Chapter 3

Interaction and Graphical Interfaces

This chapter is devoted to the developement of interactive programs and the design of user interfaces.

We are going to build the basic concepts on top of the infrastructure of the `gp` 2D graphics package introduced in the previous chapter.

As an overview we will discuss next the problems of: event treatment; interface actions with callbacks; interaction objects with multiple views, interface managers, toolkits and widget design.

Finally we will finish the chapter with an example of an actual graphics interactive program: a polygonal line editor.

3.1 Creating Interactive Programs

Using the simplicity of the function `gpevent` it is possible to develop graphical interfaces which possess great complexity and a high degree of interactivity.

In this way, the interaction will be *event driven*, which makes the implementation easier.

In the project of a good interface, we need to take into account two main elements:

- Graphical Input/Output;
- Interface Design.

The graphical input/output project consists into the decisions of how the user is going to specify the various graphical objects of the program and their behaviour.

For example: one can create a line segment in a drawing program by simply marking two distinct points on the screen, or by using a rubber banding technique, which amounts to first defining the initial point of the line segment, as an anchor, and subsequently dragging the cursor to the line endpoint, as an elastic string. Note that, both

techniques can be used to produce the same result of creating a line segment. But, the rubber band method, gives more control and visual feedback to the user.

On the other hand, the interface design consists in the creation of a global architecture of interactive objects that reflects the internal state of the program, and allows the user to interact with its parameters. This is done through widgets and an interface manager.

Continuing with the previous example, the whole interface could be made of a set of buttons for creating, deleting, and modifying line segments. They would be associated with various interactive methods, such as rubber-banding and others.

3.2 Interaction Fundamentals

The graphics interaction is basically a process by which the user manipulate objects though various logical commands. Usually, this process involves the combination of graphics output devices, such as a graphics display, with graphics input devices such as a mouse.

At the core of the interaction we have a feedback loop such that the user actions are depicted on the screen, reflecting changes of state caused in these actions.

3.2.1 Graphical Feedback

The graphical feedback essentially couples input and output, such that the graphics objects involved behave like real and active entities to the user.

One important concept is that of an *event*, that in general is caused by an input action of the user, such as moving the mouse. The event should trigger a corresponding reaction in terms of graphics output, for example: when the mouse moves, the image of the cursor on the screen changes accordingly. In that way, the user knows that the system understood the gesture and also can see the current state of the parameters (in this case the mouse location relative to the screen).

3.2.2 Logical Input Elements

We have already seen in the previous chapter the main abstractions for device independent graphics output, and the basic mechanisms for events.

The next step is to develop the concept of *logical input elements*, that provide graphics input functions. They are:

- Locator
- Buttons
- Keys

As can be seen in Figure 3.1, these logical input elements are, usually associated with the mouse and keyboard. They interface with the `gp` graphics package through the function `gpevent`.

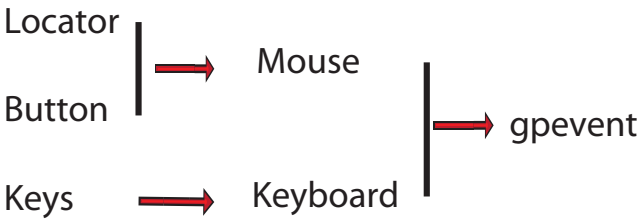

Figure 3.1 Logical Input Devices

The locator provides input for 2D coordinates relative to the window coordinate system. The buttons provide binary state values (i.e., pressed or released), while the key are associated with the ASCII character set.

3.2.3 Overview

The process of interface design entails the coupling of graphical input and output through some feedback implementation model and the construction of an architecture for an interface manager that coordinates interaction objects.

Feedback Implementation Models

The most common feedback implementation models are:

- Pooling
- Direct Event Handling
- Callbacks
- Boxed Callbacks

In the next sections, we are going to study these models in more detail.

Interface Manager Architectures

The architecture for interface design consists of a package that includes several elements for the construction of interactive programs.

The main components of an interface package are:

- Toolkit
- Interface Builder
- Run-time Manager

The toolkit contains a set of pre-packaged *widgets*, i.e., interface objects, for the various common interaction tasks, such as selecting an option from a menu or entering a text string. The interfce builder allow the user to graphically create the interface layout, while the run-time manager implements the feedback model during program execution.

In Section 3.5 we will present the architecture and implementation of a simple toolkit.

3.3 Interaction Mechanisms

In order to discuss and compare interaction mechanisms, we are going to show the pseudo-code of simple programs exemplifying their usage and implementation.

3.3.1 Non-interactive

The simplest graphics program is non-interactive. It's structure consists of an initialization to create a window and a sequence of drawing commands to display something on the screen

```
main()
{
  gpopen();
  gpwindow();
  gpviewport();
  // set gpattributes
  // execute drawing primitives
  .
  .
  gpclose();
}
```

3.3.2 Event Driven

The basic event driven interactive program uses the function `gpevent` to explicitly handle all graphics input and perform the associated output action.

```
main()
{
  gpopen();
  gpwindow();
  gpviewport();
  draw_initial_state();
  while (!quit) {
        e = gpevent();
        parse_exec_event(e) ;
  }
  gpclose();
}
```

Notice that the major implementation burden fall onto the the function `parse_exec_event`, which is responsible to explicitly handle all interaction.

```
parse_exec_event(e)
{
  switch (e) {
  case k:  // key pressed
       .
       .
  case m: // mouse movement
       .
       .
 }
}
```

As the program gets more complex and the interface more involved, this model becomes very difficult to extend and maintain. The reason is that each input event must be handled explicitly taking into account the affected objects and the state of the program. For example, when a key is pressed, it may have different meanings depending on where the mouse is located or which object is selected.

3.3.3 Callback Model

The callback model comes to the rescue of the difficulty presented by direct handling of input events. It uses events, but associates particular events to specific graphical objects or interface conditions.

For example, a particular function can be associated to a left mouse button press action. Under this model, that function is called whenever the mouse button is pressed, thus it is referred as *callback* function.

```
main()
{
  gpopen();
  gpwindow();
  gpregister("b1+", f1, d1);
  gpregister("b2+", f2, d2);
    .
    .
  gpmain_loop ()
  gpclose()
}
```

So, in the initialization fase, the user defines all callback actions through the function `gpregister`. Subsequently, the interaction loop is implemented by the function `gpmain_loop` that handles the events automatically by calling the desired actions at the appropriate times. In this way, the behavior of the interation can be changed by simply replacing the implementation of callbacks.

3.3.4 Callback with Multiple Views

The callback model can be greatly improved by establishing a link between the events and graphical objects. Note that in the general callback model, the event association is global, i.e., the same callback is activated for a particular class of event, such as a mouse button press.

The callback with multiple views model associates a local event to an action. For example, a different callback is activated depending on where the mouse button is pressed.

This model is implemented with the help of *multiple views*. The screen is tiled with different areas and they different local actions.

For example, the function

```
mvreg(1,"b1+",displ1,id1);
```

specifies that the callback `displ1(id1)` will be activated if the mouse button 1 is pressed in the screen area `v1`, while the function

```
mvreg(2,"b1+",displ2,id2);
```

specifies a similar action `displ2` for the screen area `v2`.

Of course, in this model is possible to maintain global events. This is done by a special identifier (-1) for the whole screen.

```
mvreg(-1,"=q",exit,0);  //  call exit(0) if key 'k' is pressed in any area
```

The callback with multiple views is the model that we are going to adopt to build our tookit infrastructure. Under this model the structure of an interactive program is as follows:

```
main()
{
 gpopen()
 mvopen()
 interface_setup()
 mvmain_loop ()
 gpclose(0)
}
```

The configuration of the interface is done by the function `interface_setup` that defines each view area and the corresponding callbacks, as well as, the initial state of the interface.

```
interface_setup()
{
  mvviewport(1, x, x, y, y)
  .
  mvregister (1, , x, 0)
  .
  draw_initial_state()
}
```

In the next section we will describe the implementation of the for the multiple view callback model.

3.4 Interface Objects

Graphical interface objects can be created using the multiple view callback model discussed in the previous section. The multiple viewport framework allows interface objects to be associated with areas of the screen, while the callback framework makes these objects active by an event-driven graphical feedback.

The `mvcb` package provides an integrated implementation of these frameworks.

3.4.1 Multiple Viewports

The multiple viewport framework essentially provides a tiled screen manager on top of `gp`. This is done by implementing the abstraction of *multiple views*. Each view behaves exactly like the `gp` package, but confined to a particular screen area.

The internal state of `mv` consists of a set of views, each defined by a window and viewport. There is also the notion of a *current view*, to which all the `gp` commands apply.

40a ⟨*mv internal state* 40a⟩≡

```
static int      nv;             /* number of views */
static Box*     w;      /* windows */
static Box*     v;     /* viewports */
static int      current;    /* current view */
```

Defines:
`current`, used in chunks 40b, 42, and 44b.
`nv`, used in chunks 40, 41, 44b, and 45.
`v`, used in chunks 40, 41, 44–50, 53c, 56b, and 57a.
`w`, used in chunks 40–42, 44b, and 54–57.

The main control functions of `gp` are replicated in the `mv` package to encapsulate the corresponding functionality.

40b ⟨ *mv open* 40b⟩≡

```
int mvopen(int n)
{
 if (n<=0) return 0;
 v=(Box*) emalloc(n*sizeof(Box)); if (v==0) return 0;
 w=(Box*) emalloc(n*sizeof(Box)); if (w==0) return 0;
 nv=n;
 current=0;
 for (n=0; n<nv; n++) {
  w[n].xu =  w[n].yu = 1.0;
  mvwindow(n,0.0,1.0,0.0,1.0);
  mvviewport(n,0.0,1.0,0.0,1.0);
 }
 return 1;
}
```

Defines:
`mvopen`, used in chunk 52b.
Uses `current` 40a, `mvviewport` 41c, `mvwindow` 41b, `nv` 40a, `v` 40a, and `w` 40a.

41a ⟨*mv close* 41a⟩≡

```
void mvclose(void)
{
 efree(w);
 efree(v);
}
```

Defines:
`mvclose`, used in chunk 53a.
Uses `v` 40a and `w` 40a.

41b ⟨*mv window* 41b⟩≡

```
void mvwindow(int n, real xmin, real xmax, real ymin, real ymax)
{
 if (n<0 || n>=nv) return;
 w[n].xmin=xmin;
 w[n].xmax=xmax;
 w[n].ymin=ymin;
 w[n].ymax=ymax;
}
```

Defines:
`mvwindow`, used in chunks 40b and 54.
Uses `nv` 40a, `real` 60 60, and `w` 40a.

41c ⟨*mv viewport* 41c⟩≡

```
void mvviewport(int n, real xmin, real xmax, real ymin, real ymax)
{
 if (n<0 || n>=nv) return;
 v[n].xmin=xmin;
 v[n].xmax=xmax;
 v[n].ymin=ymin;
 v[n].ymax=ymax;
}
```

Defines:
`mvviewport`, used in chunks 40b, 43, and 54.
Uses `nv` 40a, `real` 60 60, and `v` 40a.

42a ⟨*mv clear* 42a⟩≡

```
void mvclear(int c)
{
 int old=gpcolor(c);
 int n=current;
 gpbox(w[n].xmin,w[n].xmax,w[n].ymin,w[n].ymax);
 gpcolor(old);
}
```

Defines:
`mvclear`, never used.
Uses `current` 40a and `w` 40a.

The auxiliary function `mvframe` draws an outline around the view making it easier to see its area on the screen.

42b ⟨*mv frame* 42b⟩≡

```
void mvframe(void)
{
 int n = current;
 gpline(w[n].xmin,w[n].ymin,w[n].xmax,w[n].ymin);
 gpline(w[n].xmax,w[n].ymin,w[n].xmax,w[n].ymax);
 gpline(w[n].xmax,w[n].ymax,w[n].xmin,w[n].ymax);
 gpline(w[n].xmin,w[n].ymax,w[n].xmin,w[n].ymin);
}
```

Defines:
`mvframe`, used in chunk 56a.
Uses `current` 40a and `w` 40a.

The function `mvdiv` divides a rectangular area of the screen into a tiling of nx by ny views.

43 ⟨*mv divide* 43⟩≡

```
void mvdiv(int nx, int ny, real xvmin, real xvmax, real yvmin, real yvmax)
{
 int i,n;
 real dx=(xvmax-xvmin)/nx;
 real dy=(yvmax-yvmin)/ny;
 for (n=0,i=0; i<ny; i++)
 {
  int j;
  real ymax=yvmax-i*dy;
  real ymin=ymax-dy;
  for (j=0; j<nx; j++,n++) {
   real xmin=xvmin+j*dx;
   real xmax=xmin+dx;
   mvviewport(n,xmin,xmax,ymin,ymax);
  }
 }
}
```

Defines:
`mvdiv`, used in chunk 44a.
Uses `mvviewport` 41c and `real` 60 60.

The function `mvmake` applies `mvdiv` to the whole screen area.

44a ⟨*mv make* 44a⟩≡

```
void mvmake(int nx, int ny)
{
 real x,y;
 if (nx>ny) {
  x=1.0;
  y=((real)ny)/nx;
 } else {
  x=((real)nx)/ny;
  y=1.0;
 }
 mvdiv(nx,ny,0.0,x,0.0,y);
 gpviewport(0.0,x,0.0,y);
}
```

Defines:
`mvmake`, never used.
Uses `mvdiv` 43 and `real` 60 60.

The function `mvact` makes the specified view active, i.e., it becomes the current view.

44b ⟨*mv activate* 44b⟩≡

```
int mvact(int n)
{
 int old=current;
 if (n<0 || n>=nv) return old;
 gpwindow(w[n].xmin,w[n].xmax,w[n].ymin,w[n].ymax);
 gpviewport(v[n].xmin,v[n].xmax,v[n].ymin,v[n].ymax);
 current=n;
 return old;
}
```

Defines:
`mvact`, used in chunks 45 and 56a.
Uses `current` 40a, `nv` 40a, `v` 40a, and `w` 40a.

3.4.2 Callback with VIews

The callback model is implemented for multiple views bu creating a mechanism associating events with views.

For this purpose the function `mvevent` is defined.

45 ⟨*mv event* 45⟩≡

```
char* mvevent(int wait, real* x, real* y, int* view)
{
 int n; real gx,gy, tx,ty;
 char* r=gpevent(wait,&gx,&gy);
 if (r==NULL) return r;
 gpview(&gx,&gy); tx=gx; ty=gy;
 gpwindow(0.0,1.0,0.0,1.0);
 gpviewport(0.0,1.0,0.0,1.0);
 gpunview(&gx,&gy);
 *view=-1;
 for (n=0; n<nv; n++) {
  if (gx>=v[n].xmin && gx<=v[n].xmax && gy>=v[n].ymin && gy<=v[n].ymax) {
   int old=mvact(n);
   gpunview(&tx,&ty);
   *x=tx;
   *y=ty;
   *view=n;
   mvact(old);
   break;
  }
 }
 return r;
}
```

Defines:
`mvevent`, used in chunk 49a.
Uses `mvact` 44b, `nv` 40a, `real` 60 60, and `v` 40a.

The callback abstraction is implemented through a list of events patterns that are matched to views.

46 ⟨*mv callbacks state* 46⟩≡

```
typedef struct event    Event;

struct event {
 int v;
 char* s;
 MvCallback* f;
 void* d;
 Event* next;
};

static Event*   firstevent=NULL;
static int      gp_wait=1;
```

Defines:
`firstevent`, used in chunks 47 and 48.
`gp_wait`, used in chunks 48 and 49a.
Uses `next` 47 60 and `v` 40a.

For convenience we will define the following macros:

47 ⟨*mvcb macros* 47⟩≡

```
#define new(t)                  ( (t*) emalloc(sizeof(t)) )
#define streq(x,y)              (strcmp(x,y)==0)
#define V(_)                    ((_)->v)
#define S(_)                    ((_)->s)
#define F(_)                    ((_)->f)
#define D(_)                    ((_)->d)
#define next(_)                 ((_)->next)
#define foreachevent(e)         for (e=firstevent; e!=NULL; e=next(e))

static Event*   findevent       (int v, char* s);
static Event*   matchevent      (int v, char* s);
static int      match           (char *s, char *pat);
```

Defines:
D, used in chunks 48 and 49a.
F, used in chunks 48 and 49a.
findevent, used in chunk 48.
foreachevent, used in chunks 49b and 50a.
matchevent, used in chunk 49a.
new, used in chunks 48 and 62.
next, used in chunks 46, 48, and 62.
S, used in chunks 48–50.
streq, used in chunk 49b.
V, used in chunks 48–50.
Uses firstevent 46, match 50b, and v 40a.

The function `mvregister` associates a callback action to a particular event and view.

48 ⟨*mv register function* 48⟩≡

```
MvCallback* mvregister(int v, char* s, MvCallback* f, void* d)
{
 MvCallback* old;
 Event* e=findevent(v,s);
 if (e==NULL) {
  static Event* lastevent=NULL;
  e=new(Event);                          /* watch out for NULL! */
  V(e)=v;
  S(e)=s;
  F(e)=NULL;
  next(e)=NULL;
  if (firstevent==NULL) firstevent=e; else next(lastevent)=e;
  lastevent=e;
 }
 old=F(e);
 F(e)=f;
 D(e)=d;
 if (s[0]=='i' && f!=NULL) gp_wait=0;
 return old;
}
```

Defines:
`mvregister`, used in chunks 52b and 55b.
Uses `D` 47, `F` 47, `findevent` 47 49b, `firstevent` 46, `gp_wait` 46, `new` 47 60, `next` 47 60, `S` 47, `V` 47, and `v` 40a.

The `mvmainloop` is the function that actually implements the run-time callback model matching events to views.

49a ⟨*mv mainloop* 49a⟩≡

```
void mvmainloop(void)
{
 for (;;) {
  real x,y;
  int v;
  char* s=mvevent(gp_wait,&x,&y,&v);
  Event*e=matchevent(v,s);
  if (e!=NULL && F(e)(D(e),v,x,y,s))
    break;
 }
}
```

Defines:
`mvmainloop`, used in chunk 53b.
Uses `D` 47, `F` 47, `gp_wait` 46, `matchevent` 47 50a, `mvevent` 45, `real` 60 60, and `v` 40a.

The functions `findevent` and `matchevent` are used to query the list of event patterns when an event is processed.

49b ⟨*find event* 49b⟩≡

```
static Event* findevent(int v, char* s)
{
 Event* e;
 foreachevent(e) {
  if (V(e)!=v) continue;
  if (s==NULL && S(e)==NULL) break;
  if (s==NULL || S(e)==NULL) continue;
  if (streq(S(e),s)) break;
 }
 return e;
}
```

Defines:
`findevent`, used in chunk 48.
Uses `foreachevent` 47, `S` 47, `streq` 47, `V` 47, and `v` 40a.

50a ⟨*match event* 50a⟩≡

```
static Event* matchevent(int v, char* s)
{
 Event* e;
 foreachevent(e) {
   if (V(e)<0 || V(e)==v)
     if (match(S(e),s)) break;
 }
 return e;
}
```

Defines:
`matchevent`, used in chunk 49a.
Uses `foreachevent` 47, `match` 50b, `S` 47, `V` 47, and `v` 40a.

The actual pattern matching of strings is done by the auxiliary function `match`.

50b ⟨*match string* 50b⟩≡

```
static int match(char *s, char *pat)
{
 if (s==NULL) return pat==NULL;
 if (pat==NULL) return s==NULL;
 for (; *s!=0; s++, pat++) {
  if (*s!=*pat) return 0;
 }
 return 1;
}
```

Defines:
`match`, used in chunks 47 and 50a.

3.5 Toolkits

The `tk` toolkit package builds on top of the `mvcb` package to create interface objects (i.e., widgets). In this way, rectangular areas of the screen are associated with such objects and the `tk` library implements the proper feedback for each type of widget. This is done by registering specific callbacks for each active widget.

For example, a *pushbutton* widget will have as a state a binary value (on / off) and it will be materialized as a box on the screen with a text over a black or white background, depending on the current value. Every time the user clicks on the button, it changes state and the callback function informs the user program that the value has changed. Notice that the graphical feedback is handled automatically by the widget.

In summary, the toolkit creates a layer of abstraction that implements basic interface objects to be used by the application program.

3.5.1 Basic Elements

The central issue in the design of an interface toolkit is the definition of the set of widgets to be implemented and the mechanisms for creating new widgets.

Here we are going to suggest a minimum set of widgets that implement the essential functionality of a general user interface.

The minimum toolkit is composed of the following widgets: (a) Button; (b) Slider; (c) Selection; (d) Text area; and (e) Graphics canvas. A simple graphical depiction of each widget is shown in Figure 3.2.

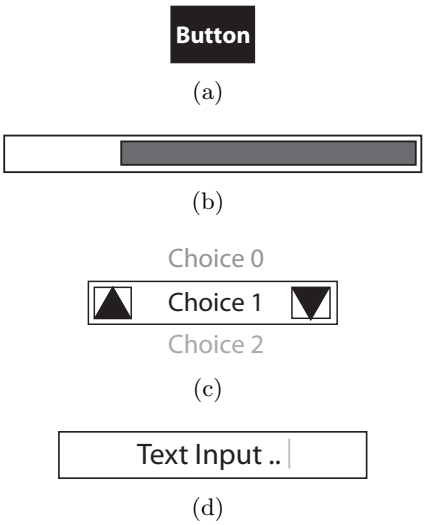

Figure 3.2 Essential Widgets: (a) Button; (b) Slider; (c) Selection; (d) Text Area.

3.5.2 The TK package

The widget API consists of functions for creating/destroying widget instances, mapping and unmapping them on the screen. These functions are:

```
w = create_widget (pos, par, fun)
destroy_widget (w)
map_widget (w)
unmap_widget (w)
```

The internal state of the package has a vector of widget pointers, the size of the vector, and the last available entry in the vector.

52a ⟨*tk local state* 52a⟩≡

```
Widget **wa = NULL;
int wn = 0;
int wi = 0;
```

Defines:
`wa`, used in chunks 52–54.
`wi`, used in chunks 52–54.
`wn`, used in chunks 52–54.
Uses `Widget` 57b.

The basic functionality of the run-time interface manager is implemented through the functions `tk_open`, `tk_close` and `tk_mainloop`, which respectively initializes the interface, terminates the interface and handles the interaction loop.

52b ⟨*tk initialization* 52b⟩≡

```
void tk_open(int n)
{
  int i;
  mvopen(n);
  wa = NEWARRAY(n, Widget *);
  for (i=0; i<n; i++)
    wa[i] = NULL;
  wn = n;
  wi = 0;
  mvregister(-1,"r",tk_redraw,NULL);
  gpflush();
}
```

Defines:
`tk_open`, never used.
Uses `mvopen` 40b, `mvregister` 48, `tk_redraw` 53c, `wa` 52a, `wi` 52a, `Widget` 57b, and `wn` 52a.

53a ⟨*tk close* 53a⟩≡

```
void tk_close()
{
  efree(wa);
  wa = NULL; wn = wi = 0;
  mvclose();
}
```

Defines:
`tk_close`, never used.
Uses `mvclose` 41a, `wa` 52a, `wi` 52a, and `wn` 52a.

53b ⟨*tk main loop* 53b⟩≡

```
void tk_mainloop()
{
  mvmainloop();
}
```

Defines:
`tk_mainloop`, never used.
Uses `mvmainloop` 49a.

The function `tk_redraw` is used to display all the current active widgets on screen.

53c ⟨*tk redraw* 53c⟩≡

```
int  tk_redraw(void* p, int v, real x, real y, char* e)
{
  int i;
  fprintf(stderr, "redraw\n"); fflush(stderr);
  for (i=0; i<wi; i++) {
    switch (wa[i]->type) {
    case TK_BUTTON:
      button_draw(wa[i], 1); break;
    default:
      error("tk"); break;
    }
  }
  gpflush();
  return 0;
}
```

Defines:
`tk_redraw`, used in chunk 52b.
Uses `button_draw` 56a, `real` 60 60, `redraw` 62, `TK_BUTTON`, `v` 40a, `wa` 52a, and `wi` 52a.

A new widget is instantiated by calling the function `tk_widget` and specifying its type and parameters.

54 ⟨*tk widget* 54⟩≡

```
Widget* tk_widget(int type, real x, real y, int (*f)(), void *d)
{
  Widget *w = widget_new(type, x, y, 0.2, f);
  if (wi >= wn)
    error("tk");
  w->id = wi++;
  wa[w->id] = w;
  mvwindow(w->id, 0, 1, 0, 1);
  mvviewport(w->id, w->xo, w->xo + w->xs, w->yo, w->yo + w->ys);
  switch (type) {
  case TK_BUTTON:
    button_make(w, d); break;
  default:
    error("tk"); break;
  }
  return w;
}
```

Defines:
`tk_widget`, never used.
Uses `button_make` 55b, `mvviewport` 41c, `mvwindow` 41b, `real` 60 60, `TK_BUTTON`, `w` 40a, `wa` 52a, `wi` 52a, `Widget` 57b, `widget_new` 55a, and `wn` 52a.

The internal function `widget_new` creates a generic widget object that should subsequently be bound to a specific widget class.

55a ⟨*new widget* 55a⟩≡

```
Widget* widget_new(int type, real x, real y, real s, int (*f)())
{
  Widget *w = NEWSTRUCT(Widget);
  w->id = -1;
  w->type = type;
  w->xo = x; w->yo = y;
  w->xs = w->ys = s;
  w->f = f;
  w->d = NULL;
  return w;
}
```

Defines:
`widget_new`, used in chunk 54.
Uses `real` 60 60, `w` 40a, and `Widget` 57b.

A new widget class is defined into the `tk` framework by specifying functions for creation and drawing, as well as the interaction mechanism which is handled through callbacks under the `mvcb` package.

As an example of creation of a new widget class, we are going to show ho to define a button widget. This is done through the functions `button_make` and `button_draw`.

55b ⟨*make button* 55b⟩≡

```
void button_make(Widget *w, char *s)
{
  mvregister(w->id,"b1+",button_pressed,w);
  mvregister(w->id,"b1-",button_released,w);
  w->d = s;
  button_draw(w, 1);
}
```

Defines:
`button_make`, used in chunk 54.
Uses `button_draw` 56a, `button_pressed` 56b, `button_released` 57a, `mvregister` 48, `w` 40a, and `Widget` 57b.

56a ⟨*draw button* 56a⟩≡

```
void button_draw(Widget *w, int posneg)
{
  char *label = w->d;
  int fg, bg;
  if (posneg) {
    fg = 1; bg = 0;
  } else {
    fg = 0; bg = 1;
  }
  mvact(w->id);
  gpcolor(fg);
  gpbox(0., 1., 0., 1.);
  gpcolor(bg);
  gptext(.2, .2, label, NULL);
  mvframe();
  gpflush();
}
```

Defines:
  `button_draw`, used in chunks 53c and 55–57.
Uses `mvact` 44b, `mvframe` 42b, `w` 40a, and `Widget` 57b.

The button behavior is defined through the callbacks `button_pressed` and `button_released` which handle respectively the events button press and release.

56b ⟨*press action* 56b⟩≡

```
int  button_pressed(void* p, int v, real x, real y, char* e)
{
  button_draw(p, 0);
  return 0;
}
```

Defines:
  `button_pressed`, used in chunk 55b.
Uses `button_draw` 56a, `real` 60 60, and `v` 40a.

57a ⟨*release action* 57a⟩≡

```
int  button_released(void* p, int v, real x, real y, char* e)
{
  Widget *w = p;
  button_draw(w, 1);
  return w->f();
}
```

Defines:
`button_released`, used in chunk 55b.
Uses `button_draw` 56a, `real` 60 60, `v` 40a, `w` 40a, and `Widget` 57b.

A widget object is defined by a data structure that contains its id, type, position and size on screen, local data and an application callback function.

57b ⟨*widget data structure* 57b⟩≡

```
typedef struct Widget {
  int id;
  int type;
  real xo, yo;
  real xs, ys;
  void* d;
  int (*f)();
} Widget;
```

Defines:
`Widget`, used in chunks 52 and 54–57.
Uses `real` 60 60.

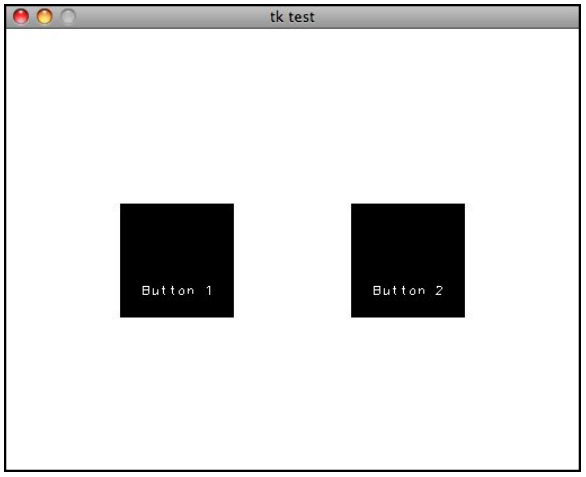

Figure 3.3 Example of interactive program using TK

3.5.3 Example

As an example of a graphics interactive program that uses the `tk` toolkit to generate its interface we show below a simple application that creates two buttons on screen, one for printing a value and the another for quitting the program. Figure 3.3 shows the interface layout of the program.

```
int main(int argc, char* argv[])
{
  Widget *w0;
  gpopen("tk test", 512, 512);
  tk_open(10);
  tk_widget(TK_BUTTON, .2, .5, but1, "Button 1");
  tk_widget(TK_BUTTON, .6, .5, but2, "Button 2");
  tk_mainloop();
  tk_close();
  gpclose(0);
}

int but1()
{
  fprintf(stderr, "Button 1 pressed\n"); fflush(stderr);
  return 0;
}
```

```
int but2()
{
  fprintf(stderr, "Button 2 pressed - quitting\n"); fflush(stderr);
  return 1;       // exits the main loop when 1 is returned.
}
```

3.6 Polygon Line Editor

As an example of the use of a graphics canvas, we show in this section the implementatio of a polygon line editor application.

Note that the program implements a rubber banding method for line input as discussed in the introduction of this chapter.

The screen of the program is depicted in Figure 3.4.

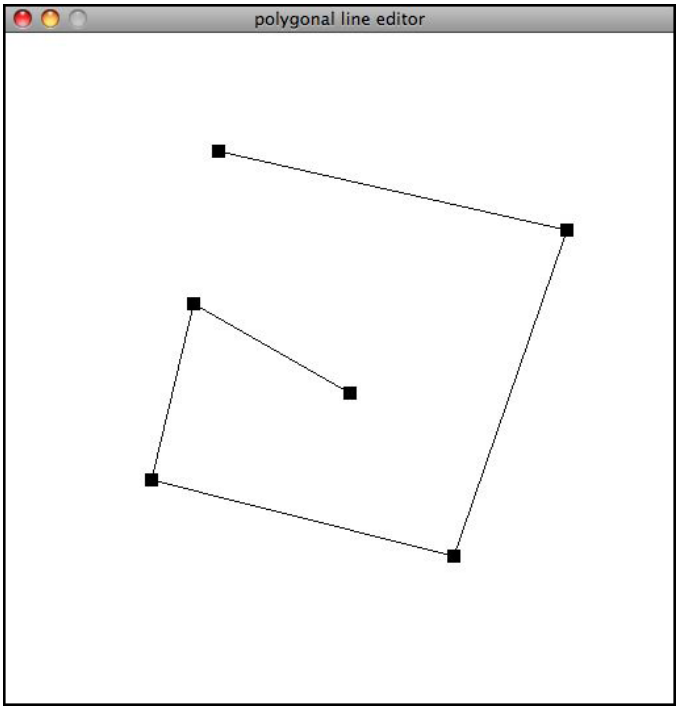

Figure 3.4 Polygon Line Editor

60 ⟨*ple state* 60⟩≡

```
#define TOL     tol

typedef struct point    Point;

struct point {
 real   x,y;
 Point* next;
 Point* prev;
};

void    redraw          (int clear);
void    delpoints               (void);
void    showpolygon     (void);
void    showspline      (void);
void    showpoints      (void);
void    addpoint                (real x, real y);
void    movepoint       (real x, real y);
void    delpoint                (real x, real y);
void    startmove       (real x, real y);
void    endmove         (real x, real y);
void    showchange      (Point* p, int c);
void    showpoint       (Point* p);
void    showside                (Point* p, Point *q);
Point* findpoint        (real x, real y);

Callback
 do_polygon,
 do_quit,
 do_redraw,
 do_addpoint,
 do_startmove,
 do_endmove,
 do_delpoint,
 do_movepoint;

#define X(p)            ((p)->x)
#define Y(p)            ((p)->y)
```

```
#define new(t)  ((t*)emalloc(sizeof(t)))
#define next(p) ((p)->next)
#define prev(p) ((p)->prev)

static Point*           firstpoint=NULL;
static Point*           lastpoint=NULL;
static Point*             moving=NULL;
static int                      showingpolygon=1;
static int                      showingpoints=1;
static real                     xmin = 0, xmax = 1, ymin = 0, ymax = 1;
static real                     aspect = 1, tol = 0.1;
```

Defines:
findpoint, never used.
firstpoint, used in chunk 62.
lastpoint, used in chunk 62.
moving, used in chunk 62.
new, used in chunks 48 and 62.
next, used in chunks 46, 48, and 62.
prev, used in chunk 62.
real, used in chunks 21b, 22, 24–27, 29, 30, 41, 43–45, 49a, 53–57, and 62.
showingpoints, used in chunk 62.
showingpolygon, used in chunk 62.
TOL, used in chunk 62.
X, used in chunk 62.
Y, used in chunk 62.

Uses addpoint 62, delpoint 62, delpoints 62, do_addpoint 62, do_delpoint 62, do_endmove 62, do_movepoint 62, do_polygon 62, do_quit 62, do_redraw 62, do_startmove 62, endmove 62, movepoint 62, redraw 62, showchange 62, showpoint 62, showpoints 62, showpolygon 62, showside 62, and startmove 62.

62 ⟨*ple functions* 62⟩≡

```
int main(int argc, char* argv[])
{
 gpopen("polygonal line editor", 512 * aspect, 512);
 gpwindow(xmin,xmax, ymin,ymax);

 gpmark(0,"B"); /* filled box mark */

 gpregister("kp",do_polygon,0);
 gpregister("kq",do_quit,0);
 gpregister("kr",do_redraw,0);
 gpregister("k\f",do_redraw,0);
 gpregister("b1+",do_addpoint,0);
 gpregister("kd",do_delpoint,0);
 gpregister("b3+",do_startmove,0);
 gpregister("b3-",do_endmove,0);
 gpregister("m3+",do_movepoint,0);

 gpmainloop();
 gpclose(0);
}

void redraw(int clear)
{
 if (clear)
   gpclear(0);
 if (showingpolygon)
   showpolygon();
 showpoints();
 gpflush();
}

void delpoints(void)
{
 firstpoint=lastpoint=NULL;     /* lazy! */
}

void addpoint(real x, real y)
{
```

```
 Point* p=new(Point);
 X(p)=x;
 Y(p)=y;
 next(p)=NULL;
 if (showingpoints) showpoint(p);
 if (firstpoint==NULL) {
  prev(p)=NULL;
  firstpoint=p;
 } else {
  prev(p)=lastpoint;   next(lastpoint)=p;
  if (showingpolygon) showside(lastpoint,p);
 }
 lastpoint=p;
}

void delpoint(real x, real y)
{
 Point* p=findpoint(x,y);
 if (p!=NULL) {
  if (prev(p)==NULL) firstpoint=next(p); else next(prev(p))=next(p);
  if (next(p)==NULL) lastpoint=prev(p);  else prev(next(p))=prev(p);
  redraw(1);
 }
}

void startmove(real x, real y)
{
 moving=findpoint(x,y);
 if (moving!=NULL) {
  x=X(moving); y=Y(moving);
  gpcolor(0);    gpplot(x,y); gpcolor(1);
  gpmark(0,"b"); gpplot(x,y);
 }
}

void movepoint(real x, real y)
{
 if (moving!=NULL) {
  showchange(moving,0);
```

```
  X(moving)=x; Y(moving)=y;
  showchange(moving,1);
 }
 else startmove(x,y);
}

void endmove(real x, real y)
{
 if (moving!=NULL) {
  gpmark(0,"B");
  redraw(0);
  moving=NULL;
 }
}

Point* findpoint(real x, real y)
{
 Point* p=firstpoint;
 for (p=firstpoint; p!=NULL; p=next(p)) {
  if ((fabs(X(p)-x)+fabs(Y(p)-y))<TOL) break;
 }
 return p;
}

void showpoints(void)
{
 Point* p;
 for (p=firstpoint; p!=NULL; p=next(p))
  showpoint(p);
 gpflush();
}

void showpolygon(void)
{
 Point* p;
 for (p=firstpoint; p!=NULL; p=next(p))
  showside(p,next(p));
 gpflush();
}
```

```
void showpoint(Point* p)
{
 gpplot(X(p),Y(p));
}

void showside(Point* p, Point *q)
{
 if (p!=NULL && q!=NULL) gpline(X(p),Y(p),X(q),Y(q));
}

void showchange(Point* p, int c)
{
 gpcolor(c);
 showpoint(p);
 if (showingpolygon) {
  showside(prev(p),p);
  showside(p,next(p));
 }
 gpflush();
}

int do_clear(char* e, real x, real y, void* p)
{
 delpoints();
 redraw(1);
 return 0;
}

int do_polygon(char* e, real x, real y, void* p)
{
 showingpolygon=!showingpolygon;
 redraw(1);
 return 0;
}

int do_quit(char* e, real x, real y, void* p)
{
 return 1;
```

```
}

int do_redraw(char* e, real x, real y, void* p)
{
 redraw(1);
 return 0;
}

int do_addpoint(char* e, real x, real y, void* p)
{
 addpoint(x,y);
 gpflush();
 return 0;
}

int do_startmove(char* e, real x, real y, void* p)
{
 startmove(x,y);
 gpflush();
 return 0;
}

int do_endmove(char* e, real x, real y, void* p)
{
 endmove(x,y);
 gpflush();
 return 0;
}

int do_delpoint(char* e, real x, real y, void* p)
{
 delpoint(x,y);
 gpflush();
 return 0;
}

int do_movepoint(char* e, real x, real y, void* p)
{
 movepoint(x,y);
```

```
  gpflush();
  return 0;
}
```

Defines:
addpoint, used in chunk 60.
delpoint, used in chunk 60.
delpoints, used in chunk 60.
do_addpoint, used in chunk 60.
do_clear, never used.
do_delpoint, used in chunk 60.
do_endmove, used in chunk 60.
do_movepoint, used in chunk 60.
do_polygon, used in chunk 60.
do_quit, used in chunk 60.
do_redraw, used in chunk 60.
do_startmove, used in chunk 60.
endmove, used in chunk 60.
findpoint, never used.
main, used in chunks 412, 417, and 419.
movepoint, used in chunk 60.
redraw, used in chunks 53c and 60.
showchange, used in chunk 60.
showpoint, used in chunk 60.
showpoints, used in chunk 60.
showpolygon, used in chunk 60.
showside, used in chunk 60.
startmove, used in chunk 60.

Uses firstpoint 60, lastpoint 60, moving 60, new 47 60, next 47 60, prev 60, real 60 60, showingpoints 60, showingpolygon 60, TOL 60, X 60, and Y 60.

3.7 Review

In this chapter we presented an architecture for interface design that has four layers, as show in Figure 3.5. The first layer is the graphics interactive program; the second layer is the interface toolkit, implemented by the `tk package` and the `mvcb` library. The third layer is the graphical input and output, implemented by the `gp` package. The fourth layer is the window system, which is plataform dependent, for example, X11 in the Linux platform, Vista in the Microsoft Windows platform and Aqua / Cocoa for the MacOS X platform.

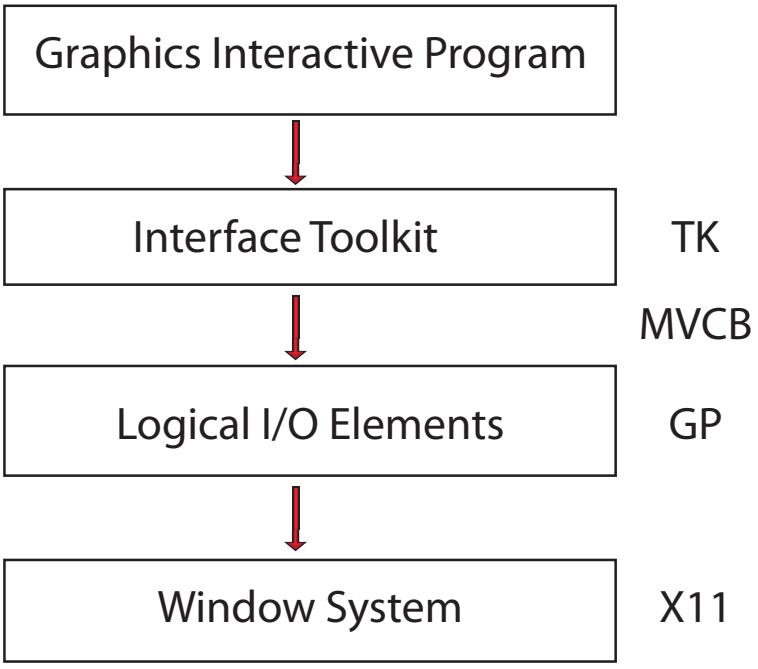

Figure 3.5 Implementation Layers

3.8 Comments and References

In this chapter, we presented the implementation of a library for interface design in Computer Graphics.

Some of the popular toolkit libraries are: QT, GTK, FLTK and Glui.

3.8.1 Summary

The external API of the MVCB library is composed of the following routines:

```
int     mvopen  (int n);
void    mvclose (void);
void    mvwindow(int n, real xmin, real xmax, real ymin, real ymax);
void    mvviewport(int n, real xmin, real xmax, real ymin, real ymax);
int     mvact   (int n);
void    mvclear (int c);
void    mvframe (void);
void    mvmake  (int nx, int ny);
void    mvdiv   (int nx, int ny, real xmin, real xmax, real ymin, real ymax);

char*        mvevent    (int wait, real* x, real* y, int* view);

void              mvmainloop(void);
MvCallback* mvregister(int v, char* s, MvCallback* f, void* d);
```

3.9 Exercises

3.1. Incorporate map and unmap operations tin the TK library.

3.2. Extend the TK library to include a slider widget.

3.3. Extend the TK library to include a choice widget.

3.4. Extend the TK library to include a text widget.