CLX — Common Lisp X Interface

July 9, 2007

Contents

1 Render Extension **5**

1.1 Overview 5
1.2 Pictures 5
1.2.1 Creating Pictures 5
1.2.2 Picture Attributes 6
1.2.3 Predefined Pictures 8
1.3 Picture Formats 8
1.3.1 Querying Picture Formats 8
1.3.2 Picture Format Attributes 9
1.4 Composition 10
1.4.1 Operators 10
1.4.2 Binary Composition 11
1.4.3 Unary Composition 12
1.4.4 Geometric Composition 12
1.5 Filters 13
1.6 Glyph Sets 14

1 Render Extension

1.1 Overview

The Render extension is an alternative X11 extension for rendering graphics. The main idea of the Render extension is to base rendering on the *composition* operation. This operation combines a *source* picture with the *destination* picture to yield a new *destination* picture; possibly under a certain mask. The operation itself is one of the Porter-Duff operations.

A *picture* is a two-dimensional array of pixels, each of which consists out of red, green, blue and alpha components. In the context of the X protocol, the pixel store of such a picture is either a window or a pixmap.

A *picture format* defines a mapping from pixel values found in the pixel store of a *picture* to the red, green, blue and alpha components. *Picture formats* come in two flavors: *direct picture formats* and *indexed picture formats*.

Composition then means to combine pixels from the destination picture with pixels from the source picture with a given Porter-Duff operator. Composition is further subject to a mask, which is another picture. This mask is either explicitly given as in `render-composite` or implicitly specified as certain shape as in `render-triangle`, `render-trapezoid` and similar functions.

1.2 Pictures

The source and destination operants of the composition operation are *pictures*. The two most important properties of a *pictures* are: The *drawable* it refers to and specifies where pixel data is coming from, and the *picture format* that specifies how pixel data from the *drawable* is decoded into component values.

Additionally a picture behaves somewhat like a graphics context in that it also features a clipping region.

1.2.1 Creating Pictures

`render-create-picture` *drawable* *Function*
`&key` *format*
(repeat `:off`*)*
(alpha-x-origin `0`*) (alpha-y-origin* `0`*)*
(clip-x-origin `0`*) (clip-y-origin* `0`*)*
(clip-mask `:none`*)*
(subwindow-mode `:clip-by-children`*)*

Creates a new picture object. *drawable* specifies the drawable for pixel store. *format* specifies the *picture format* of the pixels. Unless *drawable* is a window, the *format* argument is required.

For a description of the other init arguments, please see the appropriate accessor functions described below.

`render-free-picture` *picture* *Function*

Frees the picture *picture*.

1.2.2 Picture Attributes

`picture-drawable` *picture* *Reader*

Returns the drawable associated with *picture*.

`picture-format` *picture* *Reader*

Returns the *picture format* of *picture*.

`picture-repeat` *picture* *Accessor*

Gets (and with `setf` sets) the repeat property of the picture *picture*. If `:off` the extent of the picture is confined to the size of the `picture-drawable`. If `:on` the whole two-dimensional plane is tiled with pixel data from `picture-drawable`.

`picture-alpha-map` *picture* *Accessor*
`picture-alpha-x-orign` *picture* *Accessor*
`picture-alpha-y-orign` *picture* *Accessor*

Gets (and with `setf` sets) the alpha map attribute of the picture *picture*.

When the alpha map is `NIL` or `:none` the alpha channel of *picture* comes from *picture* itself, or is assumed to fully opaque, if *picture* does not have one.

When the alpha map is another picture, alpha bits come from the alpha channel of that other picture.

Optionally `picture-alpha-x-orign` and `picture-alpha-y-orign` might specify an integer coordinate offset of the alpha map. These offsets are added to coordinate before accessing the alpha map.

`picture-clip-mask` *picture* *Accessor*
`picture-clip-x-orign` *picture* *Accessor*
`picture-clip-y-orign` *picture* *Accessor*

Gets (and with `setf` sets) the clip mask attribute of *picture*. The clip mask can be one of:

`:none`
: No clip mask is defined.

a coordinate sequence
: The coordinate sequence specifies a rectangle list.

a pixmap
: The clip mask is taken from the *pixmap, which must be a bitmap.*

Optionally `picture-clip-x-orign` and `picture-clip-y-orign` might specify an integer coordinate offset for the clip mask.

`picture-graphics-exposures` *picture* *Accessor*

`picture-subwindow-mode` *picture* *Accessor*

`picture-poly-edge` *picture* *Accessor*

`picture-poly-mode` *picture* *Accessor*

`picture-dither` *picture* *Accessor*

`picture-component-alpha` *picture* *Accessor*

`picture-transformation` *picture* *Accessor*

Gets (and with `setf` sets) the picture transformation. The transformation is represented by a 3×3 two-dimensional array of real numbers.

When a picture is used as a source picture in a composition operation, coordinates are transformed by this transformation prior accessing the pixel store. The effect is, that to transform the picture *picture* by a transformation T, the `picture-transformation` must be set to the inverse T^{-1}.

The pixels might be subject to filtering (see `picture-filter`) to smooth the result in case of scaling or rotation.

The default is the identity matrix `#2A((1 0 0) (0 1 0) (0 0 1))`.

Example

```
(setf (picture-transformation p) #2A((1/2 0 0) (0 1/2 0) (0 0 1)))
```

makes the picture *p*, when used as source source appear twice as big as usually.

`picture-id` *picture* *Reader*

Returns the XID of the picture *picture*.

`with-picture` (*picture* `&key` *repeat*
alpha-map alpha-x-origin alpha-y-origin
clip-mask clip-x-origin clip-y-origin
subwindow-mode
graphics-exposures
filter transformation)
`&body` *body* *Macro*

Within the dynamic extend of *body* bind the given picture attributes to new values.

1.2.3 Predefined Pictures

There are several function to create special predefined pictures.

`render-create-solid-fill` *display color* *Function*

Creates a picture, that is solidly filled with color *color*.

`with-solid-fill` (*picture display r g b* `&optional` (*a* `1.0`)) `&body` *body* *Macro*

Bind *picture* to a temporary picture that is a solid fill of the color given by (r, g, b, a). The picture is valid for the dynamic extent of *body*.

`render-create-linear-gradient` *display* x_1 y_1 x_2 y_2 *spread stops* *Function*

1.3 Picture Formats

A *picture format* describes how pixel values (unsigned integers) are mapped to red, green, blue and alpha components.

Picture formats come in two flavours: *direct* and *indexed* picture formats.

A *direct* picture format maps *bytes* (in the Common Lisp sense) of the pixel value to components. Internally component values are considered to be real numbers between 0 and 1 inclusive. These values are encoded in a certain *byte* of the pixel value. If the *byte* is k bits wide, a byte value x maps to $x/(2^k - 1)$. Should the alpha *byte* have zero bits, the alpha value is assumed to be fully opaque. Should all of the red, green, and blue *byte*s be of zero width, the components are assumed to be zero.

??? What about indexed picture formats?

1.3.1 Querying Picture Formats

Picture formats cannot be created at will. Instead the X server maintains a list of supported picture formats. Use `query-picture-formats`, `find-matching-picture-format`, or `window-picture-format` to find an appropriate picture format.

The server is required to offer at least a *direct* picture format with 8 bits each of red, green, blue, alpha. As well as 8 bits of red, green, and blue and zero bits of alpha. Use the convience functions `display-rgba8-picture-format`, `display-rgb8-picture-format` or `display-a8-picture-format` to query those formats.

`find-matching-picture-formats` *display* *Function*
`&key` *depth-min depth-max depth*
red-min red-max red
green-min green-max green
blue-min blue-max blue
alpha-min alpha-max alpha
type colormap

Returns a list of picture formats on display *display*, that match the specification.

red-min, *green-min*, *green-min*, *alpha-min* specify a minimum number of red, green, blue or alpha bits. The default is *don't care*.

red-max, *green-max*, *green-max*, *alpha-max* specify a maximum number of red, green, blue or alpha bits. The default is *don't care*.

red, *green*, *green*, *alpha* specify a number of red, green, blue or alpha bits, that must be matched exactly. The default is *don't care*.

Additionally *type* specifies a picture format type (`:direct` or `:indexed`). *colormap* specifies a colormap.

Example

To find all picture formats that support eight bits of red, green, blue and alpha, use:

```
(find-matching-picture-formats
   *display* :red 8 :green 8 :blue 8 :alpha 8)
```

If you care for a particular depth, like say 32 bits, and only want a direct picture format:

```
(find-matching-picture-formats
   *display* :red 8 :green 8 :blue 8 :alpha 8
             :depth 32 :type :direct)
```

It is entirely legal to have zero bits of color components. So if you need a eight bit depth alpha map, use:

```
(find-matching-picture-formats
   *display* :red 0 :green 0 :blue 0 :alpha 8
             :depth 8 :type :direct)
```

Finally just saying

```
(find-matching-picture-formats *display*)
```

would return a list of all supported picture formats on the server `*display*`.

`window-picture-format` *window* *Function*

Returns the picture format that corresponds to the colormap of *window*.

`display-rgb8-picture-format` *display* *Function*

`display-rgba8-picture-format` *display* *Function*

`display-a8-picture-format` *display* *Function*

1.3.2 Picture Format Attributes

`picture-format-type` *picture-format* *Function*
→ *type*

Returns the type of the picture format *picture-format*, which is either `:direct` or `:indexed`.

`picture-format-id` *picture-format* *Function*

Returns the XID of the picture format *picture-format*.

`picture-format-red-byte` *picture-format* *Function*
`picture-format-green-byte` *picture-format* *Function*
`picture-format-blue-byte` *picture-format* *Function*
`picture-format-alpha-byte` *picture-format* *Function*

Returns the *byte specifier* for the red, green, blue or alpha component of pixel values according to the picture format *picture-format*.

`picture-format-depth` *picture-format* *Function*

`picture-format-display` *picture-format* *Function*

`picture-format-colormap` *picture-colormap* *Function*

`find-window-picture-format` *window* *Function*

Find the picture format which matches the given window.

1.4 Composition

1.4.1 Operators

For each pixel, the four channels of the image are computed with:

$$C = C_a F_a + C_b F_b$$

where C, C_a, C_b are the values of the respective channels and F_a and F_b come from the following table:

Operator	F_a	F_b
`:clear`	0	0
`:src`	1	0
`:dst`	0	1
`:over`	1	$1 - A_a$
`:over-reverse`	$1 - A_b$	1
`:in`	A_b	0
`:in-reverse`	0	A_a
`:out`	$1 - A_b$	0
`:out-reverse`	0	$1 - A_a$
`:atop`	A_b	$1 - A_a$
`:atop-reverse`	$1 - A_b$	A_a
`:xor`	$1 - A_b$	$1 - A_a$
`:add`	1	1
`:saturate`	$\min(1, (1 - A_b)/A_a)$	1
`:disjoint-clear`	0	0
`:disjoint-src`	1	0
`:disjoint-dst`	0	1
`:disjoint-over`	1	min(1,(1-Aa)/Ab)
`:disjoint-over-reverse`	min(1,(1-Ab)/Aa)	1
`:disjoint-in`	max(1-(1-Ab)/Aa,0)	0
`:disjoint-in-reverse`	0	max(1-(1-Aa)/Ab,0)
`:disjoint-out`	min(1,(1-Ab)/Aa)	0
`:disjoint-out-reverse`	0	min(1,(1-Aa)/Ab)
`:disjoint-atop`	max(1-(1-Ab)/Aa,0)	min(1,(1-Aa)/Ab)
`:disjoint-atop-reverse`	min(1,(1-Ab)/Aa)	max(1-(1-Aa)/Ab,0)
`:disjoint-xor`	min(1,(1-Ab)/Aa)	min(1,(1-Aa)/Ab)
`:conjoint-clear`	0	0
`:conjoint-src`	1	0
`:conjoint-dst`	0	1
`:conjoint-over`	1	max(1-Aa/Ab,0)
`:conjoint-over-reverse`	max(1-Ab/Aa,0)	1
`:conjoint-in`	min(1,Ab/Aa)	0
`:conjoint-in-reverse`	0	min(Aa/Ab,1)
`:conjoint-out`	max(1-Ab/Aa,0)	0
`:conjoint-out-reverse`	0	max(1-Aa/Ab,0)
`:conjoint-atop`	min(1,Ab/Aa)	max(1-Aa/Ab,0)
`:conjoint-atop-reverse`	max(1-Ab/Aa,0)	min(1,Aa/Ab)
`:conjoint-xor`	max(1-Ab/Aa,0)	max(1-Aa/Ab,0)

1.4.2 Binary Composition

`render-composite` *op source mask dest* *Function*
src-x src-y mask-x mask-y dst-x dst-y
width height

`render-composite` *destination source dest-x dest-y width height* *Function*
`&key` (*op* `:over`) (*mask* `:none`)
src-x src-y
mask-x mask-y

1.4.3 Unary Composition

1.4.4 Geometric Composition

The Render extension only provides for triangles and trapezoids. When a geometric shape is rendered, this shape is rendered to an implicit mask. Thus to render shape S with source picture s upon destination d is equivalent to:

```
m := create temporary picture;
Rasterize S in m;
d := (s in m) OP d;
```

The geometric composition functions have a keyword argument `:mask-format` that may be used to specify the picture format to use for the implicit mask.

Rasterization functions come both in singluar and pluaral. The singular types take spread coordinates for ease of programming, the plural variants take coordinate sequences for efficiency, in case coordinates come from some other triangulization code. For triangles tri-strips and tri-fans are supported.

In the wire protocol coordinates are passed as 16.16 fixed point numbers; this fact is hidden in that the CLX functions take just reals, which are converted as appropriate. Note however that nothing is done to numbers which are out of range.

In case high performance is an absolute must, the plural variants exsit also with a `-fast` suffix that take *coord-sequence* as a specialized array of `(signed-byte 32)` and are configured for low safety, high speed. It is suggested that these are used from specially tailored triangulation code further up in the chain. These functions are not documented here.

`render-triangle` *destination source x1 y1 x2 y2 x3 y3* *Function*
`&key` *src-x src-y* (*op* `:over`) *format*

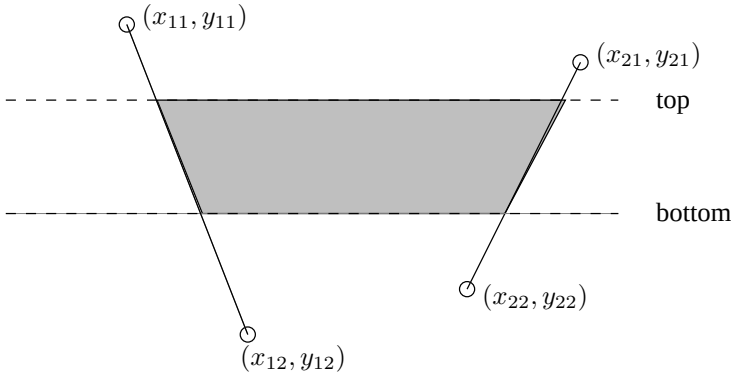
Renders a single triangle on the destination picture *destination* compositing pixels from *source*. The triangle is specified by the three points (x_1, y_1), (x_2, y_2) and (x_3, y_3) all of which can be real numbers.

`render-triangles` *destination source coord-sequence* *Function*
`&key` *src-x src-y* (*op* `:over`) *format*

`render-trapezoid` *destination source* *Function*
top bottom
x11 y11 x12 y12
x21 y21 x22 y22
`&key` *src-x src-y* (*op* `:over`) *format*

Renders a trapezoid. The top and bottom edges are given by *top* and *bottom*. The left and right edges are defined by the lines $(x_{11}, y_{11}) - (x_{12}, y_{12})$ and $(x_{21}, y_{21}) - (x_{22}, y_{22})$. All of these coordinates are real numbers. The y coordinates of the lines need not to be coincident with *top* or *bottom*.

The following diagram illustrates how trapezoids are specified.



`render-trapezoids` *destination source coord-sequence* *Function*
`&key` *src-x src-y* (*op* `:over`) *format*

`render-fill-rectangle` *picture op color x1 y1 w h* *Function*

1.5 Filters

A source *picture* might be subject to filtering prior to composing with the destination *picture*. Especially this is of importance, if the source *picture* has an affine transformation installed. Some filters (like the convolution filters) are also useful, if the transformation is the identity.

`picture-filter` *picture* *Accessor*

Returns (and with `setf` sets) the filter installed on the picture *picture*.

Filters are represented either by a single keyword or a cons of a keyword and a list of the filter arguments.

Currently the following filters are defined:

`:nearest`
: Nearest neighbor filtering

`:bilinear`
: Linear interpolation in two dimensions

Additional names may be provided for any filter as aliases. A set of standard alias names are required to be mapped to a provided filter so that applications can use the alias names without checking for availability.

`:fast`
: High performance, quality similar to `:nearest`.

`:good`
: Reasonable performance, quality similar to `:bilinear`

`:best`
: Highest quality available, performance may not be suitable for interactive use.

There is also a set of standard filters which are not required but may be provided. If they are provided, using the standard name, they must match the definition specified here.

`(:convolution` *m n* $c_{11} \ldots c_{mn}$`)`
: $m \times n$ convolution filter. The values specified in `(setf picture-filter)` are m, n and then mn filter parameters. m and n must be integers.

`(:gaussian` *r*`)`
: Gaussian blur. r is the radius in pixels (which can be fractional). A standard Gaussian 2D convolution filter will be applied.

`render-query-filters` *drawable* *Function*
→ *filters aliases*

Returns the list of filters supported by the given drawable. Each item of the list returned is a cons of the primary name of the filter and a list of aliases names.

Implementation Note: On the wire filter names are just strings. These strings happen to be lower case; so we apply `:invert` case (in the sense of `readtable-case`) and intern those strings in the keyword package.

1.6 Glyph Sets

`render-free-glyph-set` *glyph-set* *Function*

`render-create-glyph-set` *format* *Function*

`render-reference-glyph-set` *existing-glyph-set* *Function*

`render-composite-glyphs` *destination glyph-set source dest-x dest-y sequence* *Function*
`&key` *op*
src-x src-y mask-format
start end

`render-add-glyph` *glyph-set id* `&key` *x-origin y-origin x-advance y-advance data* *Function*

`render-free-glyphs` *glyph-set glyph-ids* *Function*

`render-free-glyphs` *glyph-set glyph-ids* *Function*