A Tutorial For SUIT
The Simple User Interface Toolkit

Matthew Conway
Randy Pausch
Kimberly Passarella

Computer Science Department, University of Virginia

suit@uvacs.cs.Virginia.EDU

We would like to thank those organizations who have helped fund SUIT:

National Science Foundation

Science Applications International Corporation

Virginia Center for Innovative Technology

Virginia Engineering Foundation

Welcome To SUIT

SUIT, the Simple User Interface Toolkit, is a subroutine library which helps C programmers create graphical user interfaces that may be modified interactively. SUIT acts as a window manager for screen objects such as buttons, scroll bars, and menus. As SUIT-based programs execute, users may change attributes of the screen objects including an object's location and appearance. The changes to these attributes are then saved with the program.

Thanks for SUIT are due to its original author, Nathaniel Young, and to Roderic Collins, Matt Conway, Tom Crea, Jim Defay, Pramod Dwivedi, Robert DeLine, Brandon Furlich, Rich Gossweiler, Drew Kessler, Chris Long, William McClennan, Kim Passarella and Anne Shackelford. Thanks are also due to the UVa User Interface Group and the entire UVa CS department for their help in the extensive user testing that went into SUIT and the SUIT tutorial.

This work was supported in part by the National Science Foundation, the United Cerebral Palsy Foundation, the Virginia Engineering Foundation, the Virginia Center for Innovative Technology, and SAIC.

We would like to hear from you. Feel free to send electronic mail to:

`suit@uvacs.cs.Virginia.EDU`

with any comments you might have. We are especially interested in mail describing any errors, unclear sections, or omissions you find.

IMPORTANT: Before You Run This Tutorial

1. We strongly suggest that you go through this tutorial with a friend. Learning a toolkit is always easier if there is someone else to consult.

Installation Notes:

2. There are several files you will need in order to run this tutorial. If you are running on a UNIX machine at the University of Virginia, you can install these files automatically by typing:

```
/users/suit/bin/InstallTutorial <machine type>
```

where <machine type> is one of: `sun3` `sparc`

This program will create a directory for you and will copy the following example files into that directory: ***Makefile, empty.c, poly.sui, 3cell.sui, demo.sui, poly, 3cell, and demo***.

Things You Need To Run This Tutorial

- Be sitting at a (preferably color) display
- Know the C programming language, and know how to compile C programs on your system
- Be familiar with a text editor (emacs, vi, etc.)
- Have SUIT installed
- Have a copy of the tutorial files properly installed (see above).
- If you don't understand "callback routines" or function pointers, you may wish to read the article entitled "An Introduction to External Control" included at the end of this tutorial.

A Simple SUIT Program: *poly*

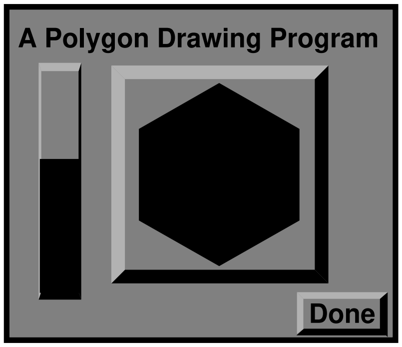


*Figure 1: **poly** contains four widgets: (Top to bottom, left to right) A label, a bounded value, a polygon, and a button.*

To see what a SUIT application looks like, change into the ***tutorial*** directory and run the program ***poly***, a simple program that draws regular polygons. (If you are working on a UNIX machine, make sure you are running the X window system first.) You should see something like Figure 1.

SUIT applications contain screen objects called *widgets*. The object on the left is a *bounded value widget* which controls the number of sides in the polygon. The object in the lower right hand corner of Figure 1 is a *button widget* which allows you to exit the program. To change the number of sides in the polygon:

1. Move the mouse cursor over the bounded value. Either press and release any mouse button (click), or hold any mouse button down and move the cursor up and down (drag). Try making the polygon have seven sides. Note that the bounded value has a minimum setting of three.

How to Modify a SUIT Interface

SUIT gives you the ability to move and resize widgets *while the application is running*.

Moving Widgets

1. Hold down both the SHIFT and CONTROL keys with one hand
2. With the other hand, move the cursor near the center of the widget you'd like to move. Press and hold down the leftmost mouse button.

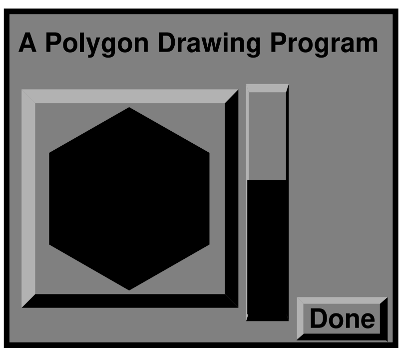


*Figure 2: The **poly** application, after moving some widgets.*

A dashed outline of the widget should appear and follow the cursor until you release the mouse button, at which point the widget will be placed at the new location, and you can release the SHIFT and CONTROL keys. Move the bounded value to the right of the polygon, so that the interface looks like figure 2.

Note that once you have moved the bounded value and released the SHIFT and CONTROL keys, you can once again use the mouse to change the bounded value's value. Click on the bounded value until the polygon has five sides.

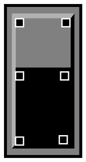
SHIFT and CONTROL: The SUIT-keys

The SHIFT and CONTROL keys are used as a signal to SUIT that you wish to interact with the interface as the interface designer (letting you move and resize things), not as a user of the application (letting you slide the bounded value up and down). Whenever you are holding down the SHIFT and CONTROL keys, you're "talking to SUIT" and whenever you are not, you're "talking to the application." SUIT uses this strange, two-key combination so that it won't interfere with any special keystrokes you might want to use in your application.

In the rest of this tutorial, we will precede any operation which requires you to use the SHIFT and CONTROL keys with the prefix "SUIT-". For example, "SUIT-a" means "hold down the SHIFT and CONTROL keys and press the letter ***a*** on the keyboard." Similarly, "SUIT-click" means "hold down the SHIFT and CONTROL keys, and click the left mouse button."

A table of the SUIT command keys appears at the end of this tutorial.

Selecting Widgets



SUIT provides several other means by which you can change the appearance of widgets in your interface. All of these, however, require you to first select the object you wish to change. To do this, SUIT-click on it.

Try selecting the bounded value. When the bounded value is selected, *handles* appear around the edges.

Handles appear on a widget when the widget is selected. Dragging a side handle will allow you to resize the widget in that direction.

Deselecting Widgets

An object may be deselected by SUIT-clicking again anywhere on the screen. Try deselecting the bounded value. The handles will go away.

Resizing Widgets

In addition to moving widgets, you can also resize them.

1. SUIT-click on the bounded value to select it.
2. SUIT-drag one of the handles. Try resizing the bounded value to make it tall and thin. Resize it again to make it square. Deselect the bounded value by clicking anywhere on the screen.

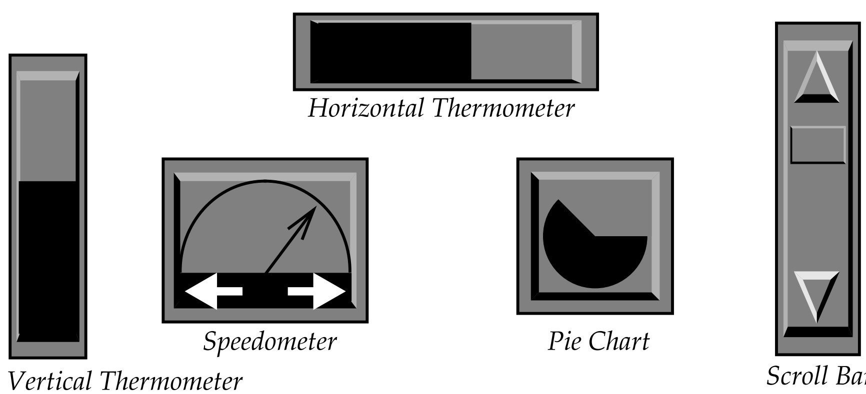
When Widgets Don’t Resize

Buttons and labels are designed to “shrink to fit”, meaning that they always remain slightly larger than the text they contain. In the upcoming section on the property editor, you will see how to change this and other properties of any widget so that you can make your buttons and labels any size you want.

Cycling Widgets Among Display Styles

The bounded value you have been manipulating allows you to specify the number of sides for the polygon; in this case, a number from 3 to 20. There are any number of ways that a bounded value like this might be displayed. In addition to the vertical thermometer, SUIT provides five built-in *display styles* for the bounded value widget:

The display styles of the bounded value widget



To change a widget’s current display style, you *cycle* the widget:

1. Move the cursor to the center of the bounded value
2. Type SUIT-c

Notice that as you cycle the bounded value from style to style, the current value does not change; the only thing that changes is the way that the value is displayed.

The Widget Library: *demo*

Now is a good time to see all the widgets in the SUIT library, available for use in any application.

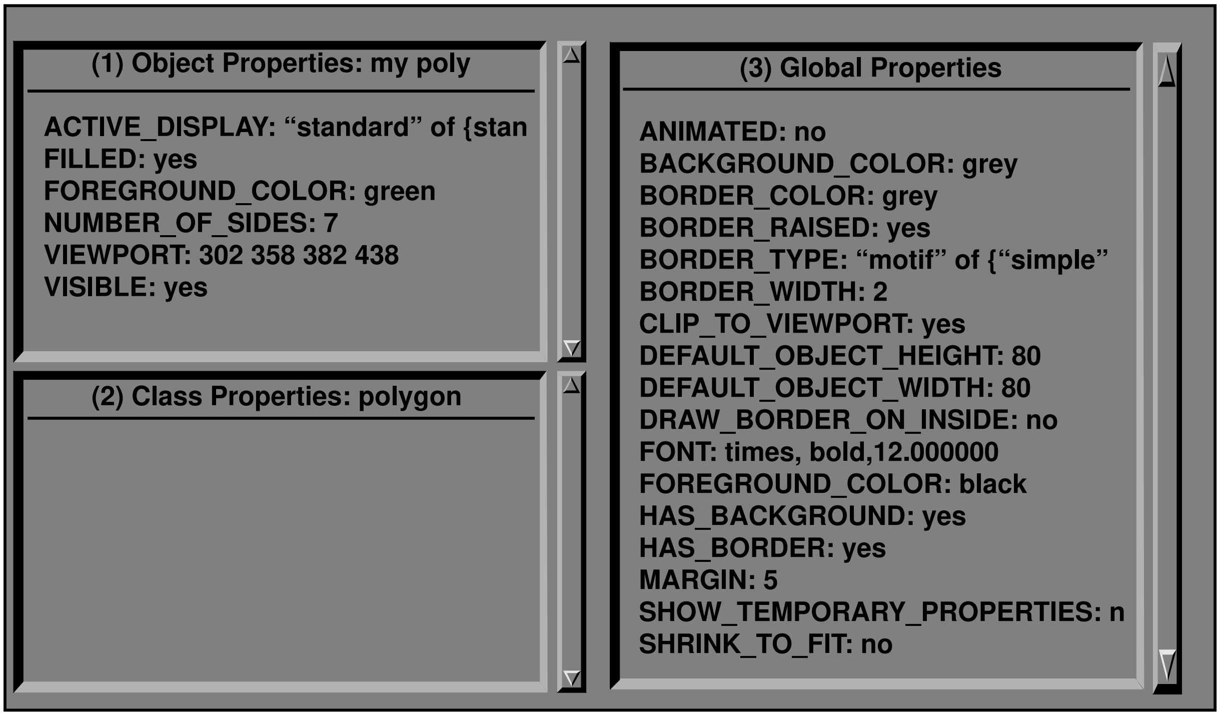
1. Exit ***poly*** by pressing the DONE button. If you get a message that says "Sorry, unable to update .sui file", you can ignore the warning.
2. Run ***demo***

Demo is a do-nothing application; it contains widgets you can poke at without causing anything to happen. For fun, try playing with the other widgets; see if you can guess what they do.

Widget Types

- **Bounded Value:** for allowing the user to select a floating point number between some minimum value and a maximum.
- **Menu:** a collection of buttons.
- **Scrollable List:** Allows the user to select a textual item from a scrolling list of choices.
- **Radio Buttons:** for choosing exactly one item out of many in a mutually exclusive fashion.
- **Text editor:** simple emacs key bindings.
- **Color Chips:** for displaying a "currently selected color"
- **Type in box**: a one line text entry widget.
- **On / off switch:** for toggling a boolean.
- **Buttons:** invoke functions when they are pressed.

Properties and the Property Editor



What if we wanted to change something about a widget other than its location, such as its color? Each SUIT widget maintains information like this in the form of a collection of variables or *properties* that govern the widget's appearance and functionality. To view or change any of the properties that a widget has, you can invoke the SUIT *property editor*.You will notice that properties, like variables, come in different types; SUIT supports a wide variety of types including double, string, boolean, and more advanced types like color, font, and enumeration.

***WARNING*: There might be a strong temptation to "play" with the properties you find in the SUIT property editor. We recommend that you not do this. There are some properties, particularly global properties that, if altered, can bring disastrous results. If the programs in this tutorial start behaving strangely after changing a global property, press the ABORT button and start again.**

1. Put the cursor over the polygon widget. Type SUIT-e (for "edit").
2. BOOLEANS: Click on the object property FILLED. It should change to NO. Notice that the polygon is not filled anymore.
3. TEXT: Click on the object property called NUMBER_OF_SIDES. A text box will appear. Type a CONTROL-k to delete the text currently in the textbox. (This means hold down the control key and press "k".) Type in the number 3 and press RETURN. Notice that the polygon now has 3 sides.
4. COLOR: If you have a color monitor, Click on the object property FOREGROUND_COLOR. Select a bright color from the choices offered and press OK. Notice that the color of the polygon has changed to the color selected on the color chips.
5. Exit the property editor by clicking on *OK*.

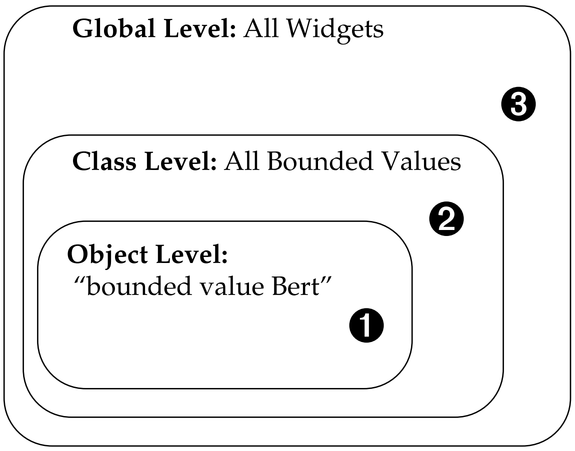
Other useful "type in box" commands are:

Delete a character	CONTROL- d
Forward a character	CONTROL- f
Back a character	CONTROL- b
Kill Line	CONTROL- k
Beginning of Line	CONTROL- a
End of Line	CONTROL- e

More About Properties

As you have seen, each screen object has various *properties* which store information about the object's appearance and functionality. If these properties were always stored with each object, it would be hard to enforce consistency. For example, if all the labels in an application were green, and you decided to make them all red, you would have to change them all one by one. SUIT addresses this problem by allowing properties to be stored at three different levels: the *object* level, the *class* level, and the *global* level. These nested levels are shown below.

Property lookup is performed from the innermost level (object) to the outermost level (global)



To find a property, SUIT starts at the object level. If the property in question has not been defined at the object level, the search is performed again at that widget's class level. If the search fails here too, SUIT searches the global level, where if it is still not found, a default value for that type of property is returned.

EXAMPLE: Suppose the code for a button widget called "Bert" initiates a property lookup on a property called FOREGROUND_COLOR. SUIT first looks to see if a "foreground color" property has been specified for Bert at the object level. If so, SUIT returns the value of the property. If not, SUIT looks to see if there is a FOREGROUND_COLOR property specified at the class level for all "buttons". If not, SUIT searches the global level for FOREGROUND_COLOR, at which point, if the property is still not found, the default value of black will be returned.

Property Inheritance

To get a better idea of how property lookup works, try the following steps:

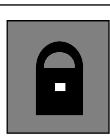
1. INVOKE THE PROPERTY EDITOR: Type SUIT-e over the ***label*** at the top of the screen that says *"SUIT".*

Note that the label has no HAS_BORDER property at the object level. This means that the label is inheriting its HAS_BORDER from a higher level -- in this case, the class level.

2. Change the class level HAS_BORDER property to YES by clicking on it.
3. EXIT THE PROPERTY EDITOR: Click on the *OK* button.

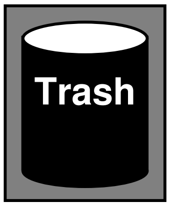
All the labels now have borders because they are all getting their boolean HAS_BORDER property from the Class level. Suppose you wanted all but one of the labels to have a border? To do this, you have the single label override his class level property by giving him a property at the object level. We'll do this by copying the label's class property to the object level.

Properties that have a lock icon next to them cannot be changed from inside the property editor.



4. COPYING PROPERTIES: Invoke the property editor again on the same label. We are now going to ***copy*** a property from the Class level to the Object level. To do this: Press and hold the mouse button down (no need to use SUIT keys here) over the HAS_BORDER property listed under Class Properties and move your mouse until the cursor is over the Object Properties box. When you release the mouse button, the property will be copied from the Class level to the Object level.
5. Change the Object level HAS_BORDER property of this label back to NO. Exit the property editor again. Notice that this label is now differ-

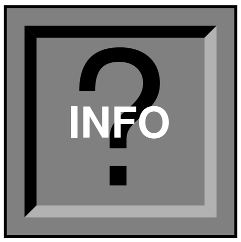
ent from the others: it has no border because this label finds its HAS_BORDER property at the object level (where HAS_BORDER is NO), not the class level (where the value is YES).



What if we wanted this label to revert to inheriting its HAS_BORDER from the Class? We need to ***delete*** the object level property to let this happen.

6. DELETING PROPERTIES: Invoke the property editor one last time on the label. Dispose of the object level HAS_BORDER property by dragging the property from the object level listing to the trash can icon. Notice that the label has gone back to inheriting its HAS_BORDER property from its class.
7. EXIT PROPERTY EDITOR: Click on *OK.*

Interactive Help: Info

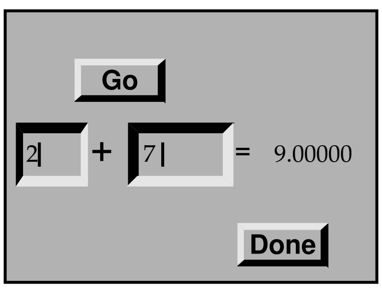


If you are unsure of the purpose of some property, you can get help from SUIT's *Info* facility. Info works like the trash can: you drag a property from its place in the property editor to the Info panel. A dialog box will appear with a message that gives a short description of the property.

1. Invoke the property editor (SUIT-e) on the *bounded value* widget.
2. Drag the GRANULARITY property to the Info icon to see a description of that property. Dismiss the help box by pressing OK.
3. EXIT PROPERTY EDITOR: Click on *OK.*
4. EXIT DEMO*:* Click on ***demo***'s *Done* Button.

Your First SUIT Application: *3cell*

You are now ready to build your own SUIT application, a three cell spreadsheet. To see an example of what your application will be like run ***3cell***.



When you click on the GO button, 3cell adds together the numbers that have been entered into the first two cells and displays the sum in the third.

The Skeleton of All SUIT Applications

Exit ***3cell*** and load ***empty.c*** into your text editor. We will build ***3cell*** starting with this template:

```
#include "suit.h"

void main(int argc, char *argv[]) {
    SUIT_init(argv[0]);
    SUIT_createDoneButton (NULL);
    SUIT_beginStandardApplication();
}
```

In order to make this program into ***3cell*** you will:

1. Add several additional screen objects:
   - One button (the GO button)
   - Two text boxes (the cells into which numbers are entered)
   - Three labels (a plus sign, an equals sign, and the result of the addition)
2. Add a *callback* function called `PerformAddition()` to the GO button. For right now, all this function will do is beep.

The exact changes to make to *empty.c* are printed on the following page.

```
#include "suit.h"
SUIT_object num1, num2, result;

void PerformAddition(SUIT_object object) {
    GP_beep();
}

void main(int argc, char *argv[]) {
    SUIT_init(argv[0]);
    SUIT_createDoneButton(NULL);
    num1 = SUIT_createTypeInBox("num 1", NULL);
    num2 = SUIT_createTypeInBox("num 2", NULL);
    SUIT_createLabel("+");
    SUIT_createLabel("=");
    result = SUIT_createLabel("result");
    SUIT_createButton("GO", PerformAddition);
    SUIT_beginStandardApplication();
}
```

Changes to the code are marked with change bars

3. Compile ***empty.c*** (on a Unix system, type ***make***) and run ***empty***.
4. Since this is the first time you have run this program, all the objects are randomly scattered across the screen. Go ahead and make the interface look like ***3cell's***.
5. Now enter numbers into the two text boxes and press the GO button. Notice that although you hear the beep, the cell for the result is not updated. That is because it has not been "attached" to the cells into which you entered numbers. The following steps will illustrate how to do this.
6. Exit ***empty*** by pressing the DONE button.
7. Modify ***empty.c*** to look as shown below. This new code will read the current values of the "type in boxes". Add them together and display the sum in the "result" label. Notice that the property names (LABEL and CURRENT_VALUE) are constants that appear in CAPITAL LETTERS.

```
#include "suit.h"
SUIT_object num1, num2, result;

void PerformAddition(SUIT_object object) {
    double temp1, temp2;
    char buffer[100];

    temp1 = atof(SUIT_getText(num1,CURRENT_VALUE));
    temp2 = atof(SUIT_getText(num2,CURRENT_VALUE));
    sprintf(buffer, "%lf", temp1 + temp2);
    SUIT_setText(result, LABEL, buffer);
}
void main(int argc, char *argv[]) {
    SUIT_init(argv[0]);
    SUIT_createDoneButton(NULL);
    num1 = SUIT_createTypeInBox("num 1",NULL);
    num2 = SUIT_createTypeInBox("num 2",NULL);
    SUIT_createLabel("+");
    SUIT_createLabel("=");
    result = SUIT_createLabel("result");
    SUIT_createButton("GO",PerformAddition);
    SUIT_beginStandardApplication();
}
```

8. Compile ***empty.c*** and run the resulting executable, ***empty***.
9. Enter numbers into the two text boxes and press the GO button. Notice

that now the cell designated to display the result is updated.

What The New Code Does:

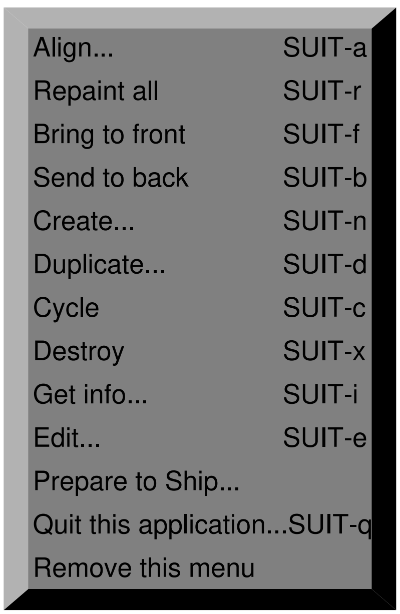
The `PerformAddition()` function begins by looking up the CURRENT VALUE property on each of the two type in boxes (`num1` and `num2`). The current value of a type in box is the text that appears in the box.These strings are converted into floats by the standard C library function `atof()`. These two floats are then added together and the result converted to a string called `buffer` via `sprintf()`. The last step takes the LABEL property of the widget called `result` and makes it equal `buffer`. This function is run each time the GO button is pressed because we registered the function `PerformAddition()` as the "callback" for the GO button when we called `SUIT_createButton ("GO", PerformAddition)`.

Interactive Widget Creation

In the previous section you created three labels by adding the following calls to ***empty.c***:

```
SUIT_createLabel("+");
SUIT_createLabel("=");
SUIT_createLabel("result");
```

Though this was not particularly difficult, it did require that you recompile. Fortunately, SUIT does provide a way for you to create widgets interactively, through the use of the SUIT command menu.



The SUIT command menu: The functions available here can also be reached through the keyboard shortcuts that appear on the menu.

The SUIT Command Menu

The *SUIT command menu* provides additional operations that can be performed on widgets: To access this menu:

1. Run your new application called ***empty***.
2. Place the cursor over an unoccupied area

Type SUIT-m. The SUIT menu will appear.

The SUIT command menu provides access to many of SUIT's interactive tools; the individual items not covered in this tutorial are discussed in the SUIT Reference Manual. To remove the menu select *Remove this menu*.

1. Bring up the SUIT menu by typing SUIT-m.
2. Choose "Create..." from the SUIT menu.
3. Select "label" from the list of widget classes.
4. Type "My 3 Cell Application" in the space provided for an "object name".
5. Select OK. A new label will appear on the screen.
6. Choose "Create..." again from the SUIT menu.
7. Create a clock widget and place it in the upper left hand corner of the application.
8. Exit ***empty*** by pressing the DONE button.

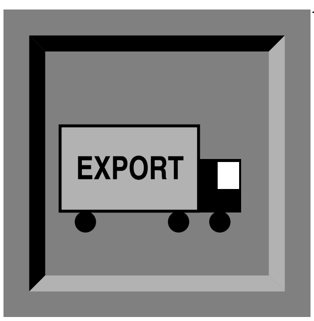
Saving the Interface: the ".sui" File

Each time you quit a SUIT application, the properties associated with each of the widgets are saved in a file called an ".sui" file.[1] Not only does this file contain all the information necessary to display your application's interface (all the widget locations and sizes) but it also keeps a record of the state the program was in when you last left it (the value of all the bounded values, the choices that were selected for each radio button in the interface, etc.). If a SUIT program starts up without an accompanying ".sui" file, the widgets appear on the screen in random locations and all other properties are given default values (booleans default to NO, colors default to black, doubles and integers are 0, etc.).

Interactive Widgets and the ".sui" file

The ".sui" file is also used for something else: the definition of widgets that are created interactively. For example, if you were to look in the file ***3cell.c***, you would notice that there is no function call to `SUIT_createLabel()` that corresponds to the new interactively created "My 3 Cell Application" label. The only reason SUIT knows to create such a label the next time the program is run is that the description of the label is kept in the ".sui" file. If you were to lose the ". sui" file any widgets created interactively would be lost as well (not to mention the description of the entire interface of the application). *Given the importance of the ".sui" file, you should treat the ".sui" file with the same care as you would the executable or the source code.*

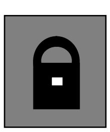
Interactive Functionality: Export



SUIT can attach program functionality interactively.

Suppose you had a widget called "Ernie" and you wanted the user to be able to control Ernie's foreground color through a set of color chips somewhere in the application. Instead of adding a `SUIT_createColorChips()` call to your code, writing the callback, and recompiling the code, you could *export* Ernie's FOREGROUND COLOR to the application's interface via the export button in the property editor. Exporting a property means having SUIT add another widget to your application interface whose sole job is to *control some property of another widget*. SUIT can do this because for every data type that SUIT manipulates (integers, booleans, strings, etc.) there is a corresponding widget that can represent that type (bounded values, check boxes, type-in boxes, and respectively). To export the polygon's FILLED property:

Properties that have a lock icon next to them cannot be changed from inside the property editor.



1. Start up the ***demo*** program.
2. Invoke the property editor (SUIT-e) on the polygon widget.
3. Drag the FILLED property from the object level listing to the export panel (looks like a moving van), in the same way you might drag a property to the trash can.
4. Select *OK* to exit the property editor.

Notice that there is now a new *on/off switch* in the application (this is the kind of widget that controls boolean data types). Click on the switch and the polygon will change from filled to unfilled and back again.

1.) So named because the name of the file is the same as the name of the program with an sui extension: the application ***empty*** will create an sui file called ***empty.sui.***

Where to Go for More Information

If you need further information, the following documents are available:

- An Introduction to External Control
- The SUIT Reference Manual
- The SUIT example Files that come with the SUIT distribution

In addition, questions addressed to:

`suit@uvacs.cs.virginia.edu`

will be answered promptly.

Summary of SUIT-keys

Operation	What it Does	Hot Key[1]
SUIT menu	*invokes the SUIT menu, which contains most of the following functions*	SUIT-m
cycle	*change a widget's display style*	SUIT-c
align	*lines up selected widgets by tops, bottoms, etc.*	SUIT-a
send to back	*selected widget goes behind all others*	SUIT-b
bring to front	*selected widget goes in front of all others*	SUIT-f
select widget	*marks a widget as selected; deselects a widget if already selected; selects all widgets if cursor is over no widget*	SUIT-s
redraw	*repaints all widgets*	SUIT-r
edit properties	*examine and alter a widget's properties*	SUIT-e
get info	*prints a widget's name, class and display style in a dialog box*	SUIT-i
open widget	*opens up a parent widget so that the children may be accessed.*	SUIT-o
close widget	*closes a parent widget that was opened with SUIT-o*	SUIT-k
create new widget	*creates a new SUIT object on the fly*	SUIT-n
destroy	*destroys a SUIT widget*	SUIT-x
version	*prints the version of SUIT you are using*	SUIT-v
duplicate	*creates a copy of a SUIT widget*	SUIT-d
write sui file	*writes the sui file to disk without exiting*	SUIT-W

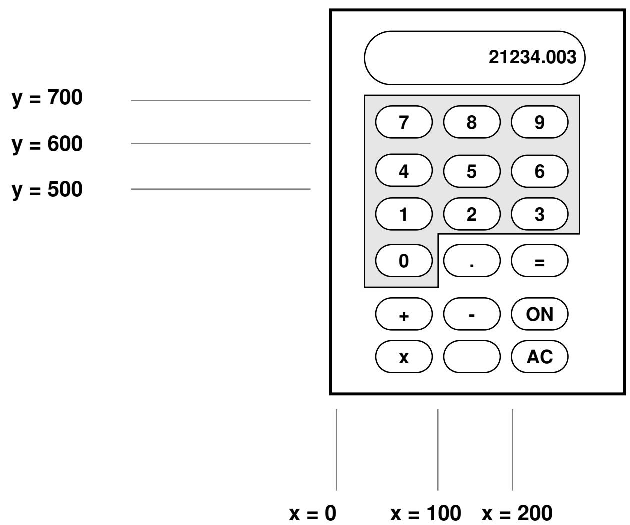
1.) "SUIT" is shorthand for holding down the SHIFT and CONTROL keys simultaneously.

Appendix: An Introduction To External Control

Motivating Example: The Calculator

This section is an introduction to a model of computing which may be new to you: the external control model. External control is a way of programming that has become the norm in windowing environments, but can be confusing at first.

Imagine that it is your job to write a graphical user interface (GUI) for the following application: a simple four function calculator.



The Simple Calculator. Numbers given are typical pixel locations.

The first thing you would do is partition the screen into its various regions, laying out where each button and screen element is supposed to go. Then, your program would be "driven" by mouse clicks -- each time the user clicks, you would find out which "button" was hit, and then do the appropriate thing. This "picking routine" often ends up being a rat's nest, as in:

```
GetMouseClick (&x, &y);
if ( ( y > 500) && ( y < 700) ) /* in number buttons */
{
if ( x < 100 ) /*  in left row of buttons */
      {
      if (y > 600)
      HandlePressOnSevenButton();
      else if (y > 500)
      HandlePressOnFourButton();
      ...
```

Needless to say, this code is hard to write, and even harder to maintain. If, for example, you wanted to "reshape" your calculator to be wide, not tall, you would have to completely rewrite this routine.

A better solution is to build a table which contains the location of each important screen component, or "widget," and a subroutine to call when the user clicks on that widget with the mouse. We can indicate which subroutine to call by using #`defines`, which are similar to Pascal CONSTs. Such a table would look something like:

xmin	**ymin**	**xmax**	**ymax**	**routineNumber**
0	600	100	700	SEVENBUTTON
0	300	100	400	PLUSBUTTON
100	300	200	400	MINUSBUTTON
And so on...				

Now, the picking routine is very simple:

```
int button = NO_BUTTON;
GetMouseClick(&x, &y);
for ( i = 0 ; i < max_table_entries ; i++ )
{
      if (  (x >= table[i].xmin) &&
            (x <= table[i].xmax) &&
            (y >= table[i].ymin) &&
            (y <= table[i].ymax) )
  button = table[i].routineNumber;
}

switch ( button ) /* like a Pascal CASE statement */
{
      case NO_BUTTON: beep(); break; /* user missed */
      case SEVENBUTTON: PressSevenButton(); break;
      case PLUSBUTTON: PressPlusButton(); break;
      case MINUSBUTTON: PressMinusButton(); break;
      ... and so on
}
```

This is much better, because the picking routine is no longer "hard wired" and all you need to do to change the location of a widget is to change its numbers in the table. This picking routine technically runs more slowly than the previous version, but it turns out that this all happens so fast that it's not a problem -- modern computers can easily loop through the hundred or so objects on the screen with no perceptible delay to the user. Given the advantages to this approach, and the fact that the user can't tell the difference, there really is no incentive for using the faster version.

As it turns out, we can even avoid the need for all the constants and the switch statement, because the C language allows us to store the routine name (technically, a pointer to where the routine is stored in memory) in the table. So, the above code can be written as:

```
/* instead of integers, the table now contains
RoutinePointers */
RoutinePointer routineToCall = NULL;
GetMouseClick(&x, &y);
for ( i = 0 ; i < max_table_entries ; i++ )
{
if ( (x >= table[i].xmin) &&  (x <= table[i].xmax) &&
      (y >= table[i].ymin) &&  (y <= table[i].ymax) )
      routineToCall = table[i].callback;
}

if ( routineToCall == NULL )
      beep();
else
      routineToCall(); /* invoke C callback routine */
```

Now the picking routine is extremely small and simple, and all we have to do to change the screen interface is change numbers in the table (N.B. this scheme does not address what happens if two screen widgets overlap; we'll assume that doesn't happen).

Changing the Table Values

But this still hasn't gotten us very far. Figuring out the locations of all the buttons is still very tedious from the programmer's point of view and the all widget locations are fixed at compile time; only a programmer can design or change the interface. However, by going to this table driven external control model, we have gained flexibility that we did not have before. As long as the X, Y pairs are in the table, there's nothing to say that they must remain fixed. We could let the designer directly manipulate the interface by moving and resizing the widgets right on the screen, a process that simply *updates the global location table by changing the location information for the given widget!*

We would like for the user to be able to drag screen widgets around with the mouse and literally "redraw" the screen interface. But this causes an ambiguity: when the user presses the mouse button on the "plus" key, did he or she mean to drag the plus button to a new location, or to actually press the button? In SUIT, we solve this by having the designer hold down the CONTROL and SHIFT keys when he or she wants to move widgets.

New External Control Loop

Putting all this together, the main external control loop for SUIT looks like this:

```
while ( TRUE )
{
GetMouseEvent(&x, &y);
if (shift and ctrl are both held down)
      talk to SUIT : drag widget on screen
else
      Scan the table and call a callback routine
}
```

Saving and Restoring State

The table of widget locations should be saved between invocations of the program. We can write the names and locations of all the widgets to a file just before the application exits so that the next time the application is run, the location values can be reloaded into the global table. In SUIT, the location table file is called an ".sui" file (say all the letters: "ess you eye"). In fact, this ".sui" file holds more than just the locations of all the widgets -- it holds any other state information that the program might wish to restore on the next invocation of the program (e.g. widget colors, fonts, etc.).

Summary

External control is a model of computation that is commonly used in event driven scenarios, which is why it is so useful in GUIs (graphical user interfaces). The thread of control in such a system lies in the hands of the user; the main loop is the server that handles the mouse and keyboard events and dispatches them to the appropriate widgets.

The most important thing that you must realize when programming with external control is that you are no longer writing a large, monolithic program where you, the author, control the sequence of what happens. Instead, you write many small "callbacks" which can be called at any time. For example, in the calculator program, the user is free to press any button at any time, and there is no "main program" which prompts the user to "press a number, then an operator, then another number, then the '=' key." You will probably be very tempted to put code in your callback routines which "prompt" or force the user's choices; this is something you should usually avoid. Instead, try to accept the programming model that the user is in charge and may press any widget at any time.