# VEX

## Video Extension to X
## Version 5.0

Todd Brunhoff

Tektronix, Inc.
Tektronix Research Laboratories

X Window System is a trademark of M.I.T.

Copyright © 1989 Tektronix, Inc.

Permission to use, copy, modify, and distribute this document for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice are retained, and that the name of Tektronix not be used in advertising or publicity pertaining to this document without specific, written prior permission. Tektronix makes no representations about the suitability of this document or the protocol defined in this document for any purpose. It is provided ‘‘as is’’ without express or implied warranty.

## 1. Overview

The spirit of this extension is to provide an X interface to the generally interesting aspects of displaying live video in windows, capturing graphics from windows and converting them to a video signal, and managing the network of connections to and from devices that may receive or produce these signals, such as video tape recorders. Of those people that have provided input, some disagree on the amount of functionality that should be provided in a server, particularly on control of video devices like VCRs. To satisfy all parties, the protocol is designed so that the server may advertise many or few limitations on its capabilities. For example, a server may advertise that it has an input device, but that it has no control over it and looks like a cable TV channel; alternatively, it can say that it can control playback speed, direction, seeking, etc.

For applications that need access to the digitized video pixels, this extension allows normal pixmaps or windows with core visual types to hold video, which in turn allows read and write access to the pixels via the core protocol or other extensions such as the X Image Extension. Applications that need nothing more than TV-in-a-window, and for hardware that does not actually digitize any pixels, VEX introduces two new visual types: **VideoColor** and **VideoGray**. Their purpose is to express the visual aspects of a visible, but untouchable video picture, hence drawing on a window of this visual type has no effect and the pixels are undefined unless a picture is present.

VEX also embraces the concept of blending; overlay is a common, degenerate instance of this. In short, a sandwich of three windows (parent, child and grandchild) are created, with the center window acting as a blender. Two new visual types are introduced for this window, **BlendStatic** and **BlendDynamic**, which correspond most closely with the core visual types **StaticGray** and **GrayScale**. Their function is to express the mapping of pixels in the blend window to another value before the blending actually takes place. 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 today, some even inexpensive, that can change the blending of two pictures over a large area by changing a single entry in a so-called blendmap. The standard overlay hardware available in several products is represented as a depth 1, **BlendStatic** visual.

This latter concept is very important to many applications that want access to video-in-a-window, hence, it is introduced here. It may be important enough in the future for other applications to use this concept, so the specifications for blending are kept as separated as possible from those that deal with video input.

Today, video input and output hardware has complex limitations and capabilities; the capabilities do not always scale in expected ways, nor do the limitations always make sense. The VEX extension can provide large amounts of information about these limitations and capabilities; and it is intended that the client, through the use of convenience routines in the Xlib implementation, will be able to easily determine what it can and can't do without the overhead of a round-trip request to the server.

## 2. Terminology

**Blend base window (BBW)**

A window is considered a blend base window, or BBW, when it the direct ancestor of a BW; a window may not be both a BBW and a BW.

**Blend window (BW)**

A window is considered a blend window, or BW, when its visual type is **BlendStatic** or **BlendDynamic.** Some graphic hardware is able to multiplex pixels from two frame buffers on a pixel by pixel basis; the simplest of these is the common overlay. Although the actual hardware to accomplish this varies, the overlay control plane can be thought of as third frame buffer, a 1 bit signifying 100% of frame buffer A and a 0 bit signifying 100% of frame buffer B, or vice versa. More sophisticated hardware may (effectively) have two or more planes that describe intermediate mixtures of the pixels, and in this form, it is also known as Alpha Channel.

Each pixel in the blend window corresponds with one pixel in the BBW and one pixel in the BTW. The blend pixel value is used to index a value in a blendmap which in turn is what is used to calculate the visible pixel. See Blendmap.

**Blend top window (BTW)**

A window is considered a blend top window, or BTW, when it the direct descendant of a BW.

**Potential blend region (PBR)**

See BBW, BW and BTW. A potential blend region are those pixels pairs in a BBW and BTW that may be blended using a BW. Specifically, it is the region of visible pixels in a BTW, excluding the border, matched with the pixels directly beneath in the BW parent and the BBW grandparent.

**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.** A blendmap is used with a BW to determine the blended value of its parent and children, the BBW and BTW. Each pixel in the BW indexes a value in the blendmap which is then used to calculate the blended value of the BBW and BTW pixels.

**Blend triple**

A blend triple consists of a BBW, BW and BTW, and must occupy positions in a window hierarchy of parent, child and grandchild, respectively. Multiple blend triples may have one or two windows in common.

**Control atoms**

A VEX server may provide controls for various aspects of video input hardware, video output hardware and video devices. Each type of control has a unique ISO Latin-1 string interned by the server and these are provided by **QueryVideo**. The correspondence of strings and control types are described in a separate document, VEX Devices and Controls (not yet written).

**Designated overlay window (DOW)**

A BBW or a BTW may become a designated overlay window through the use of the **SelectOverlay** request. Core graphic requests that specify the DOW as a destination will implicitly cause an similar request to be performed on the associated BW.

**Full-motion video**

The display of video on the screen at a rate at least as fast as that defined by the encoded video signal. For example NTSC is defined as (roughly) 30 frames per second and 60 fields per second.

**Still video**

A single, static frame acquired from a video signal.

**Video capture**

See video output.

**Video device**

An external device for producing, receiving or routing a video signal, regardless of the video signal format. For example, a videodisc player or VCR. It is identified by a VIDEODEVICE id.

**Video digitization**

A process that converts a video image from an encoded signal into a digital form having a supported depth and format.

**Video input**

Video input in general refers to an incoming signal that is decoded or digitized to produce a picture on the workstation display. In this document, video input refers to the digitizing or decoding hardware, and it is identified by a VIDEOIN id. Logically, a client will connect an output device, such as a video disk player, to a video input using the **ChangeConnectivity** request, and then use **RenderVideo** to direct in which drawable the picture will appear.

**Video input region (VIR)**

This is defined as the subset of pixels in a VW or a pixmap that is actually being updated by a **RenderVideo** request.

**Video output**

Video output in general refers to signals produced by encoding workstation graphics. In this document, video output refers to the encoding hardware, and it is identified by a VIDEOOUT id. Logically, a client will connect an input device, such as a video tape recorder, to a video output using the **ChangeConnectivity** request, and then use **CaptureGraphics** to direct from which drawable the graphics will be captured.

**Video output region (VOR)**

This is defined as the set of pixels on the screen or in a pixmap that are being encoded by a **CaptureGraphics** request, whether it is accomplished by reading pixels from a frame buffer or by analog conversion.

**Video window (VW)**

A video window is defined as any window actively displaying video or one that has the potential to display nothing but video. Specifically,

- a window having a visual type **VideoColor** or **VideoGray**.
- a window with a core visual type, and actively servicing a **RenderVideo** request.

## 3. Visuals

VEX introduces four new visual types: **VideoGray**, **VideoColor**, **BlendStatic** and **BlendDynamic**. The first two are used to create one instance of a video window (VW), and the last two are used to create a blend window (BW).

For **VideoGray** and **VideoColor** The pixel value is ignored and the displayed RGB values are taken directly from a video input. Windows created with **VideoGray** display only the luminance portion of a video signal; windows created with **VideoColor** are able to display a color picture if chrominance information is available in the video signal. A side effect of this visual type is that the visible rendition of pixels not covered by a **RenderVideo** request are undefined, including the window's border.

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; **CreateWindow** will accept a blendmap where a colormap is specified if a blend visual type is also specified; other 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.

The pixels of a window having a blend visual, a BW, are defined only for those regions obscured by a BTW, which in turn is unobscured by any other window: this is the PBR. Visible regions in the BW are undefined, including the border.

Each pixel in the BW defines the mixture of corresponding pixels in the BBW and the BTW on a one-to-one basis. 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 potential blend region (PBR) are calculated as follows:

$$dvalue = \alpha \cdot bvalue + (1-\alpha) \cdot tvalue$$

where

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

and

*dvalue* is the displayed (visible) primary
*bvalue* is the graphic primary from the BBW
*tvalue* is the video primary from the BTW
*bpixel* is the pixel in the BW
*bmap* is the array of red protocol-intensities in the blendmap for the BW.

When a window becomes a BBW, and a pixel in the PBR changes from zero BTW to non-zero BTW, and the BTW pixel has not been retained by the hardware or the server, then the BTW pixel is treated as if a ClearArea had been performed on the pixel. Similarly, when a window becomes an BBW, and a pixel in the PBR changes from zero BTW to non-zero BTW, and the BTW pixel has not been retained by the hardware or the server, then the BTW 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 a **RenderVideo** request is active for the exposed pixel, digitization for that pixel will begin immediately. If there are multiple blend triples in a window hierarchy, such that a BTW in one triple is a BBW in another, the visible pixels on the screen for the areas common to both blend triples will be composed from the topmost triple.

Because of the semantics of blend visuals, BBW and BW pixels are not meaningful unless occluded by a BTW. For this reason, the potential blend region (PBR) of the BBW and BW 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 a BBW or BW, the changes take place even though the window is occluded. However, this does not guarantee that the pixels in the BBW and BW will not be discarded by the server for some reason.

## 4. Types

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

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

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

VIDEOIO: VIDEOIN or VIDEOUT

VIDEOID: VIDEOIN, VIDEOOUT or VIDEODEVICE

When supplied to a request, VIDEODEVICE must be one supplied by the **QueryVideo** request, else **Match** error. When a VIDEOIN or VIDEOOUT is supplied to a request, it must be one returned by the **CreateVideo** request, else **Match** error.

CONNECTION: [ source, destination: VIDEOID
old-state, new-state: CARD32 ]

The CONNECTION type is used by the **QueryVideo** request to describe a transition on the directed graph of adjacency matrices (returned by **QueryVideo**) when the given source and destination are connected. The old-state and new-state element are indices into the list of adjacency matrices. The two matrices they index describe the connectivity among video

inputs, outputs, and devices before and after the source and destination are connected.

CONTROL: [ type: {**Device**, **Input**, **Output**}
name: ATOM
id: VIDEOID or CARD32
format: { 8, 16, 32 }
length: CARD32 ]

The control structure is used to represent the capabilities of the "knobs" and "dials" on various video hardware. It is returned by the **QueryVideo** request. The name in a control is a control atom predefined by the extension. The id element identifies which resource the control affects. For video devices, the type is **Device** and the id is a VIDEOID; For video input and output, the identifiers are created by the client, so the type is **Input** or **Output** and the id is a CARD32 representing the numbered video input or output, respectively. The format and length precisely describe the structure expected in a **ChangeVideoControls** request. The length is the number of bytes expected in the value element of a SETTING structure whose id and name match those in the control. The format describes the arrangement of data cells so the server may swap them properly before interpreting them. The meaning of the data and the appropriate names for controls are described in another document, VEX Devices and Controls (not yet written).

SETTING: [ name: ATOM
id: VIDEOID
value: LISTofINT8 or LISTofINT16 or LISTofINT32 ]

This type is used to change the setting of a named control associated with a named VIDEOID. Only the **ChangeVideoControls** and **QueryVideoControls** requests utilize it. The control to be changed is named by the atom and is associated with the named id. The value must have a length in bytes equal to the length advertised in the corresponding CONTROL structure provided by **QueryVideo**, else Length error.

FRACTION: [ numerator, denominator: INT32 ]

The FRACTION type is used to represent a rational number; the denominator must be strictly positive (> 0) in all requests, else **Value** error.

FRACTIONRANGE: [ base, increment: FRACTION
limit: CARD32
type: { **Linear**, **Geometric** } ]

The fractions represented by FRACTIONRANGE depend on the type, and for each type, the members of each set are as follows:

**Linear**: { base, base + 1*increment, base + 2*increment, ...,
base + limit * increment }
**Geometric**: { increment ** (base+0), increment ** (base+1), ...,
increment ** limit }

RECTANGLERANGE: [ base, limit: RECTANGLE
x-inc, y-inc: INT16
width-inc, height-inc: CARD16
type: { **Linear**, **Geometric** } ]

The RECTANGLERANGE is ultimately used in the VIDEOGEOMETRY type below. It describes the possible geometries of sources and destinations for **RenderVideo** and **CaptureGraphics** requests. The set of rectangles described by a single

RECTANGLERANGE are all combinations of four sets derived for x, y, width and height. The members of the sets are determined based on the type as follows.

| | | |
|---|---|---|
| **Linear**: | x | = { base.x, base.x + x-inc, base.x + 2*x-inc, ..., limit.x } |
| | y | = { base.y, base.y + y-inc, base.y + 2*y-inc, ..., limit.y } |
| | width | = { base.width, base.width + width-inc, base.width + 2*width-inc, ..., limit.width } |
| | height | = { base.height, base.height + height-inc, base.height + 2*height-inc, ..., limit.height } |
| **Geometric**: | x | = { x-inc ** (base.x+0), x-inc ** (base.x+1), ..., x-inc ** limit.x } |
| | y | = { y-inc ** (base.y+0), y-inc ** (base.y+1), ..., y-inc ** limit.y } |
| | width | = { width-inc ** (base.width+0), width-inc ** (base.width+1), ..., width-inc ** limit.width } |
| | height | = { height-inc ** (base.height+0), height-inc ** (base.height+1), ..., height-inc ** limit.height } |

For example, the RECTANGLERANGE whose value is

[ base = {0, 0, 320, 240}, limit = {960, 784, 320, 240}
x-inc = 16, y-inc = 1
width-inc = 0, height-inc = 0
type = **Linear** ]

would describe a total of 47,100 {x, y, width, height} sets

{0, 0, 320, 240}
{16, 0, 320, 240}
...
{960, 0, 320, 240}
{0, 1, 320, 240}
{16, 1, 320, 240}
...
{960, 1, 320, 240}
etc.
{960, 784, 320, 240}

PLACEMENT: [ frame-rate: FRACTION }
source, destination: RECTANGLERANGE
x-scale, y-scale: FRACTIONRANGE ]

The PLACEMENT type is used in the VIDEOGEOMETRY type described below. Each placement describes the possible source and destination rectangles used in **RenderVideo** and **CaptureGraphics** requests, but restricted to the scale factors described by the x-scale and y-scale elements. For video input, the frame-rate element describes the maximum nominal frame rate that the hardware can achieve in a **RenderVideo** request using a source and destination rectangle derived from the other elements. For video output, the frame-rate element describes the maximum nominal rate at which frames can be encoded. It is important to stress that this is only a nominal rate and may vary due to other factors such as hardware contention. The x-scale and y-scale FRACTIONRANGE specifies the possible ratios of

destination x pixels : source x pixels
destination y pixels : source y pixels

where the ratio is numerator (destination) divided by denominator (source). The source and destination rectangles derived must have width and height ratios described by x-scale and y-scale. For example, given a PLACEMENT of

```
frame-rate      =   [ 30, 1 ]
source          =   [ base = {0, 0, 10, 15}, limit = {639, 479, 640, 480}
                      x-inc = 1, y-inc = 1
                      width-inc = 1, height-inc = 1
                      type = Linear ]
destination     =   [ base = {0, 0, 320, 240}, limit = {960, 784, 320, 240}
                      x-inc = 16, y-inc = 1
                      width-inc = 0, height-inc = 0
                      type = Linear ]
x-scale         =   [ base = {0, 1}, increment = {2, 1}
                      limit = 6
                      type = Geometric ]
y-scale         =   [ base = {0, 1}, increment = {2, 1}
                      limit = 5
                      type = Geometric ]
```

This implies that the source rectangles may be any with the constraints that

0 <= x <= 639
0 <= y <= 479
10 <= width <= 640
15 <= height <= 480

and the destination rectangles may be any selected from the earlier example. The combined rectangles are limited to those source-destination pairs where the fractions

$$width\text{-}scale = \frac{destination\ width}{source\ width} \quad \text{and} \quad height\text{-}scale = \frac{destination\ height}{source\ height}$$

can be reduced to one of the fractions from the respective sets

```
width-scale     = [ {1,1}, {2,1}, {4,1}, {8,1}, {16,1}, {32,1}, {64,1} ]
height-scale    = [ {1,1}, {2,1}, {4,1}, {8,1}, {16,1}, {32,1} ]
```

And for this example, using any of these source-destination pairs, the hardware in this example can maintain a nominal 30 fps.

```
VIDEOGEOMETRY:      [ signal-frame-rate: FRACTION
                      signal-field-rate: FRACTION
                      signal-width, signal-height: CARD16
                      concurrent-use: CARD16
                      placement: LISTofPLACEMENT ]
```

This type describes the attributes of video inputs and video outputs and is returned by the **QueryVideo** request. The signal-frame-rate and signal-field-rate describe the number of frames per second and fields per second in the signal format. For input, the rate describes the signal received; for output, the signal produced.

For video input, the signal-width and signal-height describe the dimensions of the signal picture as if it were placed directly on the workstation screen; for video output, they describe the dimensions of a screen region whose pixels would map one-to-one to the output signal. It is important to note that the frame-buffer holding the video picture, if any, may not actually have these dimensions; but the numbers describe the extent of the source rectangle for video input, or the destination rectangle for the video output.

For a video input and output, the concurrent-use indicates the number of times the resource may be in simultaneous use, and implies the number identifiers a client may create using **CreateVideo**. The server should publish a number that the hardware can support with respect to connectivity and quality of the input or output. Clients should use this number in deciding how many resources they may use, and a video resource manager should use this

in deciding how to handle allocation among multiple clients.

Each placement element in the list describes its own range of values. The complete range of placement parameters is derived from the entire list, but parts of one element in the list may not be combined with another. So, for example, source and destination rectangles described in one placement may not be combined with scale factors described in another placement.

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

This type is used by **QueryVideo** and **QueryBlend** to describe the depth and visual type of a potential VW, BBW, BW or BTW. The visuals are described by the connection setup, the **QueryBlend** and/or **QueryVideo** requests.

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

This type is used by the **QueryBlend** request to describe the acceptable members of a single blend triple. 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 a BBW or BTW. The blend element is also an index into the list of allowed-depths returned by **QueryBlend** and indicates the depth and visual of the BW.

If **Model1Overlay** is asserted, then a window created with a depth and visual indicated by the model1 element may become a DOW. Likewise, if **Model2Overlay** is asserted, then a window created with a depth and visual indicated by the model2 element may become a DOW. If **Composite** is asserted, then pixels in the BTW and BBW occupy the same frame buffer. In this case, pixels within a PBR copied from a BTW window that are not 100% BTW are undefined. Likewise, pixels within a PBR copied from a BBW that are not 100% BBW are undefined. If **Composite** is not asserted, then BBW and BTW pixels occupy separate frame buffers. In this case, pixels copied from a BBW or BTW are guaranteed to be from that drawable alone, even in the PBR.

OWNER: [ wid: WINDOW
vid: VIDEOID ]

This type is used by the **SetOwnership** request to establish ownership for any of the video resources: input, output and devices. The specified window identifies the client, and the videoid indicates the resource of which it gains ownership.

## 5. Requests

### QueryVideo

*wid*: WINDOW
==>
*screen*: CARD8
*video-depths*: LISTofDEPTHS
*allowed-depths*: LISTofRENDERMODEL
*vdev*: LISTofVIDEODEVICE
*inattr*: LISTofVIDEOGEOMETRY
*outattr*: LISTofVIDEOGEOMETRY
*video-input-models*: LISTofBITMASK {**Pixmap**, **Window**}
*video-output-models*: LISTofBITMASK {**Pixmap**, **Window**, **Composite**}
*videoid-controls*: LISTofCONTROL
*clip-size*: LISTofRECTANLGE
*input-overlap*: BOOL
*capture-overlap*: BOOL
*io-overlap*: BOOL
*adjacency-matrices*: LISTofLISTofCARD8
*transitions*: LISTofCONNECTION
*current-state*: CARD32
*time*: TIMESTAMP

Errors: **Window**

This request returns information about video hardware for the screen associated with the specified window. The screen is returned as the first element in the reply. If there is no video hardware available for the screen, a **Match** error is returned.

The video-depths specify what depths/visuals are unique for video input and output. These are guaranteed to be different from those returned in the X connection setup; the list may be null; pixmaps are supported for each depth listed. Further, the presence of a VEX extension in a server may cause the LISTofFORMAT provided by the connection setup to be extended. The additional formats would be those that would allow **GetImage** and **PutImage** access to windows or pixmaps created with depths and visuals not published in the connection setup. The definition of DEPTH is included in the core protocol's description of the connection information.

The allowed-depths specify what depths/visuals are supported for use with video input and video output. These depths and visuals are selected from those listed in video-depths, and those provided by the X connection setup. The list is in no particular order.

The vdev lists the devices that provide signals to a video input or accept signals from a video output, such as playback or record. The list may be null and implies that the server has no control over any devices that may be attached to the video inputs or video outputs.

The inattr list the attributes of the decoder or digitizer used by the video inputs. The outattr list the attributes of the encoder used by the video outputs. The entries in both lists correspond one-to-one with the unique video input renderers and video output encoders; the length of these lists imply the number of rows in video-input-models and video-output-models. **CreateVideo** and the videoid-controls described in this request reference these resources by this implied index, starting from 0.

The list of video-input-models should be interpreted as a two dimensional array, column moving fastest and represents the relationship between the video input resources and the set of depths/visuals used to create video windows (VW). The columns are labeled left-to-right with the list of allowed depths/visuals, and the rows are labeled top-to-bottom with the list of video inputs. Each cell is a bitmask containing zero or more true bits.

**Window** is asserted if a **RenderVideo** request may specify a window with the intersecting depth/visual as a destination and the intersecting video input as the source. **Pixmap** is asserted if a **RenderVideo** request may specify a pixmap with the intersecting depth as a destination and the intersecting video input as the source. There is guaranteed to be at least one **Window** or **Pixmap** assertion in every row.

For example, the following array means that a VW for video input #1 must be depth 12, **TrueColor**; or depth 0, **VideoColor**; pixmaps of depth 24 can be used as a destination for a **RenderVideo** request. Video input #2 can only support pixmaps of depth 12.

| | | VideoColor depth 0 | TrueColor depth 12 | None depth 24 |
|---|---|---|---|---|
| *Video in* | #1 | Window | Window | Pixmap |
| | #2 | 0 | Pixmap | 0 |

The list of video-output-models should be interpreted as a two dimensional array, column moving fastest, and represents the relationship between the video output resources and the set of depths/visuals that each may capture. The columns are labeled left-to-right with the list of allowed depths/visuals, and the rows are labeled top-to-bottom with the list of video outputs. Each cell is a bitmask containing zero or more true bits.

If **Window** is asserted, then a video output can capture the intersecting depth/visual in a window, including the border. If **Composite** is asserted, and a window having the intersecting depth/visual is specified in a **CaptureGraphics** request, and the window is a member of a blend triple, then the window's contents will be captured after its primaries are blended. If **Composite** is asserted, then **Window** will be asserted for the same cell; The assertion of **Composite** is guaranteed to agree with the information returned by **QueryBlend**. If **Composite** is not asserted, but **Window** is asserted, then capture from a window having the intersecting depth/visual will always consist of unblended pixels from the source window and its descendants of the same depth, whether or not the window is a member of a blend triple. If a descendant is a member of a blend triple, the result is undefined: it is hardware dependent. If **Pixmap** is asserted, then a pixmap with the intersecting depth can be captured.

For example, if the video hardware has two video outputs, one able to capture depth 12, **TrueColor**, blended windows, and one able to capture depth 24 pixmaps, then the array would look like:

| | | TrueColor depth 12 | TrueColor depth 24 |
|---|---|---|---|
| *Video out* | #1 | Composite, Window | 0 |
| | #2 | 0 | Pixmap |

The videoid-controls describe the controls, if any, for each VIDEOID. The order of the list will, in general, match the vin, vout and vdev, although the number of controls for any one particular VIDEOID is arbitrary. The list may be null.

The clip-size list corresponds to the list of video inputs, and describes the smallest rectangle that can be used to clip a VIR partly occluded by other windows. If either the width or the height are 0, then the entire VIR must be displayed and cannot be clipped. This is meant as advisory for clients and window managers as they choose window placement.

If the input-overlap element is false, then VIRs may not overlap; true otherwise. The capture-overlap element is false if the hardware does not allow VORs to overlap; true otherwise. The io-overlap element is false if the hardware does not allow a VIR and a VOR to overlap; true otherwise. Note that io-overlap may be true, but video-output-models determines if a VIR may be captured or not.

In all cases, if clip-size, input-overlap, or io-overlap constraints are violated, the content of the violated regions is undefined; if capture-overlap or io-overlap is violated, then the

output signal for violated regions are undefined and is hardware dependent; no errors are returned to the offending request, but a **VideoViolation** event may be generated. In addition, other constraints that VEX cannot express may be violated, and a **VideoViolation** event may be generated. Requests that may violate constraints are **RenderVideo**, **CaptureGraphics**, **MapWindow**, **UnmapWindow**, **MapSubwindows**, **UnmapSubwindows**, **ConfigureWindow**, **CirculateWindow**, **DestroyWindow**, **DestroySubwindows**, **ReparentWindow**.

The adjacency-matrices should be interpreted as a list of two dimensional arrays, each array the same size, each with column moving fastest. Each array represents a possible set of connections among video input, output and devices.

The columns in each individual matrix represent sources, and are labeled left-to-right with the video devices followed by the video outputs; the rows represent destinations, and are labeled top-to-bottom with the video devices followed by the video inputs. This implies that no video input may be a source, and no video output may be a destination. Each column is a single bit: the first video device is the most significant bit, the next device is next most significant bit, and so forth through the least significant bit, and on to the next byte. Each row is

$$\frac{(devices + outputs) + 7}{8}$$

bytes in length. The bits in an array are true if the intersecting source and destination are connected as such; false, otherwise. The list of arrays is an unordered list of nodes on an directed graph.

The transitions list provides the edges to the graph. Each entry describes a single transition between two nodes given the connection of a source and destination. The nodes are adjacency matrices in the list, indexed by the old-state and new-state elements of the CONNECTION type. Each edge described is bi-directional; i.e., disconnecting the same source and destination while in the new-state will transition back to old-state. Also, other connections or disconnections may be implied by a single transition, and these edges will also appear in the transitions list.

For example, if there are four matrices and four transitions returned by **QueryVideo**

| | *matrix 0* | *sources* dev #1 | dev #2 | out #1 |
|---|---|---|---|---|
| | dev #1 | F | T | F |
| *dest.* | dev #2 | F | F | F |
| | in #1 | F | F | F |

| | *matrix 2* | *sources* dev #1 | dev #2 | out #1 |
|---|---|---|---|---|
| | dev #1 | F | T | F |
| *dest.* | dev #2 | F | F | F |
| | in #1 | T | F | F |

| | *matrix 1* | *sources* dev #1 | dev #2 | out #1 |
|---|---|---|---|---|
| | dev #1 | F | T | F |
| *dest.* | dev #2 | F | F | T |
| | in #1 | F | F | F |

| | *matrix 3* | *sources* dev #1 | dev #2 | out #1 |
|---|---|---|---|---|
| | dev #1 | F | T | F |
| *dest.* | dev #2 | F | F | T |
| | in #1 | T | F | F |

{ out1, dev2, 0, 1 }
{ dev1, in1, 0, 2 }
{ dev1, in1, 1, 3 }
{ out1, dev2, 2, 3 }

This states that device 1 and 2 are always connected as destination and source, respectively; output 1 may be connected to device 2 at any time; device 1 may be connected to input 1 at any time.

The current-state element describes the present state of connection; i.e. which matrix in the adjacency-matrices represents the current connectivity. **VideoConnectivityState** events are guaranteed to be delivered following each change caused by a **ChangeConnectivity**

request.

The time element is used to prevent race conditions between the delivery of a **VideoChange** event and the use of some other VEX requests. The value of the timestamp is constant until there is a change in the list of video devices, and is then replaced with the current server time.

**CreateVideo**

*id*: VIDEOIO
*which*: CARD16
*kind*: {**Input**, **Output**}

Errors: **Alloc**, **Value**,

This request creates the specified identifier handle for the video input or output resource indexed by which; that index must be less than the number of inputs or outputs, as appropriate, else **Value** error. If **Input** is indicated, then the identifier is a VIDEOIN; if **Output** is indicated then the identifier is a VIDEOOUT. This resource is freed on connection close for the creating client. The number of identifiers created for a single client corresponding to a particular input or output must not exceed the concurrent-use for that resource as supplied by **QueryVideo**, else **Alloc** error.

**RenderVideo**

*source*: VIDEOIN
*destination*: DRAWABLE
*src-x, src-y, dest-x, dest-y*: CARD16
*src-width, src-height, dest-width, dest-height*: CARD16
*full-motion*: BOOL
*priority*: CARD8
*subwindow-mode*: {**ClipByChildren**, **IncludeInferiors**}

Errors: **Drawable**, **Match**, **Value**

The destination must be a valid drawable id, else **Drawable** error. The specified region of the source video signal is decoded and directed to the destination. The visible portion of the destination rectangle becomes a VIR.

In general, it is intended that when a window of visual type **VideoGray** or **VideoColor** is the destination, the pixels of the video picture are inaccessible and are magically placed "over" the screen its pixels. When the destination is a window having a core visual type, the intent is that the video pixels are fully accessible, but happen to change "very frequently". However, not all hardware supports this concept and pixels may migrate between domains, become intermixed, and sometimes undefined because of a change in depth or visual. This asymmetry is defined where possible and is described here and in the description for **QueryVideo** and **QueryBlend**.

If the drawable is a window, then **Window** must be asserted in the video-input-models returned by **QueryVideo** for the intersecting visual/depth of the window and the specified video input, else **Match** error. If the destination is a pixmap, then **Pixmap** must be asserted in the video-input-models for the intersecting depth of the pixmap and the specified video input, else **Match** error.

If the window is visible, then the visible AVR pixels are displayed. If the destination is an unmapped window, then the window's pixels may be rendered, depending on the value of backing-store for the window.

The src-x, src-y, src-width and src-height describe the extent of the source rectangle in the video picture. The dest-x, dest-y, dest-width and dest-height describe the rectangle in the destination drawable to receive the video picture. The source video picture will be scaled by

$$x\ scale = \frac{dest-width}{src-width} \qquad y\ scale = \frac{dest-height}{src-height}$$

The source and destination rectangles and scaling factors must match one of those provided in a single PLACEMENT by the **QueryVideo** request for the specified video input source, else **Value** error. If the destination rectangle is valid, but some or all of it is outside the extent of the destination drawable, the result is clipped to a granularity advertised by **QueryVideo**, and a **VideoViolation** event may be generated. If the source rectangle is valid, but some or all of it is outside the extent of the video picture, the result in the corresponding region of the destination is undefined: it is hardware dependent.

The full-motion element determines whether the video input signal is continually updated or not. If it is true, then the server will try to achieve the maximum nominal rate advertised by the **QueryVideo** request for the specified scale factor and placement. If it is False, then a single frame will be rendered. It is just as if the **RenderVideo** request was immediately followed with a **StopVideo** request specifying the same drawable. However, a full-motion element of False is an aid to the server to avoid extra digitization.

Priority is a value (0 to 100 inclusive, else **Value** error), representing the percentage of video bandwidth that the server should try to preserve. A value of 0 tells the server that graphic output to the screen should have top priority and that the video decoding may occasionally miss some incoming or outgoing frames. A value of 100 gives video decoding top priority meaning that the graphics performance may be degraded. It is advisory only and does not guarantee anything.

Subwindow-mode controls the treatment of the window's children. For **ClipByChildren**, the destination is clipped by all viewable children. If **IncludeInferiors** is specified, then viewable inferiors of the destination that overlap the VIR are updated with pixels from the video picture if the destination and its inferiors have a matching visual/depth. Otherwise, the result in the inferiors is undefined.

If the destination window is clipped by siblings or children, a **VideoViolation** event may be generated if an overlap or clipping constraint is violated; see **QueryVideo**.

A **RenderVideo** request using a window W already allocated to an earlier **RenderVideo** request has the same effect as issuing a **StopVideo** request with arguments W and **Render**, followed by the new **RenderVideo** request. Similarly, a **RenderVideo** request using a video input V already allocated to an earlier **RenderVideo** request has the same effect as issuing a **StopVideo** request with arguments W' and **Render**, where W' is the window from the earlier request, followed by the new **RenderVideo** request. In either case, a **VideoOverride** event may be generated.

Clients that do not need access to digitized pixels should use a window having a **VideoGray** or **VideoColor** visual type as a destination. This allows the server to display the video picture using the most efficient means. There is no guarantee that the signal is actually digitized, even if a window with a core visual type is specified. However, **CopyArea**, **CopyPlane** and **GetImage** requests with a source region within a VIR are guaranteed to return a consistent frame with respect to adjacent fields in the video signal.

If the specified window changes size while the request is still active, the destination x and y the original request will be adjusted to follow the bit gravity of the destination, but the width, height and scale are not adjusted. If the resulting rectangle is beyond the extent of the destination, the result is clipped appropriately; a **VideoViolation** event may be generated.

Graphics may be drawn on a VW, but if the window is servicing an active **RenderVideo** request, the results are undefined.

**CaptureGraphics**

*source*: DRAWABLE
*destination*: VIDEOOUT
*src-x, src-y, dest-x, dest-y*: CARD16
*src-width, src-height, dest-width, dest-height*: CARD16
*full-motion*: BOOL
*priority*: CARD8
*subwindow-mode*: {**ClipByChildren**, **IncludeInferiors**}

Errors: **Drawable**, **Match**, **Value**

The source must be a valid drawable id, else **Drawable** error.

The specified region of the source is encoded into a video signal by the destination encoder. If the drawable is a window, then **Window** must be asserted in the video-output-models returned by **QueryVideo** for the intersecting visual/depth of the window and the specified video output, else **Match** error. If the source is a pixmap, then **Pixmap** must be asserted in the video-output-models for the intersecting depth of the pixmap and the specified video output, else **Match** error.

The src-x, src-y, src-width and src-height describe the extent of the source rectangle of the region captured. The dest-x, dest-y, dest-width and dest-height describe the rectangle in the destination video picture to receive the graphics. The source graphics region will be scaled by

$$x\ scale = \frac{dest-width}{src-width} \qquad y\ scale = \frac{dest-height}{src-height}$$

The scaling factors applied must match one of those provided by the **QueryVideo** request for the specified video output destination, else **Value** error. If the destination rectangle is valid, but some or all of it is outside the extent of the video picture, those pixels are clipped in the output video picture. If the source rectangle is valid, but some or all of it is beyond the extent of the source drawable, the result in the encoded video signal is undefined: it is hardware dependent.

The full-motion element determines whether the video output signal is constantly updated or not. If it is true, then the server will try to achieve the maximum rate advertised by the **QueryVideo** request for the specified scale factor and placement. If it is False, then a single frame will be captured. It is just as if the **CaptureGraphics** request was immediately followed with a **StopVideo** request specifying the same drawable; however, a full-motion element of False is an aid to the server to avoid extra encoding.

Priority is a value (0 to 100 inclusive, else **Value** error), representing the percentage of video bandwidth that the server should try to preserve. A value of 0 tells the server that graphic output to the screen should have top priority and that the video capture process may occasionally miss some incoming or outgoing frames. A value of 100 gives video capture top priority meaning that the graphics performance may be degraded. It is advisory only and does not guarantee anything.

Subwindow-mode controls the capture of the window's children. If **ClipByChildren** is specified, the source is clipped by all viewable children only if **Composite** is not asserted in the video-output-models for the video output destination. If **Composite** is asserted, subwindow-mode will be effectively **IncludeInferiors**, regardless of its value.

If **IncludeInferiors** is specified for the subwindow-mode, then each visible region of an inferior of the source that overlaps the VIR may captured, but only if one or both of the following are true:

- The inferior is the same depth, visual and window type as the source window.
- **Composite** is asserted in the video-output-models for the video output destination.

Otherwise, the source rectangle is clipped.

If the source rectangle is clipped by siblings or children, the appearance of the clipped regions in the encoded picture are undefined: it is hardware dependent.

A **CaptureGraphics** request using a window W already allocated to an earlier **CaptureGraphics** request has the same effect as issuing a **StopVideo** request with arguments W and **Capture**, followed by the new **CaptureGraphics** request. Similarly, a **CaptureGraphics** request using a video output V already allocated to an earlier **CaptureGraphics** request has the same effect as issuing a **StopVideo** request with arguments W' and **Capture**, where W' is the window from the earlier request, followed by the new **CaptureGraphics** request. In either case, a **VideoOverride** event may be generated.

If the specified window changes size while the request is still active, a **VideoViolation** event may be generated. Only the source x and y of the original request will be adjusted to follow the bit gravity of the source. If a portion of the new rectangle is beyond the extent of the source, the result in the output signal is undefined.

**StopVideo**

*dlist*: LISTofDRAWABLE
*action*: BITMASK {**Render**, **Capture**}

Errors: **Drawable**, **Value**

The id must be a valid drawable, else **Drawable** error. If more than one id is in error, which one is reported is arbitrary.

The video actions previously requested on any of the listed ids are stopped, even if one of the ids produces an error. If **Render** is specified, then **RenderVideo** requests on the listed ids are terminated. If **Capture**, is specified, then **CaptureGraphics** requests on the listed ids are terminated. Both may be specified. If neither **RenderVideo** or **CaptureGraphics** are currently active on a id, no error is reported.

If a **RenderVideo** on window W having a visual of type **VideoColor** or **VideoGray** is followed by a **StopVideo** on W, then the visible result on the screen is undefined, because it is hardware dependent. If a **RenderVideo** on a window W having a core visual type is followed by a **StopVideo** on W, the result in W is the pixels take on the value of the last frame acquired: they can be manipulated by other X requests.

**QueryVideoControls**

*vid*: VIDEOID
*control-list*: LISTofATOMS
==>
LISTofSETTING

Errors: **Match**, **Atom**

This request provides the current setting associated with each of the listed controls for the listed video id. The control list must be taken from the set of control names supplied by **QueryVideo**, else Atom error.

If the vid or any of the control-list generate an error, no settings are returned.

**ChangeVideoControls**

*setting*: LISTofSETTING

Errors: **Window**, **Match**, **Atom**, **Value**, **Length**

This request changes the setting of each control named in each SETTING structure. The control named by the atom and associated with the specified id is set according to the value of the data. A **VideoControl** event is generated on the named window when the control setting has actually been accomplished, failed, or was denied.

All valid settings take effect, even if some of the settings are invalid. If an error is returned, which setting generated the error is not specified.

**ChangeConnectivity**

*time*: TIMESTAMP
*old-state*, *new-state*: CARD32

Errors: **Match**, **Value**, **Access**

The time specified must be later than the time returned by the last **VideoChange** event, else **Value** error.

The state of the network transitions to new-state if the current state index is old-state. If the current state is not old-state, then no transition is made and a **Match** error is returned. However, if the connectivity of the network is already in new-state, no error is returned and the current connectivity is maintained. Moving from old-state to new-state may require several transitions along the directed graph of adjacency-matrices. If the request would violate an ownership, an **Access** error is returned.

**ChangeOwnership**

*owners*: LISTofOWNER

Errors: **Window**, **Match**

For each member of the list, the client indicated by the specified window is given ownership of the specified resource. For a video input or video output, this means that subsequent **RenderVideo** or **CaptureGraphics** requests are guaranteed not to be redirected, respectively. In addition, all controls associated with the resource may only be changed by the owning clients. The number of clients given ownership of a single video input or output must not exceed the concurrent-use for that resource as supplied by **QueryVideo**, else **Match** error.

If a **RenderVideo** or **CaptureGraphics** request is issued for a resource, and one or more instances of the resource are in use by a client that does not have ownership, then a **StopVideo** request will be implied as described by **RenderVideo** and **CaptureGraphics**, regardless of whether the issuing client has ownership for that resource or not. A **VideoOverride** request will be generated on the drawable. Under the same circumstances, if the issuing client does not have ownership, and all instances of the resource have ownership, the request will not take effect and a **VideoRequest** event will be generated. Note that a request that uses resource whose usage is not saturated will always succeed, regardless of ownership.

Similarly, for a video device, the client indicated by the specified window is given complete ownership of the specified device. **ChangeConnectivity** requests are guaranteed not to violate the connectivity of that device, i.e. whatever it is immediately connected to will remain connected. Second level connectivity is not guaranteed. In addition, all **ChangeVideoControls** requests on a device will only succeed for the owning client. If a **ChangeVideoControls** request would violate ownership, a **VideoControl** event will be generated on the resource.

If a device is not owned by a specific client, all clients may affect the device's connectivity or controls. The number of clients given ownership of a single device must never exceed one, else **Match** error. Generic ownership is achieved by designating the root window as the client-owner. Ownership can be rescinded by setting the owner to the root window or by destroying the previously designated window.

**QueryBlend**

*wid*: WINDOW
==>
*blend-depths*: LISTofDEPTHS
*allowed-depths*: LISTofRENDERMODEL

*blend-triples*: BLENDTRIPLES
*default-blendmap*: COLORMAP
*min-installed-maps, max-installed-maps*: CARD16

Errors: **Window**

This request describes the blending capabilities of the server's hardware.

The blend-depths specify what depths/visuals are unique for blending. These depths are guaranteed to be different from those returned in the X connection setup and will only include visual types **BlendStatic** and **BlendDynamic**; pixmaps are supported for each depth listed. Further, the presence of a VEX extension in a server may cause the LISTofFORMAT provided by the connection setup to be extended. The additional formats would be those that would allow **GetImage** and **PutImage** access to windows or pixmaps created with depths and visuals not published in the connection setup. The definition of DEPTH is included in the core protocol's description of the connection information.

The allowed-depths specify what depths/visuals are supported for use with in blend triples, as seen in the description of BLENDTRIPLES above. These depths and visual ids are selected from those listed in video-depths by **QueryVideo**, those listed in blend-depths, and those provided by the X connection setup. The list is in no particular order.

The list of blend triples describe the possible combinations of depths and visuals for each member of a blend triple.

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 BWs 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.

**ChangeBlendGC**

*gc*: GCONTEXT
*blendgc*: GCONTEXT
*value-mask*: BITMASK

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

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 a blend window (BW) using gc will imply similar operations to a designated overlay window (DOW) 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 | Inhieritance |
|---|---|
| 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 BW is aligned with stipple in the DOW |
| tile-stipple-y-origin | adjusted so the stipple in the BW is aligned with stipple in the DOW |
| subwindow-mode | the subwindow-mode is always **ClipByChildren**. |
| graphics-exposures | copied |
| clip-x-origin | adjusted so the clipped region in the BW is aligned with the clipped region in the DOW. |
| clip-y-origin | adjusted so the clipped region in the BW is aligned with the clipped region in the DOW. |
| clip-mask | copied |
| dash-offset | copied |
| dashes | |

Inhierited values in a gc override explicit changes via **ChangeGC** at all times.

**SelectOverlay**

*overlay, underlay*: WINDOW
*mode*: {**Select**, **Unselect**}

Errors: **Window**, **Match**

This request selects or removes the designated overlay window (DOW) for a blend triple. The two windows must be a member of the same blend triple, else **Match** error. If the mode is **Select**, the overlay window becomes the DOW for the blend triple canceling any previous designation for that triple; the specified overlay must have an allowed depth and visual, as provided by **QueryBlend**, else **Match** error. If the mode is **Unselect** then the specified blend triple will no longer have a DOW and further operations to the former DOW will not imply operations to the BW.

A single window may be a DOW for more than one blend triple. Note that it is possible for a BBW or a BTW to be a DOW. After a blend triple has been created, and before any **SelectOverlay** has been performed on it, the triple has no DOW.

Whenever a gc is used in a request that specifies a DOW as the destination, the same request is performed on the BW of the 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 BW, 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 DOW implies a **CopyArea** request with the src-x, src-y, dst-x and dst-y adjusted so the regions in the DOW align with the same regions in the BW. If the source rectangle is outside the PBR, or if the source drawable is different than the DOW, a **PolyFillRectangle** request is performed on the BW with a list of rectangles that correspond to and are aligned with the regions of the DOW 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 DOW causes a **PolyFillRectangle** request to be performed with a list of rectangles that correspond to and are aligned with the regions of the DOW that are actually modified by the request.

The following requests performed on the DOW, imply the same request on the BW with all x and y coordinates adjusted to align with the original coordinates in the DOW: **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 DOW, the same request is performed on the BW with the x and y values adjusted to align with the same region in the DOW.

Other graphic extensions may render to a DOW implying equivalent requests on the BW. However, whether an equivalent request exists or is meaningful for the BW 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).

**SelectVideoEvents**

*id*: VIDEOID or DRAWABLE
*mask*: BITMASK
{**VideoRequests**, **VideoRedirect**, **DeviceNotify,**
**SyncNotify**, **ControlNotify**}

Errors: **Drawable**, **Match**

This requests selects events for the specified resource. For **VideoRequests** and **VideoRedirect** the id specified is ignored because the event is global. Selecting **DeviceNotify** will always generate a **VideoConnectivityState**.

## 6. Core Requests

**CreateWindow**

< this is a core protocol request >

If the VEX extension is present, this request will understand four new visual types: **VideoGray**, **VideoColor**, **BlendStatic** and **BlendDynamic**. The first two are used to create one instance of a video window (VW), and the last two are used to create a blend window (BW).

This request will return a **Match** error if a BW or BTW is created that would form an illegal blend triple. Visuals specified in a **CreateWindow** request must be one supported by the screen, and may be supplied by **QueryVideo** or **QueryBlend**.

**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 visualtype appropriate for blending. These visuals are described by the **QueryVideo** request, and will be selected from the color models **BlendStatic** and **BlendDynamic**. All other core 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.

**ReparentWindow**

< this is a core protocol request >

The **QueryBlend** request specifies certain relationships between blendable depths and visual types. An attempt to reparent a BBW, BW or BTW that would violate the acceptable relationships will cause a **Match** error.

## 7. Events

Every VEX event also contains the least-significant 16 bits of the sequence number of the last request issued by the client that was (or is currently being) processed by the server.

**VideoConnectivityState**

*screen*: CARD8
*new-state*: CARD32
*time*: TIMESTAMP
*sequence*: CARD32

This event is reported to the client selecting **DeviceNotify** on the specified screen when the connectivity of video inputs, outputs and devices changes state. The new-state element is an index into the list of adjacency matrices returned by **QueryVideo**.

The time reported is the time of the last network change caused by a **ChangeConnectivity** request.

The delivery of the **VideoConnectivityState** may not happen until some time after a **ChangeConnectivity** request, implying that the sequence number will not necessarily correspond to the originating request. The sequence number will be filled in with the sequence number of the originating request for the issuing client; for all other clients, it will be set to zero.

**VideoChange**

*screen*: CARD8
*vdev*: VIDEODEVICE
*state*: {**Online**, **Offline**}
*time*: TIMESTAMP

This event is reported unconditionally to any client that has issued a **QueryVideo** request on the specified screen when the named video device has changed state. Issuing an **QueryVideo** request will yield a new device list and adjacency matrices. New video devices are guaranteed not to have the same id as devices that were once listed, except for devices that go off line and return later: these devices will retain their previous id. This event is guaranteed to be delivered before any **VideoConnectivityState** events are delivered for the new connectivity state.

The time is what would be reported by the **QueryVideo** request.

**VideoControl**

*vid*: VIDEOID
*name*: ATOM
*state*: {**Success**, **Fail**, **Denied**}
*time*: TIMESTAMP

This event is generated when **ControlNofify** is selected on the resource, or if a **ChangeVideoControls** request failed in an unexpected way, or violated ownership of a device. The vid and name are those in the SETTING given to the **ChangeVideoControls**

request. The state is **Success** if the setting was accomplished, **Fail** if the setting failed, or **Denied** if the setting would have violated ownership.

The time is the current server time.

**VideoViolation**

*window*: WINDOW
*vid*: VIDEOIO
*action*: ACTIONMASK
where ACTIONMASK is {**Scale**, **Placement**, **Clip**, **Overlap**}
*state*: {**Success**, **Subset**, **Fail**}
*time*: TIMESTAMP

This event is reported to the client selecting **Exposure** on the window. The event is generated for the specified window and video input or output when scaling, placement, clipping and overlap constraints are violated. If state is **Success**, then the listed violations caused no change in the display or video output for the specified window, although some other window may be affected; If state is **Subset**, then some subset of the unobscured regions are still being displayed or captured; if the state is **Fail**, then the video input is no longer being displayed, or the capture is no longer occurring.

The time is the current server time.

**VideoSync**

*id*: DRAWABLE
*vin*: VIDEOIN
*state*: {**Acquired**, **Lost**}
*time*: TIMESTAMP

This event is reported to the client selecting **SyncNotify** on the vin. It is generated following a **RenderVideo** request with the specified id and video input, and before a **StopVideo** request on same, whenever the sync is lost or gained. At least one **VideoSync** event is guaranteed following a **RenderVideo** request, if selected.

The time is the current server time.

**VideoOverride**

*id*: DRAWABLE
*vin*: VIDEOIN
*time*: TIMESTAMP

This event is reported to the client selecting **Exposure** on the id. It is generated following a **RenderVideo** or **CaptureGraphics** request that results in an implicit **StopVideo** on the specified id.

The time is the current server time.

**VideoRequest**

*request*: { **Render**, **Capture**}
*redirect*: BOOL
*source*: DRAWABLE
*destination*: VIDEOOUT
*src-x, src-y, dest-x, dest-y*: CARD16
*src-width, src-height, dest-width, dest-height*: CARD16
*full-motion*: BOOL
*priority*: CARD8
*subwindow-mode*: {**ClipByChildren**, **IncludeInferiors**}

This event is reported to the client selecting **VideoRequests** or **VideoRedirect**. The request element indicates whether the requests was **RenderVideo** or **CaptureGraphics**. If **VideoRedirect** is selected, every request issued using a resource for which the client does not have ownership will generate a **VideoRequest** event with redirect true. If

**VideoRequests** is selected, every **RenderVideo** or **CaptureGraphics** request will generate a a **VideoRequest** event with redirect false.