A Tutorial for Suit: The Simple User Interface Toolkit

Randy Pausch

Computer Science Report No. TR-90-29
September 1, 1990

A Tutorial For SUIT, the Simple User Interface Toolkit

Randy Pausch
Computer Science Department
University of Virginia

electronic responses:
graphics@uvacs.cs.virginia.edu

NOTE: This is a *draft* document: it would be greatly appreciated if you would send electronic mail to `graphics@uvacs.cs.virginia.edu` about any errors, unclear sections, missing information, or anything else you'd like to tell us. SUIT is currently under active development. Any questions or comments mailed to `graphics@uvacs.cs.virginia.edu` will be answered promptly. We know this draft of the tutorial does not answer all the questions you might have, and we ask that you please make use of the electronic mail address to answer questions that come up. Thanks!

SUIT, The Simple User Interface Toolkit, is a subroutine library which helps C programmers create graphical user interfaces. A C programmer can invest approximately two hours with this tutorial and be able to create SUIT programs which run on UNIX/X-Windows, IBM-PC/DOS, and Macintosh platforms. SUIT is like having a window manager for interface components – as SUIT-based programs execute, users may change the location, appearance, and functionality of screen objects such as buttons, sliders, menus, etc. All changes are saved with the program so later invocations will reflect the interface changes.

Thanks for SUIT are due to its original author, Nathaniel Young, and to Roderic Collins, Matt Conway, Jim Defay, Pramod Dwivedi, Robert DeLine, Brandon Furlich, Rich Gossweiler, Ken Kestner, and Kimberly Passarella.

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.

This tutorial begins by having you use SUIT to interactively modify an existing graphical interface. You are then given a brief overview of SUIT which is followed by examples of how to develop your own C programs using the SUIT subroutine library. After approximately two hours with this document, you should be able to construct simple SUIT programs. This document is meant to be read as you sit at a color display and try things out; if you have any questions or comments about SUIT or this tutorial, send electronic mail to `graphics@uvacs.cs.virginia.edu` and it will be answered as soon as possible.

Running an Existing SUIT Application

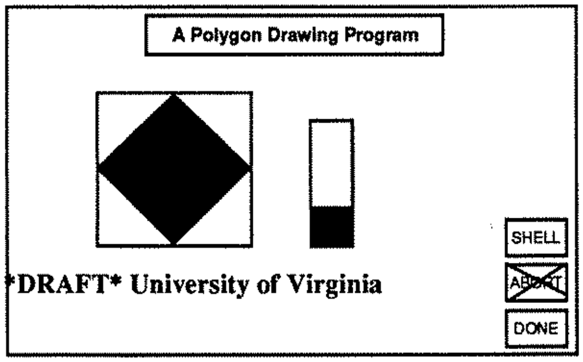
The first program you'll run will draw regular, N-sided polygons. If you're running on a PC, the program will use the entire screen. If you're running on a UNIX workstation you should be running the X window system, because the program will run in a new X window. If you're not already running X, you may be able to do so by typing:

```
startx
```

This tutorial gives command/path names for the unix version, but you should find all necessary files in the appropriate locations on whatever machine you are using. Type

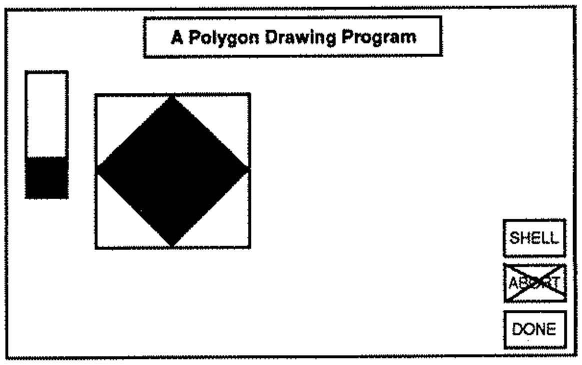
```
/users/graphics/suit/examples/poly
```

to invoke the first application. It may take 10 or 15 seconds for the application fire up; this delay will disappear in later versions. If things take longer than about 30 seconds, something is wrong and you should contact a member of the SUIT staff. When the `poly` application fires up, you should see something like:



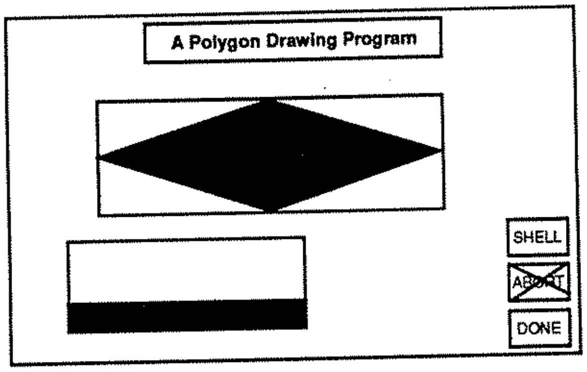
The object on the right is a *slider* and it controls the number of sides in the polygon on the left. The three buttons in the lower right hand corner appear in most SUIT applications; they allow you to gracefully exit the program (DONE), leave the program without saving your work (ABORT), or pop to a command shell (SHELL). The ABORT button is covered with a *guard*, indicating it's a dangerous command.

You can change the number of sides in the polygon by moving the mouse cursor over the slider and either clicking (pressing and releasing) any mouse button, or pressing down on any mouse button and dragging the slider up and down until you release. Try making the polygon have seven sides. You can also reposition screen objects, but doing so requires telling SUIT that you'd like to move a screen object, not interact with it. For example, you would like to move the slider around with the mouse, but you can't just click on the slider and move it around the screen, because when you click on the slider, the slider responds by sliding up or down. To tell SUIT that you'd like to move an object, rather than interact with it normally, you need to sort of "raise one hand" to tell SUIT to listen to you, and then use your other hand to manipulate the mouse. You can do this by pressing (and holding down) both the CONTROL and SHIFT keys. While you're holding down CONTROL and SHIFT with one hand, move the cursor near the center of the slider and press and hold down the *leftmost* mouse button. A dashed outline of the slider should appear and follow the cursor until you release the mouse button, and the slider will now move to the new location. You can now release the SHIFT and CONTROL keys. Try moving the slider to the upper left of the polygon, to get a picture like:



Note that once you have moved the slider and released the CONTROL and SHIFT keys, you can once again use the leftmost mouse button to adjust the slider's value. Play with both moving the slider around and adjusting its value – just remember that whenever you're holding down the CONTROL and SHIFT keys, you're talking to SUIT, and whenever you're not, you're talking to the application. As a shorthand, the rest of this document will use the notation `SUIT-click` to mean "talk to SUIT by holding down the CONTROL and SHIFT keys, and use the leftmost mouse button."

`SUIT-click` can be used to change the size or shape of an object by SUIT-click'ing on the object close to a border. Hold down CONTROL and SHIFT, move the cursor near an edge or corner you wish to adjust, and then press the leftmost mouse button down and hold it. The familiar dashed rectangle appears; you should now move the mouse outward to push the edge or corner out. To make an object smaller, you `SUIT-click` it near an edge or corner, drag out to grab the edge or corner, and then bring the edge back in. Note that you must start off slightly inside the object. Make sure you're comfortable with move and resize; don't go on until you can make the screen look something like:



Holding down both CONTROL and SHIFT is awkward. On some systems, you can avoid this by holding down the ALT key. The ALT key is often to the left of the space bar; on some keyboards, it is labeled LEFT, instead of ALT. Not all systems support using ALT instead of CONTROL and SHIFT. If your system does, you can simply hold down the ALT key, instead of CONTROL and SHIFT, whenever you want to `SUIT-click`.

You can also invoke SUIT commands via the keyboard. For example, you can explicitly force a repaint of the screen by using `SUIT-r` (that is, hold down the CONTROL and SHIFT keys and press the 'r' key on the keyboard, and then release all three keys.) Again, if your system lets you, you may be able to just hold down ALT instead of both CONTROL and SHIFT.

Changing Your Interface's Appearance

SUIT allows you to interactively change many aspects of your application's interface. For example, the slider is a screen component which displays a *bounded value* between 3 and 20. This can be displayed in several ways; SUIT allows you to switch between them. To *cycle* the slider between possible display styles, move the cursor to the center of the slider object and use the `SUIT-c` command. The bounded value can appear in several different ways, including as a slider that slides up and down, a slider that slides left and right, and a speedometer. Click on the bounded value when it's displayed as a speedometer and see what happens. Now cycle back to the up and down slider, since you'll use it in the next example.

Each SUIT object has various *properties* which you can change with SUIT's built in *property editor*. To change the slider object's properties, move the cursor to the center of the slider object and use `SUIT-e`. to invoke the property editor for that object. When interacting with the property editor, you don't need to hold down CONTROL, SHIFT, or ALT; you'll be able to just click with the mouse. One portion of the screen is labeled **Object Properties** and contains the different aspects of the bounded value object. The property editor always displays the properties in alphabetical order. The slider object has a *boolean property* called **has border.** If you click on that property (remember, just click with the left mouse button, you don't need to hold down CONTROL, SHIFT, or ALT), it will change, and you should see the slider redraw itself without a border. By clicking on the property again, you can turn the border back on. If you want to change a color property, select a color from the color chips, and then click on the property you want to be that color. Try to make the slider have a blue border.

When a SUIT object draws itself, it looks up various properties about itself, like "what color should my border be?" If these properties were always stored with each object in a program, it would be hard to enforce consistency. For example, if all the sliders were green, and you decided to make them all red, it would require changing them one by one. SUIT addresses these problems by providing a hierarchical property data structure. When an object asks a question like "what color should my border be?", SUIT first looks to see if the object has specified a value for that property. If so, SUIT returns the property's value. If the object does not have a property with that name, SUIT then looks in a data structure for the object's class (all SUIT objects belong to a *class*. All sliders are in the *bounded value* class. This name was chosen because their current value is always within minimum and maximum bounds). When SUIT looks up the **border color** property for the "bounded value" class, SUIT finds the color each bounded value should default to if it does not specify its own color.

If SUIT still doesn't find the named property at the class level, it looks in a data structure for the *global properties*. Each of these *property lists* (object, class, and global) is displayed on the screen in the property editor. If you're wondering about the area with the header **display properties,** SUIT looks in a data structure for the *current display style* before looking at the object level. In that way, you can have a bounded value object with a red border when displayed as an up & down slider, but a green border when displayed as a speedometer.

To see how this all works, click on a pink color chip and set the global property for **foreground** to pink. Now exit the property editor by clicking on the **DONE editing properties** button, and notice that all objects are now displayed in pink.

The one exception is the bounded value; it specified its own foreground color, so it doesn't default to the global property. If you want to have the bounded value default to the global property, you must destroy its specification of **foreground** at the object level. To do that, put the mouse cursor back over the slider and use `SUIT-e` to go back into the property editor. Now, move the cursor over the **foreground** property in the object properties area and press down with any mouse button. While holding the mouse button down, drag the property (which will appear as a dashed rectangle while you drag it) over to the trash can and then release the mouse button. (Make sure that your cursor is on top of the trash can when you let up.) The bounded value no longer specifies its foreground property at the object level, and falls back on the global specification and becomes pink. This might be a good time to change the global property for **background** to a dark shade of blue.

Boolean and color properties change when you click on them; for other properties, such as strings and numbers, you must click with the left mouse button on the desired property and then type in the new value. When typing in a value, you can use emacs editing commands, and you must hit `RETURN` when you're done typing the new property value. If you would like to specify a property at the object level that doesn't already exist there, you can make a copy of it from some other level. For example, if you drag a global property to the object area (as opposed to dragging it to the trash can), it makes a copy of the global property at the object level.

Try making the polygon widget have a different background color than the rest of the screen. Move the cursor to the center of the polygon object and use `SUIT-e` to get back into the property editor. Copy the *background* global property to the object level, and then change it to be green.

Now is a good time to see all the existing types of screen objects, sometimes called *widgets*, that SUIT allows you to use off-the-shelf. Exit out of the current program by clicking on the `DONE` button and then type

```
/users/graphics/suit/examples/useall
```

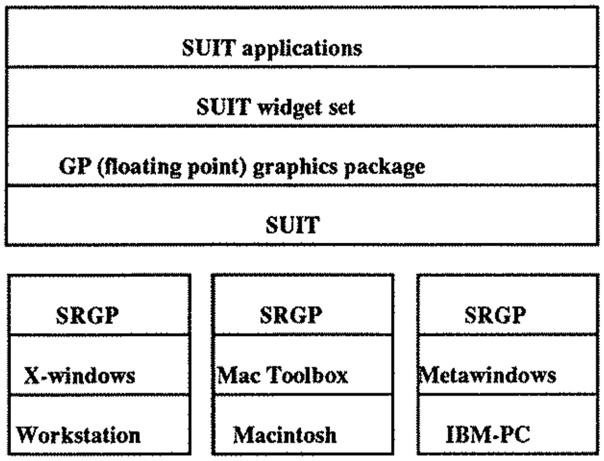
to see a program that demonstrates all the widgets in the widget library. This is a *brain-dead* application; it contains a bunch of widgets you can poke at, but nothing is going to happen because they aren't connected to any program functionality. So, you can poke at the colorchip widget, which in a real application might be hooked up to something, but in this application it doesn't do anything. The exceptions are the *menu* and *radio button* widgets, which print out the selections when they are made. This causes a console window to appear in the Macintosh version of SUIT. Suggested fun things to try with the `useall` program are:

(1) Discover the many different University of Virginia logos (cycle between alternate displays by using `SUIT-c`).
(2) Cycle the menu widget to see its vertical, horizontal, and pull-down display styles.
(3) Make a very wide but short Rotunda.
(4) Change the label at the top of the screen from "SUIT -- The Simple User Interface Toolkit" to "SUIT -- It's fun" (invoke the property editor and type in the new label).
(5) Change the colors to make things look really garrish.

When you're done, click on the `DONE` button to exit the program.

Portability

SUIT is capable of running on several different platforms because it is built in layers:



SRGP (the Simple Raster Graphics Package) has been implemented on three platforms, UNIX workstations, Macintoshes, and IBM-PCs running DOS. In each case SRGP was ported quickly by utilizing an existing graphics package. SRGP provides a common ability to draw lines, text, and other graphics primitives across all the platforms. Most SUIT graphics use a slightly more sophisticated package, GP (Graphics Package), which allows screen objects to automatically scale to any given size or shape. SUIT contains the code for maintaining the objects within an application, and a standard widget set is provided for application use. All of SUIT is written in ANSI-C, and applications can be moved from one platform to another by recompilation. Information describing the state of user interfaces, such as object positions and sizes, is stored in files with the `sui` extension. These interface description files are human readable and can be carried with an application from one platform to another.

Separation of Form and Function

As you have already seen, SUIT provides a fairly powerful set of interactive tools for modifying a program's user interface. One of the goals of SUIT is to remove user interface decisions from the programmer. We distinguish between *function*, which must be specified by programmers in C code, and *form*, which should be specified (potentially by non-programmers) using the interactive tools. Beginning SUIT programmers typically ask questions like "When I create an object, how to I specify that it should be red?" In most cases, it is better to simply create the object and then use the interactive tools to position it and change its properties. While this seems unnatural to many programmers, try to always distinguish between form and function, and remember than form should be able to specified by someone like a graphic artist using the interactive tools.

As an extreme example of this, consider that the menu widget can cycle between vertical, horizontal, and pull-down styles, and that the application is not aware that this has happened. The application decides that a set of functionality should be provided via a menu, and the *interface designer*, perhaps a graphic artist, uses the interactive tools to decide how that menu should be presented to the end user. All these decisions are stored in `sui` files that SUIT creates in the same directory as the application's executable file.

The information stored in `sui` files quickly becomes as important as C code and you will find that you should not delete a `sui` file lightly, since it may represent a fair amount of specification effort. Also, multiple `sui` files can exist for a single application. A suit program named `foo` (or `foo.EXE`, on a DOS machine) normally looks for a file called `foo.sui` but by passing the `-f bar.sui` switch, any SUIT application can be asked to use an alternate interface description stored in the file `bar.sui`. Unless told otherwise, SUIT applications expect to find their `sui` file in the same directory as the executable, *not* the directory the user was in when running the program. This means that if a program lives in a central location and is used by several users, they will (assuming they all have write permission to the file) all be using the same `sui` file. This will be fixed in later versions of SUIT.

One final note about the interactive tools: There is no written manual of the properties used by each of the widgets provided in the standard set. Programmers often need to access (usually to examine, but sometimes to change) properties of an object. There is no manual to go to "look up" the properties used by a widget; instead, the programmer should use the interactive property editor to find out the name of the relevant property. In this way, SUIT is self-documenting.

The Architecture of SUIT

The heart of SUIT is an array of screen objects, where each object is stored as:

```
[x, y, width, height, paintProc, hitProc]
```

For example, the polygon program you used had five screen objects: the polygon object, the slider object, and three buttons, each of which was an object. The (x,y,width,height) are the location and size of the screen object, the paintProc is a procedure to call to draw that object on the screen, and the hitProc is a procedure to call when the user clicks or types while the mouse is over that object. For example, the paintProc for the `DONE` button draws the word "DONE" in a rectangle, and the hitProc for the `DONE` button causes the program to exit. When SUIT needs to paint an object or process an input action by the user, it can call the appropriate procedure. A procedure which is stored in a data structure and later invoked is sometimes referred to as a *callback* because it has been registered to be "called back" at a later time.

When SUIT needs to paint the entire screen (when the program is first starting, or when you later use `SUIT-r` to force a repaint), SUIT runs down the array and calls each of the paintProcs. Picking is done in a similar fashion; when you click or type, SUIT runs down the table searching for an object the mouse's (x,y) falls within. If you are holding down CONTROL and SHIFT (or possibly just ALT), SUIT then handles the input. This is how you can move objects – SUIT recognizes that you are talking to it, paints the dashed rectangle that follows the mouse, and then changes the object's (x,y) in the array and repaints the object by calling its paintProc. When you are *not* holding down CONTROL and SHIFT (or just ALT), SUIT sends the input event on to the object by calling the object's hitproc. The basic structure of a SUIT program is:

```
create all the screen objects by putting them in the array

do forever
        begin
        get the next (x,y) mouse click
        run down the table and find an object it's within
        if user is talking to SUIT (CONTROL & SHIFT are down)
                then
                process move, resize, cycle, etc.
        else
                call that object's hitProc
        scan the table and repaint any objects that need updating
        end
```

This style of programming is sometimes called the *external control model*, because you do not explicitly direct your program's thread of control. Instead, you register pieces of functionality which are later invoked at appropriate times. Also note that, contrary to most people's intuition, hitProcs do *not* paint anything on the screen. Instead they rely on the appropriate paintprocs to be called later. This allows SUIT to be very efficient about updating the screen. For example, if the user is typing very fast in a type-in box, each character will cause the type-in object's hitProc to be called, and it will append that character to the string stored in the object's `current string` property. When SUIT later calls the paintProc, it looks up the object's `current string` property in order to know what to paint on the screen. Divorcing processing input from painting buys SUIT a great deal of flexibility. As a simple example, consider what would happen if the hitProc either painted on the screen, or itself called the paintProc. The user might get farther and farther ahead of the screen update (painting to the screen is almost always a bottleneck). By separating input processing and painting, SUIT can allow several input actions to be processed by calls to the hitProc before calling the type-in box's paintProc.

SUIT is actually somewhat clever about deciding when to repaint screen objects. For example, SUIT is smart enough to know that if you change a property, that object will require redisplay, but only if the new value for the property is different than the old one. If you want to explicitly tell SUIT that an object will require redisplay, you can call

```
SUIT_redisplayRequired(joe)
```

but only this causes SUIT to record the fact that the object `joe` should be repainted, and to perform the necessary call to joe's paintProc when SUIT deems it convenient.

Properties

Properties are stored in the *property list* attached to each object. Each item in the list is simply a `[name, value]` pair, such as `[border width, 3]` or `[current string, 'now is the time for']`. In addition to the property list attached to each object, there is a property list for each class of objects and a global property list. Each property is either `permanent` or `temporary.` Permanent properties are written to the `sui` file when the program exits, but temporary properties are not. Most properties are permanent. Properties can be examined by calling

```
Pointer SUIT_getProperty(SUIT_object obj,
        char        *propertyName,
        char *propertyType);
```

which returns a generic pointer to the area in memory containing the data for the current value of the property. Note that the `type` of all properties is explicitly specified. For many common types, convenience macros are defined, such as

```
someValue = SUIT_getInteger(someObject, "property name");
```

for example,

```
numSides = SUIT_getInteger(polygon, "number of sides");
```

Similarly, properties can be set by calling

```
void SUIT_setProperty(SUIT_object obj,
        char *propertyName,
        char *propertyType,
        Pointer propertyPtr, /* pointer to the new value */
        int propertyLength,  /* number of bytes of data */
        int level,   /* DISPLAY, OBJECT, CLASS, or GLOBAL */
        int permanence); /* PERMANENT or TEMPORARY */
```

but for common data types, macros such as

```
SUIT_setDouble(someObject, "some property", 2.345);
```

can be used. The macros assume that you are getting or setting permanent properties; if you are not, similar versions with names like `SUIT_deluxeGetInteger` can be found in the include files.

Each property is of a specific *type* which is simply a name given as a character string. SUIT already knows about many common types, such as integers, doubles (floating point numbers), text strings, colors, etc. If you want to use a new type of your own choosing, you will have to first register the type by calling `SUIT_registerType.`

Several SUIT properties are not intuitive, so we mention them briefly here. First, colors are actually a record with two fields. The first field is what the color should be on a color monitor, such as `red,` and the second tells what the color should be on a black and white screen. The interactive property editor allows you to change the portion of the record that is appropriate on whichever screen you are using. In this way, the same `sui` file can be used to specify an interface that works well on both color and monochrome screens. As a programmer, you will probably not need to worry about this, because your only use of colors will look like:

```
GP_setColor (SUIT_getColor (someObject, "foreground color"));
```

which tells the GP graphics package to set the current color to be whatever the `foreground color` property is for the current object. GP takes the color record and does the appropriate thing on whatever type of display you are using.

Another tricky property is `visibility,` which should be set by calling

```
SUIT_setVisibility(SUIT_object, boolean);
```

rather than using the normal property mechanisms, in order to cause the proper screen update.

A Simple SUIT Application

Many programs can be written by creating objects from the existing library of widget types. Let's begin by looking at the simplest possible SUIT program, which is:

```
#include "suit.h"

/* paint procs, hit procs, and other code should be added here */

void
main (int argc, char *argv[])
{
    SUIT_init(&argc, argv);

    /* object creation calls should be added here */

    SUIT_beginStandardApplication(NULL);

}
```

To get your own copy of this program, make sure you're in a directory where you want to the program to be and then type

```
cp /users/graphics/suit/template/suitprog.c .
cp /users/graphics/suit/template/Makefile .
```

The program includes `suit.h` which defines the routines and datatypes provided by SUIT and the SUIT widget set. The initialization call to `SUIT_init` allows every SUIT-based program to accept a common collection of command line options. This is implemented by passing `&argc` and `argv` parameters to `SUIT_init`, which extracts SUIT-related command line options and returns the remaining command line options to the main program. (For a complete list of the SUIT command line options, type "-suithelp" as a command line option to any SUIT-based program). The call to `SUIT_beginStandardApplication` creates the familiar SHELL, ABORT, and DONE buttons found in most SUIT applications, and then begins the infinite loop of input processing.

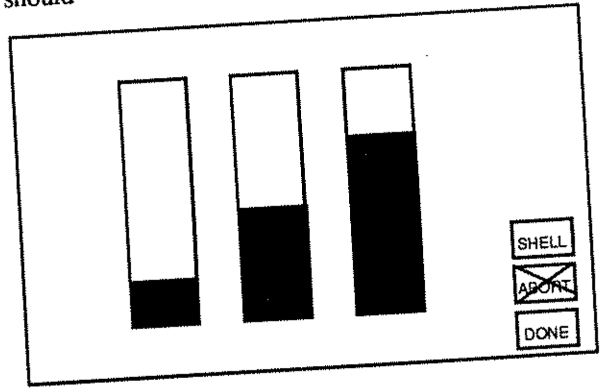
The single parameter to `SUIT_beginStandardApplication` can be used to specify a function to be called just before the program exists. In keeping with the external control model, this function will be called by SUIT when the user clicks on the DONE button, but that the function silently performs its task and does not prompt the user with a question like "Do you really want to exit?" If you are not used to the *external control model*, you will probably find it quite strange to "give up" the flow of control like this. One drawback of this model is that your program must wait for the user to actually provide input before it proceeds. If your program needs to perform real-time computation (such as a video game that needs to keep a ball bouncing around), you can replace the call to `SUIT_beginStandardApplication` with

```
SUIT_createStandardApplication(NULL); /* or your Done function */
SUIT_beginDisplay();
for (;;)
        {
        move the ball by setting some object(s) properties
        SUIT_checkAndProcessInput(0);
        }
```

If the user has already clicked or typed, `SUIT_checkAndProcessInput` processes the input (calling any necessary callback hit procedures), and then returns. If input isn't already waiting, the parameter is used as the number of 60ths of a second to wait for input. If zero is passed, the routine does not wait at all.

Let's try compiling and running this program: type `make` to compile the program, and then type `suitprog` to run it. You should see the familiar three buttons, but that's all. You can still use the property editor, but your application itself doesn't do anything, as you might have guessed from the source code.

In order to make a program that actually does something, you need to create screen objects and attach them to pieces of the program. Let's create a program that has three integers, one of which is always computed to be the average of the other two. The program should look something like:



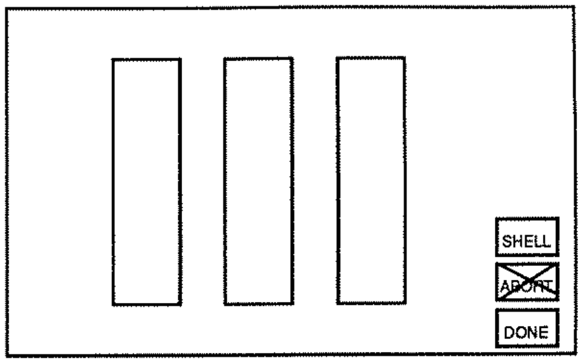
Go into your favorite text editor and modify your program to look like:

```
#include "suit.h"

SUIT_object          number1, number2, average;


void Average(SUIT_object object)

{

double     total;

total = SUIT_getDouble(number1, "current value") +
      SUIT_getDouble(number2, "current value");


SUIT_setDouble(average, "current value", total/2);

}

void
main (int argc, char *argv[])
{

   SUIT_init(&argc, argv);

   number1 = CreateBoundedValue ("number 1", Average);
   number2 = CreateBoundedValue ("number 2", Average);
   average = CreateBoundedValue ("average",  Average);

   SUIT_beginStandardApplication(NULL);

}
```

Your program now has three global variables, one for each of the three sliders. Each slider object is created via a call to `CreateBoundedValue` (remember, sliders are really of class *bounded value*, because they represent a bounded number that can be displayed as a slider, speedometer, etc.). With each call to `CreateBoundedValue,` the second parameter is a subroutine that the slider will call each time its value changes. If including the declaration of the type `SUIT_object` of the parameter `object` in the parameter list to `Average` looks unusual to you, it's a feature of ANSI-standard C.

By attaching three sliders to the `Average` function, the program guarantees that each time any slider is changed, the `average` slider's current value is recomputed. The `Average` function looks up the `current value` property for the `number1` and `number2` objects, and sets the `current value` property for the `average` object to be their numerical average. Because an object's property is changed, SUIT will know to automatically redisplay that object. So, when you click on one of the other two sliders, the `Average` procedure will be called, the computation will be performed, and SUIT will then redisplay whatever is necessary.

Type `make` to compile the program. Then type `suitprog` to run it. You should see the familiar buttons appear in the lower right hand corner, and you should also see things appearing in the lower left hand corner. When you first create an object, SUIT doesn't know where on the screen to put it. Rather than trying to guess, SUIT always puts things in the lower left hand corner, and assumes you'll move them to where you'd like them. Try moving and resizing your sliders to make the screen look like:



But which slider is which? You can figure out the name of an object by making it iconic (use SUIT-i), and then clicking on the icon to make it full size again. Make your sliders be in order (from left to right) `number1, average,` and `number2.` Now when you change either the leftmost or rightmost slider, the middle one will display their average. Note that you can still move and resize these widgets, and cycle them to make them all look like speedometers. You now understand the basics of ''widget programming.'' In widget programming, you create instances of screen objects provided by a library, and specify your own C functions to be called at appropriate times. To make sure you've got it, try adding a button to your program with a call like

```
clearButton = CreateButton("clear", myClearFunction)
```

and then write `myClearFunction` (which takes a single parameter of type SUIT_object), which sets all three sliders to zero. Note that you can either create variables of type `SUIT_object` or reference your objects by their string names, by calling the function

```
SUIT_name("some string");
```

which will return the `SUIT_object` whose name was specified to be `some string` when it was created.

Graphics

As previous stated, SUIT is a layered system. The lowest level basic graphics package, SRGP (Simple Raster Graphics Package), does all the screen drawing for SUIT. On top of this we have provided a thin layer known as GP. This additional layer is needed for two reasons, one is that SRGP does all of its drawing in screen pixel coordinates, and the other involves colors.

The problem with drawing in pixels is that each time an object paints, it would need to calculate how to lay itself out within its *viewport,* that area of the screen where you placed the object. To get around this problem, each object is given a floating point *window coordinate system* to specify graphics output primitives. SUIT initially sets this window to range from 0.0 to 1.0 in both x and y. The GP function calls, such as `GP_lineCoord` then converts these floating point coordinate calls into the appropriate screen pixel coordinates.

GP also supports color in a way that allows applications to run nicely on either black and white or color displays. To understand the problem, imagine that we were running on a color machine and specified a UVA logo with a red foreground and a green background. If we then ran the program, using the `sui` file we had created on a monochrome display, SRGP would map all non-white colors to be black, and the logo would be black on black; clearly the wrong thing. GP defines the type `SUIT_color` as a record with two fields: the color (stored as a string like "red") to be on a color display, and a boolean indicating whether to be black or white when running on a monochrome display. Note that the property editor is smart enough to determine the type of display you're working on during any given execution, and only displays the appropriate portion of this record.

The `GP_pushGraphicsState` and `GP_popGraphicsState` calls, which take no parameters, are used to save and restore the current graphics state (current color, pattern, etc.) onto a stack. It's a good programming practice to use these calls in any routine where you are painting.

GP and SRGP both *clip* output primitives that would fall outside the legal coordinates. SUIT automatically provides each object with a clip rectangle for the object's screen viewport before calling the paint procedure. Thus, when an object paints itself, it cannot draw outside its given area of the screen.

Because GP and SRGP are so similar, it would be redundant to supply detailed manuals on both. The document *SRGP for ANSI-C,* available as the postscript file `/users/graphics/srgp/doc/srgp.ps`, contains full documentation on SRGP. The file `/users/graphics/suit/src/suit/gp.h` contains the headers for the routines you will probably want to use in your widgets. You will note that most GP calls are simple renaming of SRGP calls, but they take double (floating point) *window coordinate,* not integer *screen pixel coordinate* parameters. We strongly recomment you use the GP calls and not the SRGP calls in your code.

Making Your Own Class of SUIT Objects

You now know how to use SUIT to change the location, size, and other properties of your screen objects, and how to use the standard library of screen widgets. This section will show you how to create your own application-specific types of screen widgets. The tutorial goes step-by-step through the creation of a *vector* widget, which will allow the user to interactively control a two-dimensional vector, as might be useful in a physics simulation. To see an example of the final version of the program, run

```
/users/graphics/suit/examples/vector3
```

and drag the vectors around. The left mouse button moves the head of the vector, and the rightmost mouse button moves the tail of the vector. Note that you can press down with the mouse and literally drag the ends of the vectors around. Each vector's head should be drawn as an arrow, but this program just draws it as an ellipse because its author was too lazy to figure out the angles for the arrowhead.

What you just ran is the final version of the program this tutorial will show you how to develop. The first version of the program you'll is a very simple, and later versions continue to add functionality to build up to the program you just ran.

The first version of the program has three routines, which allow us to

(1) Paint a vector widget
(2) Hit a vector widget (send user input to it)
(3) Create a vector widget

The first version of the program looks like:

```
#include "suit.h"

void
PaintVector(SUIT_object object)

{

double      headx = SUIT_getDouble(object, "head x");
double      heady = SUIT_getDouble(object, "head y");

GP_lineCoord(0.0, 0.0, headx, heady);

}

void
HitVector(SUIT_object object, SUIT_event event)

{

SUIT_setDouble(object, "head x", event.worldLocation.x);
SUIT_setDouble(object, "head y", event.worldLocation.y);

}


SUIT_object
CreateVector(char *name)

{

SUIT_object             newGuy;

newGuy = SUIT_createObject(name, "vector");
SUIT_addDisplayToObject( newGuy, "standard",
                                    HitVector, PaintVector);
return(newGuy);

}

void
main (int argc, char *argv[])

{

SUIT_init(&argc, argv);

(void) CreateVector("fred");

SUIT_beginStandardApplication(NULL);

}
```

The `PaintVector` routine is called with a single parameter which is the SUIT object to be painted. SUIT programs draw their graphics primitives with a graphics package called GP. Before SUIT calls your paintProc, it first paints your area of the screen with the background color and sets GP to use the proper foreground color. SUIT also takes care of things like drawing an object's border, if it has one. This means that simple widgets can just draw their graphics primitives. In this case, the widget wants to draw a vector from the lower left hand corner to wherever the tip of the vector is. This drawing must be specified with respect to a well-defined coordinate system.

The reason that SUIT widgets stretch when they are resized is because their graphics are drawn in a *world coordinate system*, not in screen coordinates. You can set the bounds of your world coordinate system to be whatever you wish, by default they range 0.0 to 1.0 in both x and y. By drawing in world coordinates, you allow the GP graphics package to scale and shape your graphics to fit into the current shape of the widget on the screen. In order to actually draw the line, you call `GP_lineCoord` to draw a line from (0.0,0.0) to (headx, heady). (The origin of GP's coordinate systems is the lower left.) The tail of the vector is always at (0.0, 0.0), and the widget determines where the head of the vector is by looking up two floating point properties, called `head x` and `head y`.

Whenever the user interactively moves the vector, our HitProc will change those properties for us. Whenever the user clicks on a vector widget, the HitProc is called with the current *input event*. Part of the input event record is the floating point (x,y) location where the mouse was clicked. The HitProc uses these numbers to set the `head x` and `head y` properties for the object. Setting these properties will cause SUIT to automatically call the paint procedure for the object, so the hitprocs *never* need to do any painting on the screen.

The `CreateVector` routine can be called to create a new widget of type `vector`. The first line creates the object, and the second line attaches your `HitVector` and `PaintVector` routines. The name `standard` is needed because you might create a widget with more than one way of displaying itself, and you would want to give a unique name to each pair of [hit/paint] procedures. You can run this version of the program by typing:

```
/users/graphics/suit/examples/vector1
```

Your vector will initially come up with its head at (0.0, 0.0), since your `double` properties will default to zero. As soon as you click in your widget, you'll be controlling the vector. You can move and resize your widget, and use the property editor to change your foreground and background colors.

The next version of the program you'll see allows the vector to optionally have an arrowhead (actually, just a circle, because it's easier to draw). This version also creates several instances of the vector widget. This version looks like:

```
#include "suit.h"

void
PaintVector(SUIT_object object)

{

double      headx = SUIT_getDouble(object, "head x");
double      heady = SUIT_getDouble(object, "head y");

GP_lineCoord(0.0, 0.0, headx, heady);

/********** THIS HAS BEEN ADDED **********/
if ( SUIT_getBoolean(object, "draw arrowhead") )
   GP_ellipseCoord(headx-0.05, heady-0.05, headx+0.05, heady+0.05);
/********** THIS HAS BEEN ADDED **********/

}

void
HitVector(SUIT_object object, SUIT_event event)

{

SUIT_setDouble(object, "head x", event.worldLocation.x);
SUIT_setDouble(object, "head y", event.worldLocation.y);

}

SUIT_object
CreateVector(char *name)

{

SUIT_object            newGuy;

newGuy = SUIT_createObject(name, "vector");
SUIT_addDisplayToObject( newGuy, "standard",
                                        HitVector, PaintVector);
return(newGuy);

}

void
main (int argc, char *argv[])

{

SUIT_init(&argc, argv);

(void) CreateVector("fred");
/********** THIS HAS BEEN ADDED **********/
(void) CreateVector("sally");
(void) CreateVector("debbie");
/********** THIS HAS BEEN ADDED **********/

SUIT_beginStandardApplication(NULL);

}
```

The paintproc now examines a boolean property and if it is `TRUE,` paints an arrowhead (drawn as an ellipse which is specified by its bounding rectangle). Once you ask for a property, it will be created and set to some default value. If you don't like it, you can change it with the property editor. To see how this version of the program performs, type:

```
/users/graphics/suit/examples/vector2
```

Note that none of your vectors have arrowheads. When you ask for a property for the very first time, SUIT creates it at the class level with some default value (FALSE for booleans). Use the property editor and make the vectors class default to having red arrows with arrowheads (set the `foreground` and `draw arrowhead` properties at the **class** level). Then make the `sally` object be blue, without an arrowhead, by overriding the class' defaults at her object level. You can determine which one is `sally` by using the iconize command; move the cursor over an object and use `SUIT-i` to make an open object iconic. Clicking on the icon will open the object back up.

The next (and final) version of the vector widget will tell the difference between different kinds of input. Look at the file

```
/users/graphics/suit/examples/vector3.c
```

and notice that the only difference in its paintproc is that it now looks for the `tail x` and `tail y` properties.

The major change is in the hitproc; the widget now distinguishes between input events via the left mouse button, right mouse button, and keyboard. All SUIT events are named; `left down` indicate the left mouse button went down, and `right down` indicates the rightmost button when down. If the event name is `keyboard,` the field `event.keyboard` contains the character that was pressed. The head of the vector can now be moved with the left mouse button, or the tail with the right mouse button. By typing characters at the vector, you can tell it to change its color.

Writing Your Own Programs Using SUIT

To start making your own applications, make a new directory and copy

```
/users/graphics/suit/template/suitprog.c
/users/graphics/suit/template/Makefile
```

to that directory. Now, you can edit `suitprog.c` and type `make` to recompile it.

Where to Go for Help

The files

```
/users/graphics/suit/examples/useall.c
/users/graphics/suit/src/widget/widget.h
/users/graphics/suit/src/suit/gp.h
/users/graphics/suit/src/suit/basesuit.h
```

are good starting points to find the calls you might want to make. Remember: this introductory documentation is only a *draft*, so we anticipate lots of questions and problems using the system. If you have any questions or comments, *please* mail them to `graphics@uvacs.cs.virginia.edu` and they will be answered as soon as possible.

Advanced Section

- selection
- delete props stuff
- undo in property editor