Programming Facilities for User Modification of Solid Modeling Systems

Gregory John Glass
Institute of Building Science
Carnegie-Mellon University

Abstract: A proposal is presented for reducing the disparity between the types of operations provided in most solid modeling systems and the operations available in the real world for the elements that are being modeled. The solution centers around the use of a "programmable" user interface that will permit the definition of higher level meta commands. These meta commands should tailor the actions available in a system to conform to the practices of a specific profession such as mechanical engineering or architecture. They may also provide a set of basic primitive parts that is appropriate for the needs of a specific group of users. Knowledgeable individual users will also be able to customize their working environment. The facilities for the interactive definition and use of these meta command are outlined along with an example of how they may be used.

Sujet:une approche est presentee pour reduire la disparite entre les operations disponibles sur la plupart des systemes et les operations reellement effectuees par l'utilisateur sur les objets representes. La principale idee est l'utilisation d'un interface programmable qui permet la definition de macro-commandes d'un plus haut niveau. Ces macro-commandes doivent adapter les operations fournies par le systeme aux besoins specifiques de professions comme la Mecanique ou l'Architecture. Elles doivent aussi fournir un ensemble de primitives communes pour les besoins d'un groupe d'utilisateurs. Les utilisateurs individuels potentiels doivent aussi pouvoir adapter leur cadre de travail. Les possibilites de definition interactive et d'utilisation de ces commandes seront eclairees par des exemples.

1 Introduction

Most solid modeling systems are organized to deal effectively with the characteristics of solids in general, but most design disciplines have specific methods that are based on knowledge of the nature of the types of solids that are manipulated. For example if a particular rectangular solid represents a wall the architect can restrict the types of compositions that may be generated using that rectangular solid to those that are possible with walls. Restriction of the possibilities that may be modeled during the design of an artifact makes the design task faster [2]. In order for a design system to speed the task of the designer it must take advantage of these practices. Also the designers conceptualization is cleaner because he is modeling not only the physical geometry of the design elements but also the manipulation operation are models of that which may take place on the original artifact.

In (Arbab *et al.* [1]) a related idea was proposed which consisted of a user interface that restricted the design operations to those that may take place during the manufacturing process. This might be fine for the design of machined parts but for some design products this places an unreal restriction on the design process because it would force the designer to develop the design in an opposite order from that which is required to come up with a successful solution. For example when designing a ship: first the accommodations, the holds and the machinery spaces must be laid out in order to determine their overall size. Then the hull and the keel may be designed. But in manufacturing the ship the first thing that is laid down is the keel. It would be very difficult for a navel architect to design a ship if first he had to design the keel then the hull and then the interior spaces. In order to change the size of the engine room he would have to have a cutting torch operator and dismantle part of his design.

It is clear that special knowledge, not only of the modeled elements but also of the operations, is a useful thing to embed into a design system. This paper proposes that a programming capability be incorporated into solid modeling systems that would permit the modification of the system to accommodate different design practices. Presented is an example of how a user would define a new method. This will illustrate the methodology used in the system and

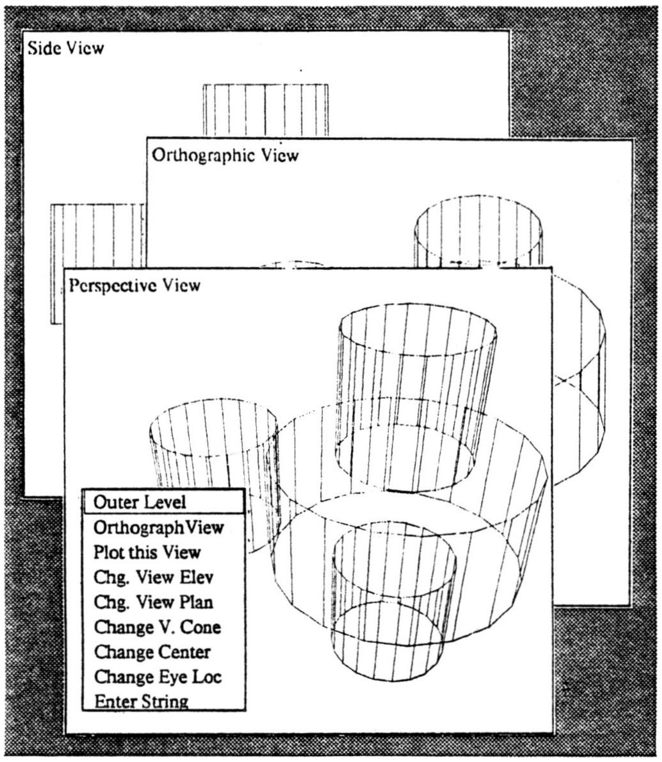


Figure 1: Typical Screen During Interaction

the character of the interaction This capability is currently being added to a solid modeling system that has been developed at Carnegie-Mellon University. An overview of the purpose of this system which is known as VEGA and its organization may be found in [10]. More information of the details of the implementation of the system's programming facilities beyond those that are presented here may be found in [3]. A view of what the screen looks like during a typical interaction session for the current system is shown in figure 1.

2 Requirements of a Design Programming Environment

Most integrated programming environments [9, 4, 6, 7] have been implemented to support an existing programming language such as Lisp or Pascal. These languages are text based and therefore require an environment that is appropriate for that medium. Because the purpose of our programming environment was to add extensions and make modifications to a menu based interactive design system we wanted to develop a new language syntax that followed the style of interaction that is used in the design system. It was thought that by careful design of the interaction command language it could be the same as the programming language. In doing this we wanted to decrease the amount of typing in of information and commands that is required of the user. Since we tailered the language to fit the interaction environment the system design and implementation was simplified.

A design programming environment must be simple for users/designers to generate definitions and it must possess sufficient power to permit construction of a rich set of operators. In this proposed environment simplicity is insured by having the programming operations closely follow the syntax of the design operations. When the user is creating a new command definition he interactively performs the sequence of design operations that make up the command and these are remembered by the system. The power of the programming language comes about from three factors: having a complete set of control operators, being able to define functions with parameters and the ability to define structured variables.

There are three parts to a project such as this: the syntax of the language, the semantics of the operations and the character of the user interaction. In the current system the syntax closely fallows that of the FORTH programming language with the addition of several pre-defined types that that are useful for modeling. As in FORTH, argument passing is done using a stack. The primary virtue of having the stack is that the interactive user is presented with a simple environment for doing operations without having to name elements. For example, the current top shape or the current top value closely corresponds to the word at the current location used in EMACS [5] or the current class and current object of the Smalltalk80 system. These promote the use of modeless interaction and nieve users seem to find this to be a natural way of relating to design programs [8]. Observations of students learning programming languages seems to confirm that when variables are introduced the students confusion is increased.

There are three alternative ways to structure the argument stack: have one stack that is used for all elements without type information stored with the data, or there can be one stack with the type of information stored also, or there may be multiple stacks, each of which contain only elements of the same type. Each of the techniques that incorporate type information may be used to facilitate run time error checking. The VEGA system was implemented using a system of multiple stacks. There are two reasons for introducing the extra complication of having more than one stack. With the use of only one stack in designing meta commands the user must manage the data for every operation. The order that the different information is entered becomes very important and during hand simulation required information was often buried in the stack requiring separate operations to restore order. As users organize their information conceptually based on type. The other advantage of multiple stacks is that it provides a level of abstraction that parallels this concept. Operations may then take place based on this information.

In a language that the user may interactively extend it is difficult to definitively state the semantics but pre-defined are all of the normal geometric operators such as union and intersection of shapes.

This system encourages programming using an *example-based* strategy that is similer to that used for the definition of *keyboard macros* in EMACS. The difference is that as a command is added to the definition not only is the current stat of the process displayed as it is effected by that command but also that content of the new definition is displayed. The design of this system was heavily influenced by the Tinker system [4] which is a environment for defining Lisp programs by example and the Smalltalk system. The differents between Tinker and our system are accounted for mainly because we wanted the programming system to be as close a possible to the interaction environment and our desire to deal with the problems of working in 3-D space.

As was previously stated the user intercation utilizes menus for selections of commands. It is beyond the scope of this paper to go into much detail on the workings of these parts but presentation of a example should clarify the way that they work together.

3 The Example

For an example we will define a procedure that will create a roller that is placed next to an existing roller and is centered about a given point. This situation is shown in figure 2.

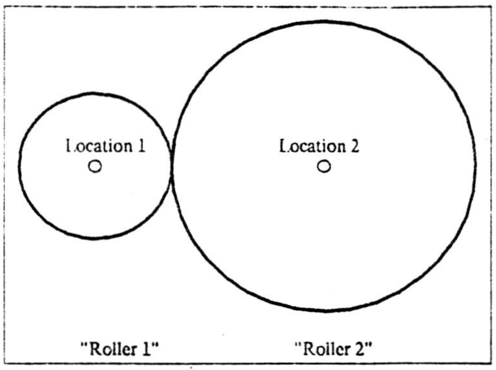


Figure 2: Diagram of the Task

As input this operation requires the location of the new roller here known as location 2 and the name of the existing roller, "Roller 1", that this one is to touch and the name that is to be assigned to the new roller, "roller 2". This will utilize an existing function, called Make-Roller, that makes a roller given a name for the new roller, a center location and the radius. Performance of the task requires finding the center of the existing roller and its radius, then finding the distance between the existing roller's center and the new location. Then by subtracting the existing roller's radius from this distance the radius is found that is required to define the new roller.

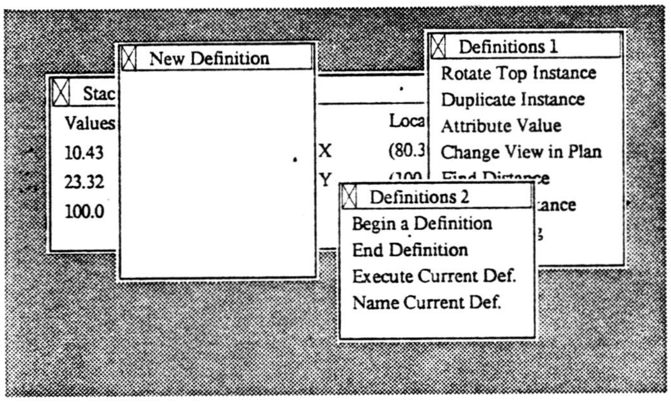


Figure 3: Start a New Definition

The first task in defining this procedure would be to supply some example date that may be used as test input during the definition phase. Then the user selects the *Begin Definition* command from a menu of existing definitions. As is shown in figure 3 this displays a new window that will be used to contain the display of the component definitions as they are added to the new definition.

Then the user executes the sequence of actions that are to be added to the new definition. As each action is selected from the menus they are executed and they are also added to the new definition as is shown in figure 4. In this case the code that is required to define the new roller in the syntax of our language is shown below:

```
Select-Name-Instance           ; fetch the instance who's name is
                               ;  on top of string stack and push
                               ;  onto instance stack
Duplicate-Instance             ; duplicate the top entry
                               ;  on the instance stack
Duplicate-Location             ; duplicate the loc. for "roller 2"
Center-Location                ; Finds the center of the shape of the
                               ;  top instance
Distance                       ; find the distance between the top
                               ;  two locations and push
                               ;  this onto value stack
                               ;  pop the locations
Push-Const-String "Radius"     ; push this string
Duplicate-Instance             ; duplicate TOS instance
Attribute-Value                ; get the value of the radius
Pop-Instance                   ; don't need "roller one" any more
Difference                     ; find difference between distance and
                                  radius
Make-Roller                    ; make a roller given these parameters
```

As can be seen the structure of the language requires the specification of a sequence of instruction that utilize the stack for all arguments. The stack conditions after each of these component operations are:

VEGA CODE	String	Value	Instance	Location
Given Stack Conditions	"Roller 1"			Location2
	"Roller 2"			
Select-Name-Instance	"Roller 2"	-	Rol1	Location2
Duplicate-Instance	"Roller 2"	-	Rol1	Location2
	-	-	Rol1	
Duplicate-Location	"Roller 2"	-	Rol1	Location2
	-	-	Rol1	Location2
Center-Location	"Roller 2"	-	Rol1	Location1
	-	-	-	Location2
	-	-	-	Location2
Distance	"Roller 2"	Dist	Rol1	Location2
Push-Const-String "Radius"	"Radius"	Dist	Rol1	Location2
	"Roller 2"	-	-	-
Duplicate-Instance	"Radius"	Dist	Rol1	Location2
	"Roller 2"	-	Rol1	-
Attribute-Value	"Roller 2"	Rad	Rol1	Location2
	-	Dist	-	-
Pop-Instance	"Roller 2"	Rad	-	Location2
	-	Dist	-	-
Difference	"Roller 2"	Rad 2	-	Location2
Make-Roller	-	-	Rol2	-

When all of the operations are added to the new definition the *End Definition* operation is selected. This makes the new defihition the currently defined definition. This current definition may be executed like any other operation by the *Execute Current Def.* command.

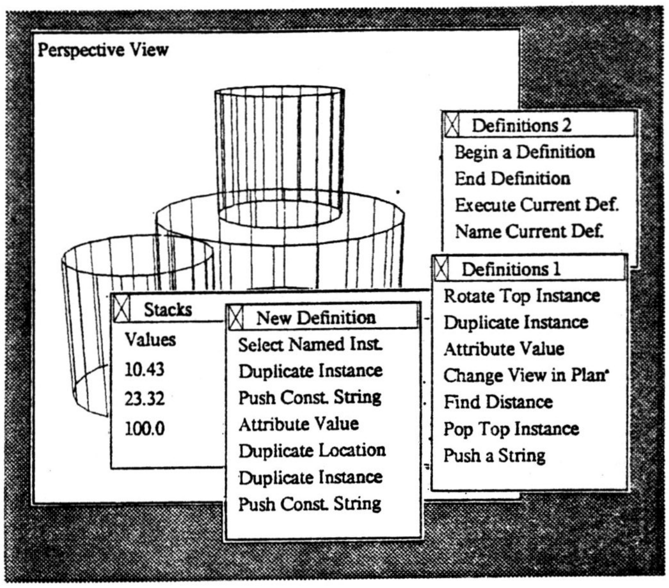


Figure 4: Add Commands to Definition

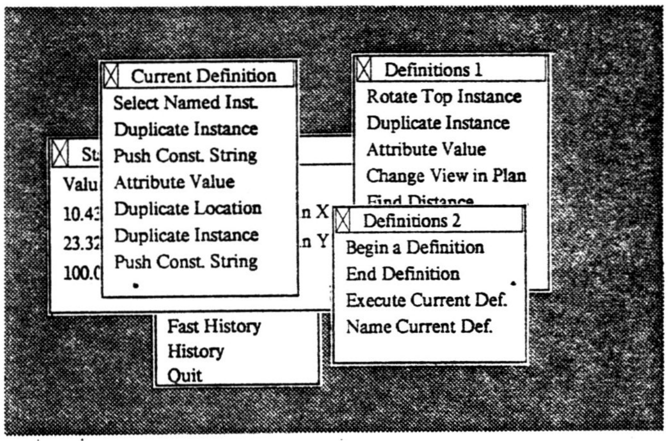
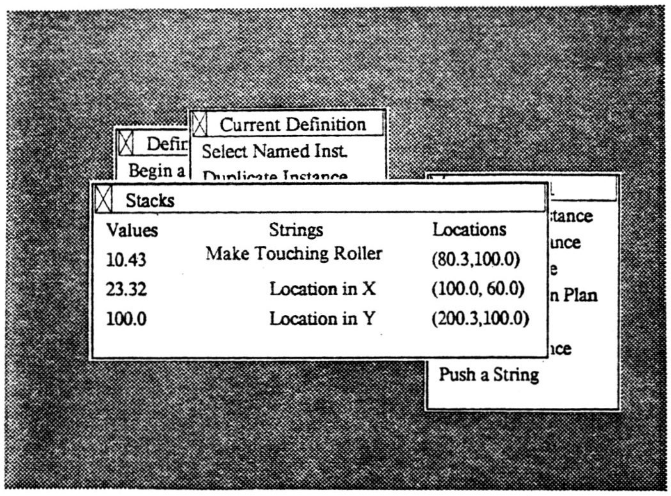


Figure 5: End the Definition

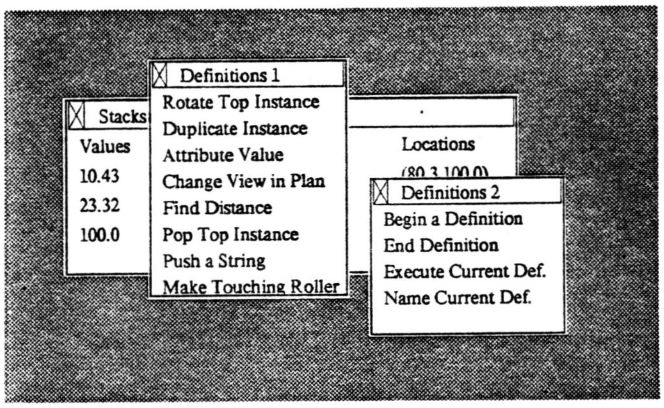
It is thought that many procedures will be defined and used for a short period of time and not used any more. The current definition concept deals with that case but if the user would like to save this procedure then a name for it must first be pushed onto the string stack as is shown in figure 6.

When the string that will become the name of the operation is defined on top of the string stack then the user selects the *Name Current Def.* operation which will add this name to the list of definitions and it will remove the window that shows the current definition from the screen as is shown in figure 7.



Values	Strings	Locations
10.43	Make Touching Roller	(80.3,100.0)
23.32	Location in X	(100.0, 60.0)
100.0	Location in Y	(200.3,100.0)

Figure 6: Push a String for the Name



Values	Locations
10.43	(80.3,100.0)
23.32	
100.0	

Figure 7: Save the Definition

One stack that is not used in this example but is used quite a bit is the boolean stack. This is used for conditional execution of code. There is no block structuring in this language so each collection of components that would be in a block if expressed in a language such as pascal must be defined as a separate definition. For an example of block structuring and conditional execution we might take the following simple operation in pascal:

```
read(a);
read(b);
if a = b then begin
    a := a * a;
    a := a + b
end;
```

In our language we would first have to define the instructions that are here found inside the begin end block as a new definition. In this case it might be called ***action***:

```
Duplicate Value
Multiply Value
Add Value
```

Notice that since the operations are selected from a menu it is not an advantage to use short names for identifiers.

The next part of the code is the entry of the data, the expression evaluation and the conditional execution of ***action***. The code for that section in our language is:

```
Enter Value
Enter Value
Equal Value
Push Const String "action"
If True
```

The *Equal Value* test takes the top two values on the value stack, finds if they are equal and if so pushes ***True*** onto the boolean stack else ***False*** is pushed. Then the two values are removed from the stack. The *If True* operation executes the definition named by the top of the string stack if the top of the boolean stack is true.

4 Execution environment for meta commands

First some terms that we will use must be defined. *Execution time* is the time during which user specified actions are being executed. *Methods* are actions that may be initiated by the user at execution time[1]. There are two types of methods: *primitive*, those that are pre-defined within the system, and *secondary*, those that the user has defined[2]. Provided with each method, be it a primitive or a secondary, is a unique identifier which we call a *method identifier*. The execution package must maintain a directory that contains the *method identifiers*. During execution we will have to search through our list of methods matching on the identifiers. This will be a source of execution time slow down.

Because secondaries may call other secondaries there must be a *Return Method Stack* - RMS that will contain a pointer to a directory entry for the next method that is to be executed at the end of the currently executing one. The depth of this stack corresponds to the level of nesting of the meta command calls. Since a method might detect an error during its execution and we want to be able to modify

[1]The use of the word method for this comes from the Smalltalk systems

[2]These two terms come from the Forth language

values at this time and then resume execution we also need an *Error Method Stack*. This will contain a pointer to the method that contained the error. At this location execution will start again after the correction of a definition or the data on a stack.

The heart of the execution environment is the inner interpreter. The inner interpreter contains the procedures that get the next action and call the correct function. If a secondary is being executed then the interpreter will take the next command from the definition of that secondary rather then try to get one from the user as it would for a primitive. This code is organized as a loop. Normally when primitives are being executed the loop will try to get the next command from the user. This will continue until execution of a secondary is requested then the loop will take its commands from the secondary's definition rather then from the user. At the end of the secondary's definition the method that is pointed to by the RMS will provide the command. The RMS will also be popped by one. If the RMS is empty then the interpreter will again take its commands from the user.

By the use of this simple approach the user may define new commands while using the system. The commands are not stored in a file and then executed by reading the file as in some systems but they may be stored in files as a whole to be read back in and used at a later time. We may do this using the string that the user specifies to be the identifier for this meta command. The identifying string is what is used as the label on the menu selection button for this meta command.

5 Conclusion

Adding a programming capability to a solid modeling system permits the modeling of the operations that are relevant for design elements as well as their geometry and attributes. This also allows the designer to customize his working environment to facilitate his design task. In this way also the system may be adapted for a new class of user. In the system presented attempts to simplify the programming task by having the programming language be the same as the user interaction language. The requirement of having the user type in names for identifiers was minimized by having a menu based system.

Acknowledgements

Thanks are due to Chuck Eastman and the other members of the CAD-Graphics Laboratory of IBS at CMU for many constructive suggestions during the development of this system.

[1] Arbab, F., Lichten, L., Melkanoff, M.
Toward CAM - Oriented CAD.
In *Proceedings of the 19th Design Automation Conference.*
Sigda, 1982.

[2] Glass, G. J.
Automated Part Location in the Design of Assemblies.
Technical Report CSL-83-4, Institute of Building Science, Carnegie-Mellon Univ., January, 1983.

[3] Glass, G. J.
Programming Facilities for User Modification of VEGA.
Technical Report CSL-83-11, Institute of Building Sciences, Carnegie-Mellon Univ., April, 1983.

[4] Lieberman, H.
Constructing Graphical User Interfaces By Example.
In *Proceedings Graphics Interface '82*, pages 295-302.
Canadian Man-Computer Communications Society, Toronto, Ontario, May, 1982.

[5] Meyrowitz, N., van Dam, A.
Interactive Editing Systems: Part 1 and Part 2.
Computing Surveys 14(3):321-415, September, 1982.

[6] Sandewall, E.
Programming in an Interactive Environment: The LISP Experience.
Computing Surveys 10(1):35-71, 1978.

[7] Shapiro, E., Collins, G., Johnson, L., Ruttenberg, J.
Pases: A Programming Environment for Pascal.
ACM Sigplan Notices 16(8):50-57, 1981.

[8] The Learning Research Group.
The Smalltalk-80 System.
Byte 6(8):36-48, August, 1981.
The whole issue is devoted to Smalltalk-80.

[9] Tesler, L.
The Smalltalk Environment.
BYTE 6(8):90, August, 1981.

[10] Woodbury, R., Glass, G.
The VEGA Solid Modeling System.
In *Proceedings Graphics Interface '83.* Canadian Man-Computer Communications Society, May, 1983.