Making New Themes
For Kvantum

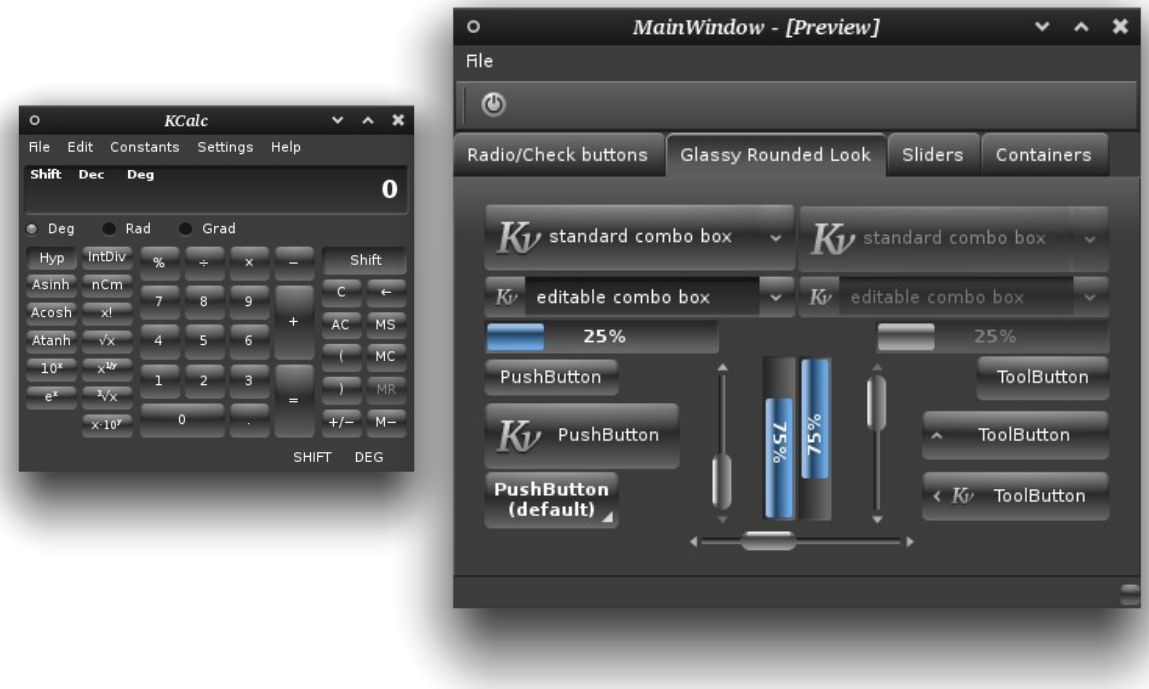
Please install Kvantum, read the file “Theme-Config”, and create the configuration folder before reading this document! A basic knowledge of Inkscape is also presupposed here.

Making new themes may take time but its logic is not complex. Each Kvantum theme consists of a configuration file (explained in “Theme-Config”) and an SVG image. Both should have the same name – ***MY_THEME***, for example – and be put into the same folder *“~/.config/Kvantum/**MY_THEME**”*. This document explains how you could create an SVG image for your new theme.

But let us first try an alternative theme that is already included in the source. Its name is ***Glassy***. The folder *“doc/**Glassy**”* contains the config file *“**Glassy**.kvconfig”* and the SVG image “***Glassy**.svg”*. If you install ***Glassy*** and change the active theme to it with **Kvantum Manager**, the folder “Glassy” will be created inside *“~/.config/Kvantum/”*, the above-mentioned files will be put into it, and the file “*~/.config/Kvantum/kvantum.kvconfig*” will contain:

theme=***Glassy***

Now run any Qt or KDE application and see the difference:



Scrollbar and buttons have a glassy look with rounded edges, tabs, progressbars and line-edits also have rounded edges, tabs are left aligned and the active tab is attached to the tab widget. If you open “*Glassy.svg”* with Inkscape, you will find just a few objects in it. Kvantum first searches that image for

the widget parts and if it does not find the relevant object names, it will go to the image of the default theme. That is similar to what Kvantum does with the configuration files, as was explained in "Theme-Config".

For instance, to not show scrollbar grip indicators of the default theme, invisible rectangles with names "grip-normal", "grip-focused" and "grip-pressed" are created in *"Glassy.svg"*. On the other hand, in the same image, there is no object for the interior of progressbar patterns but just objects for their frame, so that the new frame is used alongside the default interior.

The number of objects you create inside your SVG image depends on how much you want your theme to be different from the default one. The easiest way is to start with the default SVG image itself. The file *"default.svg"* in the "doc" folder is the image for the default theme. It contains useful comments on various objects. *(Please do not use the image with the same name in the folder "style/themeconfig/" because it is cleaned with **SVG Cleaner** and not only does not contain any comment, the groupings of its objects could also be misleading!)* You could change the objects one by one in whatever way you prefer, delete those objects you do not want to change, put invisible rectangles in place of those you want to omit, and even add new objects.

Do not forget that the look of your theme is determined by its config file too. Also note that your theme will be used together with a color scheme of your choice. Therefore, select colors and gradients carefully, so that they match your color scheme. Yes! There may be a lot of work to do but it is what you pay for being able to control virtually every aspect of each widget.

After you have finished your work with the image, first back it up and then, preferably, clean it with *SVG Cleaner*. It is a nice tool that can reduce the size of an SVG image considerably. In this way, the memory footprint will be minimized. If *SVG Cleaner* is not in the repository of your Linux distro, you could get its latest source from *https://github.com/RazrFalcon/SVGCleaner*. If you have used the cloning menu-item of Inkscape (which links similar objects) as far as possible, the image will have the minimum size.

To make your theme available to others, put these three files in a folder named ***MY_THEME***:

MY_THEME.svg (the SVG image)
MY_THEME.kvconfig (the Kvantum config file)
MY_THEME.colors (a KDE color scheme)

In this way, your users could install and choose your theme easily with "Kvantum Manager", which is a simple GUI made for that purpose.

That is the basic logic behind making themes for Kvantum. Now, we pay attention to some details in the following sections:

Elements
Interior and Frames
Indicators
Flat indicators and Hight Contrast
States
Orientations
Inactiveness

The Default (Push) Button
Inheritance and Alignment
Patterns
Junctions for Tab Widgets
Floating Tabs
Translucency and Shadow for Menus and Tooltips
Blurring for Menus and Tooltips
Window Translucency
Maximum Corner Roundness (Frame Expansion)

Elements

Each section of the config file – except for the General, GeneralColors and Hacks sections – determines the look of a widget by setting the elements that are used to draw it (see Sections Table in "Theme-Config"). Usually, there are three kinds of elements, namely, *frame*, *interior* and *indicator*. Some widgets may not need all of them and some may need more.

The basic names of elements are mostly optional but there are a few exceptions, i. e. the names of dial elements (dial, dial-notches, dial-handle) MDI titlebar buttons (mdi-maximize, mdi-restore, mdi-minimize, mdi-close, mdi-shade, mdi-menu), and the default button indicator (button-default-indicator).

Interior and Frames

The *names* (or *id* strings) of the rectangular objects, that are used to draw the frame and interior elements of a widget, depend on its state. There are five states at most: *normal*, *focused*, *pressed*, *toggled*, and *disabled*. For each state, there are at most nine rectangular objects: one for the interior and eight for the frame.

Each interior object should have a name (id) with this format:

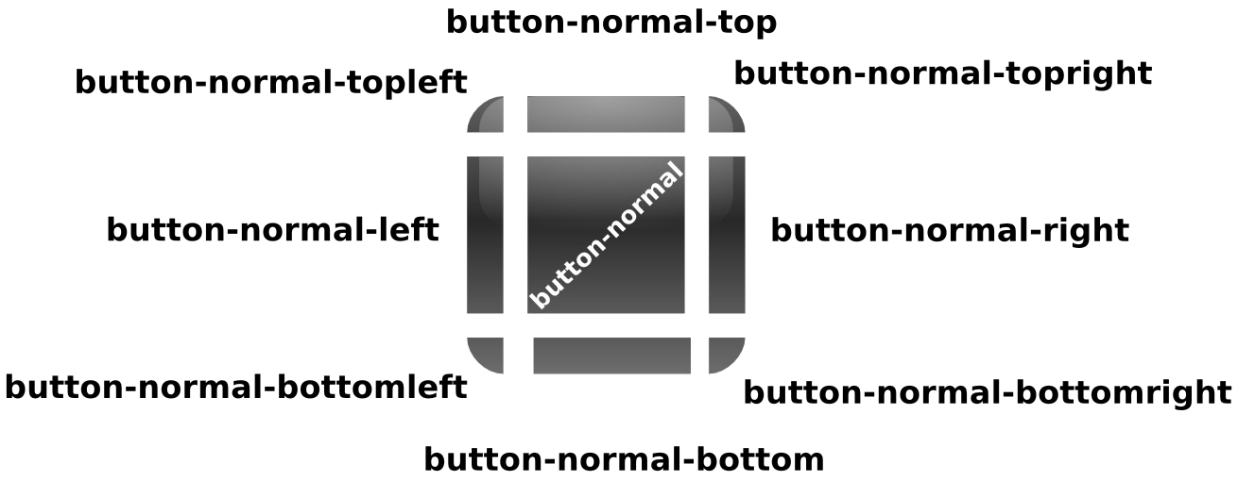
iNAME-STATE

Here, iNAME is set as the string value of "interior.element" in the config file, and STATE is the state of the widget the object represents. The name of each frame object should be:

fNAME-STATE-POSITION

Where fNAME is set as the string value of "frame.element" in the config file, and POSITION could be *top*, *bottom*, *left*, *right*, *topleft* , *topright* , *bottomleft* or *bottomright*.

For example, the following image shows the names of the nine objects that together draw the normal state of a button widget, whose frame and interior names are both "button". It could be defined in the "PanelButtonCommand" or "PanelButtonTool" section of the config file.



Not all widgets have frame. For example, windows do not have frame and it will make no difference whether you set *"frame=true"* and define a frame element under the *Window* section.

Also some framed widgets may not exactly obey the frame widths provided under their corresponding sections. For example, the frame of a tooltip has a uniform thickness on all sides, which is equal to the maximum of the frame widths defined under the *Tooltip* section. Or if there is not enough space for a button, all of its frame widths will be set to 3px at most.

Indicators

An indicator is a sign or icon on a widget that shows some action is available or informs the user of something about that widget.

For instance, some tool buttons have "arrow indicators," which will show a drop-down menu if pressed. Arrow indicators also appear on combo boxes to show that other options are available. Or the handle of a scrollbar slider may have an indicator that can make it easier to find it. The close button of a tab can be seen as an indicator too. And so on.

This is the list of all indicators Kvantum draws:

Section	Indicator
TreeExpander	Indicators showing that a tree branch is expanded or not.
IndicatorSpinBox	Up/down and plus/minus indicators for spin widgets.
HeaderSection	The sorting indicator for headers in item views.
DropDownButton	Indicator that shows a drop menu is available.
Tab	Tab close button and also tab-tear indicator.
IndicatorArrow	Left/right/up/down arrows used in various widgets.

Scrollbar	Indicators for scrolling.
ScrollbarSlider	A decorative indicator on the slider of a scrollbar. Also glows at its top and bottom.
Toolbar	The handle of a floatable toolbar; also toolbar separators.
SizeGrip	Window resize indicator.
PanelButtonCommand	An indicator for the default push button, another one showing that the button has a drop menu, and also arrow indicators for tool buttons.
SliderCursor	That handle of a slider (a volume control, for example).
TitleBar	Maximize/restore/minimize/close/shade/menu buttons of the titlebar of a QmdiSubWindow.
MenuItem	The tear-off indicator for detachable menus; also the menu-item separators and submenu/scroller arrows. As an exception, if there is no submenu/scroller arrow element for menu-items in the SVG image, those of IndicatorArrow will be used.
Splitter	An indicator for the handle of a splitter. Also a decorative indicator on it.

“Flat” Indicators and High Contrast

Like other widgets, tool or push buttons can have different backgrounds and their text colors should be set appropriately to have enough contrast with their background colors. But unlike other widgets, tool and push buttons can be flat, in which case no background (panel) is drawn for their normal state. In such cases, Kvantum automatically sets their text color to that of the widget behind them (toolbar, menubar or any container whose text color can be set). It will also use “flat” indicators instead of the usual ones *if* they exist in the SVG image and *if* the usual indicators do not have enough contrast with the widget behind flat buttons.

The names of flat indicator objects are made by adding the string **“flat-”** to the beginning of the names of usual indicators. For example, if the name of the indicator element under the *PanelButtonCommand* section is “arrow”, extra objects with names “***flat-****arrow-up-normal”*, “***flat-****arrow-down-normal”*, …, “***flat-****arrow-right-focused”, “****flat-****arrow-left-pressed”*, etc. could be added to the SVG image. Also the default button indicator, whose name is always *button-default-indicator*, can have a flat counterpart with the name ***flat-button-default-indicator***.

If Kvantum does not found the “down-normal” objects of “flat” indicators, it will use the usual ones for drawing the indicators of flat buttons. “Flat” indicators are good only when there is a high contrast between the background color of buttons and that of widgets behind them, for example, when dark buttons with white texts and indicators are used together with light containers.

To determine whether there is a high contrast, Kvantum relies on the value of *text.normal.color*. Therefore, ***apart from textless widgets, only the interior elements of those widgets that accept state-***

specific text colors can have a high contrast with the window or base background. For example, menubars, toolbars and buttons have state-specific text colors but generic frames or tab frames do not (see the note in the explanation of the key *text.normal.color* in the file *Theme-Config.pdf)*. As a result, the interior colors of toolbars, menubars or buttons can have a high contrast with the window color (the value of *window.color* under the section *GeneralColors*), provided that their normal text colors are set correctly in the config file. However, the interior colors of generic frames or tab frames (if they have any interior element at all) should NOT have a high contrast with the window color because *text.normal.color* has no meaning for them.

States

As mentioned before, there are five states at most: *normal*, *focused*, *pressed*, *toggled*, and *disabled*. You do not need to draw any object for the disabled state of interiors or frames because they are automatically created based on the normal state by reducing its opacity.

However, the disabled states of all *menu-items* and *menubar-items* and also those of most *indicators* should be included in the SVG image because, for example, we may want disabled indicators to be totally invisible or have a neutral color.

Not all widgets have all the possible states. For example, toolbars only have the normal and disabled states, line-edits can only be in a normal, focused or disabled state, etc. On the other hand, the SVG elements used for drawing frame focus rectangle (under the *Focus* section) cannot have any state because they are used for distinguishing some widgets that already have keyboard focus.

You could know about the possible states by examining the image *"doc/default.svg"* with Inkscape. Not drawing redundant objects not only saves your time but also reduces the memory usage.

Orientations

Some widgets, like scrollbars, can be oriented both vertically and horizontally; some others, like tabs, have even more orientations. Even if you use gradients, you will need to draw objects only for one of the possible orientations, which may be different based on which orientation a widget most commonly has in various applications. There is no consensus about that but these are the orientations you should use when you draw objects for Kvantum:

Widget	Orientation
Scrollbar (slider, groove, indicator, grip)	Vertical
Slider groove (like in volume controls)	Vertical
Slider Handle	Vertical with tick marks to the right of the slider. *(The handle will be rotated or mirrored only if its width and height are different.)*
Splitter Handle	Vertical (which means that the splitter itself is horizontal technically)

Progressbar (groove, pattern/indicator)	Horizontal
Tab	Horizontal (and top)
Toolbar	Horizontal
Toolbar Handle (for floatable toolbars)	Vertical (the toolbar itself is horizontal)
Toolbar Separator (between toolbar buttons)	Vertical (the toolbar itself is horizontal)

Kvantum automatically draws the other orientation(s) for each of the above widgets. There is only one exception and it is the arrow indicators of the "IndicatorArrow" section, all of whose orientations should be included. It is better to draw all orientations of the arrow indicators of the "MenuItem"section too but if they are missing, those of "IndicatorArrow" will be used.

Inactiveness

The window containing a widget may not have keyboard focus, in which case it is said to be inactive. Inactive windows are usually distinguished from the active one by their title-bars.

In Kvantum, inactiveness of a widget means that its window is inactive. Inactiveness can be considered as a sub-state so that, for each state of an SVG object, an "inactive" counterpart can be added. The name of such objects should have the string *"-inactive"* after their state strings. For example:

E-normal**-inactive**(-top/-bottom/...)
E-focused**-inactive**(-top/-bottom/...)
E-pressed**-inactive**(-top/-bottom/...)
E-toggled**-inactive**(-top/-bottom/...)
E-disabled**-inactive**(-top/-bottom/...)

Where "E" is the name of the element that object draws, as it appears in the config file.

This feature is completely optional and is not used in the default theme. If "inactive" objects are present, they will be used for drawing widgets on inactive windows; otherwise the usual objects will be used for drawing widgets on both active and inactive windows.

The Default (Push) Button

The default push button is the one that has the keyboard focus. There are two (optional) ways to make it distinct: (1) adding a default button indicator; and/or (2) giving a default frame to it.

The default button indicator has no state and the name of its SVG object is always ***"button-default-indicator"***. When drawn on the button, its size is equal to the value of the key *"indicator.size"* under the *PanelButtonCommand* section and its place is on the right or left bottom corner of the widget for LTR or RTL layout direction respectively.

The names of the default frame SVG objects are made by adding the string ***"-default"*** to the end of the name of button frame element under the *PanelButtonCommand* section. For example, if the latter is "button", the default button SVG objects should be named as *"button-default-top"*, *"button-default-topleft"*, *"button-default-topright"*, etc. Since the interior element (*"button-default"*, in this example) is not used in drawing, you do not need to include it. Like the default indicator, the default frame does not have any state.

Inheritance and Alignment

Although the key *inherits* can be used under various widget sections for not repeating identical properties, its use can result in an interesting visual effect too. For example, if this key is used under the *ComboBox* section as "inherits=PanelButtonCommand", and provided that no frame width or text margin is specified, the frame widths and text margins of combo-boxes will be equal to those of push-buttons. As a result, if a combo-box is located adjacent to a push-button horizontally and if both of them either have icon or are iconless, they will look aligned; in other words, their top as well as bottom borders will be on the same level. The same thing can be said about tool-buttons (under the section *PanelButtonTool*), line-edits and spin-boxes.

Therefore, if you want the horizontally adjacent widgets to look aligned as far as possible, you could rely on the key *inherits* and set it to PanelButtonCommand, as the best candidate for inheritance.

Patterns

A pattern is an image used for tiling the interior of an element. The interior is tiled with it when, at least, one of the keys *interior.x.patternsize* or *interior.y.patternsize* has a positive value under the corresponding section. (The absence of these keys means no tiling.) These keys show how the interior is tiled with the pattern in the horizontal and vertical directions, respectively. If one of them is zero, there will be no tiling in its direction.

The SVG object used for tiling is the interior object itself unless there is another SVG object whose name is made by adding ***"-pattern"*** to the end of the name of the interior object, in which case, the pattern is drawn as a tiled layer over the interior background. Needless to say, in the latter case, the pattern object should have some translucency for the background to be seen behind it.

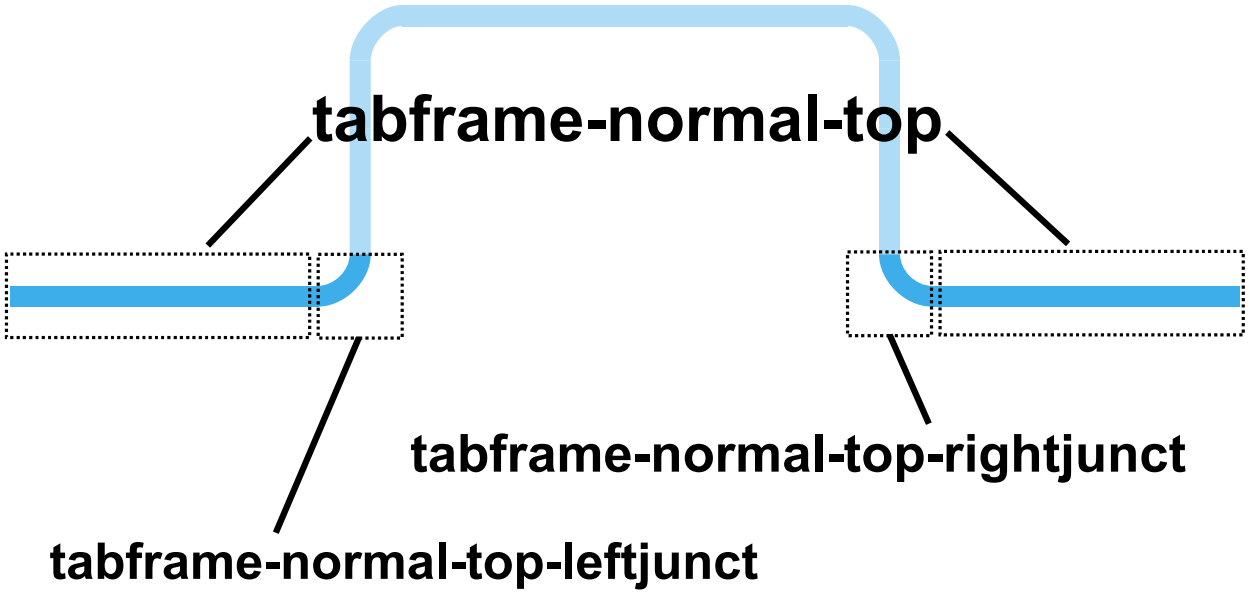
For example, if the Window section has an interior element called "window" and if, at least, one of the pattern sizes is greater than zero, the SVG object "window-normal" will be used for tiling the background of windows and dialogs (see Interior and Frames). However, if an object with the name "window-normal-pattern" is also present, the background will be drawn with "window-normal" *without tiling* and then, "window-normal-pattern" will be used for tiling over it.

Also note that some widgets never accept patterns, even when the pattern size keys have positive values for them. They are widgets, like grouped toolbar buttons, spinbox buttons and view-items , for which a pattern does not have much meaning.

Junctions between Attached Active Tabs and Tab Widgets

If the key *attach_active_tab* is true, active tabs will be attached to their tab widgets. In fact, the frame element from the section *TabFrame* will be cut under an active tab. Of course, the bottom frames of tabs should be drawn so that they appear really attached to their tab widgets. If the frames in the sections *Tab* and *TabFrame* are thin, everything will be all right. But in case of thick frames or when you want to customize the two junctions between the left and right frames of the tab, on the one hand, and the cut frame of the tab widget, on the other hand, you could add extra SVG objects, whose names are those of the frame elements of *TabFrame* plus the two strings "**-leftjunct**" and "**-rightjunct**".

Provided that the frame name under TabFrame is *tabframe*, the following image shows two of these extra objects for the top frame of a tab widget.



Three other pairs of such objects for left, right and bottom frames should also be drawn appropriately if such junctions are used at all.

"Floating" Tabs

In some applications, the tab-bar has no tab widget or the latter is in the "document mode", so that tabs seem "floating". If you have used tab junctions or chosen shapes suitable for attaching tabs to a tab widget, you might want to choose different shapes for the tab interior and (bottom) frames in such cases. You could do so by adding another set of objects, whose names are made by adding the string **"floating-"** to the beginning of the names of the original tab objects.

For example, if the interior and frame elements under the *Tab* section are named "tab", the names of the extra "floating" objects will be *"**floating-**tab-normal"*, *"**floating-**tab-normal-left"*, … , *"**floating-**tab-*

toggled”, *“**floating-**tab-toggled-left”*, etc.

This feature is optional, of course. If Kvantum finds an ***interior*** object for the normal floating tab (*“floating-tab-normal”* in the above example), it will use the floating objects for drawing tabs when there is no tab widget or when it is in the document mode; otherwise, it will use ordinary tabs in all places.

Translucency and Shadow for Menus and Tooltips

If compositing is enabled, menus and tooltips can be translucent and/or have shadow. The following explanation is for menus but it applies to tooltips as well.

Let us suppose that the name of the menu element is “menu”, as is the case with the default theme. So, the names of its corresponding SVG objects are *menu-normal*, *menu-normal-top*, *menu-normal-topleft*, *menu-normal-left*, etc. If these nine objects have translucency, the menus will be translucent when compositing is available.

To have shadow, we should set the key *menu_shadow_depth* to a positive value and also add another group of frame objects, whose names include the word “shadow” as the second word in their names, i.e. *menu-**shadow**-top*, *menu-**shadow**-topleft*, *menu-**shadow**-left*, etc. The shadow and the menu frame should together be divided into these eight objects. For example, *menu-**shadow**-top* draws the shadow for the top part of menus and when menus have frame, it also includes the top part of their frame. The object *menu-normal* is used, with or without shadow, for drawing the interior of menus.

The keys *menu_shadow_depth* and *tooltip_shadow_depth*, in the General section of the configuration file, control the width of menu and tooltip shadows respectively.

If the key *composite* is set to false or the environment does not support compositing, menus and tooltips will be drawn without translucency and shadow and with a frame width of, at most, 2px.

Blurring for Menus and Tooltips: “Shadow Hint” Rectangles

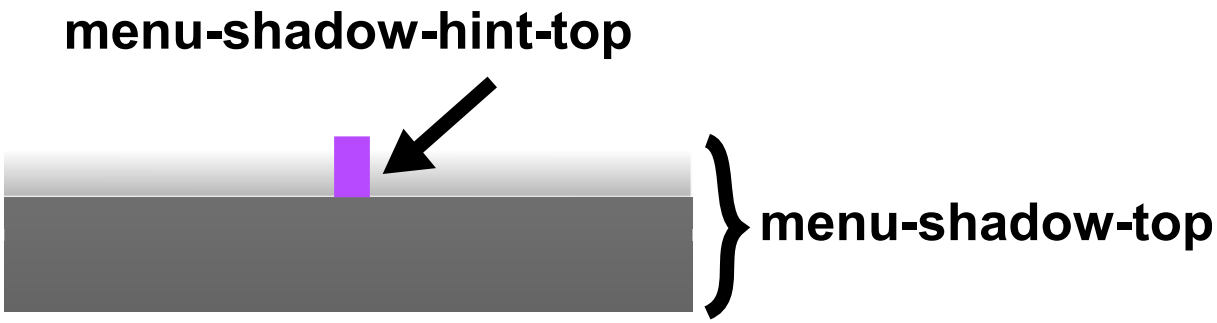
If (pop-up) blurring is enabled (which is possible only under KDE), the regions behind translucent menus and tooltips will be blurred. But for blurring not to include their shadows, extra “shadow-hint” rectangles should be appropriately drawn to inform Kvantum about pure shadows.

For example, suppose that the name of the menu element is “menu”. Then, there are four “shadow-hint” rectangles with these names:

*menu-**shadow-hint**-top*

*menu-**shadow-hint**-bottom*
*menu-**shadow-hint**-left*
*menu-**shadow-hint**-right*

They determine the height or width of the *purely shadowy* parts of the top, bottom, left and right frames respectively. For example, the height of *menu-**shadow-hint**-top* is that of *menu-**shadow**-top* minus the height of the top frame included in it, etc. The following image shows this:



Therefore, only the heights of *menu-shadow-hint-top* and *menu-shadow-hint-bottom* are important, while for *menu-shadow-hint-left* and *menu-shadow-hint-right*, only the widths are pertinent.

The same is true for tooltips, of course. So, there can be eight “shadow-hint” rectangles in total.

Even if you do not use blurring with your theme, it is a good idea to include these eight rectangles when your menu and tooltip objects are translucent because the user might enable blurring with **Kvantum Manager** later.

Window Translucency

Whole windows and dialogs can be made translucent. That needs compositing, a true value for *translucent_windows* in the configuration file and a translucent SVG image for the interior element of the Window section. As is the case with menus and tooltips, there will be no translucency if compositing is not enabled in the configuration file or not supported by the environment.

Some applications are not compatible with window translucency and may show totally transparent windows or even crash. They are usually Qt video playing applications (although some Qt-based video players, like VLC, support window translucency). You could exclude them by adding the names of their executable files to the *opaque* key.

Maximum Corner Roundness and Frame Expansion

Although the four SVG objects used for drawing corner frames (-topleft, -topright, -bottomleft, -bottomright) can be quite curved, the degree of corner roundness depends on the frame widths too and so, it cannot be high.

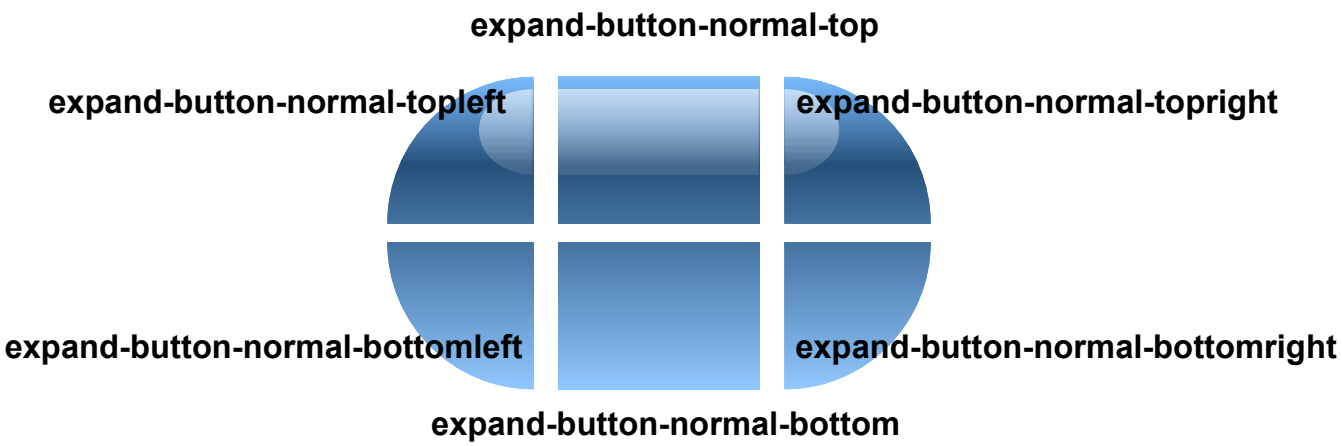
However, Kvantum has a key that can expand frames, namely ***"frame.expansion"***. If its value is greater than zero under any widget section , the frames of that widget will be expanded until the corner frames meet each other either horizontally or vertically, depending on the aspect ratio of the actual widget, *provided that at least the height or the width of the actual widget is less than or equal to the value of "frame.roundness"*. The expansion is so that the corner frame objects become equal squares even when they are not drawn as SVG squares.

You could use this key to make widget corners as rounded as possible. By giving a positive value to *"frame.expansion"*, you could not only decide which widgets have extremely rounded corners but also set size limits, beyond which, they should have ordinary corners (because too big widgets would look weird with completely rounded corners).

The SVG objects used for drawing the completely rounded corners can be the usual ones, in which case you would not need to add anything to your SVG image. But, except for totally flat objects (which do not have color gradient), you might want to add extra objects for maximally rounded widgets. If so, you should name them by adding the prefix ***"expand-"*** to the beginning of the usual object names. For example, for maximally rounded buttons with the frame element named as "button" under the *PanelButtonCommand* section, the names of the objects that are used specifically for complete rounding are *"expand-button-normal-top"*, *"expand-button-normal-topleft"*, *"expand-button-normal-topright"*, …, *"expand-button-focused-top"*, *"expand-button-focused-topleft"*, *"expand-button-focused-topright"*, etc. The interior, left or right objects are not used in drawing and you do not need to include them. However, Kvantum looks for the ***top objects*** (whose names end with "-top") and only if it founds them, it will use the other *"expand-"* objects. Of course, you should draw the corner objects rounded and give all objects appropriate color gradients if any.

In case of *"expand-"* objects, the color gradients of opposite frame objects should be complementary, so that when they are adjacent to each other due to frame expansion, a smooth gradient forms.

The following image can serve as an example:



As you can see, in the above image, there are no objects for the interior, the right frame or the left frame. It is obvious why the interior object is redundant. As for the right and left frames, if the height of widget is greater than its width, Kvantum first rotates its rectangle by 90 degrees, draws it by using the available objects and then rotates it by 90 degrees again but in the opposite direction, so that right and left objects are not needed (this is only a rough description). The reason is that the widget looks more natural in this way.

The key *"frame.expansion"* can also be used for purposes other than corner rounding. The main idea is to make the corner objects as big as possible so that the left and right (or top and bottom) sides of widgets get their shapes.

* * *

Anyway, by renaming *"default.svg"* to *"**MY_THEME**.svg"*, putting it alongside the file *"**MY_THEME** .kvconfig"* in *"~/.config/Kvantum/**MY_THEME**"*, and playing with them, you could learn more about theme-making than by reading any document.

After every change you make to the SVG image or config file, you could see how various widgets look by clicking on the **Preview** button of **Kvantum Manager** or by entering the command *kvantumpreview* in terminal.