BLEND

An Extension to X for Overlays and Blending
Version 1.3

Todd Brunhoff

Tektronix, Inc.
Tektronix Research Laboratories

1. Overview

This extension attempts to express the concept of blending, and the common, degenerate instance of this, called overlays. In the realm of image processing, one might blend two pictures to produce a third or replace one of the original two. BLEND abstracts a hybrid of this idea, and provides the mechanism to combine two discrete pixel sets, which may reside in different frame buffers or in different parts of the same logical or physical pixel, visually mixing them on a pixel-by-pixel basis; however, no new set of pixels is produced and the original images remain unchanged. Common hardware today can display one pixel or the other, but some hardware can blend the color primaries more continuously, such as 70% – 30%, or 50% – 50%.

In short, two InputOutput windows are created which are non-inferiors of each other; these contain the two pixel sets. The blending for any particular pair of drawables is represented in one of two ways, depending on hardware and software. Either by a new drawable type, called a translucent, or by special attributes of certain pixels that allow them to become transparent. Either or both representations may be available in a single server.

Two new visual types are introduced, **BlendStatic** and **BlendDynamic**, which correspond most closely with the core visual types **StaticGray** and **GrayScale**. Their function is to describe the properties of a blendmap, a colormap-like structure, that expresses the mapping of pixels values to a blending value. In the same way that colormap hardware is able to change large portions of a display by changing one or a few colormap entries, there exists hardware that can change the blending of two pictures over a large area by changing a single entry in a so-called blendmap. Common overlay hardware that uses a control plane to designate whether the overlay or underlay pixel is visible is represented as a depth 1 translucent with a **BlendStatic** visual. Other overlay hardware uses "magic pixels" that allow an underlay pixel to be seen; this is represented as a depth *n* overlay window, with two visuals, a **BlendStatic** visual for the blending properties and a core visual type for color properties.

2. Terminology

Blendmap

A blendmap is identical in structure to the core protocol's colormap type, and in fact, can be manipulated in the same way by most of the colormap-related requests. Its visual type is either **BlendStatic** or **BlendDynamic**. In a blend triple, a blendmap is used with a translucent to determine the blended value of its underlay and overlay. Each pixel in the translucent indexes a value in the blendmap which is then used to calculate the blended value of the underlay and overlay pixels. In a blend pair, a blendmap is used with the overlay to determine the blended value of its pixel and that of the overlay. Each pixel in the overlay still indexes an entry in the window's colormap, but it also indexes a value in a blendmap which is then used to calculate the blended value of the underlay and overlay pixels.

Blend pair

A blend pair consists of two InputOutput windows, an underlay and overlay window; the latter has transparent pixels, defined by a blendmap.

Blend set

A blend set is a generic term used to refer to a blend pair or triple. The term "blend set" has been substituted for the phrase "blend pair or blend triple" in this document.

Blend triple

A blend triple consists of two InputOutput windows, an underlay and overlay window, and a third drawable called a translucent.

Overlay

An overlay is defined as a window having an associated translucent (a blend triple), or having transparent pixels (a blend pair). The name overlay is independent of the window stacking order, hence, an overlay may actually be beneath an underlay in stacking order.

Potential blend region (PBR)

See underlay, overlay and translucent. A potential blend region is those pixel pairs in an underlay and an overlay that may be blended, whether they are a blend pair or a blend triple. If the overlay is above the underlay in the window stacking order, it is the region of visible pixels in a overlay, excluding the border, matched with coincident pixels in the underlay, excluding the border. If the underlay is above the overlay, then it is the region of visible pixels in a underlay, excluding the border, matched with coincident pixels in the overlay, excluding the border. Because the PBR is defined by a subset of the visible pixels in the upper-most window in a blend set, it is immaterial whether other windows occlude the bottom-most window.

Translucent

The mixture of the windows in a blend triple is defined by a new drawable called a translucent and a blendmap. The size and location of the translucent is bound to the overlay window, changing as the overlay changes. Each pixel in the translucent corresponds with a pixel in the overlay window that has the same x and y coordinate; these pixels correspond with a pixel in the underlay window having the same absolute screen coordinate. The translucent pixel value is used to index a value in a blendmap which in turn is used to calculate the visible pixel.

Transparent pixel

In a blend pair, the mixture of windows is defined by pixels in the overlay window, and a blendmap. Each pixel in the overlay corresponds with one pixel in the underlay having the same absolute screen coordinate. The overlay pixel is used to index a value in a blendmap which in turn is used to calculate the visible pixel.

Underlay

An underlay is defined as a window having an associated overlay window. The name underlay is independent of the window stacking order, hence, an underlay may actually be above an overlay in stacking order.

3. Visuals

BLEND introduces two new visual types: **BlendStatic** and **BlendDynamic**. For **BlendDynamic**, a pixel value indexes a blendmap to produce independent blend values; the blend values can be changed dynamically. **BlendStatic** is treated in the same way except the blendmap has predefined read-only blend values, which are server dependent.

A colormap and a blendmap are, for the most part, the same object in X. The **CreateColormap** request creates both colormaps and blendmaps, depending on the visual type; **CreateTranslucent** will accepts a blendmap although the protocol type is really a colormap; other core requests accept a blendmap in place of a colormap as appropriate. Unlike **StaticGray** and **GrayScale**, which primary drives the blending hardware is defined as the red component; thus, the client should always store the desired value in the red component.

Although a translucent has a visual type, the pixels in the translucent are not displayed, but instead affect the display of the underlay and overlay pixels. For a blend triple, each pixel in the translucent defines the mixture of corresponding pixels in the underlay and overlay on a one-to-one basis. For a blend pair, each transparent pixel in the overlay defines the mixture of the transparent pixel with its corresponding pixel in the underlay.

The blending equally affects the visible red, green and blue primaries for a pixel. The state of each pixel's r, g and b in the PBR are calculated as follows:

$$dvalue = \alpha \cdot uvalue + (1-\alpha) \cdot ovalue$$

where

$$\alpha = \frac{bmap\,[\,bpixel\,]}{65535}$$

and

dvalue is the displayed (visible) primary
uvalue is the primary from the underlay
ovalue is the primary from the overlay
bpixel is the pixel in the translucent (blend triple) or overlay (blend pair)
bmap is the array of red protocol-intensities in the blendmap.

Simply put, if the pixel indexes a blendmap entry that has all ones in the red component, then the displayed pixel will be the underlay's pixel. If the pixels indexes a zero, then the overlay's pixel is displayed.

When a window becomes an underlay, and a pixel in the PBR changes from zero underlay to non-zero underlay, and the underlay pixel has not been retained by the hardware or the server, then the underlay pixel is treated as if a **ClearArea** had been performed on the pixel. Similarly, when a window becomes an overlay, and a pixel in the PBR changes from zero overlay to non-zero overlay, and the overlay pixel has not been retained by the hardware or the server, then the overlay pixel is treated as if a **ClearArea** had been performed on the pixel. If possible, expose events for all exposed pixels in each window will be coalesced.

If there are multiple blend sets viewable on a screen such that an overlay in one set is an underlay in another, the visible pixels on the screen for the areas common to both blend sets will be composed from the topmost set.

Because of the semantics of blend visuals, a PBR of the bottom-most window in a blend set will always behave as if backing-store were set to **WhenMapped** or **Always**, at the server-implementor's discretion. Thus, when graphic operations change the PBR in the occluded member of a blend set, the changes take place even though the window is occluded. However, this does not guarantee that the pixels in that window will always be retained.

4. Types

BLENDTRIPLE: [model1, model2, blend: CARD16
capability: BITMASK
{**Model1Overlay**, **Model2Overlay**, **Composite**}]

BLENDPAIR: [model1, model2: CARD16
capability: BITMASK
{**Model1Overlay**, **Model2Overlay**, **Composite**}
visual-id: VISUALID]

BLENDTRIPLE and BLENDPAIR are used by the **QueryBlend** request to describe the acceptable members of a blend set. The *model1* and *model2* elements are indices into the list of allowed-depths returned by **QueryBlend** and indicate the depth and visual that may be used to create an underlay or overlay. The *blend* element in a blend triple is also an index into the list of allowed-depths and indicates the depth and visual of the translucent. For the blend pair, the visual-id describes the blend visual type of the overlay window and blendmap (Note that the overlay window also has a core visual type for its color properties).

If **Model1Overlay** is asserted, then a window created with a depth and visual indicated by the *model1* element may become an overlay for the blend set. Likewise, if **Model2Overlay** is asserted, then a window created with a depth and visual indicated by the *model2* element may become an overlay for the blend set. A blend triple only has one overlay; i.e. of both **Model1Overlay** and **Model2Overlay** are asserted, the structure describes two potential blend triples. If **Composite** is asserted, then pixels in the underlay and overlay occupy the same frame buffer. In this case, pixels within a PBR copied from an overlay window that are not 100% overlay are undefined. Likewise, pixels within a PBR copied from an underlay that are not 100% underlay are undefined. If **Composite** is not asserted, then underlay and overlay pixels occupy separate frame buffers. In this case, pixels copied from the PBR of an underlay or overlay are guaranteed to be defined.

RENDERMODEL: [depth: CARD8
visual-id: VISUALID]

This type is used by **QueryBlend** to describe the depth and visual type of a potential underlay, overlay or translucent. The visuals are described by the connection setup and the **QueryBlend** request.

TRANSLUCENT: 32-bit value (top three bits guaranteed to be zero)

This type is used to provide a handle on the translucent resource in a blend triple. If a request requires this type, but the specified id does not name a defined TRANSLUCENT, then a **Pixmap** error is generated.

TRANSLUCENT-EVENT: {**Exposure**, **ColormapChange**}

A translucent may need expose events because its associated overlay window may receive exposure events for areas the translucent may not need to update. **ColormapNotify** events inform a client of the state of the blendmap associated with the translucent.

5. Requests

QueryBlend

wid: WINDOW
==>
blend-depths: LISTofDEPTHS
allowed-depths: LISTofRENDERMODEL
blend-triples: LISTofBLENDTRIPLE
blend-pair: LISTofBLENDPAIR
flags: BITMASK {**HardwareTriple**, **HardwarePair**}
default-blendmap: COLORMAP
min-installed-maps, *max-installed-maps*: CARD16
backing-stores: {**Never**, **WhenMapped**, **Always**}

Errors: **Window**

This request describes the blending capabilities supported by the server.

The blend-depths specify what depths/visuals are unique for blending. These depth structures normally list only the visual types **BlendStatic** and **BlendDynamic**, but may also list core visual types or visual types from other extensions that may describe an overlay or underlay. Pixmaps are supported for each depth listed. The presence of a BLEND extension in a server may cause the LISTofFORMAT provided by the connection setup to be extended. The additional formats will be those that would allow **GetImage** and **PutImage** access to windows, pixmaps or translucents created with depths not published in the core protocol connection setup. The definition of DEPTH can be found in the core protocol's description of the connection information.

The allowed-depths specify the depths/visuals that are supported for use in blend sets, as seen in the description of BLENDTRIPLES and BLENDPAIRS above. These depths and visual ids are selected from those listed in in blend-depths, those provided by the X connection setup, and possibly those provided by other extensions. The list is in no particular order.

Each item in the list of blend triples describe the possible combinations of depth and visual for each member of a blend triple. Each item in the list of blend pairs describe possible depths and visuals for each member blend pairs. Either list of blend sets may be null, but not both.

If asserted, the **HardwarePair** flag indicates that blend pairs are supported in hardware. If it is not asserted, then it implies that any published blend pairs are implemented in software, and that graphic operations on the overlay may be very slow in order to properly emulate blend pair semantics.

If **HardwareTriple** is asserted, it indicates that blend triples are supported in hardware. If it is not asserted, then it implies that any published blend triples are implemented in software, and that graphic operations on the translucent or the overlay may be very slow in order to properly emulate blend triple semantics.

The default-blendmap is available for clients with minimal blending requirements. Its depth and visual type is that of the first entry in blend-depths. Clients that create translucents with this same depth and visual may want to allocate from this map by default.

Min-installed-maps specifies the number of blendmaps that can be guaranteed to be installed simultaneously (with **InstallColormap**), regardless of the number of entries allocated in each map. Max-installed-maps specifies the maximum number of blendmaps that might possibly be installed simultaneously, depending on their allocations. Multiple **BlendStatic** blendmaps with identical contents but differing in resource ID should be considered as a single blendmap for the purposes of this number. For the typical case of a single hardware blendmap, both values will be 1.

Backing-stores indicates when the server supports backing stores for this screen on translucents, although it may be storage limited in the number of translucents (and windows) it can support at once.

CreateTranslucent

translucent: TRANSLUCENT
drawable: DRAWABLE
depth: CARD8
visual: VISUALID
value-mask: BITMASK
value-list: LISTofVALUE

Errors: **IDChoice**, **Drawable**, **Match**, **Alloc**

This request creates a translucent and assigns the identifier *translucent* to it. The depth and visual must be one of the those supported for blend triples by the root of the specified drawable, else Value error. The height and width of the translucent are undefined until it becomes a member of a blend triple; the contents of the translucent are undefined until its overlay is mapped and perhaps not until the overlay becomes visible, depending on the value of backing-store. The value-mask and value-list specify attributes of the translucent that are to be initialized. The possible values are:

Attribute	Type
background-pixmap	PIXMAP
background-pixel	CARD32
bit-gravity	BITGRAVITY or **CopyFromOverlay**
backing-store	{**NotUseful**, **WhenMapped**, **Always**}
backing-planes	CARD32
backing-pixel	CARD32
exposures	BOOL
blendmap	COLORMAP or **DefaultBlendmap**

The default values when attributes are not explicitly defined are:

Attribute	Type
background-pixel	zero
bit-gravity	**CopyFromOverlay**
backing-store	**NotUseful**
backing-planes	all ones
backing-pixel	zero
event-mask	SETofTRANSLUCENT-EVENT
blendmap	**DefaultBlendmap**

If background-pixmap is given, it overrides the default background-pixmap. The background pixmap and the translucent must have the same root and the same depth, else **Match** error. Any size pixmap can be used, although some sizes may be faster than others. If background-pixel is given, it overrides the default background-pixmap and any background-pixmap given explicitly, and a pixmap of undefined size filled with background-pixel is used for the background. Range checking is not performed on the background-pixel value; it is simply truncated to the appropriate number of bits. The background tile origin is always the translucent origin.

When no valid contents are available for a translucent and PBRs are either visible or the server is maintaining backing store, the server automatically tiles the regions with the translucent's background. **Exposure** events are then generated for the regions.

The bit-gravity defines which region of the translucent should be retained if the associated overlay is resized.

A backing-store of **WhenMapped** advises the server that maintaining contents of obscured regions when the translucent's blend triple is mapped would be beneficial. A backing-store of **Always** advises the server that maintaining contents even when the blend triple is unmapped would be beneficial. In this case, the server may generate an expose event when the translucent becomes a member of a blend triple. A value of **NotUseful** advises the server that maintaining contents is unnecessary, although a server may still choose to maintain contents while the blend triple is mapped. While the server maintains contents, expose events will not normally be generated, but the server may stop maintaining contents at any time.

The backing-planes indicates (with bits set to 1) which bit planes of the translucent hold dynamic data that must be preserved in backing-stores and during save-unders. The backing-pixel specifies what value to use in planes not covered by backing-planes. The server is free to save only the specified bit planes in the backing-store or save-under and regenerate the remaining planes with the specified pixel value. Any bits beyond the specified depth of the translucent in these values are simply ignored.

The *blendmap* specifies the blendmap that reflects the desired blending of the translucent. Servers capable of supporting multiple hardware blendmaps may use this information. The blendmap must have the same visual type as the translucent, else **Match** error. If **DefaultBlendmap** is specified, the default blendmap for the screen is used. However, the translucent must have the same depth and visual type as the default blendmap, else **Match** error.

Translucents may be used in any request in which a drawable is appropriate. For **GetGeometry**, the the depth is the number of bits per pixel for the translucent. The x, y, and borderwidth will always be zero. If the translucent is a member of a blend triple, the width and height will be that of the overlay window, otherwise both will have the value of one.

ClearArea, although it accepts only a window, may be used on a translucent. The specified rectangle is tiled with a plane-mask of all ones and function of **Copy**.

The event-mask defines which events the client is interested in for this translucent.

ChangeTranslucentAttributes

translucent: TRANSLUCENT
value-mask: BITMASK
value-list: LISTofVALUE

Errors: **Pixmap**, **Match**, **Alloc**

The value-mask and value-list specify which attributes are to be changed. The values and restrictions are the same as for **CreateTranslucent**. Setting a new background, whether by background-pixmap or background-pixel, overrides any previous background. Changing the background does not cause the translucent contents to be changed.

Changing the backing-store of a translucent whose overlay window is obscured to **WhenMapped** or **Always**, or changing the backing-planes or backing-pixel, or save-under of a mapped window may have no immediate effect.

Changing the colormap of a window by defining a new map, generates a **ColormapNotify** event. Changing the blendmap of a translucent with a PBR may have no immediate effect on the screen (see **InstallColormap** request in this document). The order in which attributes are verified and altered is server-dependent. If an error is generated, a subset of the attributes may have been altered.

ChangeBlendGC

gc: GCONTEXT
blendgc: GCONTEXT
value-mask: BITMASK

Errors: **GContext**, **Match**, **Alloc**

For the description of this request, a graphic gc is the gc specified in a core protocol request (a complete list appears in the description of SelectOverlay). A blend gc is a gc associated with a graphic gc by means of this request; the blend gc is used in implied graphic requests to the translucent (also described in SelectOverlay).

This request changes which blend gc is associated with a graphic gc, and the way components in the former are inherited from the latter. Operations on an overlay using *gc* will imply similar operations to its underlay using *blendgc*. This cancels any previous association that the graphic gc has with another blend gc. The id specified by *blendgc* must not itself be a graphic gc with an associated blend gc, else **Match** error.

The value mask indicates which values in the blend gc are inherited from the associated graphic gc. In general, if the bit for a gc component is zero, then the blend gc values are used; if a one bit is asserted for a gc component, then that component in the blend gc is inherited from the graphic gc. Some of the components are transformed when they are inherited. This causes a special validation of the blend gc to occur every time the graphic gc is validated or when a **ChangeBlendGC** is issued. The values are inherited as follows:

Component	Inheritance
function	copied
plane-mask	copied
foreground	copied
background	copied
line-width	copied
line-style	copied
cap-style	copied
join-style	copied
fill-style	if the fill-style is **Tiled**, then it is replaced by **Solid**, otherwise it is copied
fill-rule	copied
arc-mode	copied
tile	never copied
stipple	copied
tile-stipple-x-origin	adjusted so the stipple in the blend gc is aligned with stipple in the graphic gc
tile-stipple-y-origin	adjusted so the stipple in the blend gc is aligned with stipple in the graphic gc
subwindow-mode	the subwindow-mode is always **ClipByChildren**.
graphics-exposures	copied
clip-x-origin	adjusted so the clipped region in the translucent is aligned with the clipped region in the overlay.
clip-y-origin	adjusted so the clipped region in the translucent is aligned with the clipped region in the overlay.
clip-mask	copied
dash-offset	copied
dashes	

Inherited values in a blend gc override explicit changes via **ChangeGC** at all times. Nevertheless, explicit changes to a blend gc are remembered, such that changing the value mask so that a gc component is no longer inherited causes the remembered explicit value to be installed.

AssociateBlendPair

overlay, *underlay*: WINDOW
blendmap: COLORMAP or **DefaultBlendmap**

Errors: **Window**, **Colormap**, **Pixmap**, **Match**

This request creates an association of two windows and forms a blend pair. The visual and depth of the underlay and overlay must match a pair published by **QueryBlend**, else **Match** error. The specified blendmap describes the mapping of overlay pixel values to blending factors. Servers capable of supporting multiple hardware blendmaps may use this information. The blendmap's visual type and the depth and visual type of the underlay and overlay must match a blend pair published by **QueryBlend**, else **Match** error. If **DefaultBlendmap** is specified, the default blendmap for the screen is used. However, the default blendmap must have the same depth as the overlay and the visual type of the matching blend pair, else **Match** error. Both windows and the blendmap must belong to the same screen, else match error.

If the overlay and underlay match a previously specified blend set, the newer pair takes precedence canceling the previous designation. An underlay may only be a member of one blend set at one time, otherwise the conflicting request generates a **Match** error.

AssociateBlendTriple

overlay, underlay: WINDOW
translucent: TRANSLUCENT
implicit: BOOL

Errors: **Window**, **Pixmap**, **Match**, **Alloc**

This request creates an association of two windows and a translucent, forming a blend triple. The visual and depth of the underlay, overlay and translucent must match a triple published by **QueryBlend**, else **Match** error. If *implicit* is true, then any graphic operation to the overlay causes an similar graphic operation to the translucent as described below. If it is false, then no implicit operations will be performed for this triple.

A translucent may be a member of more than one blend triple, but only if the overlay window is the same in all triples. If the overlay and underlay match a previously specified blend set, the newer triple takes precedence canceling the previous designation. An underlay may only be a member of one blend set at one time, otherwise the request that causes the conflict generates a **Match** error.

If a gc is used in a graphic request that specifies an overlay as a destination, and the overlay belongs to a blend triple whose *implicit* is true, a similar request is performed on the translucent of that blend triple using an associated blend gc. If one has not been designated with **ChangeBlendGC**, a default gc is used whose depth is the same as the translucent, and whose elements are inherited from those in the specified gc as described by the **ChangeBlendGC** request, except for five elements:

Component	Value
function	**Copy**
plane-mask	all ones
foreground	1
background	0
tile	None

A **CopyArea** request that specifies source and destination rectangles that fall at least partly in the overlay implies a **CopyArea** request with the src-x, src-y, dst-x and dst-y adjusted so the regions in the overlay align with the same regions in the translucent. If the source rectangle is outside the PBR, or if the source drawable is different than the overlay, a **PolyFillRectangle** request is performed on the translucent with a list of rectangles that correspond to and are aligned with the regions of the overlay that are actually modified by the **CopyArea** request.

A **CopyPlane** or **PutImage** request that specifies a destination rectangle that falls at least partly in the overlay causes a **PolyFillRectangle** request to be performed with a list of rectangles that correspond to and are aligned with the regions of the overlay that are actually modified by the request.

The following requests performed on the overlay, imply the same request on the translucent with all x and y coordinates adjusted to align with the original coordinates in the overlay: **PolyPoint**, **PolyLine**, **PolySegment**, **PolyRectangle**, **PolyArc**, **FillPoly**, **PolyFillRectangle**, **PolyFillArc**, **PolyFillRectangle**, **PolyText8**, **PolyText16**, **ImageText8** and **ImageText16**.

A **ClearArea** does not specify a gc, but if the request is performed on the overlay, the same request is performed on the translucent with the x and y values adjusted to align with the same region in the overlay.

Other graphic extensions may render to a overlay implying equivalent requests on the translucent. However, whether an equivalent request exists or is meaningful for the translucent is left for the foreign extension to specify.

ListInstalledBlendmaps

window: WINDOW
==>
bmaps: LISTofCOLORMAP

Errors: **Window**

This request returns a list of the currently installed blendmaps for the screen of the specified window. The order of blendmaps is not significant, and there is no explicit indication of the required list (see **InstallColormap** request).

CreateColormap

< this is a core protocol request >

This request is used to create blendmaps for the VEX extension. Identical to the core protocol's colormap, blendmaps are created with **CreateColormap** using a visual type appropriate for blending. These visuals are described by the QueryVideo request, and will be selected from the color models **BlendStatic** and **BlendDynamic** protocol colormap requests can operate on the blendmap (FreeColormap, CopyColormapAndFree, **InstallColormap**, **UninstallColormap**, AllocColor, AllocNamedColor, AllocColorCells, AllocColorPlanes, FreeColors, StoreColors, StoreNamedColor, QueryColors and LookupColor). Because blendmaps are a resource separate from colormaps, **ListInstalledBlendmaps** must be used to inspect the installed blendmaps.

InstallColormap

< this is a core protocol request >

If the specified colormap has a blend visual type, then the object is really a blendmap and the request makes this blendmap an installed map for its screen. All PBRs associated with this blendmap immediately display their blended pixels correctly. As a side effect, additional blendmaps might be implicitly installed or uninstalled by the server, but have no effect on installed colormaps. Which other blendmaps get installed or uninstalled is server-dependent except that the required list must remain installed.

If the specified blendmap is not already an installed map, a **ColormapNotify** event is generated on every window having that blendmap as an attribute. In addition, for every other blendmap that is installed or uninstalled as a result of the request, a **ColormapNotify** event is generated on every window having that blendmap as an attribute.

At any time, there is a subset of the installed maps that are viewed as an ordered list and are called the required list. The length of the required list is at most M, where M is the min-installed-maps specified for the screen by the QueryVideo request. The required list is maintained as follows. When a blendmap is an explicit argument to **InstallColormap** , it is added to the head of the list; the list is truncated at the tail, if necessary, to keep the length of the list to at most M. When a blendmap is an explicit argument to **UninstallColormap** and it is in the required list, it is removed from the list. A blendmap is not added to the required list when it is installed implicitly by the server, and the server cannot implicitly uninstall a blendmap that is in the required list.

Initially the default blendmap for a screen is installed (but is not in the required list).

UninstallColormap

< this is a core protocol request >

If the specified colormap has a blend visual type, then the object is really a blendmap and if bmap is on the required list for its screen, it is removed from the list (see **InstallColormap** request in this document). As a side effect, bmap might be uninstalled, and additional blendmaps might be implicitly installed or uninstalled. Which blendmaps get installed or uninstalled is server-dependent except that the required list must remain installed.

If bmap becomes uninstalled, a **ColormapNotify** event is generated on every window having bmap as an attribute. In addition, for every other blendmap that is installed or uninstalled as a result of the request, a **ColormapNotify** event is generated on every window having that blendmap as an attribute.

6. Events

ColormapNotify

<this is a core event>

This event is reported to clients selecting **ColormapChange** on the translucent. It is generated with value True for *new* when the blendmap attribute of the translucent is changed and is generated with value False for *new* when the blendmap of a translucent is installed or uninstalled. In either case, the state indicates whether the blendmap is currently installed.

Expose

<this is a core event>

This event is reported to clients selecting **Exposure** on the translucent. It is generated under the same conditions as described by the core protocol.