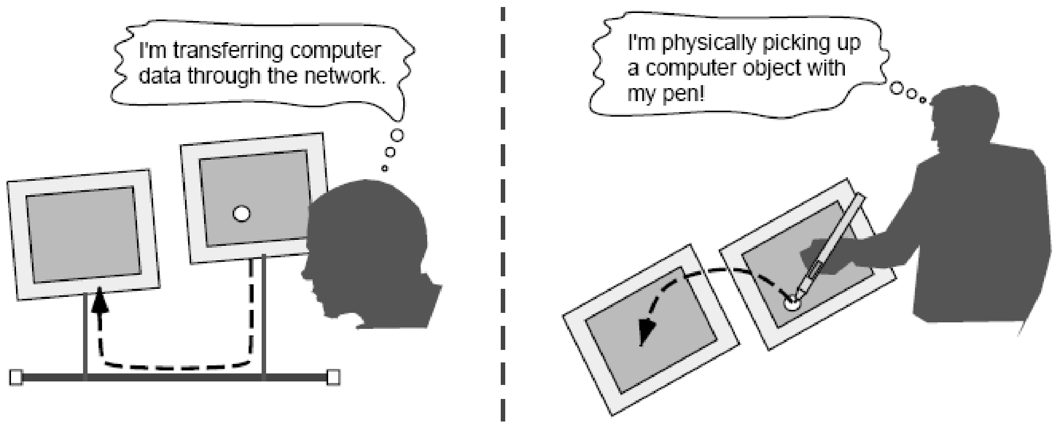
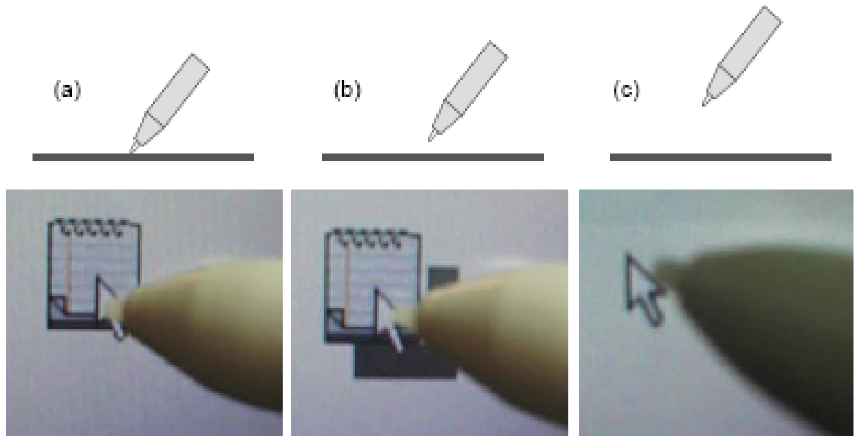

I'm transferring computer data through the network.
I'm physically picking up a computer object with my pen!

fig. The conceptual difference between remote copy and Pick-and-Drop

6.1 From Drag-And-Drop to Pick-And-Drop

Pick-and-Drop is a direct manipulation technique that is an extrapolation of drag-and-drop, a commonly used interaction technique for moving computer objects (e.g., an icon) by a mouse or other pointing devices. With the traditional drag-and-drop technique, a user first “grabs” an object by pressing a mouse button on it, then “drags” it towards a desired position on the screen with the mouse button depressed, and “drops” it on that location by releasing the button. This technique is highly suitable for a mouse and widely used in today’s graphical applications. However, simply applying the drag-and-drop to pen user interfaces presents a problem. It is rather difficult to drag an object with a pen while keep the pen tip contacted on the display surface. It is often the case that a user accidentally drops an object during the drag operation, especially when dragging over a large display surface.

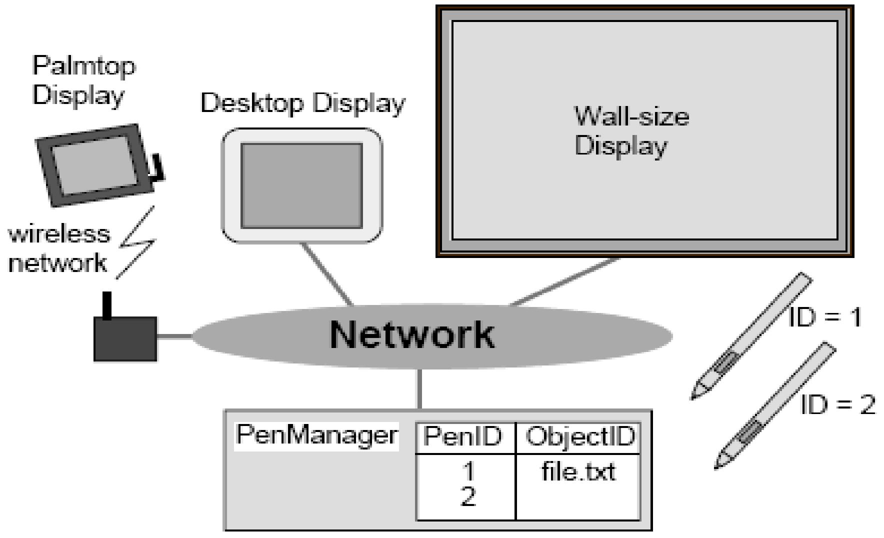
Pick-and-Drop method started as useful alternative to drag-and-drop for overcoming this problem. With Pick-and-Drop, the user first *picks up* a computer object by tapping it with the pen tip and then lifts the pen from the screen. After this operation, the pen virtually *holds* the object. Then, the user moves the pen tip towards the designated position on the screen without contacting display surface. When the pen tip comes close enough to the screen, a shadow of the object appears on the screen as show in the figure below as a visual feedback showing that the pen has the data. Then, the user taps the screen with the pen and the object moves from the pen to the screen at the tapped position. This method looks much more natural than that of drag-and-drop. In our real lives, we regularly pick up an object from one place and drop it on another place, rather than sliding it along the surface of something.



Pen and icons: (a) the pen contacts the display, (b) the pen lifts up but remains close to the screen, (c) the pen is away from the screen

6.2 Pen-IDs

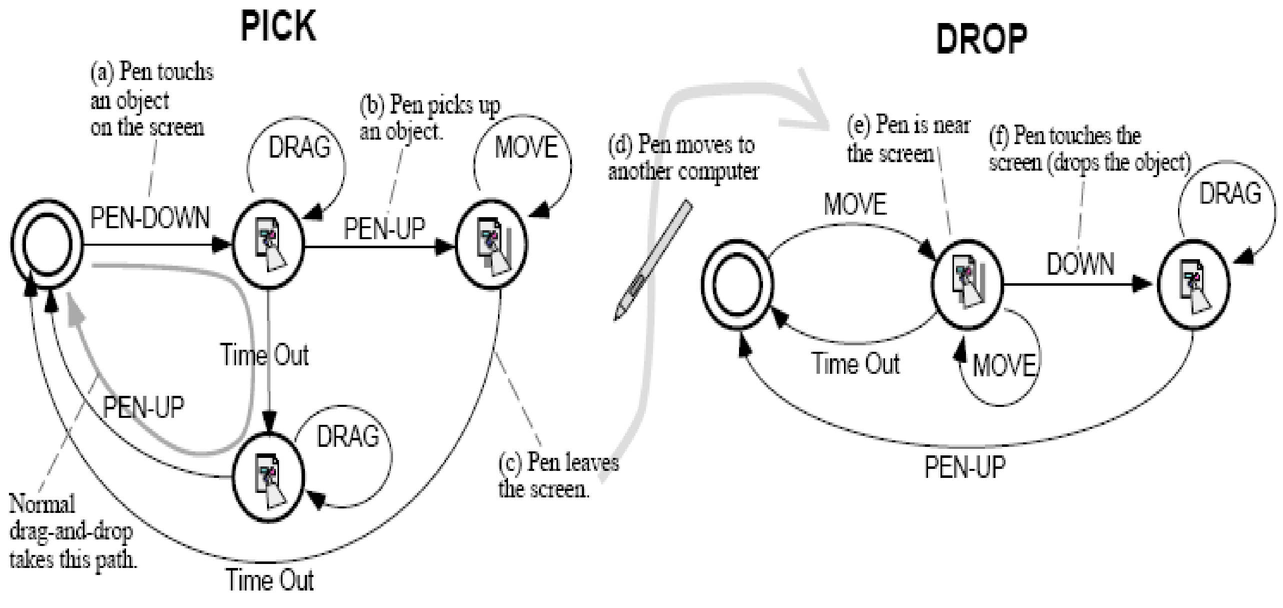
Storing data on a pen, however, makes the pen device heavy and unwieldy. The multi-computer Pick-and-Drop is developed without making such modifications to the pen by introducing the concept of *Pen IDs*. In this design, each pen is assigned a unique ID. This ID is readable from the computer when a pen is closer enough to its screen. There are currently combination of modifier buttons (attached to the pen as a side switch) to represent IDs. All computers must be connected to the network (either wired or wireless). There is a server called the "pen manager" on the network as show in the figure below.



System configuration

When a user taps an object (typically an icon) on the screen with the pen, the pen manager binds its object ID to the pen ID. This binding represents a situation in which the pen *virtually* holds the object (even though the pen itself does not contain any storage). When the user moves the same pen towards the other display, the pen manager supplies the type of the bound object to the display. Then the shadow of the data appears on the display below the current pen position. At this moment, the pen does not touch the screen.
Finally, when the user touches the display with the pen, the pen manager asks the first computer to transfer the data to second computer.
Since each pen has its own ID, simultaneous Pick-and-Drop operations by more than one pen can overlap. This feature would be useful in a collaborative setting. Pick-and-Drop can also coexist with the normal drag-and-drop by using a time-out. The system distinguishes between these two operations by measuring the period of time between pen-down and pen-up. When a user touches an object with the pen and drags it without lifting the pen tip, it initiates a drag-and-drop instead of a Pick-and-Drop.

The state transition of Pick-and-Drop is shown below.


PICK
DROP
(a) Pen touchs an object on the screen
(b) Pen picks up an object.
(c) Pen leaves the screen.
(d) Pen moves to another computer
(e) Pen is near the screen
(f) Pen touches the screen (drops the object)
PEN-DOWN
DRAG
PEN-UP
MOVE
Time Out
PEN-UP
DRAG
Normal drag-and-drop takes this path.
Time Out
MOVE
Time Out
MOVE
DOWN
DRAG
PEN-UP

The state transition diagrams of Pick-and-Drop

6.3 Object Shadows

When a pen holding data approaches a screen, a shadowed object appears on the screen to indicate that the pen has the data. This visual feedback is useful to know what kind of data the pen is holding without having to drop it. A pen's proximity to the screen can be sensed by combining the motion event and a time-out. When a user moves a pen close to the screen, the screen begins reading motion events from the pen. If motion events occur continuously, the system regards the pen as being near the screen. When a pen leaves the screen, motion events seize and the system can detect it again by setting a time-out. This technique is used for both the Pick and the Drop operations.

Example Applications
Since Pick-and-Drop is a natural extension to drag-and-drop, which is a commonly used direct manipulation technique, we should be able to apply this technique to various situations in many user interface systems. The following are some experimental applications that have identified.

6.4 Information Exchange between PDAs and Kiosk Terminals

The simplest usage of Pick-and-Drop is to support the exchange of information between two co-workers. When two people need to transfer a file or a short text segment between computers, they can simply pick it up from one's PDA display and drop it on the other's display as shown in the figure below. These two PDAs are communicating via wireless networks. It is also possible to pick up information from a kiosk terminal in a public space or an office. The terminals are installed at public spaces in the laboratory such as the coffee corner, and continuously display information. Pick-and-Drop capability to this system enables people to pick up URL information from the terminal and drop it to his/her PDA.

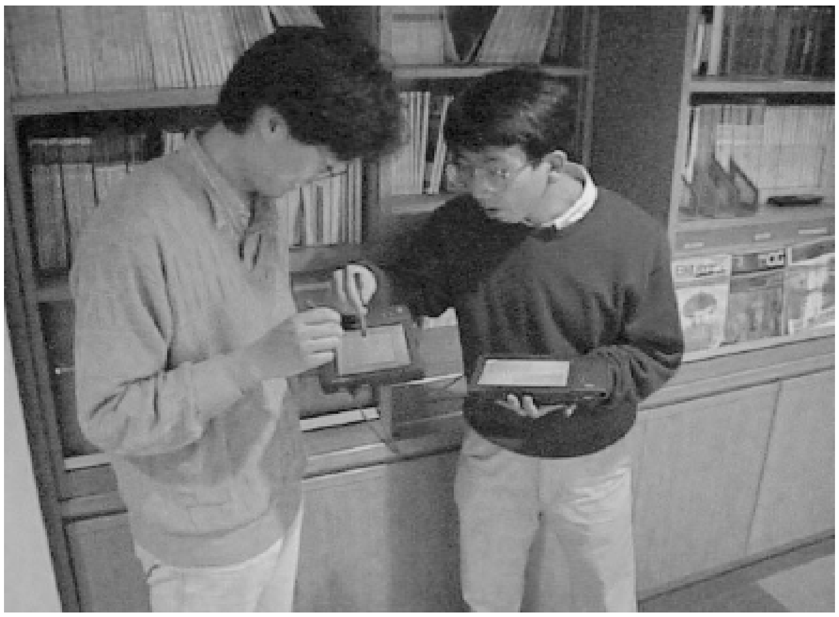


fig. Information exchange between PDAs

6.5 Picking up Paper Icons

Another possible way to extend the concept of multi display user interfaces is to support information exchange between computers and non-computer objects. For example, it would be convenient if we could freely pick up printed icons on a paper document and drop it on the computer screen.
This prototype system called *Paper-Icons* allows Pick-and-Drop between a paper object and a computer display as show in the figure below. The user can pick up an object from a printed page and drop it on a display. The page is placed on a pen sensitive tablet and a camera is mounted over the tablet. The camera is used to identify the opened page by reading an ID mark printed on it. The user can freely flip through the booklet to find a desirable icon. The system determines which icon is picked based on the page ID and the picked position on the tablet.

The Paper-Icons style is quite suitable for selecting “clip art” or “color samples” from a physical book. If the user is accustomed to a frequently used book, he/she can flip through pages very quickly by feeling the thickness of the book.

7. Problems with Direct Manipulation

Direct manipulation systems have both virtues and vices. For instance, the immediacy of feedback and the natural translation of intentions to actions make some tasks easy. The matching of levels of thought to the interface language - semantic directness - increases the ease and power of performing some activities at a potential cost of generality and flexibility. But not all things should be done directly. For example, a repetitive operation is probably best done via a script, that is, through a symbolic description of the tasks that are to be accomplished.
Some problems that can be identified in Direct Manipulation interfaces are as follows:

1. Direct manipulation interfaces have difficulty handling variables, or distinguishing the depiction of an individual element from a representation of a set or class of elements.

2. Direct manipulation interfaces have problems with accuracy, for the notion of mimetic action puts the responsibility on the user to control actions with precision, a responsibility that is sometimes best handled through the intelligence of the system and sometimes best communicated symbolically.

3. A more fundamental problem with direct manipulation interfaces arises from the fact that much of the appeal and power of this form of interface comes from

its ability to directly support the way we normally think about a domain. A direct manipulation interface amplifies our knowledge of the domain and allows us to think in the familiar terms of the application domain rather than in those of the medium of computation. But if we restrict ourselves to only building an interface that allows us to do things we can already do and to think in ways we already think, we will miss the most exciting potential of new technol*ogy:* to provide new ways to think of and to interact with a domain. Providing these new ways and creating conditions that will make them feel direct and natural is an important challenge to the interface designer.

4. Direct manipulation interfaces are not a panacea. Although with sufficient practice by the user many interfaces can come to feel direct, a properly designed interface, one which exploits semantic and articulatory directness, should decrease the amount of learning required and provide a natural mapping to the task. But interface design is subject to many tradeoffs. There are surely instances when one might wisely trade off directness for generality, or for more facile ways of saying abstract things. The articulatory directness involved in pointing at objects might need to be traded off against the difficulties of moving the hands between input devices or of problems in pointing with great precision.

5. It is important not to equate directness with ease of use. Indeed, if the interface is really invisible, then the difficulties within the task domain get transferred directly into difficulties for the user. Suppose the user struggles to formulate an intention because of lack of knowledge of the task domain. The user may complain that the system is difficult to use. But the difficulty is in the task domain, not in the interface language. Direct manipulation interfaces do not pretend to assist in overcoming problems that result from poor understanding of the task domain.

6. Certain kinds of abstraction that are easy to deal with in language seem difficult in a concrete model of a task domain. When we give up the conversation metaphor, we also give up dealing in descriptions, and in some contexts, there is great power in descriptions. As an interface to a programming task, direct manipulation interfaces are problematic. We know of no really useful direct manipulation programming environments. Issues such as controlling the scope of variable bindings promise to be quite tricky in the direct manipulation environments. Basically, the systems will be good and powerful for some purposes, poor and weak for others. In the end, many things done today will be replaced by direct manipulation systems. But we will still have conventional programming languages.

7. On the surface, the fundamental idea of a direct manipulation interface to a task flies in the face of two thousand years of development of abstract formalisms as a means of understanding and controlling the world. Until very recently, the use of computers has been an activity squarely in that tradition.

So the exterior of direct manipulation, providing as it does for the direct control of a specific task world, seems somehow atavistic, a return to concrete thinking. On the inside, of course, the implementation of direct manipulation system is yet another step in that long, formal tradition. The illusion of the absolutely manipulable concrete world is made possible by the technology of abstraction.

8. Future? After Direct Manipulation—Direct Sonification

Direct Sonification interface allows musicologists to browse musical data sets in novel ways. The data set (in the users' language often called a *collection*) is used by musicologists in their research. It contains over 7000 tunes, where each tune is represented by its score and a number of properties, such as tonality and structure. The traditional format for a collection is a printed book with various indexes. A common problem that musicologists have to deal with is to determine if tunes they collect in their field work exist in a particular collection and, if so, how they are related to other tunes in the collection, e.g., in chronology, typology.

8.1 Browsing

Browsing has become a popular term in recent years with the emergence of hypertext systems and the World Wide Web, but the concept of browsing goes well beyond these fields of application. There are many ways integrating text, sound, images, and video to provide richer and more interesting systems that would allow us to use more of our natural abilities. Marchionini and Shneiderman [1988] defined browsing as:

1. "an exploratory, information seeking strategy that depends upon serendipity"
2. "especially appropriate for ill-defined problems and for exploring new task domains"

This is the case when musicologists are searching for tunes in a collection. Tunes collected through fieldwork can often be different from older original versions. They can still be the same tunes but with the addition of an individual performer's style. This makes it difficult to use normal computer-based search algorithms [´O Maid´ın 1995]. Humans have an outstanding ability to recognize similarities in this domain, which suggests that in a good solution we should make use of our auditory abilities.

8.2 Browsing with Sound Support

In everyday listening, one is often exposed to hundreds of different sounds simultaneously and is still able to pick out important parts of the auditory scene. With musical sounds, or tunes, many different factors affect our ability to differentiate and select between the sources. Using instrumental sounds, the timbre, envelope, tonal range, and spatial cues support the formation of *auditory streams*. The tunes themselves also assist the formation of streams, as music has its own inherent syntactic and semantic properties. It is also interesting to note the “cocktail party” effect, i.e., that it is possible to switch one’s attention at will between sounds or tunes. Albers and Bergman added sounds to a web browser, but kept the use of sound at a fairly low level of interactivity. Various “clicks” were used when users clicked *soft buttons* and selected menus. To indicate system events, such as data transfer, launch of “plug-ins” and, for errors, he used “pops and clicks,” sliding sounds, and breaking of glass sounds. For feedback about content, various auditory icons were used to indicate what kind of file a hyperlink was pointing to and the file size of the content indicated by piano notes (activated when the cursor was on a hyperlink). He also created hybrid systems using combinations of auditory icons, auralization, and sound spatialization to enhance operator performance in mission control work settings.

LoPresti & Harris’ *loudSPIRE* system added auditory display to a visualization system. This system is an interesting hybrid as it used at three different layers for sonification. System events were represented by electronic-sounding tones associated with computers; data set objects were represented by percussive or atonal auditory icons parameterized for object properties; domain attributes were represented by themes of orchestral music, harmonious tonal sounds, and parameterized for attribute value of a region.
Begault [1994] demonstrated the use of 3D sound spatialization for use in cockpits and mission control, in order to enhance speech perception. Kobayashi and Schmandt showed that multiplestream speech perception can be enhanced through 3D sound spatialization, including the existence of a spatial/temporal relation for recall of position within a sample of streamed speech, i.e., that the auditory content can be mapped to spatial memory.

With multiple auditory streams it is interesting to note the problem with differences in the individual ability to differentiate between multiple sound sources. A metaphor for a user controllable function that makes it visible to the user is the application of an *aura* [Benford and Greenhalgh 1997], which in this context, is a function that indicates the user’s range of interest in a domain. The aura is the receiver of information in the domain.

8.3 Sonic Software

Normal multimedia PCs cannot play multiple sound files concurrently. This would, of course, prohibit the desired development. To work around this problem, new intermediate drivers for the sound devices were developed. The problem with existing drivers is that when a sound is to be played, the operating system allocates the physical sound device exclusively. To solve this problem, the intermediate drivers have to read sound files and transform them into a common output format. Sound spatialization was implemented to assist the users in differentiating and locating tunes.With sampled sounds, 3D spatialization can be used, but currently there is no existing support for 3D spatialization of MIDI synthesizer sounds on PC sound cards. Only stereophonic *"pan"* with difference in loudness between the left and right channel is available on standard sound cards [CreativeLabs 1996; Microsoft 1996]. The problem with different speeds and formats of source files applies to both sound files (such as, WAV) and sound-controlling files (such as, MIDI). As the users had expressed a preference for melody lines with MIDI-controlled synthesizer sounds, all further implementation work focused on stereophonic spatialization with only the difference in loudness between the left and right channel as a cue for auditory spatial location.

The users found that they sometimes wanted the aura on, sometimes off, as this allowed them to shift their focus between the neighborhoods of tunes to finer differentiation between just a few tunes. The number of tunes within the aura can vary due to the location of the cursor in relation to the density of the data set. Therefore an on–off function was added and the radius of the aura was made user controllable. The interfaces in many standard applications from some of the larger software developers have become overloaded and complicated in the interaction sequences. Through a simplified interaction sequence, users can work efficiently and with a high degree of satisfaction. The results also show that through tight coupling of the interaction, we can create a more engaging interface. By shifting some of the load from the visual to the auditory modality, we can perceive more information and make better use of our natural ability to recognize complex and "fuzzy" patterns through seeing and hearing.

Audibility is the concept of how well a system can use auditory representation in the human–computer interaction. If the audibility is good, the users will perform their work better, faster, with fewer errors, and a higher degree of satisfaction. If the use of sound in the user interface can provide more affordances, or affordances that are complementary to the visual interface, we have a system with good audibility. This is also important for users with different abilities. By using sonic representations (or auditory display) in the human–computer interaction, the resulting applications will potentially be usable to visually impaired people.

Further investigations in perception and cognition at high levels of environmental complexity are required. Many guidelines are based on extremely isolated experiments. Hence, it is difficult to apply such guidelines in real-work settings. To get more realistic models for what we, as human beings, can process, combinations of seeing, hearing, and interaction should be studied.

9. Conclusion

Direct manipulation and its descendants are thriving. Visual overviews accompanied by user interfaces that permit zooming, filtering, extraction, viewing relations, history keeping and details on-demand can provide users with appealing and powerful environments to accomplish their tasks. Most users want comprehensible, predictable and controllable interfaces that give them the feeling of accomplishment and responsibility. Direct Manipulation can helps users to give such interfaces.

10. References

1] Edwin L. Hutchins, James D. Hollan, and Donald A. Norman “Direct Manipulation Interfaces” HUMAN-COMPUTER INTERACTION, 1985, Volume 1, pp. 311-338

2] Ben Shneiderman “Direct Manipulation for Comprehensible, Predictable and Controllable User Interfaces”

3] Jun Rekimoto “Pick-and-Drop: A Direct Manipulation Technique for Multiple Computer Environments”

4] ROBERT J. K. JACOB “A Specification Language for Direct-Manipulation User Interfaces” ACM Transactions on Graphics, Vol. 5, No. 4, October 1986, Pages 283-317.

5] Ben Shneiderman “Direct Manipulation Vs Interface Agents” Interactions, November-December 1997

6] Mikael Fernstrom and Caolan MCNamara “After Direct Manipulation—Direct Sonification” ACM Transactions on Applied Perception, Vol. 2, No. 4, October 2005, Pages 495–499.

7] Scott Sona Snibbey “A Direct Manipulation Interface for 3D Computer Animation”

8] Robert St. Amant and Thomas E. Horton “Tool-based direct manipulation environments”

9] Francois Guimbretiere, Andrew Martin and Terry Winograd “Benefits of Merging Command Selection and Direct Manipulation” ACM Transactions on Computer-Human Interaction, Vol. 12, No. 3, September 2005, Pages 460–476.

10] Wolfgang Preea, Gustav Pornbergera, Hermann Sikorab “Construction Techniques of Graphic, Direct-Manipulation User Interfaces” EUROGRAPHICS ’91

11] Hao-wei Hsieh and Frank M. Shipman III “VITE: A Visual Interface Supporting the Direct Manipulation of Structured Data Using Two-Way Mappings”