# NexTK / NtkWidget –

# a replacement for Tk?!

Arnulf Wiedemann [1]

D-86931 Prittriching

Germany

## Abstract

The current implementation of Tk has a lot of leftovers of the long history it has, including a lot of dependencies from X11 code which it often calls directly.

Therefore and for other reasons George Peter Staplin started some years ago an implementation of Tk9 later on renamed to NexTk together with megaimage, ntk and freetypeext.

The basic idea behind NexTk is using images for the widgets and also for all parts of a widget.

NtkWidget is based on that implementation with some differences and with additional use of GLFW (Graphic Library FrameWork) and using the openGL library for displaying images with the same interfaces used for linux, windows and MacOS.

## History of NexTk

George Peter Staplin started about 2 years ago with an implementation of a new system for handling windows and widgets called Tk9. After some time the name was changed to NexTk for several reasons. NtkWidget is based on NexTk, so I am first describing the components of NexTk, the differences to the existing Tk implementation and some of the additional features, as well as some future features of ntkWidget.

In the next few chapters the architecture and structure of Tk, NexTk, ntkWidget and ttk(Tile) will be presented in a short overview for each of the parts.

## Architecture of Tk

From its history Tk is tied very much to unix like systems and X11. The support for windows and MacOS came after Tk existed for a lot of years.

The basic widgets in Tk are implemented completely in C, the parts of these basic widgets like borders etc. are built and displayed using X11 (client-)functions. Fonts are handled using system fonts. Options of widgets are handled by a relative complex mechanism using an option database.

The most commonly used geometry managers for Tk are grid, pack and place.

[1] Email: arnulf@wiedemann-pri.de

There is currently no alpha blending available for Tk (in the base/core) and no free rotation of texts

## Structure of Tk

Tk is mostly using an X11-client for displaying stuff on a screen as well as handling mouse and key events. The calls for this functionality is mostly hard-wired in the Tk basic widget layer. For windows it is using the Win32 API and GDI. Tk for MacOS X can run with X11 or with a native MacOS X framework.

On top of the basic widget layer there are C-coded simple widgets like frame, button, entry, text …, where text cannot be called a simple widget :). The callback functionality is implemented as Tcl commands written in C-code. Part of the option handling and more complex widgets are built on top of these simple widgets using mostly Tcl code.

## Architecture of NexTk (1)

The main difference between Tk and NexTk is, that a widget is not directly built using X11 function calls and modifying it, instead every widget and also the parts of it are built as an image. Parts of a widget are for example the window background, border, text etc. and also more complex widgets as for example a scrollbar, which itself is also built from images for the different parts of it.

The code for building the widgets is Tcl code (not C-code as for Tk). Experiments have shown, that it also not necessary for all of that code to write it in C as the C code is only about 2-3 percent faster than the Tcl code, which for GUI's normally doesn't matter.

For building the different images, which make a widget, megaimage is used.

For selecting fonts and building images for texts,  freetypeext is used.

From the above mentioned geometry managers for Tk only grid is supported partially at the moment, pack and place are planned.

There is support for alpha blending and for free rotating of  texts.

## Components of NexTk

NexTk has 4 main components for building  complex widgets and displaying the widgets on a screen as well as handling mouse button events and key events (that is when hitting a key on the keyboard).

- megaimage
- objstructure

- freetypeext
- ntk

### *megaimage:*

megaimage is a C-implemented Tcl package for creating and modifying pixel images. Creating pixel images means that you can fill a memory region, which represents a pixel image with bytes representing color values and an alpha value. For color values 3 bytes for r(ed), g(reen) and b(lue) called the rgb value are used and the 4th byte for a pixel contains the alpha value. The alpha value is representing the “transparency” of the pixel. Transparency is in the range of completely transparent to not transparent at all. There are functions for simply filling rectangles and polygons with pixels and for placing text in a certain font and for rotating texts. For texts the freetypeext component is used.

### *freetypeext:*

Freetypeext is an extension based on the open source freetype font library, which allows to build pixel images for texts suitable for megaimage. Rotation of texts is also implemented here and some parts in megaimage.

### *objstructure extension:*

Objstructure extension is a C-implemented Tcl library, which implements an object like system with option variables and call backs to be used with ntk.

### *ntk:*

Ntk is a collection of Tcl-implemented widgets which are offering similar widgets as in Tk. Here is where the base widgets are built

For details on NexTk see: http://wiki.tcl.tk/16320

For the objstructure extension see: http://wiki.tcl.tk/21111

## Architecture of NexTk (2)

As already mentioned every pixel is represented as 4 bytes, 3 bytes for the rgb value and one byte for the alpha factor. The blending functions also take care of the alpha factor and make the appropriate calculations.

The ntk package also provides code for handling root windows (the X11 desktop background), where the first and all other Tcl toplevel windows of ntk are put and for handling mouse and key events. The first toplevel window has to be created at the start of the application, to be able to put widgets in it.

The first version of ntk was on the low level also directly using X11 functionality for displaying widgets, but the current version now makes use of openGL for displaying the images of ntk.

For those who are not so familiar with graphic stuff, openGL is an open source graphic library which was originally implemented by SGI and is now open source and is widely used and adapted for different HW platforms. It is for example also used in Tcl3D.

The current version of NexTk is running on linux and on windows (win32).

## Architecture of ntkWidget

My starting point with ntkWidget was about 8 months ago, when George Peter Staplin (GPS) was looking for other developers for the NexTk project. At that time the ntk part was still written with the structure extension, a predecessor of objstructure. Looking into the source code I had at that time the feeling that for me it would be much easier to understand and further develop ntk using itcl classes and objects with its inheritance for implementing the functionality provided in ntk. So I started in agreement with GPS to implement ntk using itcl. During the implementation I was missing some useful features in itcl, which would be helpful for implementing ntk, so I added some functionality to itcl, as I was already doing a reimplementation of itcl for other reasons (to make it easier to maintain it with the modified Tcl core of Tcl version 8.5). This version is now known as itcl-ng see: http://wiki.tcl.tk/19873 but it is not yet released

After having the first version of the modified ntk running called ntkWidget, I was not yet completely happy with having at that time only X11 for displaying the wigdets on a screen, so the SW did only run on linux/(unix). So I was looking for open source packages which would allow to have the package running on linux/unix, windows and MacOS. The investigations (google search) brought me to GLFW (**G**rahic **L**ibrary **F**rame**W**ork) a C-code package written by Camilla Berglund and others, which was easy to adapt see: http://glfw.sourceforge.net

That package provided handling of root windows and handling of mouse and key events as well as joystick events and was using openGL for displaying graphic stuff. The API functions provide a platform independent handling of the events and root windows, as openGL provides for displaying. It has backends for X11, win32 and MacOS.

The only problem I was seeing was, that there is only one root window, as the library was mostly used for programming games. So I contacted the developers, if there are any plans for multiple root windows. There were plans, but not in the near future, so I decided to implement it myself and doing so I also replaced the threading in the package by using the Tcl event loop for dispatching. I also adapted the code to fit to the Tcl core code and to use as much of Tcl core functionality as possible. That package is now called GLMWFW (**G**raphic **L**ibrary **M**ulti **W**indow **F**rame**W**ork) and is available as a Tcl extension package (not yet released). It has a Tcl command with subcommands (coded as a Tcl namespace ensemble) for calling the needed functionality.

GLMWFW has at the moment only the linux X11 backend running, but the adaption of the other parts for win32 and MacOS should not be very difficult (I am lacking the development environment for that).

With these packages I have built the ntkWidget package. The widgets are built using itcl-ng classes and objects, the handling of root windows and events is done using GLMWFW and the displaying of the images blended with ntkWidgetImage is done using openGL. ntkWidgetImage is a package based on a modified version of megaimage and freetypeext for preparing text areas with a certain font.

There is some handling of themes (also used in NexTk), which is different from the handling of themes in ttk (Tile).

For those, who will argue, but I don't want to write my programs with itcl, there is no problem, all the API functionality is available as "normal" Tcl namespace ensemble commands, the namespace ::ntk is used for that.

## Structure of ntkWidget

As already mentioned, ntkWidget is using GLMWFW as the platform independent layer towards the different operating systems. This includes use of openGL for displaying.

Building of the images is done using ntkWidgetImage with its blending functions. There are two commands which are commonly used in all widgets: grid and render. Grid is the ntkWidget grid manager, which behaves similar to the Tk grid manager, also some of the options and commands are a little bit different.

Render puts the different widget parts together including text areas/images. It uses ntkWidgetImage.

On top of that there are the widgets like frame, label, button, entry etc., details later on.

Every widget is an itcl class, some of the widgets appear in the hierarchy of other widgets.

## Building Tk like widgets

Widgets like label, button, entry etc. consist of widget parts. A widget part is:

- a window, normally an rectangular area with a background color
- the borders, which are also rectangular areas with a certain width and height. The width of the borders can also be 0, if the border is not visible.
- a rectangular area for a text with a certain font, which is for example the contents of a label or a button.

There are more complex widgets like a scrollbar. A scrollbar consists for example of two arrows, which can be used to move the scrolling region up or down, left or right, and the scrollbar thumb, which has identical functionality as the arrows and there is the trough, which is a container, where the arrows and the thumb are placed and the thumb can be moved along the trough. There are also borders for a scrollbar.

All these widget parts are preparing as a result an image, if the renderer is handling it. For building the image the renderer is calling the draw callback of the widget/widget part, which builds the image. The different widget parts (images) are brought together using the renderer. The Renderer uses blending functions from ntkWidgetImage, which also take care of the alpha factor (transparency).

As the root window is also handled as a widget/image, the renderer is doing the same as for every other widget here.

## Rendering of a widget (label)

The renderer is called for the toplevel widget or any other widget in the widget hierarchy and starts the rendering process. It first prepares the widget's image it is called for and then it looks, if there are any child widgets for that widget and calls the rendering function recursively for all the child widgets. Every call to the renderer creates an image for the widget/widget part. Every result returned from the rendering of the child is an image which is put at the appropriate coordinates of the widget currently handled using ntkWidgetImage blending functions. So all the different images are brought together, to build the final image of the widget, which is what is shown on the screen.

Child widgets are for example the border widget parts, a text area etc.

The final displaying on the screen is done using openGL functions.

The blending is done in memory in the background and is not visible on the screen. I have tried to use different buffering functionalities available in openGL, but all had problems. Some were not available without extra packages installed on the machine, others had only a very limited number of buffers (max. 4), which is not enough to do the rendering as described above.

## ntkWidget Hierarchy

All the ntkWidget base widgets have a common class hierarchy part. There is a helpers class which contains parts common to all widgets like functions for validating option values.

The renderer mentioned already above is a common part, which can also be used outside the class hierarchy. This is realized by itcl procs which are class methods that can be used without an object.

The grid manager has also already been mentioned and is realized the same way as the renderer using itcl procs. gridData and gridManager are classes which contain specific data for widget objects.

The base of all widgets is window, a class which is a container for handling the final image of the widget as well as containing the options on the geometry and the offset of the widget within the area the widget is using. Additionally the background color and other similar stuff is placed here.

Theme is a class which can handle different styles of widget parts. It can for example handle the layout of a scrollbar depending on the selected style. This is different from what ttk (tile) is doing and only has a loose relationship to it.

All base widgets of ntkWidget, for example button,  are inheriting the widget classes described so far.

## ntkWidget base widgets

Here is the collection of ntkWidget base widgets at the moment, with a short description. Details can be found in the wiki see:  http://wiki.tcl.tk/20156

The description of he base widgets in the wiki is on many places not yet complete, as also the user APIs are not yet final.

### *Base widgets of ntkWidget*

- Box
- Button
- Checkbox
- Clock
- Entry
- Frame
- Label
- Listbox
- Menu
- Scrollbar
- Spinbox
- Text
- Toplevel

## ntkWidget commands

The ntkWidget commands are realized using an ensemble command ::ntk which has a lot of subcommands. Each of the subcommands has a list of options. As the options of the subcommands are not yet complete and final, I will not list them here.

ntk commands:

- button
- checkbox
- clock
- entry
- focus
- frame
- grid

- input
- label
- listbox
- menu
- render
- scroilbar
- spinbox
- text
- theme
- toplevel
- window
- keyPress
- keyRelease
- mousePress
- mouseRelease
- motion
- isControlKey
- widgetImage
- bgerror

## ntkWidget code example

```
package require ntkWidget
# set the root windows title
::ntk::glmwfw::Glmwfw setWindowTitle $::ntk::_win "NtkWidget Button Demo 1"

proc flipName {win} {
    set name [$win cget -text]
    switch $name {
    Arnulf {
        $win configure -text Wiedemann
      }
    Wiedemann {
```

```
        $win configure -text Arnulf
      }
   }
}

set rotate 0
if {[llength $argv] > 0} {
    set rotate [lindex $argv 0]
}
set wPath [ntk button .w -width 100 -height 100 -xoffset 50 -yoffset 50 \
-text Arnulf -textcolor [list 0 0 0 255] -bd 2 -bg [list 255 255 255 255]]

$wPath configure -command [list flipName $wPath] -rotate $rotate

# pack the widget using ntkWidget grid manager
ntk grid $wPath
```

## Ttk style

Here comes a short introduction into ttk, as an implementation of a similar mechanism will replace the theme class mentioned above. To understand that mechanism it is necessary to get a rough feeling for ttk and what it is doing.

Ttk offers two mechanisms to influence the look and feel of a widget for example a button: a theme style and a theme layout.

The theme style and layout of a button is called TButton for example.

Let's compare the default and the clam theme style of a button called TButton.

There are some (not all) options, which mostly influence the "look and feel" of a button:

### *Default style Tbutton options:*

- -anchor center
- -padding [list 3 3]
- -width -9
- -relief raised
- -shiftrelief 1

### *Clam style of Tbutton options:*

- -anchor center
- -padding 5

- -width -11
- -relief raised

One can see that the padding is depending on the theme and has also influence on the width. The minus value will influence the width of the bare button without padding and border. And there is an additional option for the default theme style.

Here are some options, which influence the colors of a button depending on the theme style:

### *Default style Tbutton colors:*

- -window #ffffff
- -frame #d9d9d9
- -disabledfg #a3a3a3

### *Clam theme style Tbutton colors:*

- -window #ffffff
- -frame #cdad5d
- -disabledfg #999999

The colors of a button are different depending on the theme selected. There are a lot of other options not shown here.

## Ttk layout

Using ttk also the layout of a button may look different depending on the theme layout used.

### *Default layout TButton:*

- Background
    - [list]
- Border
    - Focusring
        - Padding
            - Text
                - [list]

### *Clam layout TButton:*

- Background

        - [list]
- Border
    - Focusring
        - Padding
            - Text
                - [list]

By accident the layout is identical for both cases, but let us assume the following:

### *Hypothetical default layout TButton:*

- Background
    - [list]
- Padding
    - Focusring
        - Text
            - [list]

Now with the default layout the button has no border.

## Ttk states

Ttk has different states depending for example on the mouse button is over the button, the button is pressed etc.

- active
- alternate
- background
- disabled
- focus
- invalid
- pressed
- readonly
- selected

Depending on the state it is possible to further handle the look of a widget. This is done using a style map call.

### *Style map Tbutton:*

- -background [list disabled #d9d9d9 active #ececec]
- -foreground [list disabled #a3a3a3]
- -relief [list {pressed !disabled} sunken]

This means, depending on the state the color for the background or foreground is selected. And the relief is displayed sunken, if the state is pressed and not disabled. So several states can be set for a widget and the combination determines what to do. The mapping rules above are parsed in the order they are found in the configuration file and the match, which fully satisfies the selection criteria is used.

## Ttk theme names

These are the themes currently realized in Ttk:

- alt
- aqua
- clam
- classic
- default

## Ttk like features

Background, border, padding, focusring, text etc. are the widget parts, which are used to construct a button. The look and feel of these parts is influenced by the theme style options and colors. So the same widget button has split off the look and feel from the button code. The look and feel is dependent on the theme style and the them style layout. Clam style would look identical on all platforms, but default style can have different values depending on the platform it is running on.

As a consequence the “native” look and feel is driven by setting the theme style options, colors and layout according to the rules of the platform it is running on and there is a really identical code for the button widget on all platforms without having to look for flags on every platform.

Also the look of a widget can be driven on its state without having to modify the widget's code.

And for bringing a button to a new not yet known platform, there is only the need for preparing the information for the default theme and then a button will look like it should on that platform.

## ntkWidget and Ttk like features

From what has been presented up to here one can see that the mechanisms used in ttk can be easily used for ntkWidget in replacing the current theme class with some classes giving the functionality described for ttk above.

The methods in the current ntkWidget theme class can be adopted to handle Ttk like styles. The renderer can be easily adapted to handle layouts. There is already a similar mechanism there widget parts -> layout elements. The methods in the current ntkWidget theme class can be adopted to handle Ttk state info.

I am working on a version of theme handling for ntkWidget, which is mostly based on the ideas found in the source code of ttk (tile) from Joe English, but my version will be completely written in Tcl also using itcl-ng classes as a base. It is mostly implementing the handling and use of theme style and theme layout for doing layout and styles of ntkWidget. The missing part is the state info. The files describing the style and layout information have a similar format as the ttk ones, but there is no compilation necessary, as all the needed code is written in Tcl (itcl). I have the feeling , that this implementation will only have 30-50% of lines of code compared to the C-implementation, as the use of classes and objects reduces the code extremely.

As the ntkWidgets are constructed from widget parts, there is also no general problem using the ttk layout parts as parts of ntkWidget widgets and building a widget using layout information and rendering that with the renderer using the layout and style information provided.

## Itcl-ng addons for ntkWidget

There are some enhancements implemented in itcl-ng, which are useful - besides other places - for ntkWidget:

- methodvariable
- option
- delegate method
- delegate proc
- delegate option

A methodvariable is a variable, which automatically has a setter/getter method defined with the same name as the variable name. For example with [methodvariable status] in the class body of an itcl-ng class you can do the following: [$object status] will return the contents of variable status and [$object status blocked] will set the contents of variable status to blocked. It is also possible to set a callback function being called when the variable is set, so you can for example do some validation.

An option is a variable which is handled in a special way. All option variables are elements of an object specific array variable itcl_options which is not class but object specific. one can set and get the contents of an option using the builtin configure and cget methods of itcl-ng.

Additionally you can – via parameters in the option declaration – define a method for validating the value before setting the option variable and you can define methods called before setting or getting the value when the configure or cget method for the option is called. That allows you for example to call a redraw function callback when setting the option to a new value. Instead of having a method name as parameter one can also use a variable name and then fill the variable with the method name later on. This is used for setting dynamically callbacks for geometry managers depending on which one is used (grid/pack/place)

Delegate method/proc allows one to delegate the call to a method to for example another objects method (this one can also have a different name). It is very similar to “forwarding”. This allows you to use for example defined itcl-ng widget classes without the need to inherit from them.

Delegate option allows one to delegate the setting and getting of an option to the option of a different object. This is the same strategy as for delegated method/proc.

## Itk and ntkWidget

There might be the question, why not use itk, the megawidget part coming together with itcl. There are some points for not doing that in my opinion.

All widgets are derived from a widget named Archetype, so everybody needs that as a base for widgets, otherwise there is no option handling possible. In ntkWidget you could also build widgets without itcl-ng only using the ntkWidgetImage, render and grid commands without having to inherit any of the ntkWidget classes.

The inheritance of options is somehow a little bit strange, at least for me it does look so. All options are tied to the superclass Archetype, in ntkWidget they are tied to the object of a widget, which I think makes more sense.

Itk is based on the Tk widgets, so same problem concerning hard-wired X11 functionality as Tk. There is also no support for themes, which I think is important for the future.

## Status

NexTk and ntkWidget are both work in progress and there is still a huge amount  of work to be done, before there is chance to have something similar to Tk.

The developers of NexTk and ntkWidget (George Peter Saplin and myself) are working together, so both parts can inherit form the efforts the other part is doing.

There exist demos for NexTk see the NexTk wiki page.

There exist demos for ntkWidget see the ntkWidget wiki page. The demos do not include the ttk like themes, as these are not yet integrated.

## Conclusions

NexTk and ntkWidget could be both a replacement for Tk  in the future, that is my personal opinion. There is a lot of work to be done before that can happen and I am sure there will be a lot of discussion on if that is the right way to go or not.

As  NexTk and ntkWidget are developed after about twenty years of Tk, the technology is more modern and the developers have tried to avoid some of the known problems of Tk. That should also make it easier to maintain the code base in the future.

A big problem is, that there are only two people working in that area, additional developers would speed up the implementation.

What has to be discussed is, if some known incompatibilities to Tk are accepted (example: colors are now lists of r, g, b and alpha value), but that assumes, that there will be a complete reference implementation in the future ☺.

## ntkWidget „hello world“

Looking at the Tcl wiki you will find a Tcl starkit for downloading, which you can use for starting small demo examples.