CanvasPlus: an improved modular canvas implementation for Tk

J.Lima

September 15, 2005

Abstract

The standard canvas implementation in Tk lacks some important features, like complex object encapsulation, viewing transformations (zoom) and some object transformations (rotations), which make it an unappropriate tool to implement modern drawing applications. Adding these features in an add-hoc fashion to standard canvas implementation can solve some particular application problem, but will increase the canvas complexity and will always have a limited scope. A different approach, described in this article, consists of a modular canvas implementation, which comprises: a core providing a basic drawing interface and bindings to events; a set of application specific modules which implement the higher level commands. These can range from the very simple, found in the standard canvas implementation, to more structured commands which are necessary to implement complex 2D cad applications

Contents

1 Introduction

The *CanvasPlus* is a Tk widget targeted for 2D CAD applications. Its development was based on the idea of developing a suite of EDA tools using

Tcl/Tk based GUI. That is, applications built in either C or C++ featuring Tcl/Tk GUI. In addition, the Tcl scripting capabilities embedded into the application could be used for configuration and macro processing.

Tcl/Tk is a suitable framework for application development which supports both GUI construction and scripting capabilities. However, full featured drawing applications require specialized widgets. For simple drawing applications, the standard Tk canvas widget may be sufficient, but, as the complexity of the objects being manipulated increases, more advanced and specialized widgets are necessary.

Tcl was considered very strong candidate to be used as a scripting platform but also as a GUI construction tool due to its natural bound to Tk. Even after considering other, more advanced toolkits, for instance, Qt[3], wxWindows[4] or Gtk[5], the Tcl/Tk option is still attractive because, although less featured, is more lightweight. However, the choice of Tcl/Tk as a GUI platform implies the development of a new canvas widget.

Another argument in favor of Tcl/Tk is the fact that the GUI building is shifted from the compiled languages (C or C++) to the scripting environment. This is a big advantage because it is usually much simpler, and the process for improving the GUI will be much more dynamic. Experience shows that building the GUI is time consuming task. And often poorly designed GUIs frustrate the intent of an application. So, it is preferable to allow users to reconfigure the look and feel of the GUI, or even completely redesign it.

The *CanvasPlus* widget, should provide a framework for layered, 2D, interactive drawing applications. Besides that, no more assumptions should be made regarding the nature of the supported applications. Such general terms pose obvious implementation problems. To overcome them, a modular, incremental development model was adopted.

The design was also driven from the following key ideas.

- The canvas functionality is implemented mostly in low level (C/C++)
- Details of data representation and handling should never be performed in TCL.
- Standard viewing operations (zoom, pan) and standard geometrical transformations should be implemented efficiently in low level and available with the same semantics for every application.
- TCL code should be used only for configuration and GUI building.
- However, TCL code should have access to object properties.

2 Overview

The system consist of a core module dealing with Tk and X internals and a set of extension modules implementing application specific drawing modes. The core module supports only a basic set of drawing operations and geometric transformations. It provides the API for extension writers and the infrastructure to load and switch between drawing modes.

Every specialized canvas share a common subset of drawing operations, all have to deal with the burden of Tk and X details, and all have to

interact somehow with the TCL scripting environment. All this issues are dealt only once by the core of the *CanvasPlus* implementation, while the drawing modes extensions will deal with application details only.

This approach can considerably reduce the effort required to develop specialized canvases for layered 2D drawing applications.

The *CanvasPlus* widget doesn't provide a specialized canvas implementation. Instead, it provides a framework to develop and use specialized, application centric canvases.

3 The Modular Approach

When implementing the *CanvasPlus* widget, one could choose to implement a very generic widget, customizable by a Tcl script or, alternatively, implement different application specific widgets with minimum configuration options. The first option would lead to very complex code, both in C and in Tcl. The second option would lead to code duplication and wouldn't promote the reuse of the widget for other purposes.

A third approach is to implement the canvas widget in a modular fashion: a basic canvas widget with minimum functionality, and thus very generic, dealing with all Tk and X internals; and a set of high level application specific modules.

The modular approach to implement the *CanvasPlus* has two main advantages over the monolithic approach. The first one is that one can adapt the basic canvas module to a particular application simply by writing a higher level module, which can be made portable between architectures, since it doesn't include GUI dependent code. The second is that it will promote code re-usability because the generic code is shared amongst all canvas extensions.

As a result, a core module implements a canvas widget which virtualizes a drawing canvas with a physical (in true physical dimensions) Cartesian coordinate system where extensions can draw using configurable pens. Extensions can interact with the canvas in either direction: they can draw and configure the canvas properties; they will be notified to process certain class of events.

Housekeeping operations, like, for instance, redrawing the canvas in response to expose events, are handled by the canvas core transparently. Also zoom and pan operations are handled by the core.

Application specific operations, like, for instance, placing a transistor in a schematic sheet, are handled by extension modules.

Figure 1 illustrates the concept just described.

4 Canvas Drawing Modes

Connected with the *CanvasPlus* modular approach, there is the concept of *drawing mode*. The *drawing mode* consists of certain canvas vocabulary and the associated semantics that are loaded by a particular high level *CanvasPlus* module.

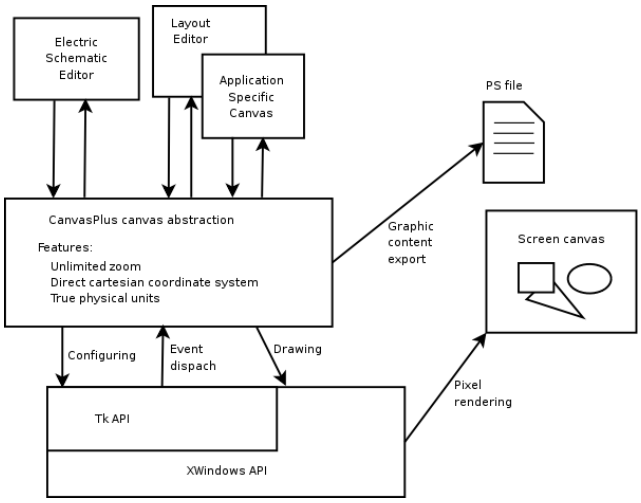

Figure 1: CanvasPlus modular architecture

To emulate the standard Tk canvas, a high level *CanvasPlus* module would implement the following commands:

```
addtag, bbox, bind, canvasx, canvasy,
cget, configure, coords, create, dchars,
delete, dtag, find, focus, gettags,
icursor, index, insert, itemcget,
itemconfigure, lower, move, postscript,
raise, scale, scan, select, type, xview,
or yview
```

Of course, nobody would care to use the *CanvasPlus* to emulate the standard Tk canvas. For that, it would be better to just use the later.

The idea of *CanvasPlus* is to implement higher level drawing environments which are application centric. The simplest scenario one can imagine is the one in which each application uses a single extension module. However, sometimes, several applications share some common functionality. In these cases, the best would be to use a stack of extension modules and each application would load its required module combination, thus improving the code reuse.

To load a particular mode, one uses the *CanvasPlus* command `setmode`. This command takes as argument the mode name. It will work as long as the required extension modules have been previously loaded with the `load` command. The following example illustrates this idea.

```
load ./modetest.so
canvas+ .c
.c setmode modetest
```

The `setmode` command returns a TCL interpreter which shall be used for further interaction with the canvas, as described in the following section.

5 Canvas Commands Interface

Unlike the standard Tk canvas — where the commands to draw or interact with the canvas contents are performed by canvas subcommands, like for instance,

```
.c create rectangle 10 10 130 60 -fill red
```

which creates a rectangle in the `.c` canvas — in the *CanvasPlus*, the subcommand approach is used only for a set of core widget commands like the `configure` or `cget`, while the commands to interact with its content are executed inside a dedicated interpreter which is bound to the *CanvasPlus* in runtime.

For example, once one initializes a drawing mode in the canvas with the command

```
set canvas_interp [.c setmode <mode name>]
```

the above task to create a rectangle would be done with

```
set script {create rectangle 10 10 130 60 -fill red}
$canvas_interp eval $script
```

This subtle syntax difference has several advantages. First of all, one can easily write multi-line scripts which are independent of the canvas where they are to be executed, in the sense that the canvas name doesn't appear in the script. Try to do that with the normal canvas approach. Second, in the context of a `bind` procedure, the code can have access to global variables with details associated to the triggering event. These variables are global only in the canvas interpreter, and different canvas will not interfere with each other. This is a much more elegant approach than the traditional way to access event details in the `bind` command.

A complete example for the usage of the bind command is shown in the listing bellow. The usual Tcl/Tk way of doing it is preserved almost intact. The only difference lays on the bound procedures which evaluate the code in the context of the `$I` interpreter, and use the local interpreter variables `$_X_` and `$_Y_`.

```
set I [$C setmode modetest1]
$I eval {set moving 0}
$C bind all <ButtonPress-1> picOrDrop
$C bind all <ButtonRelease-1> picOrDrop
$C bind all <Motion> move

proc move {} {
    $::I eval {
        if { $moving } {
            @move $prevX $prevY $_X_ $_Y_
            @puts move $_I_ $prevX $prevY $_X_ $_Y_
            set prevX $_X_
            set prevY $_Y_
        }
    }
}
```

```
proc picOrDrop {} {
    $::I eval {
        if { $moving } {
            set moving 0
            @puts droping
        } else {
            set moving 1
            set prevX $_X_
            set prevY $_Y_
            @puts pic $_X_ $_Y_
        }
    }
}
```

6 Extensions Writer Interface

More important than the improvements and additional features on the Tcl/Tk side, is the C/C++ API. In fact, one of the ideas of *CanvasPlus* is to allow a better integration of C and C++ into TCL based graphic applications than it was possible with bare Tcl/Tk. For that one needs a carefully designed API which must meet the following requirements:

- Provide the simplest yet powerful set of drawing primitives
- Handle events behind the scenes through the use of well documented hooks
- Hide Tk internals by stressing the completeness of the API.
- Provide C as well as C++ interface

Typically a extension writer for *CanvasPlus* will implement a a new drawing mode. The concept of drawing mode is flexible and powerful enough for the majority of the cases. It is not possible to describe the complete API in this article. But there is minimum framework which is necessary for the simplest extension which must be known

The following API functions are mandatory for every drawing mode extension:

```
static ClientData allocProc(CanvasPlus *canvas)
static void freeProc(ClientData cd)
static int initProc(CP_Mode *mode)
static void drawProc(ClientData cd, CP_DrawOps *ops, CP_Area *a)
static CP_Item *closestProc(ClientData cd, CP_Point *pt, double dist)
```

The function names are self explanatory. `allocProc` and `freeProc` are called when the drawing mode is created and deleted respectively. While the `initProc` is called whenever the mode becomes the active mode.

The `drawProc()` hook is used to give one the opportunity to draw the graphics content. The *CanvasPlus* core module will take care of invoking the hook right when it's necessary. The pointer to structure of type `CP_DrawOps` is used to access the drawing operations. One may wonder why this approach is necessary. With this approach no assumptions are

made regarding where the graphics contents is to be drawn. The *CanvasPlus* core can arrange to send the content to a printer or exporting to a file.

The `closestProc()` hook is used for processing GUI events. More exactly to find the item closest to the pointer position.

In addition, the following structures must be initialized:

```
static CP_RegisterInfo _info[] = {
  CP_ENTRY("@puts", putsProc),
  CP_ENTRY("@rect", rectProc),
  CP_ENTRY("@elipse", arcProc),
  CP_ENTRY("@move", moveProc),
  CP_LAST_ENTRY
};

static CP_ModeOps _operations = {
  allocProc,
  initProc,
  freeProc,
  drawProc,
  closestProc
};
```

The `CP_RegisterInfo` array is an array of structures that contain the information necessary to register procedures in the Tcl interpreter associated with the drawing mode. The '@' caracter in the beginning of each procedure name obeys to a convention adopted by the author to distinguish from other procedures.

The macros `CP_ENTRY()` and `CP_LAST_ENTRY` are convenient ways of filling the structure array without caring with the details.

The structure `CP_ModeOps` holds pointers to the drawing mode interface hooks.

The following function is called by Tcl when the library is loaded and will register the module calling `CPDefineMode()`. This is the standard way for the Tcl `load` command to initialize library code [2][1]. This is the only symbol which must be external. The code for this function, apart from its name, should be the same for most applications

```
int Modetest_Init(Tcl_Interp *interp)
{
  if ( CPDefineMode("modetest1", _info, &_operations)!=TCL_OK )
    return TCL_ERROR;

  return TCL_OK;
}
```

7 Conclusions

A *CanvasPlus* prototype has been developed and is usable, although some important features are still missing. Once a real-world extension is writ-

ten, the *CanvasPlus* may already be used for development purposes. For the time being, one shall highlight the *CanvasPlus* key ideas:

- Modular design approach.
- The use of coordinates with true physical meaning,
- Tcl drawing interface based in a slave interpreter, instead of using a sub-command approach.

7.1 Work in progress

One of missing features in *CanvasPlus* is an anti-aliasing font renderer. This is a must, because of the zoom feature in present in the canvas. Apparently, the best option would be to just use the font selection and rendering mechanism present in Tk. This approach may work if the system has only scalable fonts installed. If this is not the case, you could simply take care not selecting non-scalable fonts, or even provide a mechanism to reject them. In the author opinion, none of this solutions is satisfactory, because, with respect to font resizing the Tk API doesn't make the job easy and doesn't give any guarantees on the actual rendered font size.

For these reasons alternative font selection and rendering mechanisms are being developed.

In the mean time an extension to implement vector drawing application is being studied.

References

[1] Brent B. Welch, *Practical Programming in TclTk*. Prentice Hall PTR, 3rd Ed, 2000.

[2] John K. Ousterhout, *Tcl and the Tk Toolkit*. Addison Wesley, 1993.

[3] QT toolkik homepage: http://www.trolltech.com

[4] wxWidgets homepage: http://www.wxwindows.org/

[5] Gtk homepage: http://www.gtk.org