from v19 R6

ogWidgets

Components common

Programmer's guide
Widgets and Pickers, the "•" anonymous

Version	Date	Writer	Comments
1.0.00	Wednesday, 18 January 2023	OG	first version, Master Candidate

Protée
IT, app design

Protée sarl
31, rue Sainte-Marthe
31000 TOULOUSE

Tél : +33 (0)970 46 56 46
Mob : +33 (0)6 3718 5941
www.protee.org / info@protee.org

Overview

Introduction

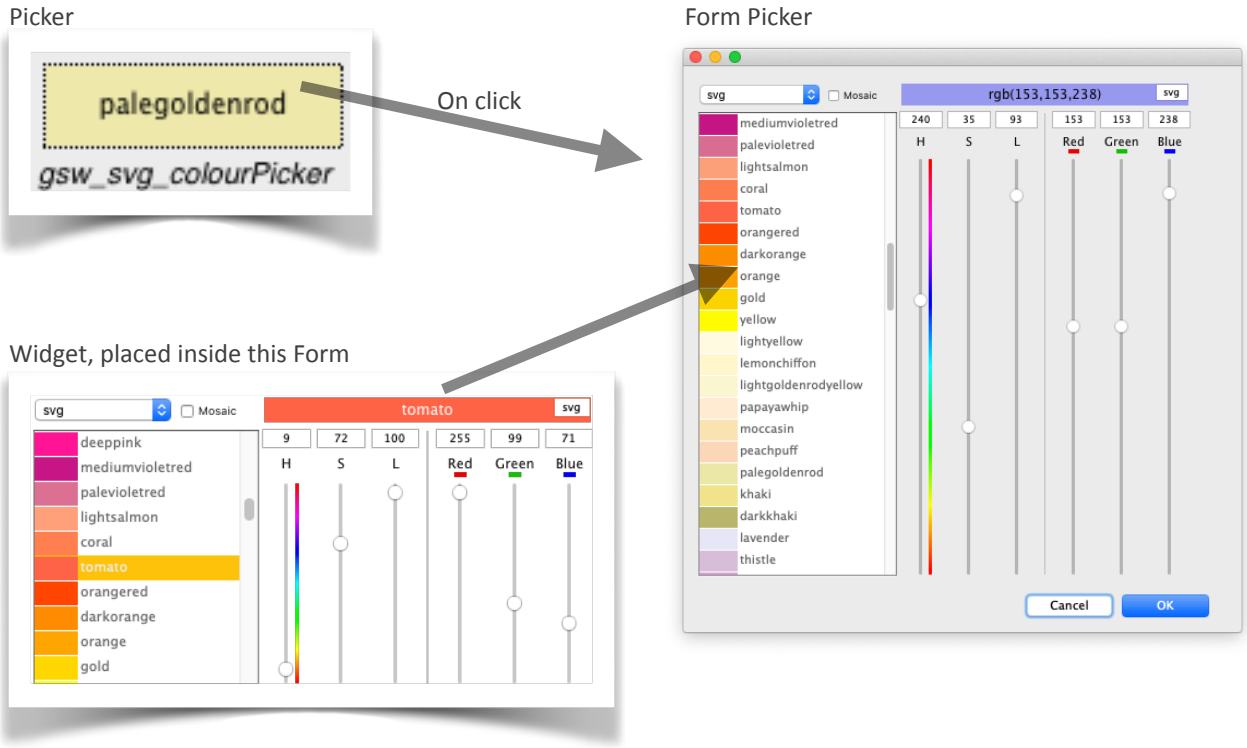
The *ogWidgets* are components as a pool of widgets to bury often used objects and to avoid to see all the “logic” behind and focus on the utilisation, with options and parameters. This approach is very efficient and drives the way you code, gives very homogeneous interfaces. At last, updates but also improvements are supposed to be very less sensitive in the host database.

ogWidgets is designed in an extremely regular and generic way. In this sense, most methods can be used for all widgets and pickers : getters, setters are the same. Here you will find all the Widgets and Pickers residing in all the Protée components database.
This generic document will drive you inside all methods that are common to our widget’s organisation. For technical reason, it is not possible to have those methods common for multiple component’s database.
So we will assume in this documentation that all method are prefixed with “wo•_” and we assume that this “•” will depend of the component you use and has to be replaced with: “s” for svgWidget, “t” for ogTaguthi, “g” for ogTools, “d” for ogDevTools, and so one…

Form objects

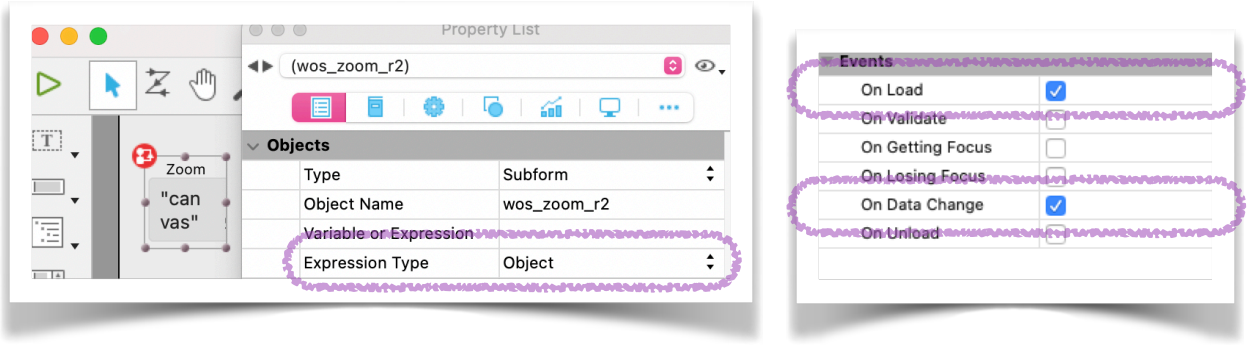
What is the difference between a widget and a picker? Ok, let assume this:

- a widget is a complex set of basic objects to build an interface area.
- a picker is a higher level composition of widgets, and where some of them can open a form with a more complex widget inside.



Form objects initialisation

All the widget and pickers instances are to be created with the Variable Type set on “**Object**”, or “**None**”. The widget embedded initialisation will make it as an “object”. But if you set the type here on object, then the initialisation by svgWidgets will be quicker and is more reliable.
The events must trigger at least “On Load”, and usually “On Data Change”.



Widgets and Pickers, about

As the 4D component’s based environment is not what I would expect related to bounded variable changes, I developed my own environment to deal with component. The cost to pay is you must always call two generic methods in the host instances, “wo•_resize” and “wo•_redraw”.

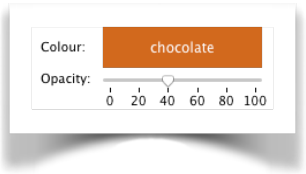
In addition we introduce the notion, like in java, of « overload » which provides a very powerful way to change the parameters. This object given technology by 4D makes possible to design a totally generic component, with no to use of any process variable, for a benefice in memory.

- Each widget/picker has a general initialisation object. This object is like a default parameters for each widget/picker, modifiable, either directly or by overload (specified properties).
- At each created instance, a copy of the general sub-object is made to get a full new set of settings.
- The instance object is obviously modifiable to have instances with different behaviours.
- It is possible to send a partial object to the instance, and by the « overload » mechanism, the settings will only be updated with the partial object sent, others properties remained unchanged.

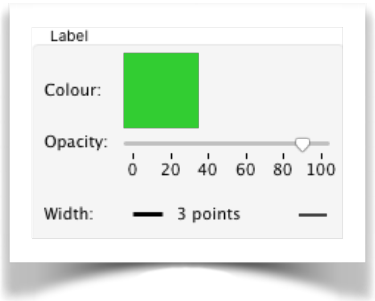
<>ob_GSW_main	{"version":"1.0.00","top":100,"windowType":3,"...able":true},"fontPicker":{"label":"Font","face":
fillPicker	{"label":"Fill","colour":"black","opacity":100,"is_displayText":true,"is_editable":true}
fontPicker	{"label":"Font","face":"Calibri","size":12,"is_col...Text":true,"style":0,"align":0,"is_editable":true}
align	0
colour	"grey"
face	"Calibri"
is_colour	False
is_displayText	True
is_editable	True
label	"Font"
size	12
style	0
markerPicker	{"marker":0,"is_editable":true}
rotatePicker	{"label":"Rotate","colour":"lightslategrey","type":1,"rotate":0,"is_editable":true}
strokePicker	{"label":"Stroke","colour":"black","opacity":10...,"marker":0,"is_marker":true,"is_editable":true}
svg_colourPicker	{"colour":"black","colourName":"","colourRGB...t":true,"is_autoAccept":true,"is_editable":true}
svg_colourWidget	{"colour":"black","colourName":"","colourRGB":"","is_editable":true}
svgActions	{"label":"Actions","action":"SELECT","is_editable":true}
svgEditor	{"label":"svgEditor","documentName":"New d...4,"width":535,"height":130},"is_editable":true}
svgTools	{"label":"Tools","tool":"SELECT","is_sticked":false,"is_editable":true}
top	100
version	"1.0.00"
widthPicker	{"colour":"black","is_displayText":true,"width":0,"is_editable":true}
windowType	3
zoomPicker	{"label":"Zoom","colour":"lightslategrey","type":0,"zoom":100,"is_editable":true}

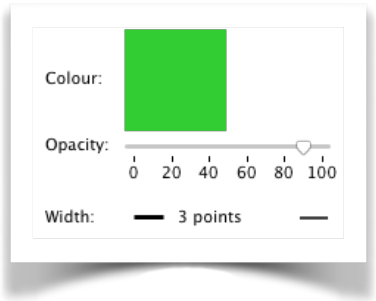
Layout

The widget/picker's host instance frame is free to size, and the content is automatically resized to fill correctly the area you have chosen for your instance.



Some pickers include a Group Box, you can choice the label displayed, and even remove it with an empty label. Those pickers are redrawn accordingly to the Group Box present or not, ie here the colour square is recalculated just having the label empty.





Inside each instances

Here is the typical code you will find into instances.

- On Load: is generated after the instance initialisation, and even a -On Load event is generated
- On Data Change: is generated each time something has changed in the widget/picker.

Example here with a "wos_fill", made with one sub-picker (color) and a ruler.

```
C_LONGINT($evt)
$evt:=Form event

Case of
      : ($evt=On Load)
            C_OBJECT($ob)
            $ob:=Self->
            $ob.is_displayText:=True
            $ob.label:=""
            wo•_resize (Border Dotted)
            $ob.colour:="none"
            $ob.opacity:=40
            wo•_redraw

      : ($evt=On Data Change)
            // From inside
            C_OBJECT($ob)
            $ob:=Self->
            // or: $ob:=wox_getWidget

// From outside
// $colour:=wox_getWidget("wos_colour3")

            // then do what you want with it!
            $colour:=$ob.colour
            $opacity:=$ob.opacity
            $is_displayText:=$ob.is_displayText
            $label:=$ob.label
End case
```

Each time you modify some object properties, at the end when you're done, you must either call
- "wo•_resize" if you resized the instance or if one property or more which touch the disposition inside the widget (here is_displayText or label).
- "wo•_redraw" if you changed one property or more which touch the datas displayed (here colour or opacity).

Access to the object parameter

For all the generic methods, you can pass an optional parameter to indicate which widget/picker you want to address. When empty, this is the current widget/picker you are in from the host instance.

From inside

From inside a host instance widget/picker, there are easy ways to access the object variable:
- $vJ_widget:=Self->
- $vJ_widget:=OBJECT Get pointer(Object current)->
- $vJ_widget:=wo•_getWidget // No parameter

From outside

From outside a host instance widget/picker, there are those ways to access the object variable:
- $vJ_widget:=OBJECT Get pointer(Object named;"widget_instance_name")->
- $vJ_widget:=OBJECT GET VALUE("widget_instance_name") is a better option
- $vJ_widget:=wo•_getWidget("widget_instance_name")

4D and component (sub-forms) management

The 4D's behaviour is a kind of shitty, probably understandable for oldies when computers were slow.
- The only components 4D knows when opening a form are the ones on page 0 and page 1, all the components residing in extra pages are just unknown.
- All the page 0 and 1 components are triggered inside first with an On Load event, then in the host instance.
- Components residing in extra pages will be triggered when you go to that page, only for the inside On Load (no host event), and only after you give back the hand to 4D.
- That is why it's good practice in component's design to generate a -On Load (with minus) event when receiving On Load. Then going to an extra page will components receive an event to let you initialize them.
- If you try to use $ptr:=**OBJECT Get pointer**(Object named;"myWidget"), then $ptr is Null, so there is no way to initialise by yourself components in extra pages before you had landed manually on them.

Object parameters

wo•__storage_prefs

Description

All components use one interprocess object as default values for each host created instance. This main interprocess object is accessible with this method.

wo•__storage_prefs -> vJ			
Parameter	**Type**		**Description**
vJ	object	↩	Gives back the main prefs object of ogWidgets

Property	Type	Default	Contents
t_name	text	svgWidgets	Component name
t_version	text	x.y.zz	Current version of svgWidgets
t_lip_app	text	wos	License app name
l_top	number	100	pixels from top as non usable area for Form windows
al_windowType	collection		“Window type” array: Plain form window; Sheet form window; Pop up form window; Palette form window
l_windowType	number	3	Default window type
t_popup_idle	text	Idle	Label in popup for the “Idle” state.
t_font_default	text	"Lucida Grande"; "Segoe UI"	Current default font
r_fontOffset_coef	number	0.7	Default colour widget in mosaic (else Listbox)
l_svg_scale	number	1	Scale for svg display. You can use 2 or 4 for retina screen.
ui	object		See below

wo•__storage_widgets

Description

All components use this storage object as default values for each created instance. This main object is accessible with this method. Properties are the default values for widget/pickers. They are default values and are listed in each proper component documentation.

wo•__storage_widgets -> vJ			
Parameter	**Type**		**Description**
vJ	object	↩	Gives back the main widgets object of svgWidgets

Methods

Component

wo•_initRegister

Description

This method initialise the full component. You have to call this method before using any of the other methods of *ogWidgets*. You must provide a correct fixed serial, otherwise all the widgets inside will be in the non-editable mode (read only) after 30 minutes.

wo•_initRegister ({host_name} ; {serial})			
Parameter	**Type**		**Description**
{host_name}	text	→	Host name associated to the serial.
{serial}	text	→	Serial of your component.

wo•__releaseNotes_get

Description

This method gives you back informations about *ogWidgets*.

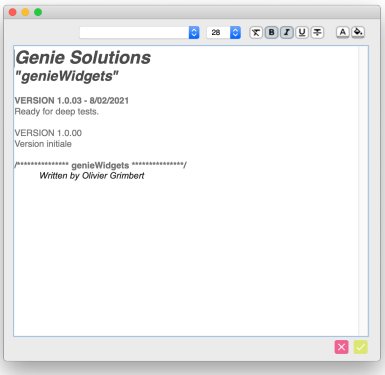
vJ := wo•__releaseNotes_get			
Parameter	**Type**		**Description**
vJ	Object	↩	Informations about svgWidgets. In ob, .product, .version and .notes are available.

wo•__releaseNotes_form

Description

This method opens a form with the releaseNotes.txt text, ready to display or to modify.

wo•__releaseNotes_form ({is_editable})			
Parameter	**Type**		**Description**
{is_editable}	Boolean	→	Read only or editable. Default false.



Instances: data and options getters

All widgets and pickers instances have a name. With this name, you can access all of them individually.
When you are inside the instance, which means inside the method of the instance, and you want to access this own instance, there is no need to give that name, this is done by default.
That is why in all the following methods, the {widget} parameter is optional.

From outside, you can use all those methods but giving this « widget » name to the methods.

wo•_getWidget

Description

This method gives back the object of a widget. If the object does not exist, no error is throw and the resulting object is Null. From inside a host instance, you can access it simply with “Self->”

wo•_getWidget ({widget}) -> vJ_widget			
Parameter	**Type**		**Description**
{widget}	text	→	Widget name, or current widget
vJ_widget	object	↩	Object for this host instance widget

wo•_setWidget

Description

This method allows you to overwrite the destination widget with the properties in vJ_source, letting other untouched. The overwrite is not recursive. This method then is followed with a redraw.

wo•_setWidget ({vJ_source} ; {widget})			
Parameter	**Type**		**Description**
vJ_source	object	→	Object with properties to overload in the widget instance
{widget}	text	→	Widget name, or current widget

wox_vJ_overload

Description

This method allows you to overwrite the target object with the properties in vJ_source, but only for the property names given in parameters, and letting other untouched. The overwrite is not recursive.

wox_vJ_overload (vJ_source ; vJ_target ; {property_name1} ; {…})			
Parameter	**Type**		**Description**
vJ_source	object	→	Object with properties to overload in the widget instance
vJ_target	object	→	Object target
property_name	text	→	List of property names, to overwrite in the widget instance

wo•_setWidgetProperties

Description

This method allows you to overwrite the destination widget with the properties in vJ_source, but only for the property names given in parameters, and letting other untouched. The overwrite is not recursive.
This method then is followed with a redraw.

wo•_setWidgetProperties (vJ_source ; {widget} ; {property_name1} ; {...})			
Parameter	**Type**		**Description**
vJ_source	object	→	Object with properties to overload in the widget instance
{widget}	text	→	Widget name, or current widget
property_name	text	→	List of property names, to overwrite in the widget instance

wo•_redraw

Description

The wo•_redraw method let you redraw the content based on the value stored in the widget, but first outside with the data referred with the pointer if used. You can use it when you modify yourself the value from outside the widget, to refresh it.

wo•_redraw ({widget})			
Parameter	**Type**		**Description**
{widget}	text	→	Widget name, or current widget

wo•_resize

Description

This method is very important and as to be used in almost all widgets on event « on load ». This will:

- resize correctly the widget content to the instance size in the form
- remove any used border for layout purpose. As it is easier to see your widget with a border in a layout, don't worry as this method will set it to « border none » when called without parameters.

wo•_resize ({border} ; {widget})			
Parameter	**Type**		**Description**
{border}	longint	→	Border to apply to widget. By default, « border none » is used. Pass a value <0 to get the border unchanged.
{widget}	text	→	Widget name, or current widget

wo•_getSize

Description

This method allows you to get the recommended size of a widget, based on a layout name.

wo•_getSize (layout ; ptr_width ; ptr_height ; {widget})			
Parameter	**Type**		**Description**
layout	text	→	One of the layout available. For now, “tight” and “basic”.
ptr_width	pointer	↩	recommended width
ptr_height		↩	recommended height
{widget}	text	→	Widget name, or current widget

wo•_setExternalLink

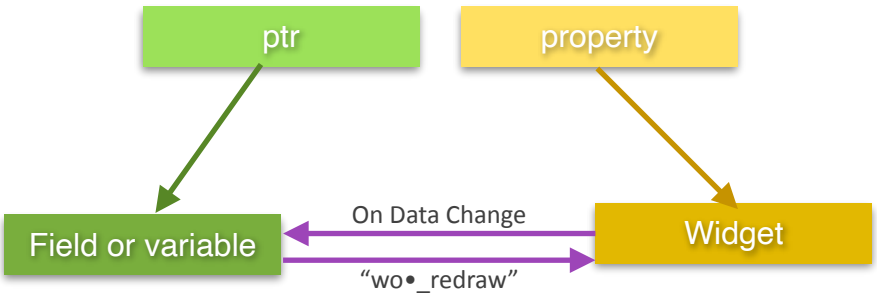
Description

This method is very important to update automatically a variable or a field to a property of a widget.

This will:
- update the property and (redraw) the widget when this method is called
- update the widget from the pointer at each redraw with “wox_redraw” call.
- update the pointed object when the widget is modified - event “On Data Change”.

This is useful to link a field with a widget: all the code is inside the widget method.

wo•_setExternalLink (ptr ; property ; {widget})			
Parameter	**Type**		**Description**
ptr	pointer	→	Pointer to a variable or a field to update automatically.
property	text	→	Property name to sync with the pointed variable or field.
{widget}	text	→	Widget name, or current widget



Implementation

The way you implement an ogWidgets depends on if you put it on a Form on page 1 or on a page higher than 1.

Page 1 is the most simple, but for pages higher, 4D subForm event manager is just shitty, as every time you land on that page, a On Load event is triggered, and when you leave that page, a On Unload event is triggered too, making the eventually stored data (Dom for xml, svg, list…) being cleared. Here is how to deal with that in order to navigate through pages, and keep the svgEditor consistent with its own datas.

- -On Load event is generated only once, that guarantees you will initialise the widget only once.
- The inside widget On Unload event is disableable, to let the widget keep the Dom not closed.
  - but, in the host On Unload event, you will have to close the Dom by yourself.