NAME

background – manage terminal background

SYNOPSIS

```
urxvt --background-expr 'background expression'
      --background-border
      --background-interval seconds
```

QUICK AND DIRTY CHEAT SHEET

Just load a random jpeg image and tile the background with it without scaling or anything else:

```
load "/path/to/img.jpg"
```

The same, but use mirroring/reflection instead of tiling:

```
mirror load "/path/to/img.jpg"
```

Load an image and scale it to exactly fill the terminal window:

```
scale keep { load "/path/to/img.jpg" }
```

Implement pseudo-transparency by using a suitably-aligned root pixmap as window background:

```
rootalign root
```

Likewise, but keep a blurred copy:

```
rootalign keep { blur 10, root }
```

DESCRIPTION

This extension manages the terminal background by creating a picture that is behind the text, replacing the normal background colour.

It does so by evaluating a Perl expression that *calculates* the image on the fly, for example, by grabbing the root background or loading a file.

While the full power of Perl is available, the operators have been design to be as simple as possible.

For example, to load an image and scale it to the window size, you would use:

```
urxvt --background-expr 'scale keep { load "/path/to/mybg.png" }'
```

Or specified as a X resource:

```
URxvt.background.expr: scale keep { load "/path/to/mybg.png" }
```

THEORY OF OPERATION

At startup, just before the window is mapped for the first time, the expression is evaluated and must yield an image. The image is then extended as necessary to cover the whole terminal window, and is set as a background pixmap.

If the image contains an alpha channel, then it will be used as-is in visuals that support alpha channels (for example, for a compositing manager). In other visuals, the terminal background colour will be used to replace any transparency.

When the expression relies, directly or indirectly, on the window size, position, the root pixmap, or a timer, then it will be remembered. If not, then it will be removed.

If any of the parameters that the expression relies on changes (when the window is moved or resized, its position or size changes; when the root pixmap is replaced by another one the root background changes; or when the timer elapses), then the expression will be evaluated again.

For example, an expression such as `scale keep { load "$HOME/mybg.png" }` scales the image to the window size, so it relies on the window size and will be reevaluated each time it is changed, but not when it moves for example. That ensures that the picture always fills the terminal, even after its size changes.

EXPRESSIONS

Expressions are normal Perl expressions, in fact, they are Perl blocks – which means you could use multiple lines and statements:

```
scale keep {
   again 3600;
   if (localtime now)[6]) {
      return load "$HOME/weekday.png";
   } else {
      return load "$HOME/sunday.png";
   }
}
```

This inner expression is evaluated once per hour (and whenever the terminal window is resized). It sets *sunday.png* as background on Sundays, and *weekday.png* on all other days.

Fortunately, we expect that most expressions will be much simpler, with little Perl knowledge needed.

Basically, you always start with a function that ‘‘generates’’ an image object, such as `load`, which loads an image from disk, or `root`, which returns the root window background image:

```
load "$HOME/mypic.png"
```

The path is usually specified as a quoted string (the exact rules can be found in the perlop manpage). The *$HOME* at the beginning of the string is expanded to the home directory.

Then you prepend one or more modifiers or filtering expressions, such as `scale`:

```
scale load "$HOME/mypic.png"
```

Just like a mathematical expression with functions, you should read these expressions from right to left, as the `load` is evaluated first, and its result becomes the argument to the `scale` function.

Many operators also allow some parameters preceding the input image that modify its behaviour. For example, `scale` without any additional arguments scales the image to size of the terminal window. If you specify an additional argument, it uses it as a scale factor (multiply by 100 to get a percentage):

```
scale 2, load "$HOME/mypic.png"
```

This enlarges the image by a factor of 2 (200%). As you can see, `scale` has now two arguments, the `200` and the `load` expression, while `load` only has one argument. Arguments are separated from each other by commas.

Scale also accepts two arguments, which are then separate factors for both horizontal and vertical dimensions. For example, this halves the image width and doubles the image height:

```
scale 0.5, 2, load "$HOME/mypic.png"
```

IF you try out these expressions, you might suffer from some sluggishness, because each time the terminal is resized, it loads the PNG image again and scales it. Scaling is usually fast (and unavoidable), but loading the image can be quite time consuming. This is where `keep` comes in handy:

```
scale 0.5, 2, keep { load "$HOME/mypic.png" }
```

The `keep` operator executes all the statements inside the braces only once, or when it thinks the outcome might change. In other cases it returns the last value computed by the brace block.

This means that the `load` is only executed once, which makes it much faster, but also means that more memory is being used, because the loaded image must be kept in memory at all times. In this expression, the trade-off is likely worth it.

But back to effects: Other effects than scaling are also readily available, for example, you can tile the image to fill the whole window, instead of resizing it:

```
tile keep { load "$HOME/mypic.png" }
```

In fact, images returned by `load` are in `tile` mode by default, so the `tile` operator is kind of

superfluous.

Another common effect is to mirror the image, so that the same edges touch:

```
mirror keep { load "$HOME/mypic.png" }
```

Another common background expression is:

```
rootalign root
```

This one first takes a snapshot of the screen background image, and then moves it to the upper left corner of the screen (as opposed to the upper left corner of the terminal window)– the result is pseudo-transparency: the image seems to be static while the window is moved around.

COLOUR SPECIFICATIONS

Whenever an operator expects a ''colour'', then this can be specified in one of two ways: Either as string with an X11 colour specification, such as:

```
"red"               # named colour
"#f00"              # simple rgb
"[50]red"           # red with 50% alpha
"TekHVC:300/50/50"  # anything goes
```

OR as an array reference with one, three or four components:

```
[0.5]               # 50% gray, 100% alpha
[0.5, 0, 0]         # dark red, no green or blur, 100% alpha
[0.5, 0, 0, 0.7]    # same with explicit 70% alpha
```

CACHING AND SENSITIVITY

Since some operations (such as `load` and `blur`) can take a long time, caching results can be very important for a smooth operation. Caching can also be useful to reduce memory usage, though, for example, when an image is cached by `load`, it could be shared by multiple terminal windows running inside urxvtd.

`keep { ... }` caching

The most important way to cache expensive operations is to use `keep { ... }`. The `keep` operator takes a block of multiple statements enclosed by `{}` and keeps the return value in memory.

An expression can be ''sensitive'' to various external events, such as scaling or moving the window, root background changes and timers. Simply using an expression (such as `scale` without parameters) that depends on certain changing values (called ''variables''), or using those variables directly, will make an expression sensitive to these events – for example, using `scale` or `TW` will make the expression sensitive to the terminal size, and thus to resizing events.

When such an event happens, `keep` will automatically trigger a reevaluation of the whole expression with the new value of the expression.

`keep` is most useful for expensive operations, such as `blur`:

```
rootalign keep { blur 20, root }
```

This makes a blurred copy of the root background once, and on subsequent calls, just root-aligns it. Since `blur` is usually quite slow and `rootalign` is quite fast, this trades extra memory (for the cached blurred pixmap) with speed (blur only needs to be redone when root changes).

`load` caching

The `load` operator itself does not keep images in memory, but as long as the image is still in memory, `load` will use the in-memory image instead of loading it freshly from disk.

That means that this expression:

```
keep { load "$HOME/path..." }
```

Not only caches the image in memory, other terminal instances that try to `load` it can reuse that in-memory copy.

REFERENCE

COMMAND LINE SWITCHES

−−background−expr perl-expression
: Specifies the Perl expression to evaluate.

−−background−border
: By default, the expression creates an image that fills the full window, overwriting borders and any other areas, such as the scrollbar.

  Specifying this flag changes the behaviour, so that the image only replaces the background of the character area.

−−background−interval seconds
: Since some operations in the underlying XRender extension can effectively freeze your X−server for prolonged time, this extension enforces a minimum time between updates, which is normally about 0.1 seconds.

  If you want to do updates more often, you can decrease this safety interval with this switch.

PROVIDERS/GENERATORS

These functions provide an image, by loading it from disk, grabbing it from the root screen or by simply generating it. They are used as starting points to get an image you can play with.

load `$path`
: Loads the image at the given `$path`. The image is set to plane tiling mode.

  If the image is already in memory (e.g. because another terminal instance uses it), then the in-memory copy is returned instead.

load_uc `$path`
: Load uncached − same as load, but does not cache the image, which means it is *always* loaded from the filesystem again, even if another copy of it is in memory at the time.

root
: Returns the root window pixmap, that is, hopefully, the background image of your screen.

  This function makes your expression root sensitive, that means it will be reevaluated when the bg image changes.

solid `$colour`
solid `$width`, `$height`, `$colour`
: Creates a new image and completely fills it with the given colour. The image is set to tiling mode.

  If `$width` and `$height` are omitted, it creates a 1x1 image, which is useful for solid backgrounds or for use in filtering effects.

clone `$img`
: Returns an exact copy of the image. This is useful if you want to have multiple copies of the same image to apply different effects to.

merge `$img` ...
: Takes any number of images and merges them together, creating a single image containing them all. The tiling mode of the first image is used as the tiling mode of the resulting image.

  This function is called automatically when an expression returns multiple images.

TILING MODES

The following operators modify the tiling mode of an image, that is, the way that pixels outside the image area are painted when the image is used.

tile `$img`
: Tiles the whole plane with the image and returns this new image − or in other words, it returns a copy of the image in plane tiling mode.

  Example: load an image and tile it over the background, without resizing. The `tile` call is

superfluous because `load` already defaults to tiling mode.

```
tile load "mybg.png"
```

mirror `$img`
: Similar to tile, but reflects the image each time it uses a new copy, so that top edges always touch top edges, right edges always touch right edges and so on (with normal tiling, left edges always touch right edges and top always touch bottom edges).

  Example: load an image and mirror it over the background, avoiding sharp edges at the image borders at the expense of mirroring the image itself

```
mirror load "mybg.png"
```

pad `$img`
: Takes an image and modifies it so that all pixels outside the image area become transparent. This mode is most useful when you want to place an image over another image or the background colour while leaving all background pixels outside the image unchanged.

  Example: load an image and display it in the upper left corner. The rest of the space is left ''empty'' (transparent or whatever your compositor does in alpha mode, else background colour).

```
pad load "mybg.png"
```

extend `$img`
: Extends the image over the whole plane, using the closest pixel in the area outside the image. This mode is mostly useful when you use more complex filtering operations and want the pixels outside the image to have the same values as the pixels near the edge.

  Example: just for curiosity, how does this pixel extension stuff work?

```
extend move 50, 50, load "mybg.png"
```

VARIABLE VALUES

The following functions provide variable data such as the terminal window dimensions. They are not (Perl−) variables, they just return stuff that varies. Most of them make your expression sensitive to some events, for example using `TW` (terminal width) means your expression is evaluated again when the terminal is resized.

TX
TY Return the X and Y coordinates of the terminal window (the terminal window is the full window by default, and the character area only when in border-respect mode).

  Using these functions makes your expression sensitive to window moves.

  These functions are mainly useful to align images to the root window.

  Example: load an image and align it so it looks as if anchored to the background (that's exactly what `rootalign` does btw.):

```
move -TX, -TY, keep { load "mybg.png" }
```

TW
TH Return the width (`TW`) and height (`TH`) of the terminal window (the terminal window is the full window by default, and the character area only when in border-respect mode).

  Using these functions makes your expression sensitive to window resizes.

  These functions are mainly useful to scale images, or to clip images to the window size to conserve memory.

  Example: take the screen background, clip it to the window size, blur it a bit, align it to the window position and use it as background.

```
clip move -TX, -TY, keep { blur 5, root }
```

FOCUS
: Returns a boolean indicating whether the terminal window has keyboard focus, in which case it returns true.

  Using this function makes your expression sensitive to focus changes.

  A common use case is to fade the background image when the terminal loses focus, often together with the `-fade` command line option. In fact, there is a special function for just that use case: `focus_fade`.

  Example: use two entirely different background images, depending on whether the window has focus.

```
FOCUS ? keep { load "has_focus.jpg" } : keep { load "no_focus.jpg" }
```

now
: Returns the current time as (fractional) seconds since the epoch.

  Using this expression does *not* make your expression sensitive to time, but the next two functions do.

again `$seconds`
: When this function is used the expression will be reevaluated again in `$seconds` seconds.

  Example: load some image and rotate it according to the time of day (as if it were the hour pointer of a clock). Update this image every minute.

```
again 60;
rotate 50, 50, (now % 86400) * -72 / 8640, scale keep { load "myclock.png"
```

counter `$seconds`
: Like `again`, but also returns an increasing counter value, starting at 0, which might be useful for some simple animation effects.

SHAPE CHANGING OPERATORS

The following operators modify the shape, size or position of the image.

clip `$img`
clip `$width`, `$height`, `$img`
clip `$x`, `$y`, `$width`, `$height`, `$img`
: Clips an image to the given rectangle. If the rectangle is outside the image area (e.g. when `$x` or `$y` are negative) or the rectangle is larger than the image, then the tiling mode defines how the extra pixels will be filled.

  If `$x` and `$y` are missing, then `0` is assumed for both.

  If `$width` and `$height` are missing, then the window size will be assumed.

  Example: load an image, blur it, and clip it to the window size to save memory.

```
clip keep { blur 10, load "mybg.png" }
```

scale `$img`
scale `$size_factor`, `$img`
scale `$width_factor`, `$height_factor`, `$img`
: Scales the image by the given factors in horizontal (`$width`) and vertical (`$height`) direction.

  If only one factor is given, it is used for both directions.

  If no factors are given, scales the image to the window size without keeping aspect.

resize `$width`, `$height`, `$img`
: Resizes the image to exactly `$width` times `$height` pixels.

fit `$img`
fit `$width`, `$height`, `$img`
: Fits the image into the given `$width` and `$height` without changing aspect, or the terminal size. That means it will be shrunk or grown until the whole image fits into the given area, possibly leaving

borders.

cover `$img`
cover `$width`, `$height`, `$img`
: Similar to `fit`, but shrinks or grows until all of the area is covered by the image, so instead of potentially leaving borders, it will cut off image data that doesn't fit.

move `$dx`, `$dy`, `$img`
: Moves the image by `$dx` pixels in the horizontal, and `$dy` pixels in the vertical.

  Example: move the image right by 20 pixels and down by 30.

```
move 20, 30, ...
```

align `$xalign`, `$yalign`, `$img`
: Aligns the image according to a factor – `0` means the image is moved to the left or top edge (for `$xalign` or `$yalign`), `0.5` means it is exactly centered and `1` means it touches the right or bottom edge.

  Example: remove any visible border around an image, center it vertically but move it to the right hand side.

```
align 1, 0.5, pad $img
```

center `$img`
center `$width`, `$height`, `$img`
: Centers the image, i.e. the center of the image is moved to the center of the terminal window (or the box specified by `$width` and `$height` if given).

  Example: load an image and center it.

```
center keep { pad load "mybg.png" }
```

rootalign `$img`
: Moves the image so that it appears glued to the screen as opposed to the window. This gives the illusion of a larger area behind the window. It is exactly equivalent to `move -TX, -TY`, that is, it moves the image to the top left of the screen.

  Example: load a background image, put it in mirror mode and root align it.

```
rootalign keep { mirror load "mybg.png" }
```

  Example: take the screen background and align it, giving the illusion of transparency as long as the window isn't in front of other windows.

```
rootalign root
```

rotate `$center_x`, `$center_y`, `$degrees`, `$img`
: Rotates the image clockwise by `$degrees` degrees, around the point at `$center_x` and `$center_y` (specified as factor of image width/height).

  Example: rotate the image by 90 degrees around its center.

```
rotate 0.5, 0.5, 90, keep { load "$HOME/mybg.png" }
```

COLOUR MODIFICATIONS

The following operators change the pixels of the image.

tint `$color`, `$img`
: Tints the image in the given colour.

  Example: tint the image red.

```
tint "red", load "rgb.png"
```

  Example: the same, but specify the colour by component.

```
tint [1, 0, 0], load "rgb.png"
```

shade `$factor`, `$img`
: Shade the image by the given factor.

contrast `$factor`, `$img`
contrast `$r`, `$g`, `$b`, `$img`
contrast `$r`, `$g`, `$b`, `$a`, `$img`
: Adjusts the *contrast* of an image.

  The first form applies a single `$factor` to red, green and blue, the second form applies separate factors to each colour channel, and the last form includes the alpha channel.

  Values from 0 to 1 lower the contrast, values higher than 1 increase the contrast.

  Due to limitations in the underlying XRender extension, lowering contrast also reduces brightness, while increasing contrast currently also increases brightness.

brightness `$bias`, `$img`
brightness `$r`, `$g`, `$b`, `$img`
brightness `$r`, `$g`, `$b`, `$a`, `$img`
: Adjusts the brightness of an image.

  The first form applies a single `$bias` to red, green and blue, the second form applies separate biases to each colour channel, and the last form includes the alpha channel.

  Values less than 0 reduce brightness, while values larger than 0 increase it. Useful range is from −1 to 1 − the former results in a black, the latter in a white picture.

  Due to idiosyncrasies in the underlying XRender extension, biases less than zero can be *very* slow.

  You can also try the experimental(!) `muladd` operator.

muladd `$mul`, `$add`, `$img` # EXPERIMENTAL
: First multiplies the pixels by `$mul`, then adds `$add`. This can be used to implement brightness and contrast at the same time, with a wider value range than contrast and brightness operators.

  Due to numerous bugs in XRender implementations, it can also introduce a number of visual artifacts.

  Example: increase contrast by a factor of `$c` without changing image brightness too much.

  ```
  muladd $c, (1 - $c) * 0.5, $img
  ```

blur `$radius`, `$img`
blur `$radius_horz`, `$radius_vert`, `$img`
: Gaussian-blurs the image with (roughly) `$radius` pixel radius. The radii can also be specified separately.

  Blurring is often *very* slow, at least compared or other operators. Larger blur radii are slower than smaller ones, too, so if you don't want to freeze your screen for long times, start experimenting with low values for radius (<5).

focus_fade `$img`
focus_fade `$factor`, `$img`
focus_fade `$factor`, `$color`, `$img`
: Fades the image by the given factor (and colour) when focus is lost (the same as the `-fade`/`-fadecolor` command line options, which also supply the default values for `factor` and `$color`. Unlike with `-fade`, the `$factor` is a real value, not a percentage value (that is, 0..1, not 0..100).

  Example: do the right thing when focus fading is requested.

  ```
  focus_fade load "mybg.jpg";
  ```

OTHER STUFF

Anything that didn't fit any of the other categories, even after applying force and closing our eyes.

keep { ... }

This operator takes a code block as argument, that is, one or more statements enclosed by braces.

The trick is that this code block is only evaluated when the outcome changes – on other calls the `keep` simply returns the image it computed previously (yes, it should only be used with images). Or in other words, `keep` *caches* the result of the code block so it doesn't need to be computed again.

This can be extremely useful to avoid redoing slow operations – for example, if your background expression takes the root background, blurs it and then root-aligns it it would have to blur the root background on every window move or resize.

Another example is `load`, which can be quite slow.

In fact, urxvt itself encloses the whole expression in some kind of `keep` block so it only is reevaluated as required.

Putting the blur into a `keep` block will make sure the blur is only done once, while the `rootalign` is still done each time the window moves.

```
rootalign keep { blur 10, root }
```

This leaves the question of how to force reevaluation of the block, in case the root background changes: If expression inside the block is sensitive to some event (root background changes, window geometry changes), then it will be reevaluated automatically as needed.