SIMPLIFYING GUI CONSTRUCTION BY EMBEDDED SCRIPTING

Lawrence A. Stabile

ABSTRACT

This article describes the InteractBox system, an object-oriented abstraction for defining common GUI items such as menus, radio boxes, and field editors. InteractBox is a single C++ class that defines a number of methods for creating graphical objects and combining them using various layout styles. This set of methods also defines an embedded scripting language. Variables may be passed by reference to the methods. The values of these variables are set based on actions taken by the user, which presents a simple and clear interface to the program.

The InteractBox system defines two derived classes: ControlBox (non-modal) and DialogBox (modal). ControlBox defines a single virtual function that the programmer supplies in a derived class. This function serves as the only callback in the system. When this function is invoked, it can test variables that have been passed by reference to determine what items have been selected and what data values need to be read. Reducing callbacks to a single function greatly simplifies the control interface.

InteractBox is built on top of the InterViews library. This paper describes the functionality of InteractBox, including an extended example taken from the field of real-time control.

Lawrence A. Stabile is a software architect at CenterLine Software, Inc., Cambridge, Massachusetts. *This work was performed while the author was at the Enterprise Computing Institute, Hopkinton, Massachusetts (formerly the Center for High-Performance Computing of Worcester Polytechnic Institute). This research is sponsored by the Advanced Research Projects Agency (ARPA), contract number DABT63-91-C-0016. The views and conclusions presented in this document are those of the author and do not represent the official policies, either expressed or implied, of ARPA or the United States government.*

INTRODUCTION

Many tools exist to assist in the construction of graphical user interfaces (GUIs). Such tools may be classified into two overall categories: programmatic tools that provide a set of language constructs, classes, or library routines; and visual tools that provide a GUI for the GUI construction, allowing the programmer to build an on-screen version of the intended interface. This paper focuses on GUI construction of the programmatic variety. Too often, GUI libraries make relatively simple items very tedious to program, extending the development time for a program by diverting too much energy toward interface issues, to the neglect of the intended program functionality. Visual GUI techniques are most useful when the GUI to be constructed is complex or has many aesthetic constraints that would force too many iterations when constructed purely via text-based languages.

The InteractBox system, described herein, was motivated by our experience with the InterViews package [Linton92]. InterViews comprises a set of C++ classes for defining and composing GUI objects. One can build menus, field editors, dialog boxes, and the like, and more complex objects by using lower-level features of the system. InterViews, while comprehensive, often makes relatively simple items difficult to construct. For example, a dialog box consisting of a few buttons and edit fields can require many lines of code and significant debugging time. Some of the complexities encountered in using InterViews are discussed by Bashir and Sternlicht.

InteractBox adds a layer on top of InterViews which abstracts away much of the tedium encountered in building sets of standard items. InteractBox defines an embedded scripting language via a set of method calls that describe the construction of GUI items, control their layout, and allow the passage of variables and function objects to effect an interface between the GUI items and the application semantics. The scripting language takes advantage of C++ overloading and type sensitivity to make the system flexible and easy to use. InteractBox is not intended as a full replacement for InterViews, but as a complement that simplifies the construction of often-used items.

InteractBox differs from other forms of scripting languages in that it is completely defined within the application programming language; that is, it is simply a set of statements and data structures defined in the programming language, which themselves form a self-contained description system. This is often referred to as an embedded language.

In contrast, for example, WCL (Widget Creation Library) [Smyth92] allows the programmer to specify graphical appearance in resource files, which are then interpreted by library code linked with the application program. Tcl/Tk [Ousterhout94] is a system for GUI design that relies on a separate scripting language (Tcl), in which several kinds of widgets may be specified. The scripting language is then interpreted by an application program to produce the desired interface. These are valuable methods of GUI development but still require a significant design cycle for common items. InteractBox provides its facilities completely within a programming language (C++), without sacrificing the conciseness or ease-of-use normally associated with separate scripting systems. Performance is also retained by providing the services at the programming language level.

The InteractBox system is part of the Insight project [Stabile93] [Stabile94], designed for the debugging and performance assessment of parallel and distributed systems. Insight itself is multithreaded, and it was required that InterViews accommodate that environment. Since a small portion of the functionality of InteractBox relies on a threaded environment, we briefly describe it in the Appendix to this article.

While InteractBox is not yet publicly available, we hope to make it so in late 1995. Currently InteractBox may be used only with the InterViews library.

INTERACTBOX FUNCTIONALITY

This section describes the functions available via the InteractBox system and illustrates their use by way of a simple example.

CONCEPTS

The InteractBox system consists of three C++ classes, InteractBox, DialogBox, and ControlBox. InteractBox is the top-level abstract base class and defines the methods that form the embedded scripting language. DialogBox provides a modal dialog box capability (control returns to the program only after the user has provided input), whereas ControlBox provides a non-modal system (control returns to the program immediately after window creation). Each utilizes the common scripting language defined by the InteractBox abstract base class.

Following are the definitions for the three classes. In this case, we have included only those methods in InteractBox that are needed for the current example; the following sections describe in detail all of the available methods.

```
class InteractBox
{
 public:
  InteractBox& operator() (const char*);
                // Field editor accepting integers.
  InteractBox& edit_field (const char* name, int& value);
  InteractBox& checkbox (const char* name, bool& value);
  InteractBox& end ();                    // Terminate an environment.
  InteractBox& window_name (const char*);  // Set name of created window.
      // Radio buttons. Button names follow as
      // operator () (char*); ordinal position of name
      // sets value. Terminate with end().
  InteractBox& radio_buttons (const char* name, int& value);
      // Callback for ControlBox class
  virtual void central_control () = 0;
      // Create and display window
  virtual bool show () = 0;
  // Many more methods...
```

```
};

class DialogBox: public InteractBox
{
 public:
  DialogBox ();
  virtual ~DialogBox ();
  bool show (Window* = 0);
  void central_control () {}  // Nop for this class
};

class ControlBox: public InteractBox
{
 public:
  ControlBox ();
  virtual ~ControlBox();
  bool show (Window* = 0);
  void central_control() = 0;  // Programmer must define in derived class
  bool pop_db (DialogBox*);
};
```

The bulk of the functionality is defined by `impl` class pointers in the private sections of the classes shown. The set of definitions required to use the system is thus very small.

The scripting language provided by InteractBox is simply a set of methods, whose sequence of invocations defines a script execution. Some methods define manipulable graphic objects, such as buttons or menus. These methods also accept data to define the visual appearance of the item, and/or by-reference variables that are set by the interface and utilized by the application program. Consider the following code, which shows a simple script that builds a (modal) dialog box consisting of a checkbox, a set of three radio buttons, and an integer edit field:

```
main()
{
  enum Color { red, green, blue };
  bool lights_on = false;
  Color color = red;
  int intensity = 10;

  DialogBox db;

  db.window_name ("Illumination Chooser");
  db.checkbox ("Lights On", lights_on);
  db.radio_buttons ("Color", (int&) color)
      ("Red")
      ("Green")
```

```
        ("Blue");
    db.end();
    db.edit_field ("Intensity", intensity);

    db.show();
    // Variables may now be accessed, e.g.,...
    cout << "Lights On = " << lights_on
         << "Color = " << color
         << "Intensity = " << intensity
         << endl;
}
```

Upon execution of `db.show()`, the graphic shown in Figure 1 is produced.

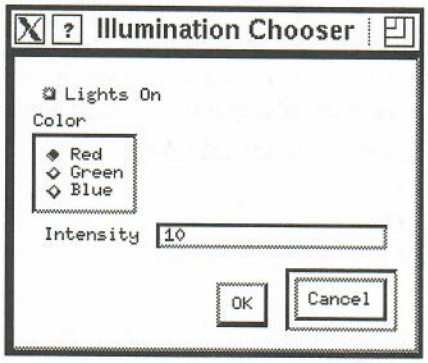

FIGURE 1: *ILLUMINATION CHOOSER DIALOG BOX*

When the user pushes the OK button after selecting the desired items, `InteractBox::show()` returns. At this point the variables passed to the script methods are set by-reference to the values selected by the user. The program can then use these values accordingly.

Let us examine this example in a little more detail. First, variables are declared for receiving the values of interest from the dialog box. These are given initial values that appear in the window. Next, we declare the dialog box itself (`db`). No window appears until we instantiate it with `db.show()`, later. The code between its declaration and `db.show()` sets attributes and establishes the data variables as the receivers of information from the user. `db` is next given a window name, "Illumination Chooser."

The establishment of the Lights On checkbox passes our first by-reference variable, `lights_on`. The system uses the initial value to set the screen appearance; in this case the variable is `false`, so the checkbox is shown as unset. The user may toggle the checkbox on the screen as often as desired; the setting takes effect only when the dialog box is closed (by pressing the OK button). This is in contrast to ControlBoxes, which cause an immediate action upon a screen event.

Next, we define a set of three radio buttons to select color. `db.radio_buttons` establishes the label for the graphic; the `color` variable is passed for the initial value and to receive the user's setting. We cast to an `int` reference to remove the need for programmer-defined type sensitivity in the DialogBox system itself. While we could have designed using templates, the extra complexity did not seem worth the small amount of extra type safety gained.

Note that no semicolon follows `db.radio_buttons ("Color", (int&) color)`. The `radio_buttons()` method, which returns an `InteractBox&`, sets the state of the `db` object to expect strings, via the overloaded `operator()`. Each string represents the name for a particular radio button. An `end()` method demarks the end of the definition. A radio button choice made by the user then sets the `int` reference variable to the ordinal position of the passed names, the first name being assigned zero. So, since Red is chosen in Figure 1, the variable `color` is assigned zero, or enum value `red`. Had the user chosen Green, `color` would be one, or enum value `green`, and so on.

The scripting language is defined to manipulate the state of the InteractBox system, even though some items, like the radio buttons, have a convenient "function call syntax." We thus could have written the radio button sequence with no semantic change, as follows:

```
db.radio_buttons ("Color", (int&) color);
      (*this)("Red");
      (*this)("Green");
      (*this)("Blue");
db.end();
```

Such flexibility is an important advantage of the system, since arbitrary code may be used in place of the constants (for example, loops to set up computed lists), as we will see in later sections.

The last field set up by the Illumination Chooser is an integer edit field, to receive an intensity. Once again, the initial value is used to show the graphic, and a value typed by the user then appears in the `intensity` variable, once the dialog box is closed.

The last call, `db.show()`, creates the dialog box window, as shown. The buttons OK and Cancel are built in. Pressing OK causes `db.show()` to return `true`; Cancel causes it to return `false`. Only a `true` return causes the by-reference data variables to take on the settings provided by the user. Upon a Cancel, the variables are left alone, and `db.show()` returns `false`.

Environments

Some of the InteractBox methods initiate a scope, which creates an environment that is active until an `end()` method is encountered. Environments have the following uses:

- Setting fonts, colors, and other visual attributes.
- Nesting layout control, such as defining vertical and horizontal boxes. For example, by default, InteractBox orients the specified items in a vertical manner. While this suffices for many simple cases, at times more control over layout is desired. For this

purpose several script methods are provided, which define enclosing environments to affect layout and appearance.

- Passing arguments of indefinite number. Several GUI items accept lists of labels or other indicators (see, for example, the radio button definition above).

Consider the following InteractBox methods:

```
InteractBox& vertical ();
InteractBox& horizontal ();
InteractBox& foreground (const char*);
```

`vertical()` orients all graphics vertically between it and a subsequent `end();` `horizontal()` has the obvious corresponding meaning. `foreground()` sets the foreground color.

Environments may be nested; the following example created a vertical stack of two sets of horizontal labels, with the "a" colored red:

```
vertical();
  horizontal();
    foreground ("red");
      label ("a");
    end();
    label ("b");
  end();
  horizontal();
    label ("c");
    label ("d");
  end();
end();
```

READOUTS

Up to this point we have shown how variables are used to get values from the user. The only time the programmer set them was to specify an initial value for a graphic. It is desirable to be able to display variables as well as set them; readouts provide this capability. An example of a readout is the InteractBox method `meter`:

```
InteractBox& meter (const char* name,
                    double& value,
                    double min_val, double max_val);
```

When invoked with the following bindings, this method displays an analog-style meter as illustrated in Figure 2.

```
name = "Balls per Minute"
value = 65
min_val = 0
max_val = 100
```

FIGURE 2: *ANALOG METER READOUT*

The meter label is given by `name`, and the minimum and maximum values by `min_val` and `max_val` (0 and 100, in this case). The value indicated by the needle (and shown numerically) is supplied by the reference variable `value`. Because it is a reference, the meter can respond to changes in the value of the variable during the running of the program. In the case of a simple scalar variable (as shown above), the InteractBox system uses a timer thread to sample the variable (the Appendix gives an overview of our thread architecture). In this manner any kind of simple variable can be passed to a readout, and its value is monitored with no further effort.

Similar capability is supplied by sampling functions, as shown in this form of the meter method:

```
InteractBox& meter (const char* name,
                    Function<double>* value,
                    double min_val, double max_val);
```

The `Function` template denotes functions of no arguments, returning type `double`. A functional object defines a method `operator()`, as follows:

```
template<class ReturnType> class Function
{
  virtual ReturnType operator() () = 0;
};
```

To use the `meter` method, the programmer supplies a function object, which the InteractBox system calls at each sampling time-point. It is the function's responsibility to compute and return the appropriate value. Sampling functions are useful when the desired value is based on a demand-driven model, that is, where the required value is not kept up-to-date in a variable by the application during its normal execution.

More complex applications may require a finer degree of refresh control. We currently provide this capability through special-purpose objects that act as interfaces between the application and the InteractBox system. The application can call a `refresh()` method to inform the InteractBox system that the object in question needs to be redrawn.

The meter method is illustrative of good use of C++ method overloading: all forms of the method have a similar structure, simplifying changes made by the programmer. For example, if the variable-based form was used, and later it was desired to change to the

functional form, all that needs to be done is to define the appropriate function object and pass it to the method in place of the variable. The remaining arguments, as well as the method name, are unchanged.

ControlBoxes

While modal dialog boxes are useful for requesting information, a non-modal interface is also needed, where activating graphic items causes some immediate program action. Such an object springs from the class ControlBox. A ControlBox is built in the same manner as a DialogBox, with the exception that the method `InteractBox::show()` returns immediately after displaying the window. The window remains visible until the user explicitly closes it.

ControlBox defines a pure virtual function, `ControlBox::central_control()`, which acts as the one and only callback in the class. Whenever any item in the ControlBox is changed by the user—a button pressed, a menu selected, etc.—`central_control()` is called. Variables passed by-reference to the fields that created the elements of the ControlBox are set accordingly and can then be tested by code defined in `central_control()`. Consider the InteractBox definition of the method button:

```
InteractBox& button (const char* name, bool& value);
```

We can create a button by invoking

```
bool a_button;
button ("A Button", a_button);
```

When the user presses the button graphic, `central_control()` is called. The variable `a_button` then assumes the value `true`; all other simple buttons have their variables set to `false` during this time. A simple cascade of `if-then-else` tests allows the checking of buttons and states in a single procedure. This is illustrated by the following example, which defines a panel of doorbells for a house containing three apartments:

```
class DoorBells: public ControlBox
{
  public:
    DoorBells ();
    void central_control ();
  private:
    bool apt_a_bell;
    bool apt_b_bell;
    bool apt_c_bell;
};

DoorBells::DoorBells ()
  : apt_a_bell (false), apt_b_bell (false), apt_c_bell (false)
{
  button ("Apt A -- Smith", apt_a_bell);
```

```
  button ("Apt B -- Jones", apt_b_bell);
  button ("Apt C -- Harrison", apt_c_bell);
  show();
};

void DoorBells::central_control ()
{
  if (apt_a_bell)
    // do Apt A bell action
  else if (apt_b_bell)
    // do Apt B bell action
  else if (apt_c_bell)
    // do Apt C bell action
};
```

When a `DoorBells` object is instantiated, a window appears containing the three defined buttons. Suppose the user selects the "Apt B -- Jones" button. `DoorBells::central_control()` is called. `apt_a_bell` and `apt_c_bell` are `false`, while `apt_b_bell` is `true`, so its action is selected.

While it may seem cumbersome to funnel all control through one procedure, we offer instead that it is a cleaner and more understandable method than the use of multiple individual callbacks. One problem is that callbacks as such do not work well in C++; they must really be objects upon which some method (for example, `operator()`) is invoked. While nothing is wrong with this, the setup can be painful when one has to implement a large number of buttons, menus, etc. Often, one's code is cluttered with callback functions that do nothing more than set a flag.

In addition, the ability to derive new classes from the ControlBox is enhanced by the fact that only a single virtual function need be defined by a programmer for the purpose of implementing all control operations.

A natural by-product of the by-reference variable method of communicating values is that both the modal DialogBox system, and the non-modal ControlBox system use the same data-passing style; it is only the control mechanism that changes from one to the other. Other items with state, such as checkboxes and radio buttons, operate similarly to simple buttons, but their states do not change upon the end of the event. Items that impart a continuous value, such as sliders, call `central_control()` whenever the user alters the graphic—for example, each slider movement causes a call and so provides immediate feedback.

The InteractBox system also refreshes any defined readouts after `central_control()` is called, so that the readouts may be updated immediately with any new values based on the user selection. For example, we can use this capability to produce a very compact definition for a meter that follows a slider position:

```
double slider_value = 5;
vertical();
  meter ("Slider Value", slider_value, 0, 10);
  slider (slider_value, 0, 10);
end();
```

This code defines a slider ranging from 0 to 10, and a meter above it for the same range. Simply by sharing the variable `slider_value`, the meter automatically follows the slider. Of course, `central_control()` is called whenever the user moves the slider so any application-specific action can be performed.

Glyph Objects

The method `glyph_object` allows the programmer to display pre-built InterViews objects as part of a set of InteractBox objects. In this case some features of InteractBox are lost, such as storage management, the use of environments, and the handling of by-reference variables.

```
InteractBox& glyph_object (Glyph*);
InteractBox& glyph_object (InteractBox*);
void replace (InteractBox* new_box);
```

Passing another InteractBox as a glyph object allows the programmer to control dynamically the kinds of items in an InteractBox display. For example, one may wish to provide a scrollable list of buttons to activate processes. The number of processes may change with time. So, a separate InteractBox object is provided for this list (call it `ProcessList`) and passed as a glyph object to its container. When a new list is required, `ProcessList::replace()` is called, passing it the new object. The old one deletes itself from the container and installs the new one.

Embedded Scripting Language

The scripting approach to GUI construction has several advantages:

- Few classes are required; the functionality is centrally defined. This also helps to reduce "include-file load."
- Class derivation for enhancement is easily supported and encouraged. Just one small include file is required, making InteractBox objects easy to insert wherever desired.
- Scripts are compact and easy to understand. Use of by-reference variables to pass information greatly reduces "callback clutter" and is a natural and simple interface.
- Scripts meld well with other program features; the full power of C++ is available, while preserving the perspicuity of the embedded language.

By defining an embedded scripting language, we abstract away many of the complexities of InterViews. Consider, for example, the definition of radio buttons. Using InterViews directly, code similar to the following is required:

```
class RadioChoice
{
 public:
  void radio_choice_0 ();
  void radio_choice_1 ();
  void radio_choice_2 ();
  int get_radio_choice ();
 private:
  int choice;
};

void RadioChoice::choice_0 () { choice = 0; }
void RadioChoice::choice_1 () { choice = 1; }
void RadioChoice::choice_2 () { choice = 2; }

int RadioChoice::get_choice () { return choice; }

declareActionCallback(RadioChoice)
implementActionCallback(RadioChoice)

main ()
{
  RadioChoice* rc = new RadioChoice;
  WidgetKit& wk = *WidgetKit::instance();
  LayooutKit& lk = *LayoutKit::instance();
  TelltaleGroup* group = new TelltaleGroup;
  PolyGlyph* box = lk.hbox();
  Button* c0  = wk.radio_button (group, "Choice 0",
                      new ActionCallback(RadioChoice)(rc,
                      &RadioiChoice::choice0));
  c0->state()->set (TelltaleState::is_chosen,
                    true);
  box->append (c0);
  box->append (lk.hspace (4.0));
  box->append (wk.radio_button (group, "Choice 1",
                    new ActionCallback (RadioChoice)(rc,
                    &RadioiChoice::choice1)));
  box->append (lk.hspace (4.0));
  box->append (wk.radio_button (group, "Choice 2",
                    new ActionCallback (RadioChoice)(rc,
                    &RadioiChoice::choice2)));
```

```
    box->append (lk.hspace (8.0));
    // ... Display box ...
    // ... Get Item Number from rc ...
    // ... etc ...
}
```

Several types of objects are required, as well as callback definitions for each possible radio button choice. Of course, code like this is what is really used "behind the scenes" in the InteractBox system; it is the goal of the system to hide this level of detail from the programmer.

The methods defined by InteractBox serve as the embedded scripting language; in the remainder of this section, we briefly describe all of the available methods.

CONTROLS

InteractBox provides the following controls:

```
    // Field editor accepting integers.
InteractBox& edit_field (const char* name, int& value);
    // Field editor accepting strings.
InteractBox& edit_field (const char* name, char*& value);
    // Field editor accepting doubles.
InteractBox& edit_field (const char*, double&);
InteractBox& checkbox (const char* name, bool& value);
enum ButtonStyle { LeftArrow, RightArrow, UpArrow, DownArrow };
InteractBox& button (const char* name, bool& value);
InteractBox& button (const char* name, ButtonStyle, bool& value);
    // Horizontal slider
InteractBox& slider (double& value, double min_val, double max_val);
InteractBox& slider (double& value, double min_val, double max_val,
  // Horizontal slider; sets value, accepts fractions of min/
InteractBox& slider (double& value, double min_val, double max_val,
                     double large_scroll_frac,
                     double small_scroll_frac);
  // Vertical slider
InteractBox& vertical_slider (double&, double min_val, double max_val);
InteractBox& vertical_slider (double&, double min_val, double max_val,
                              double large_scroll_frac,
                              double small_scroll_frac);
  // Radio buttons. Button names follow as operator () (char*);
  // ordinal position of name sets value. Terminate with end().
InteractBox& radio_buttons (const char* name, int& value);
  // Menus.
  // Menu methods accept a sequence of arguments of either
  //    (1) a name for the entry or
  //    (2) a pair (name, state), where state is a bool.
```

```
    // In case (2) a menu with a checkbox state is created.
  InteractBox& menu (const char* name, int& value);
    // Pulldown menu. Entry names follow as operator () (char*)
    // ordinal position of name sets value. Terminate with end().
    // Used within a menu() method to create a pull-right menu
    // with the given name. Parent menu() method
    // defines value; names added here are added to that.
  InteractBox& pullright (const char* name);
```

Edit fields come in three varieties, for entering `int`, `double`, and `string` values. By using type-sensitive overloading, the programmer need not be concerned with explicit conversion from characters to the desired format. When using edit fields in a ControlBox, `central_control()` is called when the user presses Return in the edit area.

The checkbox, as described earlier, presents a simple Boolean state toggle.

Buttons either are of the usual rectangular variety or may be of one of four arrow styles, as indicated by the `ButtonStyle` enum.

Sliders may be horizontal or vertical. A double value is set, which is in the range `min_val` to `max_val`. The versions that accept scroll fractions allow the programmer to specify how much of the range is traversed when the main scroll area is selected (`large_scroll_frac`) or when a scroll arrow is selected (`small_scroll_frac`).

Radio buttons, as described earlier, maintain a value based on the ordinal position of the passed list of button names.

Menus are pulldowns, with pullright as a suboption. All pullright names are added to the sequence begun by a menu method. The reference variable value is maintained based on the ordinal position of the names in the entire set. Any menu entry can be passed arguments of the form `(const char* name, bool state&)`. This form maintains the variable `state` via a checkbox menu entry; each selection of the menu by the user toggles the state variable.

READOUTS

InteractBox also defines components that display information:

```
    // Analog-look panel meter which reads a double value from
    // reader_fcn. Range 0-10.
  InteractBox& meter (const char* name, Function<double>* reader_fcn);
    // Panel meter with settable range.
  InteractBox& meter (const char* name, Function<double>* reader_fcn,
                      double min_val, double max_val);
    // Panel meter with settable range which reads from
    // the double value.
  InteractBox& meter (const char* name, double& value,
                      double min_val, double max_val);
  // Analog-look meter, thermometer-style.
```

```
InteractBox& linear_meter (const char* name,
                           Function<double>* reader_fcn);
InteractBox& linear_meter (const char* name,
                           double& value);
   // Multiple, divided bargraph.
   // Double reference variables are passed in row-major order,
   // where each row is a set of bar divisions.
   // Terminated by an end().
InteractBox& bar_meter (const char* name, int nbars,
                        int ndivs_per_bar,
                        double min_val = 0,
                        double max_val = 100);
   // Alterable label, read from READER_FCN.
InteractBox& label (const char* name, Function<char*>* reader_fcn);
   // Indicator light; state determined by reader_fcn or value
InteractBox& light (const char* name, Function<bool>* reader_fcn);
InteractBox& light (const char* name, bool& value);
```

Meters and linear meters provide simple analog readouts, as described earlier.

A `bar_meter` expands on the thermometer-style linear meter by allowing several variables to be displayed at once. The programmer sets the number of bars to display via the argument `nbars` and the number of divisions per bar via `ndivs_per_bar`. The total number of variables displayed is thus `nbars * ndivs_per_bar`. Each bar is passed as a set of arguments (`double&`) following the `bar_meter` call; a variable for each division is passed in turn for each bar. A terminating `end()` marks the end of the list.

A label in this section may change dynamically, via the `char*` reader function.

Indicator lights show a simple on/off state, which is passed via either a variable or a function.

Layout

InteractBox provides a number of layout methods:

```
InteractBox& vertical ();
InteractBox& horizontal ();
InteractBox& fixed (double width, double height);
InteractBox& natural (double width, double height);
InteractBox& spread ();
InteractBox& no_spread ();
InteractBox& glue ();
InteractBox& end ();
```

All layout specifiers are environments, except for `glue()`, and thus require a terminating `end()` following their list of subobjects.

Vertical and horizontal layouts align their subobjects on the left and top, respectively.

`fixed()` and `natural()` force the object size, in the InterViews manner.

`spread()` causes all subsequent objects within `horizontal()` and `vertical()` layouts to have `glue()` placed between them, allowing for general flexibility. `no_spread()` suppresses this.

`glue()` simply places an InterViews glue object at the indicated spot.

The `end()` method terminates all environments (think of it as a right brace) and also marks the end of a list of arguments to script statements that require them.

ATTRIBUTES

With InteractBox, it is possible to set the following attributes:

```
  // Set current test font
InteractBox& font (const char* font_name);
  // Set foreground color
InteractBox& foreground (const char* color_name);
  // Set background color
InteractBox& background (const char* color_name);
  // Set named color
InteractBox& color (const char* name, const char* value);
```

All of these are environments and set the indicated property by name.

MISCELLANEOUS

The remainder of the InteractBox methods are as follows:

```
InteractBox& operator() (const char*, int&);
InteractBox& operator() (const char*, char*&);
InteractBox& operator() (const char*, double&);
InteractBox& operator() (const char*);
InteractBox& operator() (Function<double>*);

InteractBox& enclosure ();
InteractBox& label (const  char*);   // Immutable label.
InteractBox& window_name (const char*);// Set name of created window.
InteractBox& space (double points);     // Add space.
InteractBox& glyph_object (Glyph*);
InteractBox& glyph_object (InteractBox*);
  // Create message window, displaying message.
void informer (char* message);
  // Create confirmation window.
const char* confirmer (const char* message, const char* question);
  // Replace self in parent InteractBox with NEW_BOX.
void replace (InteractBox* new_box);
```

The functional operators all provide arguments to script elements that require variable-length lists of arguments.

`enclosure()` is an environment that draws a box around its subobjects, in the foreground color.

`label()` is simply a constant label string.

`window_name()` sets the name of the X window created by `show()`.

`space()` inserts a space of the given point size.

`glyph_object()`, as described earlier, allows the addition of direct InterViews objects, including other InteractBoxes. `replace()` is called on a child InteractBox that has been added to a parent via `glyph_object()`. It removes itself from the parent, places the new InteractBox passed in its old spot, then deletes itself.

`informer()` simply pops up a window with an informative message and an OK button for dismissal, which is useful for error messages.

`confirmer()` pops up a window with the given message and asks the user the given question. It contains Yes and No buttons, returned as the Boolean value.

A Comprehensive Example

In this section, we examine a comprehensive example of the use of the InteractBox system, by defining a control panel for a testing facility in the fictitious Large Rotating Machine Company. The control panel we are building is shown in Figure 3. The panel consists of three facilities, arranged in a horizontal layout. Each facility shows similar activities; they differ in the set of variables controlled and in the facility names.

We start with the class definition for the overall set of test facilities:

```
class TestFacilityControlBox: public ControlBox
{
 public:
  TestFacilityControlBox ();
  virtual void central_control();
  void facility (char* machine_name,
                 Vars& vars);
  void production_line (Vars& vars);
  void output_rate (Vars& vars);
  void sensors (Vars& vars);
  Vars vars[3];
};
```

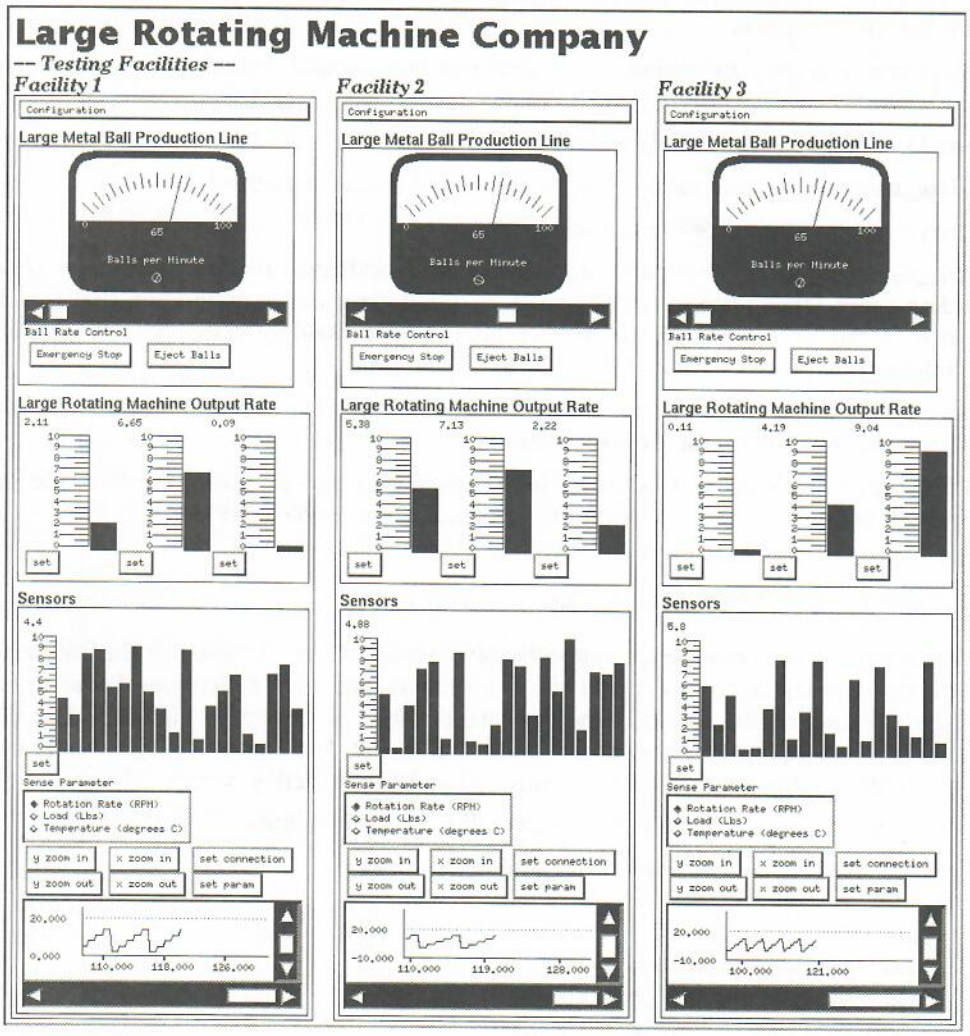

FIGURE 3: *CONTROL PANEL FOR THE TEST FACILITIES OF THE "LARGE ROTATING MACHINE COMPANY."*

The `facility()` method constructs a single panel column. It in turn calls `production_line()`, `output_rate()`, and `sensors()` to build each of its components. Each facility senses and controls a set of `Vars`, defined below:

```
struct Vars
{
  Vars ();
  bool eject_button;
  bool stop_button;
```

```
  double bpm;
  int configuration;
  int configuration_state;
  int sensors;
};

Vars::Vars ()
     :
       eject_button (false),
       stop_button (false),
       bpm (0),
       configuration (0),
       configuration_state (0),
       sensors (0)
{}
```

Each variable defined in the structure represents a state or numeric value of interest to those monitoring and controlling a production line. The `central_control()` method examines all control variables and takes appropriate action:

```
void TestFacilityControlBox::central_control()
{
  int i;
  for (i = 0; i < 2; i++)
    if (vars[i].stop_button)
      ; // Stop facility i
    else if (vars[i].eject_button)
      ; // Eject balls from unit at facility i
  // etc...
}
```

Notice that the `central_control()` method does not need to update readouts based on sensed variables; this is done automatically by the readout system. This is shown in more detail below.

Our next task is to define the contstructor for the `TestFacilityControlBox`:

```
TestFacilityControlBox::TestFacilityControlBox ()
{
  horizontal();
  {
    font ("*-r-*-sans-34-*");
      label ("Large Rotating Machine Company");
    end();
    glue();
  }
  end();
  font ("*-bold-i-*-*-20-*");
```

```
    label ("-- Testing Facilities --");
  end();
  horizontal();
  {
    facility ("Facility 1", vars[0]);
    space (20.0);
    facility ("Facility 2", vars[1]);
    space (20.0);
    facility ("Facility 3", vars[2]);
    glue();
  }
  end();

  show();
}
```

Note that in this example we make liberal use of braces to assist in indentation; they are for editing ease and syntactic clarity only and do not affect the semantics of the code.

Since the top-level facility definition is assumed to be oriented vertically, we use the InteractBox default layout for the top level. We next define the main label, "Large Rotating Machine Company," in a suitable large, bold font. We add some `glue()` here to allow sizing flexibility in the horizontal direction. Below the main label we add "--Testing Facilities --" in yet another font. Then, we are ready to define the three facility columns. We change to a horizontal layout, call the `facility()` method for each named facility, and pass it a reference to its set of `Vars`.

Each facility is constructed using the following definition:

```
void TestFacilityControlBox::facility (char* machine_name,
                                       Vars& vars)
{
  TestFacilityControlBox& t = (*this);

  vertical();
  {
    font ("*-bold-i-*-*-20-*");
      label (machine_name);
    end();
    enclosure();
    {
      vertical();
      {
        t.menu ("Configuration", vars.configuration);
        {
          t ("M0");
          t ("M1", vars.configuration_state);
```

```
            t.pullright ("M2");
            {
              t ("P0");
              t ("P1");
            }
            t.end();
            t ("M3");
          }
          t.end();
          space (10.0);
          production_line (vars);
          space (10.0);
          output_rate (vars);
          space (10.0);
          sensors (vars);
        }
        end();        /* vertical */
      }
      end();        /* enclosure */
    }
    end();        /* vertical */
}
```

In this code, we begin with a label ("Facility 1", etc.), then define an `enclosure()` that draws a box around the set of facility instruments. We then define a configuration menu and three instrument clusters, `production_line()`, `output_rate()`, and `sensors()`, separated by `space()`. Each instrument cluster is passed its set of `Vars` for sensing and control.

The code for `TestFacilityControlBox::production_line` is shown next:

```
void TestFacilityControlBox::production_line (Vars& vars)
{
  font ("*-adobe-helvetica-bold-*");
    label ("Large Metal Ball Production Line");
  end();
  enclosure();
  {
    vertical();
    {
      meter ("Balls per Minute", vars.bpm, 0, 100);
      space (2.0);
      glue();
      slider (vars.bpm, 0, 100);
      space (2.0);
      label ("Ball Rate Control");
```

```
      horizontal();
      {
        button ("Emergency Stop", vars.stop_button);
        space (5.0);
        button ("Eject Balls", vars.eject_button);
      }
      end();
      horizontal();
        space (250);
      end();
    }
    end();
  }
  end();
}
```

Here, we construct a meter for the "Balls per Minute" rate of production and a slider to control this rate. Note that since the meter and the slider share the variable `vars.bpm`, the meter automatically tracks changes made to the slider position by the user. The `central_control()` function is responsible for any actual machine controls made for such changes.

We also define two buttons for ejecting the finished metal balls and stopping the line in the event of an emergency.

The "Large Rotating Machine Output Rate" is our next section:

```
void TestFacilityControlBox::output_rate (Vars& vars)
{
  Randomfcnobj* fcnobj = new Randomfcnobj;
  font ("*-adobe-helvetica-bold-*");
    label ("Large Rotating Machine Output Rate");
  end();
  enclosure();
  {
    horizontal();
    {
      linear_meter ("Linear Meter 1", fcnobj);
      linear_meter ("Linear Meter 2", fcnobj);
      linear_meter ("Linear Meter 3", fcnobj);
    }
    end();
  }
  end();
}
```

This section does only sensing and constructs a set of three linear meters for this purpose. `Randomfcnobj` is simply a function class that returns a random value.

Last we define the sensors. This method defines items that mainly do sensing, but there is also some control so that the user may select which items to examine.

```
void TestControlBox::sensors (Vars& vars)
{
  font ("*-adobe-helvetica-bold-*");
    label ("Sensors");
  end();
  enclosure();
  {
    vertical();
    {
      space (5.0);
      bar_meter ("Bar Meter", 20, 1);
      for (int i = 0; i < 20; i++)
        (*this) (new Testfcnobj);
      end();
      space (2.0);
      radio_buttons ("Sense Parameter", vars.sensors)
        ("Rotation Rate (RPM)")
        ("Load (Lbs)")
        ("Temperature (degrees C)");
      end();
      glyph_object (new WindowlessLineGraph);
    }
    end();
  }
  end();
}
```

Of interest here is the `bar_meter()` call: it defines a set of 20 bars, each with one division. The loop then follows to supply each of the 20 bars with a function object. Here we show how well the InteractBox system combines the properties of a scripting language (in which constants are of primary virtue) and a programming language (where computed values are revered).

The radio buttons following the bar meter allow the user to select a factory variable to examine, setting `vars.sensors`.

The final item illustrates the use of `glyph_object()`, in which we pass a `WindowlessLineGraph` that we had already built for other purposes. This graph allows monitoring of time-varying variable from another process, as established via its "set connection" button. It is completely independent of the InteractBox system and shows how InteractBox and Interviews can work together.

Usage Experience

The major use for the InteractBox system has been within the Insight project at ECI [Stabile93] [Stabile94], a comprehensive system for performance monitoring and debugging in parallel and distributed systems. InteractBox forms the core of the GUI and is used in virtually all of the dialog and control boxes. Complex graphics are normally built directly in InterViews, then added as primitives to InteractBox or used as glyph objects.

Insight provides program instrumentation in the form of a library of calls that may be added to a program or inserted into a running program as probes. InteractBox is sufficiently simple that we have incorporated a version of it into the Insight library so that application developers can add their own measurement and test panels within their programs.

Other small projects have also used the InteractBox system as a quick way to define a GUI.

Feedback from programmers utilizing the system indicates that ease-of-use is the most appreciated feature. These programmers need to create GUI items at various points in a program, and InteractBox allows them to do that with minimum fuss and bother.

Future Work

Extensibility

The InteractBox system already provides a degree of extensibility in that the programmer can define new derived classes, adding statements to the scripting language, and can insert InterViews objects directly using `glyph_object()`.

There is a need, however, to be able to extend the system by adding new primitives, without modifying and rebuilding the InteractBox system itself. We are currently designing an abstract framework for this, which consists of a set of classes that exposes the InteractBox implementation, then allows the programmer to register new sets of classes defined under this framework. An important constraint is that the ease-of-use goal of the original system not be sacrificed.

Interfaces to Other Toolkits

The InteractBox system is reasonably easy to implement, and its concepts should be easily applicable to other toolkits such as Motif.

More Control over Layout and Appearance

While the InteractBox system provides a reasonable set of primitives for many common GUI constructions, it is still lacking in some areas. In particular, not all of the InterViews layout objects are yet supported, nor are other mechanisms such as shading inactive areas in a dialog box.

Error Handling

The InteractBox system does not currently handle errors gracefully. In particular, nesting errors due to improper `end()` placement are ignored; typically some anemic form of the desired display appears, which the programmer must then correct by inspection. Errors due to improper types are infrequent, due to the use of type-safe C++ interfaces.

It is natural to consider the use of C++ exceptions to catch errors such as nesting violations (for example, calling `show()` without closing out all environments with `end()` methods).

Class Structure

It has been noted that the DialogBox class may require a `central_control()` method, should the programmer wish to provide items such as sliders with a readout for feedback. Our use of DialogBoxes was such that this functionality was not needed, so this deficiency did not come to light. Adding a simple extension can permit this functionality.

More generally, it appears that the division between DialogBox and ControlBox is somewhat artificial. This design was driven by our needs at the time: a DialogBox was conceived as a means to receive input from a user, in line with program flow, whereas a ControlBox was meant to be a more general device for building permanent control panels.

A better structure might be to maintain all of the required semantics in a single class, which then may be subclassed by programmers as needed; a standard set of such subclasses could be provided with the system. This model would cause the top-level class (InteractBox) to become "fat"—for example, providing different versions of `show()` that distinguished between modal and non-modal behavior.

We believe, however, that it is very important to maintain only a small set of classes, for ease of extension by a programmer. This is the main advantage of the scripting system and should not be lost.

Conclusion

InteractBox represents a new way to "objectify" a GUI toolkit. Unlike other object-oriented GUIs, it does not rely on the use of large numbers of classes to accomplish its goals. By restricting the class structure, and relying instead on a "flat" structure in which a language is embedded, we have shown that great simplifications are possible.

InteractBox has saved us large amounts of time building the "little things"—simple menus, buttons, etc., which are needed at many points within an application. The InterViews code needed for such objects is coded and debugged once, when it is added as a primitive to InteractBox; then it may be reused with no further thought.

ACKNOWLEDGMENTS

Gary Gu implemented the initial InteractBox class, which gave us the seed from which to grow and extend the system. I also thank Rob Boudrie, Jim Grier, and Dyung Le for their valuable contributions.

APPENDIX: MUTILTHREADED DESIGN ISSUES

The InteractBox system is part of the Insight project [Stabile93] [Stabile94], designed for the debugging and performance assessment of parallel and distributed systems. Naturally, a GUI is an important component of such a system, since graphical display is essential to understanding the behavior of concurrent programs.

Insight itself is multithreaded and uses a thread class developed in C++ for this purpose. The thread class acts as an interface between the programmer and the underlying OS thread support. InterViews is not multithreaded, nor is it designed to operate in such a context, so some measure of thought is required to use it effectively in that environment.

THE PROBLEMS

The initial problem is that InterViews controls the user interface event loop. In Insight, not everything having to do with graphics is initiated via the user; events may enter from processes under test, and we often expect some graphical change without our hands touching the mouse. So, we need a way to initiate graphics operations "from the outside."

A second problem is that InterViews, being single-threaded, is not happy when multiple threads perform operations on common data structures. Some method is needed to control access to common InterViews data structures and in general to serialize graphical operations.

An additional self-imposed constraint is that we wish to modify InterViews as little as possible, so that we may keep up with new versions more easily.

SOLUTIONS

These problems are solved in the following manner. First, we set up a thread and IO handler:

- InterViews is given its own thread upon starting a session. This works without great travail, since we only start one session. Once this thread is running, we are free to create other threads for whatever we require, such as process connections.
- A class `Wakeup` derived from `IOHandler` is defined; its method is invoked by InterViews when the given `fd` is sent some data:

```
inputReady (int fd)
```

InterViews is informed of this class via the method call:

```
Dispatcher::instance().link
    (fds[0], Dispatcher::ReadMask,
    new Wakeup (this, reply_fds[1]))
```

- A `pipe` is created; one end is the input to the above handler, the other end we retain to send data to the handler.

With this mechanism in place, we may send messages to the InterViews thread and expect them to be received in the InterViews execution context. The messages can be of two kinds:

Refresh
: These indicate that certain items that require a real-time update (monitoring meters or graphs, for example) must be updated. All such objects are known to the system, so upon receipt of a refresh message, we call the refresh functions for those items. The refresh function is often optimized to take advantage of the fact that we know the graphic image is in fact intact, so we can erase and redraw only those portions that need updating, for example, updating the bar in a bar graph or the needle on an analog-style meter. As an aside, these involve damaging the appropriate area, calling `draw()` so that they have a graphic effect, and then assuring that the screen is actually updated. We attempt here to optimize these operations so that minimal blitting occurs.

Execute
: This message is an instance of the class `RefreshExecutable`. The handler calls `operator()` on this object, which may then perform arbitrary graphics operations. The normal use of `RefreshExecutables` is to update graphical items in asynchronous mode, that is, to redraw them immediately rather than waiting for the refresh clock. However, `RefreshExecutable` objects may be used for any purpose requiring the context of the InterViews thread.

A thread is created at program startup to act as a refresh timer. This thread periodically wakes up and sends a Refresh message to the InterViews thread. The interval can be set by a programmer and defaults to one second. This timed refresh was defined for those items that update quickly enough that redrawing upon each input would flood the system. We have found that updating once per second is fast enough for humans to follow readout updates but infrequent enough that machine load is minimal.

The above design has served us well, allowing both synchronous and asynchronous graphical operations and freeing us from the InterViews user-event loop. In fact, in yet another thread runs a text-based command interpreter, mainly for developers, and it coexists with InterViews with no trouble.

In addition, the above mechanism requires no changes to InterViews, although use of some low-level (but nevertheless public) features of InterViews is required.

REFERENCES

1. Bashir, Imran, and David M. Sternlicht. "A Tale of Two Toolkits: Xt versus InterViews." *THE X RESOURCE* 7 (1993).
2. Heller, Dan. *Motif Programming Manual.* O'Reilly & Associates, Inc. 1991.
3. [Linton92] Linton, Mark A. et al. *InterViews Reference Manual.* Version 3.1. December 1992.
4. [Ousterhout94] Ousterhout, John K. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.
5. [Smyth92] Smyth, David E., and Adrian Nye. "The Widget Creation Library." *THE X RESOURCE* 2 (1992).
6. [Stabile93] Stabile, L.A. "Insight: Observation of Parallel and Distributed Systems." CHPC TR93-004. WPI Center for High-Performance Computing, 1993.
7. [Stabile94] Stabile, L.A. et al. "The Insight User's Manual." WPI Center for High-Performance Computing, 1994.