Extending X for Double-Buffering, Multi-Buffering, and Stereo

Jeffrey Friedberg
Larry Seiler
Jeff Vroom

Version 3.3
January 11, 1990

The Multi-Buffering *extension described here was a draft standard of the X Consortium prior to Release 6.1. It has been superseded by the Double Buffer Extension (DBE). DBE is an X Consortium Standard as of Release 6.1.*

Introduction

Several proposals have been written that address some of the issues surrounding the support of double-buffered, multi-buffered, and stereo windows in the X Window System:

- *Extending X for Double-Buffering,* Jeffrey Friedberg, Larry Seiler, Randi Rost.
- *(Proposal for) Double-Buffering Extensions*, Jeff Vroom.
- *An Extension to X.11 for Displays with Multiple Buffers,* David S.H. Rosenthal.
- *A Multiple Buffering/Stereo Proposal*, Mark Patrick.

The authors of this proposal have tried to unify the above documents to yield a proposal that incorporates support for double-buffering, multi-buffering, and stereo in a way that is acceptable to all concerned.

Goals

Clients should be able to:

- Associate multiple buffers with a window.

- Paint in any buffer associated with a window.
- Display any buffer associated with a window.
- Display a series of buffers in a window in rapid succession to achieve a *smooth* animation.
- Request simultaneous display of different buffers in different windows.

In addition, the extension should:

- Allow existing X applications to run unchanged.
- Support a range of implementation methods that can capitalize on existing hardware features.

Image Buffers

Normal windows are created using the standard **CreateWindow** request:

```
CreateWindow
    parent        : WINDOW
    w_id          : WINDOW
    depth         : CARD8
    visual        : VISUALID or CopyFromParent
    x, y          : INT16
    width, height : INT16
    border_width  : INT16
    value_mask    : BITMASK
    value_list    : LISTofVALUE
```

This request allocates a set of window attributes and a buffer into which an image can be drawn. The contents of this *image buffer* will be displayed when the window is mapped to the screen.

To support double-buffering and multi-buffering, we introduce the notion that additional image buffers can be created and bound together to form groups. The following rules will apply:

- All image buffers in a group will have the same visual type, depth, and geometry (ie: width and height).
- Only one image buffer per group can be displayed at a time.
- Draw operations can occur to any image buffer at any time.
- Window management requests (**MapWindow**, **DestroyWindow**, **ConfigureWindow**, etc...) affect all image buffers associated with a window.
- Appropriate resize and exposure events will be generated for every image buffer that is affected by a window management operation.

By allowing draw operations to occur on any image buffer at any time, a client could, on a multi-threaded multi-processor server, simultaneously build up images for display. To support this, each buffer must have its own resource ID. Since buffers are different than windows and pixmaps (buffers are not hierarchical and pixmaps cannot be displayed) a new resource, **Buffer**, is introduced. Furthermore, a **Buffer** is also a **Drawable**, thus draw operations may also be performed on buffers simply by passing a buffer ID to the existing pixmap/window interface.

To allow existing X applications to work unchanged, we assume a window ID passed in a draw request, for a multi-buffered window, will be an *alias* for the ID of the currently displayed image buffer. Any draw requests (eq: **GetImage**) on the window will be relative to the displayed image buffer.

In window management requests, only a window ID will be accepted. Requests like **QueryTree**, will continue to return only window ID's. Most events will return just the window ID. Some new events, described in a subsequent section, will return a buffer ID.

When a window has backing store the contents of the window are saved off-screen. Likewise, when the contents of an image buffer of a multi-buffer window is saved off-screen, it is said to have backing store. This applies to all image buffers, whether or not they are selected for display.

In some multi-buffer implementations, undisplayed buffers might be implemented using pixmaps. Since the contents of pixmaps exist off-screen and are not affected by occlusion, these image buffers in effect have backing store.

On the other hand, both the displayed and undisplayed image buffers might be implemented using a subset of the on-screen pixels. In this case, unless the contents of an image buffer are saved off-screen, these image buffers in effect do not have backing store.

Output to any image buffer of an unmapped multi-buffered window that does not have backing store is discarded. Output to any image buffer of a mapped multi-buffer window will be performed; however, portions of an image buffer may be occluded or clipped.

When an unmapped multi-buffered window becomes mapped, the contents of any image buffer buffer that did not have backing store is tiled with the background and zero or more exposure events are generated. If no background is defined for the window, then the screen contents are not altered and the contents of any undisplayed image buffers are undefined. If backing store was maintained for an image buffer, then no exposure events are generated.

New Requests

The new request, **CreateImageBuffers**, creates a group of image buffers and associates them with a normal X window:

```
CreateImageBuffers
        w_id          : WINDOW
        buffers       : LISTofBUFFER
        update_action  : {Undefined,Background,Untouched,Copied}
        update_hint    : {Frequent,Intermittent,Static}
        =>
        number_buffers : CARD16

        (Errors: Window, IDChoice, Value)
```

One image buffer will be associated with each ID passed in *buffers*. The first buffer of the list is referred to as buffer[0], the next buffer[1], and so on. Each buffer will have the same visual type and geometry as the window. Buffer[0] will refer to the image buffer already associated with the window ID and its contents will not be modified. The displayed image buffer attribute is set to buffer[0].

Image buffers for the remaining ID's (buffer[1],...) are allocated. If the window is mapped, or if these image buffers have backing store, their contents will be tiled with the window background (if no background is defined, the buffer contents are undefined), and zero or more expose events will be generated for each of these buffers. The contents of an image buffer is undefined when the window is unmapped and the buffer does not have backing store.

If the window already has a group of image buffers associated with it (ie: from a previous **CreateImageBuffers** request) the actions described for **DestroyImageBuffers** are performed first (this will delete the association of the previous buffer ID's and their buffers as well as de-allocate all buffers except for the one already associated with the window ID).

To allow a server implementation to efficiently allocate the buffers, the total number of buffers required and the update action (how they will behave during an update) is specified "up front" in the request. If the server cannot allocate all the buffers requested, the total number of buffers actually allocated will be returned. No **Alloc** errors will be generated – buffer[0] can always be associated with the existing displayed image buffer.

For example, an application that wants to animate a short movie loop may request 64 image buffers. The server may only be able to support 16 image buffers of this type, size, and depth. The application can then decide 16 buffers is sufficient and may truncate the movie loop, or it may decide it really needs 64 and will free the buffers and complain to the user.

One might be tempted to provide a request that inquires whether *n* buffers of a particular type, size, and depth *could* be allocated. But if the query is decoupled from the actual allocation, another client could sneak in and take the buffers before the original client has allocated them.

While any buffer of a group can be selected for display, some applications may display buffers in a predictable order (ie: the movie loop application). The *list order* (buffer[0], buffer[1], ...) will be used as a hint by the server as to which buffer will be displayed next. A client displaying buffers in this order may see a performance improvement.

update_action indicates what should happen to a previously displayed buffer when a different buffer becomes displayed. Possible actions are:

Undefined — The contents of the buffer that was last displayed will become undefined after the update. This is the most efficient action since it allows the implementation to trash the contents of the buffer if it needs to.

Background The contents of the buffer that was last displayed will be set to the background of the window after the update. The background action allows devices to use a fast clear capability during an update.

Untouched The contents of the buffer that was last displayed will be untouched after the update. Used primarily when cycling through images that have already been drawn.

Copied The contents of the buffer that was last displayed will become the same as those that are being displayed after the update. This is useful when incrementally adding to an image.

update_hint indicates how often the client will request a different buffer to be displayed. This hint will allow smart server implementations to choose the most efficient means to support a multi-buffered window based on the current need of the application (dumb implementations may choose to ignore this hint). Possible hints are:

Frequent An animation or movie loop is being attempted and the fastest, most efficient means for multi-buffering should be employed.

Intermittent The displayed image will be changed every so often. This is common for images that are displayed at a rate slower than a second. For example, a clock that is updated only once a minute.

Static The displayed image buffer will not be changed any time soon. Typically set by an application whenever there is a pause in the animation.

To display an image buffer the following request can be used:

```
DisplayImageBuffers
    buffers      : LISTofBUFFER
    min_delay    : CARD16
    max_delay    : CARD16

    (Errors: Buffer, Match)
```

The image buffers listed will become displayed as simultaneously as possible and the update action, bound at **CreateImageBuffers** time, will be performed.

A list of buffers is specified to allow the server to efficiently change the display of more than one window at a time (ie: when a global screen swap method is used). Attempting to simultaneously display multiple image buffers from the same window is an error (**Match**) since it violates the rule that only one image buffer per group can be displayed at a time.

If a specified buffer is already displayed, any delays and update action will still be performed for that buffer. In this instance, only the update action of *Background* (and possibly *Undefined*) will have any affect on the contents of the displayed buffer. These semantics allow an animation application to successfully execute even when there is only a single buffer available for a window.

When a **DisplayImageBuffers** request is made to an unmapped multi-buffered window, the effect of the update action depends on whether the image buffers involved have backing store. When the target of the update action is an image buffer that does not have backing store, output is discarded. When the target image buffer does have backing store, the update is performed; however, when the source of the update is an image buffer does not have backing store (as in the case of update action *Copied*), the contents of target image buffer will become undefined.

min_delay and *max_delay* put a bound on how long the server should wait before processing the display request. For each of the windows to be updated by this request, at least *min_delay* milli-seconds should elapse since the last time any of the windows were updated; conversely, no window should have to wait more than *max_delay* milli-seconds before being updated.

min_delay allows an application to *slow down* an animation or movie loop so that it appears synchronized at a rate the server can support given the current load. For example, a *min_delay* of 100 indicates the server should wait at least 1/10 of a second since the last time any of the windows were updated. A *min_delay* of zero indicates no waiting is necessary.

max_delay can be thought of as an additional delay beyond *min_delay* the server is allowed to wait to facilitate such things as efficient update of multiple windows. If *max_delay* would require an update before *min_delay* is satisfied, then the server should process the display request as soon as the *min_delay* requirement is met. A typical value for *max_delay* is zero.

To implement the above functionality, the time since the last update by a **DisplayImageBuffers** request for each multi-buffered window needs to be saved as state by the server. The server may delay execution of the **DisplayImageBuffers** request until the appropriate time (e.g. by requeuing the request after computing the timeout); however, the entire request must be processed in one operation. Request execution indivisibility must be maintained. When a server is implemented with internal concurrency, the extension must adhere to the same concurrency semantics as those defined for the core protocol.

To explicitly clear a rectangular area of an image buffer to the window background, the following request can be used:

```
ClearImageBufferArea
    buffer      : BUFFER
    x, y        : INT16
    w, h        : CARD16
    exposures   : BOOL

    (Errors: Buffer, Value)
```

Like the X **ClearArea** request, *x* and *y* are relative to the window's origin and specify the upper-left corner of the rectangle. If *width* is zero, it is replaced with the current window width minus *x*. If *height* is zero it is replaced with the current window height minus *y*. If the window has a defined background tile, the rectangle is tiled with a plane mask of all ones, a function of *Copy*, and a subwindow-mode of *ClipByChildren*. If the window has background *None*, the contents of the buffer are not changed. In either case, if *exposures* is true, then one or more exposure events are generated for regions of the rectangle that are either visible or are being retained in backing store.

The group of image buffers allocated by a **CreateImageBuffers** request can be destroyed with the following request:

```
DestroyImageBuffers
    w_id        : WINDOW

    (Error: Window)
```

The association between the buffer ID's and their corresponding image buffers are deleted. Any image buffers not selected for display are de-allocated. If the window is not multi-buffered, the request is ignored.

Attributes

The following attributes will be associated with each window that is multi-buffered:

```
displayed_buffer : CARD16
update_action    : {Undefined,Background,Untouched,Copied}
update_hint      : {Frequent,Intermittent,Static}
window_mode      : {Mono,Stereo}
buffers          : LISTofBUFFER
```

displayed_buffer is set to the *index* of the currently displayed image buffer (for stereo windows, this will be the index of the left buffer – the index of the right buffer is simply *index*+1). *window_mode* indicates whether this window is *Mono* or *Stereo*. The ID for each buffer associated with the window is recorded in the *buffers* list. The above attributes can be queried with the following request:

GetMultiBufferAttributes
    w_id : WINDOW
    =>
    displayed_buffer : CARD16
    update_action : {Undefined,Background,Untouched,Copied}
    update_hint : {Frequent,Intermittent,Static}
    window_mode : {Mono,Stereo}
    buffers : LISTofBUFFER

    (Errors: Window, Access, Value)

If the window is not multi-buffered, a **Access** error will be generated. The only multi-buffer attribute that can be explicitly set is *update_hint*. Rather than have a specific request to set this attribute, a generic set request is provided to allow for future expansion:

SetMultiBufferAttributes
    w_id : WINDOW
    value_mask : BITMASK
    value_list : LISTofVALUE

    (Errors: Window, Match, Value)

If the window is not multi-buffered, a **Match** error will be generated. The following attributes are maintained for each buffer of a multi-buffered window:

    window : WINDOW
    event_mask : SETofEVENT
    index : CARD16
    side : {Mono,Left,Right}

window indicates the window this buffer is associated with. *event_mask* specifies which events, relevant to buffers, will be sent back to the client via the associated buffer ID (initially no events are selected). *index* is the list position (0, 1, ...) of the buffer. *side* indicates whether this buffer is associated with the left side or right side of a stereo window. For non-stereo windows, this attribute will be set to *Mono*. These attributes can be queried with the following request:

GetBufferAttributes
    buffer : BUFFER
    =>
    window : WINDOW
    event_mask : SETofEVENT
    index : CARD16
    side : {Mono,Left,Right}

    (Errors: Buffer, Value)

The only buffer attribute that can be explicitly set is *event_mask*. The only events that are valid are **Expose** and the new **ClobberNotify** and **UpdateNotify** event (see Events section below). A **Value** error will be generated if an event not selectable for a buffer is specified in an event mask. Rather than have a specific request to set this attribute, a generic set request is provided to allow for future expansion:

```
SetBufferAttributes
    buffer        : BUFFER
    value_mask    : BITMASK
    value_list    : LISTofVALUE

    (Errors: Buffer, Value)
```

Clients may want to query the server about basic multi-buffer and stereo capability on a per screen basis. The following request returns a large list of information that would most likely be read once by Xlib for each screen, and used as a data base for other Xlib queries:

```
GetBufferInfo
    root          : WINDOW
    =>
    info          : LISTofSCREEN_INFO
```

Where **SCREEN_INFO** and **BUFFER_INFO** are defined as:

```
    SCREEN_INFO    : [ normal_info : LISTofBUFFER_INFO,
                       stereo_info : LISTofBUFFER_INFO ]

    BUFFER_INFO    : [ visual      : VISUALID,
                       max_buffers : CARD16,
                       depth       : CARD8 ]
```

Information regarding multi-buffering of normal (mono) windows is returned in the *normal_info* list. The *stereo_info* list contains information about stereo windows. If the *stereo_info* list is empty, stereo windows are not supported on the screen. If *max_buffers* is zero, the maximum number of buffers for the depth and visual is a function of the size of the created window and current memory limitations.

The following request returns the major and minor version numbers of this extension:

```
GetBufferVersion
    =>
    major_number   : CARD8
    minor_number   : CARD8
```

The version numbers are an escape hatch in case future revisions of the protocol are necessary. In general, the major version would increment for incompatible changes, and the minor version would increment for small upward compatible changes. Barring changes, the major version will be 1, and the minor version will be 1.

Events

All events normally generated for single-buffered windows are also generated for multi-buffered windows. Most of these events (ie: **ConfigureNotify**) will only be generated for the window and not for each buffer. These events will return a window ID.

Expose events will be generated for both the window and any buffer affected. When this event is generated for a buffer, the same event structure will be used but a buffer ID is returned instead of a window ID. Clients, when processing these events, will know whether an ID returned in an event structure is for a window or a buffer by comparing the returned ID to the ones returned when the window and buffer were created.

GraphicsExposure and **NoExposure** are generated using whatever ID is specified in the graphics operation. If a window ID is specified, the event will contain the window ID. If a buffer ID is specified, the event will contain the buffer ID.

In some implementations, moving a window over a multi-buffered window may cause one or more of its buffers to get overwritten or become unwritable. To allow a client drawing into one of these buffers the opportunity to stop drawing until some portion of the buffer is writable, the following event is added:

```
ClobberNotify
        buffer      : BUFFER
        state       : {Unclobbered,PartiallyClobbered,FullyClobbered}
```

The **ClobberNotify** event is reported to clients selecting *ClobberNotify* on a buffer. When a buffer that was fully or partially clobbered becomes unclobbered, an event with *Unclobbered* is generated. When a buffer that was unclobbered becomes partially clobbered, an event with *PartiallyClobbered* is generated. When a buffer that was unclobbered or partially clobbered becomes fully clobbered, an event with *FullyClobbered* is generated.

ClobberNotify events on a given buffer are generated before any **Expose** events on that buffer, but it is not required that all **ClobberNotify** events on all buffers be generated before all **Expose** events on all buffers.

The ordering of **ClobberNotify** events with respect to **VisibilityNotify** events is not constrained.

If multiple buffers were used as an image FIFO between an image server and the X display server, then the FIFO manager would like to know when a buffer that was previously displayed, has been undisplayed and updated, as the side effect of a **DisplayImageBuffers** request. This allows the FIFO manager to load up a future frame as soon as a buffer becomes available. To support this, the following event is added:

```
UpdateNotify
        buffer      : BUFFER
```

The **UpdateNotify** event is reported to clients selecting *UpdateNotify* on a buffer. Whenever a buffer becomes *updated* (e.g. its update action is performed as part of a **DisplayImageBuffers** request), an **UpdateNotify** event is generated.

Errors

The following error type has been added to support this extension:

Buffer A value for a BUFFER argument does not name a defined BUFFER.

Double-Buffering Normal Windows

The following pseudo-code fragment illustrates how to create and display a double-buffered image:

```
/*
 * Create a normal window
 */
CreateWindow( W, ... )

/*
 * Create two image buffers.  Assume after display, buffer
 * contents become "undefined".  Assume we will "frequently"
 * update the display.  Abort if we don't get two buffers,
 */
n = CreateImageBuffers( W, [B0,B1], Undefined, Frequent )
if (n != 2) <abort>

/*
 * Map window to the screen
 */
MapWindow( W )

/*
 * Draw images using alternate buffers, display every
 * 1/10 of a second.  Note we draw B1 first so it will
 * "pop" on the screen
 */
while animating
{
	<draw picture using B1>
	DisplayImageBuffers( [B1], 100, 0 )

	<draw picture using B0>
	DisplayImageBuffers( [B0], 100, 0 )
}

/*
 * Strip image buffers and leave window with
 * contents of last displayed image buffer.
 */
DestroyImageBuffers( W )
```

Multi-Buffering Normal Windows

Multi-buffered images are also supported by these requests. The following pseudo-code fragment illustrates how to create a a multi-buffered image and cycle through the images to simulate a movie loop:

```
/*
 * Create a normal window
 */
CreateWindow( W, ... )

/*
 * Create 'N' image buffers.  Assume after display, buffer
 * contents are "untouched".  Assume we will "frequently"
 * update the display.  Abort if we don't get all the buffers.
 */
n = CreateImageBuffers( W, [B0,B1,...,B(N-1)], Untouched, Frequent )
if (n != N) <abort>

/*
 * Map window to screen
 */
MapWindow( W )

/*
 * Draw each frame of movie one per buffer
 */
foreach frame
        <draw frame using B(i)>

/*
 * Cycle through frames, one frame every 1/10 of a second.
 */
while animating
{
        foreach frame
                DisplayImageBuffers( [B(i)], 100, 0 )
}
```

Stereo Windows

How stereo windows are supported on a server is implementation dependent. A server may contain specialized hardware that allows left and right images to be toggled at field or frame rates. The stereo affect may only be perceived with the aid of special viewing glasses. The *display* of a stereo picture should be independent of how often the contents of the picture are *updated* by an application. Double and multi-buffering of images should be possible regardless of whether the image is displayed normally or in stereo.

To achieve this goal, a simple extension to normal windows is suggested. Stereo windows are just like normal windows except the displayed image is made up of a left image buffer and a right image buffer. To create a stereo window, a client makes the following request:

```
CreateStereoWindow
    parent         : WINDOW
    w_id           : WINDOW
    left, right    : BUFFER
    depth          : CARD8
    visual         : VISUALID or CopyFromParent
    x, y           : INT16
    width, height  : INT16
    border_width   : INT16
    value_mask     : BITMASK
    value_list     : LISTofVALUE

    (Errors: Alloc, Color, Cursor, Match,
        Pixmap, Value, Window)
```

This request, modeled after the **CreateWindow** request, adds just two new parameters: *left* and *right*. For stereo, it is essential that one can distinguish whether a draw operation is to occur on the left image or right image. While an internal mode could have been added to achieve this, using two buffer ID's allows clients to simultaneously build up the left and right components of a stereo image. These ID's always refer to (are an alias for) the left and right image buffers that are currently *displayed*.

Like normal windows, the window ID is used whenever a window management operation is to be performed. Window queries would also return this window ID (eg: **QueryTree**) as would most events. Like the window ID, the left and right buffer ID's each have their own event mask. They can be set and queried using the **Set/GetBufferAttributes** requests.

Using the window ID of a stereo window in a draw request (eg: **GetImage**) results in pixels that are *undefined*. Possible semantics are that both left and right images get drawn, or just a single side is operated on (existing applications will have to be re-written to explicitly use the left and right buffer ID's in order to successfully create, fetch, and store stereo images).

Having an explicit **CreateStereoWindow** request is helpful in that a server implementation will know from the onset whether a stereo window is desired and can return appropriate status to the client if it cannot support this functionality.

Some hardware may support separate stereo and non-stereo modes, perhaps with different vertical resolutions. For example, the vertical resolution in stereo mode may be half that of non-stereo mode. Selecting one mode or the other must be done through some means outside of this extension (eg: by providing a separate screen for each hardware display mode). The screen attributes (ie: x/y resolution) for a screen that supports normal windows, may differ from a screen that supports stereo windows; however, all windows, regardless of type, displayed on the same screen must have the same screen attributes (ie: pixel aspect ratio).

If a screen that supports stereo windows also supports normal windows, then the images presented to the left and right eyes for normal windows should be the same (ie: have no stereo offset).

Single-Buffered Stereo Windows

The following shows how to create and display a single-buffered stereo image:

```
/*
 * Create the stereo window, map it the screen,
 * and draw the left and right images
 */
CreateStereoWindow( W, L, R, ... )

MapWindow( W )

<draw picture using L,R>
```

Double-Buffering Stereo Windows

Additional image buffers may be added to a stereo window to allow double or multi-buffering of stereo images. Simply use the the **CreateImageBuffers** request. Even numbered buffers (0,2,...) will be left buffers. Odd numbered buffers (1,3,...) will be right buffers. Displayable stereo images are formed by consecutive left/right pairs of image buffers. For example, (buffer[0],buffer[1]) form the first displayable stereo image; (buffer[2],buffer[3]) the next; and so on.

The **CreateImageBuffers** request will only create pairs of left and right image buffers for stereo windows. By always pairing left and right image buffers together, implementations might be able to perform some type of optimization. If an odd number of buffers is specified, a **Value** error is generated. All the rules mentioned at the start of this proposal still apply to the image buffers supported by a stereo window.

To display a image buffer pair of a multi-buffered stereo image, either the left buffer ID or right buffer ID may be specified in a **DisplayImageBuffers** request, but not both.

To double-buffer a stereo window:

```
/*
 * Create stereo window and map it to the screen
 */
CreateStereoWindow( W, L, R, ... )

/*
 * Create two pairs of image buffers.  Assume after display,
 * buffer contents become "undefined".  Assume we will "frequently"
 * update the display.  Abort if we did get all the buffers.
 */
n = CreateImageBuffers( W, [L0,R0,L1,R1], Undefined, Frequently )
if (n != 4) <abort>

/*
 * Map window to the screen
 */
MapWindow( W )

/*
 * Draw images using alternate buffers,
 * display every 1/10 of a second.
 */
while animating
{
        <draw picture using L1,R1>
        DisplayImageBuffers( [L1], 100, 0 )

        <draw picture using L0,R0>
        DisplayImageBuffers( [L0], 100, 0 )
}
```

Multi-Buffering Stereo Windows

To cycle through *N* stereo images:

```
/*
 * Create stereo window
 */
CreateStereoWindow( W, L, R, ... )

/*
 * Create N pairs of image buffers.  Assume after display,
 * buffer contents are "untouched".  Assume we will "frequently"
 * update the display.  Abort if we don't get all the buffers.
 */
n = CreateImageBuffers( W, [L0,R0,...,L(N-1),R(N-1)], Untouched, Frequently )
if (n != N*2) <abort>

/*
 * Map window to screen
 */
MapWindow( W )

/*
 * Draw the left and right halves of each image
 */
foreach stereo image
        <draw picture using L(i),R(i)>

/*
 * Cycle through images every 1/10 of a second
 */
while animating
{
        foreach stereo image
                DisplayImageBuffers( [L(i)], 100, 0 )
}
```

Protocol Encoding

The official name of this extension is "Multi-Buffering". When this string passed to **QueryExtension** the information returned should be interpreted as follows:

major-opcode Specifies the major opcode of this extension. The first byte of each extension request should specify this value.

first-event Specifies the code that will be returned when **ClobberNotify** events are generated.

first-error Specifies the code that will be returned when **Buffer** errors are generated.

The following sections describe the protocol encoding for this extension.

TYPES

BUFFER_INFO

4	VISUALID	visual
2	CARD16	max-buffers
1	CARD8	depth
1		unused

SETofBUFFER_EVENT

	#x00008000	Exposure
	#x02000000	ClobberNotify
	#x04000000	UpdateNotify

EVENTS

ClobberNotify

1	see *first-event*	code
1		unused
2	CARD16	sequence number
4	BUFFER	buffer
1		state
	0 Unclobbered	
	1 PartiallyClobbered	
	2 FullyClobbered	
23		unused

UpdateNotify

1	*first-event*+1	code
1		unused
2	CARD16	sequence number
4	BUFFER	buffer
24		unused

ERRORS

Buffer

1	0	Error
1	see *first-error*	code
2	CARD16	sequence number
4	CARD32	bad resource id
2	CARD16	minor-opcode
1	CARD8	major-opcode

21	unused

REQUESTS

GetBufferVersion

1	see *major-opcode*	major-opcode
1	0	minor-opcode
2	1	request length
→		
1	1	Reply
1		unused
2	CARD16	sequence number
4	0	reply length
1	CARD8	major version number
1	CARD8	minor version number
22		unused

CreateImageBuffers

1	see *major-opcode*	major-opcode
1	1	minor-opcode
2	3+n	request length
4	WINDOW	wid
1		update-action
	0 Undefined	
	1 Background	
	2 Untouched	
	3 Copied	
1		update-hint
	0 Frequent	
	1 Intermittent	
	2 Static	
2		unused
4n	LISTofBUFFER	buffer-list
→		
1	1	Reply
1		unused
2	CARD16	sequence number
4	0	reply length
2	CARD16	number-buffers
22		unused

DestroyImageBuffers

1	see *major-opcode*	major-opcode
1	2	minor-opcode
2	2	request length
4	WINDOW	wid

DisplayImageBuffers

1	see *major-opcode*	major-opcode
1	3	minor-opcode
2	2+n	request length
2	CARD16	min-delay
2	CARD16	max-delay
4n	LISTofBUFFER	buffer-list

SetMultiBufferAttributes

1	see *major-opcode*	major-opcode
1	4	minor-opcode
2	3+n	request length
4	WINDOW	wid
4	BITMASK	value-mask (has n bits set to 1)
	#x00000001 update-hint	
4n	LISTofVALUE	value-list

VALUEs

1		update-hint
	0 Frequent	
	1 Intermittent	
	2 Static	

GetMultiBufferAttributes

1	see *major-opcode*	major-opcode
1	5	minor-opcode
2	2	request length
4	WINDOW	wid
→		
1	1	Reply
1		unused
2	CARD16	sequence number
4	n	reply length
2	CARD16	displayed-buffer
1		update-action
	0 Undefined	
	1 Background	
	2 Untouched	
	3 Copied	
1		update-hint
	0 Frequent	
	1 Intermittent	
	2 Static	
1		window-mode
	0 Mono	
	1 Stereo	
19		unused
4n	LISTofBUFFER	buffer list

SetBufferAttributes

1	see *major-opcode*	major-opcode
1	6	minor-opcode
2	3+n	request length
4	BUFFER	buffer
4	BITMASK	value-mask (has n bits set to 1)
	#x00000001 event-mask	
4n	LISTofVALUE	value-list

VALUEs

4	SETofBUFFER_EVENT	event-mask

GetBufferAttributes

1	see *major-opcode*	major-opcode
1	7	minor-opcode
2	2	request length
4	BUFFER	buffer
→		
1	1	Reply
1		unused
2	CARD16	sequence number
4	0	reply length
4	WINDOW	wid
4	SETofBUFFER_EVENT	event-mask
2	CARD16	index
1		side
	0 Mono	
	1 Left	
	2 Right	
13		unused

GetBufferInfo

1	see *major-opcode*	major-opcode
1	8	minor-opcode
2	2	request length
4	WINDOW	root
→		
1	1	Reply
1		unused
2	CARD16	sequence number
4	2(n+m)	reply length
2	n	number BUFFER_INFO in normal-info
2	m	number BUFFER_INFO in stereo-info
20		unused
8n	LISTofBUFFER_INFO	normal-info
8m	LISTofBUFFER_INFO	stereo-info

CreateStereoWindow

1	see *major-opcode*	major-opcode
1	9	minor-opcode
2	11+n	request length
3		unused
1	CARD8	depth
4	WINDOW	wid
4	WINDOW	parent
4	BUFFER	left
4	BUFFER	right
2	INT16	x
2	INT16	y
2	CARD16	width
2	CARD16	height
2	CARD16	border-width
2		class
	0 CopyFromParent	
	1 InputOutput	
	2 InputOnly	
4	VISUALID	visual
	0 CopyFromParent	
4	BITMASK	value-mask (has n bits set to 1)
	encodings are the same as for CreateWindow	
4n	LISTofVALUE	value-list
	encodings are the same as for CreateWindow	

ClearImageBufferArea

1	see *major-opcode*	major-opcode
1	10	minor-opcode
2	5	request length
4	WINDOW	buffer
2	INT16	x
2	INT16	y
2	CARD16	width
2	CARD16	height
3		unused
1	BOOL	exposures