The Little Application Framework

Tim Budd

March 13, 1995

1 Introduction

This document describes `LAF`, the Little Application Framework. LAF is intended to be a simple, platform independent framework for developing programs in C++, mainly for instructional purposes. The intent in using LAF, as in all frameworks, is to provide a mechanism to more easily create relatively complex applications by hiding details of implementation. Versions of LAF are available for the macintosh (developed using the Symantic C++ compiler), and the PC (developed using the Borland C++ compiler). Versions for other platforms are under development.

Like most object-oriented application frameworks, the creation of a new application using LAF makes extensive use of inheritance. There are seven main classes provided by LAF, three of which are used mainly as a basis for subclassing. These seven classes are `application`, `button`, `menu`, `menuItem`, `staticText`, `editText`, and `debugBox`. The three classes normally subclassed are `application`, `button`, and `menuItem`. These will all be described in more detail in subsequent sections.

Fundamentally, an application is created by subclassing the class `application`, and overriding certain methods. The arguments required by the constructor for the parent class differ sightly depending upon which platform is targeted. Section 7 will describe in more detail the platform specific features of the LAF.

The main event loop is started by invoking the method `run`, inherited from class `application`. This method does not return. Execution is terminated when the user selects the `quit` menu item, closes the application window, or invokes the `quit()` method inherited from class `application`.

The simplest application possible is shown in Figure 1. Notice that creating the application consists of (a) defining the application class, (b) creating an instance of the class, and (c) invoking the `run` method for the instance of the application class. In this particular case the application does nothing more than display a window on the users screen, and wait for the user to quit the application.

```
class simpleApp : public application {
public:
   simpleApp() : application("simple application") { }
};

void main() {
   simpleApp theApp;

   theApp.run();
};
```

Figure 1: A Simple Application

2 The class `application`

The functionality provided by the class `application` is shown in Figure 2. The following sections will describe these methods in greater detail.

2.1 Starting and Stopping the Application

The function `initialization` is used to initialize features of an application prior to execution. Subclasses should redefine this function if they have application specific initialization to perform. This method is invoked automatically by the framework, and need not be directly executed by the user. Several methods, notably `attachButton` `attachMenu`, `attachEditText` and `attachStaticText`, can *only* be invoked from within the `initialization` method.

The function `run` starts the application. Normally this is the last function called in the procedure `main`. The function does not return. The method `quit` can be called to halt the running application.

2.2 Scheduling Update Events

The methods `update()` or `clearAndUpdate()` are invoked whenever the user desires to schedule a screen refresh. Most often this is necessary following a response to an event, such as a mouse-down event, or a key-press event. The clear method first clears the screen before the refresh. In order to perform the refresh, the application `paint()` method will be invoked. The user typically does not invoke the `paint()` method directly.

2.3 Redrawing the Screen Image

The method `paint()` is used to redraw the screen image. This method is usually overridden in application classes to provide application specific behavior. This

initialization() run() quit()	initialize the application begin execution for the application halt the application
update() clearAndUpdate()	schedule window for updating clear window, schedule for updating
paint()	redisplay the screen
mouseButtonDown(int, int) keyPressed(char)	coordinates of mouse down event character of key
top() botton() left() right() height() width()	coordinate of top of screen coordinate of bottom of screen coordinate of left of screen coordinate of right of screen height of window width of window
textPosition(int &, int &) setPosition(int x, int y) print(char *)	return current text position set the position of the pen print text at current position
circle(int x, int y, int r) point(int x, int y) line(int, int, int, int) rectangle(int, int, int, int)	circle centered at x,y with radius r point (small filled circle) draw line rectangle, upper left and lower right
setPen(color, lineStyle, width)	set pen characteristics
gridOn(color, int) gridOff()	set grid with given distances turn off grid
attachButton(...) attachMenu(...) attachStaticText(...) attachEditText(...)	attach button to window attach menu to window attach static text object attach edit text object

Figure 2: Methods Provided by class `application`

function should not be directly called by the user, instead the method is invoked in response to a call on the method `update()` or `clearAndUpdate()`.

2.4 Mouse and Key-Press Events

When the button on the mouse is pressed the method `mouseButtonDown` is invoked. The two integer arguments represent the coordinates of the cursor, relative to the application window, at the time the mouse was pressed. The default behavior associated with this method is to do nothing. This method must be subclassed by the application class to perform any action.

Notice that the mouse down event does not usually perform any actions which directly modify the image on the screen. (In the PC version of LAF, it cannot perform such actions). Instead, the mouse down function typically records whatever information may be necessary, then invokes either the method `update()` or `clearAndUpdate()` in order to force a screen redrawing.

If a coordinate grid is being used (see next section) then note that the mouse coordinates are changed to the nearest grid point.

When the user presses a key the method `keyPressed` will be invoked. This method takes as argument the character representing the key.

2.5 Window Coordinates

The methods `top()`, `bottom()`, `left()`, and `right()` return the basic screen coordinates of the application window, expressed in pixels. Note that these values are given in the global coordinate system, where 0,0 is the upper left-hand portion of the screen. Almost all other functions which use coordinates use the application relative number, where 0,0 is the upper left-hand portion of the application window.

The methods `height()` and `width()` yield the height and width of the screen (the difference between top and bottom, and between right and left, respectively).

2.6 Graphics and Drawing Functions

A number of simple printing and drawing routines are available as methods in class `application`. Normally Calls on graphics functions are executed during screen update (the notable exception is the method `print`). This means that such calls are made from the `paint()` method, or from a method invoked by the `paint()` method. This restriction is more or less firm depending upon the platform, but is nevertheless a good rule of thumb. (The restriction is firm in the PC version of LAF, slightly less firm in the Macintosh version).

The coordinates for drawing are the same as for mouse down events. This means that the value 0,0 represents the upper left corner, and x-axis values

increase as one moves right, while y-axis values increase as one moves *down*. The latter is often counterintuitive to the beginning programmer.

In the function `circle` the first two arguments represent the x and y values of the center, while the third argument represents the radius. The function `point` draws a circle of radius 1.

In the function `line` the first two arguments represent the starting location, while the second two represent the ending point.

For the function `rectangle` the first two arguments represent the upper left corner of the rectangle, while the third argument represents the width (change in x-axis) and the last integer argument represents the height (change in y-axis).

2.7 Printing

The method `setPosition` can be used to move to a given position in the window. The method `print` then prints a text value in the given position. Newlines are honored in the text, so multiple rows of text can be printed. The method `textPosition` can be used to determine the current text position; the arguments are passed by reference.

Unlike the other graphics functions, which must be invoked from within the `paint` method from a function executed by the `paint` method, calls on `print` *cannot* be made in the `paint` method. (This may be a bug, or a mac-specific item. Need to check on this.)

2.8 Drawing Simple Polygons

There are a number of methods that can be used to draw simple polygons. Note that the coordinates used in these drawing functions represent the local coordinates for the window, not the global screen coordinates. The following command, for example, will print a circle of width 20 at the exact center of the screen.

```
circle(width() / 2, height() / 2, 20);
```

2.9 Setting the Pen Characteristics

The method `setPen` can be used to set the characteristics of the pen used in the draw operations. The first argument is the pen color. The pallet of colors available differs on different platforms, as summarized in the following chart:

PC	black, blue, green, cyan, red, magenta, brown, gray, white, brightBlue, brightGreen, brightCyan, brightRed, rightMagenta, brightYellow, brightGray
Mac	blackColor, whiteColor, redColor, greenColor, blueColor, cyanColor, magentaColor, yellowColor

The second argument to the `setPen` function is the pattern. Again, the set of values differs depending upon the platform:

PC	solidLine, dashedLine, dottedLine, nullLine
Mac	white, black, gray, ltGray, dkGray

The last argument to the `setPen` method is the pen width, in pixels.

2.10 A Coordinate Grid

The method `gridOn` turns on a display of a coordinate grid of dots. When the grid is on, a pixel of the given color is displayed at each coordinate location which is a multiple of the size given by the second argument. Mouse down events are changed to reflect the closest grid coordinate point.

3 Buttons

A button is formed by subclassing the class `button`, creating an instance of the subclass, and attaching the button to the window. The subclass must redefine the method `pressed`, which will be invoked each time the button is pressed.

To attach a button to a window the method `attachButton` is invoked. Because of order of initialization constraints, this method must *only* be invoked within the `initialization` method for an application. There are six arguments to this method.

attachButton(button *, char * title, int x, int y, int width, int height)

The first argument is a pointer to the button object. The second argument is the text to display on the face of the button. Notice this text is not considered to be part of the button itself, but merely part of the way the window displays the button. The four integer arguments represent the upper left corner of the button, and the width and height of the button.

An illustration of the creation of a button will be given in the examples provided at the end of this document.

4 Menus

Menus are in many ways similar to buttons. There are two menu classes supplied with the little application framework, `menu` and `menuItem`. The first of these represents a category on the menu bar, while the second represents a specific item from the category. When a menu item is selected by the mouse, the `selected` method for the associated menu item is executed. Thus, in a manner similar to the way in which behavior for buttons is specialized, application

setText(char *)	set contents of text buffer
<< char	append char to text buffer
<< char *	append string to text buffer
<< int	append integer to text buffer
text()	return contents of text buffer
size()	return number of characters in buffer

Figure 3: Functions defined for static and editable text boxes

specific behavior for menu items is provided by subclassing the `menuItem` class and redefining the method `selected`.

Menus are typically created in the `initialization` method for an application. Menu items are attached to their associated menus by invoking the method `addMenuItem(menuItem *)`. This is method takes as argument a pointer to the menu item. Once all items have been inserted, the menu itself is attached to the application by invoking the method `attachMenu(menu *)`. An example provided later in this document illustrates the use of menus and menu items.

5 Text Boxes

There are two kinds of text boxes supported by the LAF. These are static text boxes, which can be assigned to but not edited, and editable text boxes. Figure 3 shows the operations permitted on these data types. Both forms of boxes will be provided with scroll bars should the text they maintain not fit in the space provided.

A static text box is used to display text that is not being edited. The initial text in the box is defined by an argument to the constructor. Two other arguments to the constructor are optional. The first is an integer argument, and represents the width of the rectangular box used to surround the text. The default value is zero (that is, no visible box). The third argument indicates whether the text should be left justified (value `leftText`, right justified (`rightText`), or centered (`centerText`). The default value is left justified.

As with buttons, a static text box must be attached to the application window. This is accomplished using the method `attachStaticText`, which takes arguments similar to the method `attachButton`.

attachStaticText(staticText *, int x, int y, int width, int height)

Text can be added to both static and editable text boxes using the stream operators <<. Characters, strings and integers are recognized as arguments to these operators. Output routines for other data types can be easily written.

Editable text boxes are similar to static text boxes, except that editing operations are permitted.

6 Examples

Figure 4 illustrates a simple program which draws a circle of radius 20 each time the mouse is clicked. Because the graphics routines can only be executed from inside the `paint` method, the location of the mouse click is saved in a pair of global variables. Since there is no application specific initialization in this case, the `initialization` routine is not declared. Note that the application variable itself is simply declared as a local variable in the `main` program.

Note that the programs in this section are illustrated using the Macintosh version of LAF. On the PC and other platforms the constructor for the application would be slightly different. These differences are described in Section 7.

The next example program, shown in Figure 5, is slightly more complex. This program illustrates the structure of a typical application and the use of buttons. The program will display the text "hello world" each time the mouse is pressed. Two buttons are provided in the upper right corner. One simply quits the application (duplicating the action of both the go-away box and the standard "quit" menu item.) The second button clears the screen.

This program illustrates a common phenomenon. Because the methods for the buttons need to refer to methods from the application class, the application object itself must be declared as a global variable. (In this case, the variable `theApp`). Methods for the buttons then can modify the application by invoking methods through this global variable. For this reason the declaration of the global variable appears before the class definitions for the buttons, which in turn must appear before the definition of the method `initialization`, in which they are attached to the window.

Figure 6 illustrates a program similar to the previous, this time using menu items rather than buttons for the quit and clear actions. The structure of the program is, however, very similar. The text of the quit and clear buttons is followed by a slash and a letter. On the macintosh, this will automatically establish a short-cut key sequence for selecting the associated menu item.

Figure 7 illustrates the use of edit boxes. An edit box is created in the upper left corner of the window. Key presses outside the edit box are caught and appended to the end of the edit text. The user can also edit the text directly, in the usual fashion. When the user clicks the mouse outside of the edit box, the current contents of the box are copied on to the window. Static text boxes are similar, only the text cannot be modified in a static text box.

```
/*
   circle program
*/

# include "maclaf.h"

class circleApp : public application {
public:
   circleApp() : application("hello world") { };
   virtual void mouseButtonDown(int, int);
   virtual void paint();
};

int savex = 0;
int savey = 0;

void circleApp::mouseButtonDown(int x, int y)
{
   savex = x; // save coordinates of mouse down
   savey = y;
   update(); // update without erase
}

void circleApp::paint()
{
   circle(savex, savey, 20); // draw a circle of size 20
}

void main() {
   circleApp theApp; // create the application
   theApp.run();     // run it
}
```

Figure 4: A Program the Responds to Mouse Events

```
/*
   hello program with buttons
*/

# include "maclaf.h"

class helloApp : public application {
public:
   helloApp() : application("hello world") { };
   virtual void initialization();
   virtual void mouseButtonDown(int, int);
};

void helloApp::mouseButtonDown(int x, int y)
{
   setTextPosition(x, y);
   print("hello\nworld");
   update(); // update without erase
}

helloApp theApp;

class quitButton : public button {
public:
   void pressed() { theApp.quit(); }
};

class clearButton : public button {
public:
   void pressed() { theApp.clearAndUpdate(); }
};

void helloApp::initialization()
{
   app.attachButton(new quitButton, "quit", 5, 5, 50, 20);
   app.attachButton(new clearButton, "clear", 5, 30, 50, 20);
}

void main() { theApp.run(); }
```

Figure 5: A Hello World Program with Buttons

```
/*     hello program with menus     */

# include "maclaf.h"

class helloApp : public application {
public:
   helloApp() : application("hello world") { };
   virtual void initialization();
   virtual void mouseButtonDown(int, int);
};

void helloApp::mouseButtonDown(int x, int y)
{     setTextPosition(x, y);
   print("hello\nworld");
   update(); // update without erase }

helloApp theApp;

class quitItem : public menuItem {
public:
   quitItem() : menuItem("quit/Q") { }
   void selected() { theApp.quit(); }
};

class clearItem : public menuItem {
public:
   clearItem() : menuItem("clear/C") { }
   void pressed() { theApp.clearAndUpdate(); }
};

menu theMenu("options");

void helloApp::initialization()
{    theMenu.addMenuItem(new quitItem);
   theMenu.addMenuItem(new clearItem);
   app.attachMenu(& theMenu); }

void main() { theApp.run(); }
```

Figure 6: A Hello World Program with Menus

```
/*
   edit box program
*/

# include "maclaf.h"

class editApp : public application {
public:
   editApp() : application("edit box application") { };
   virtual void initialization();
   virtual void mouseButtonDown(int, int);
   virtual void keyPressed(char);
};

editBox ebox(" ");

void editApp::mouseButtonDown(int x, int y)
{
   setTextPosition(x, y); // move to current location
   print(ebox.text()); // print contents of edit box
   update(); // update without erase
}

void editApp::keyPressed(char c)
{
   ebox << c;    // append character to edit box
   update();    // force refresh of image
}

editApp theApp;

void editApp::initialize()
{
   theApp.attachEditText(&ebox, 10, 10, 100, 100);
}

void main() { theApp.run(); }
```

Figure 7: Application Program with Edit Boxes

7 Platform Specific Features

In this section we will describe the features of LAF which are specific to each of the platforms to which it has been ported.

7.1 PC LAF

7.2 Mac LAF

On the macintosh the constructor for the application class requires only the name of the window, as shown in Figures 5 through 7.

One method provided by the `application` class on the macintosh LAF which is not provided by other version is `eventTick`. This function is executed once every time an event is recognized. By scheduling an update (and thus forcing a new event) within this function an animation-like effect can be created. An example is shown in Figure 8. This program draws a circle moving from the upper left to the lower right, like a ball.

```
/*
   animation example
*/

# include "maclaf.h"

class aniapp : public application {
public:
   aniapp() : application("animation example") { }
   virtual void eventTick();
   virtual void paint();
};

int savex = 0;
int savey = 0;

void aniapp::eventTick()
{
   savex += 5;
   savey += 10;
      // stop animation at 300
   if (savex < 300) update();
}

void aniapp::paint()
{
   circle(savex, savey, 20); // draw a circle
}

void main() { aniapp theApp; theApp.run(); }
```

Figure 8: A Simple Animation