Double Buffer Extension Library

X Consortium Standard

Ian Elliot, Hewlett-Packard Company
David P. Wiggins
X Consortium, Inc

Double Buffer Extension Library: X Consortium Standard

by Ian Elliot
David P. Wiggins
X Consortium, Inc

X Version 11, Release 7.7

Version 1.0

Table of Contents

Chapter 1. Introduction

The Double Buffer Extension (DBE) provides a standard way to utilize double-buffering within the framework of the X Window System. Double-buffering uses two buffers, called front and back, which hold images. The front buffer is visible to the user; the back buffer is not. Successive frames of an animation are rendered into the back buffer while the previously rendered frame is displayed in the front buffer. When a new frame is ready, the back and front buffers swap roles, making the new frame visible. Ideally, this exchange appears to happen instantaneously to the user and with no visual artifacts. Thus, only completely rendered images are presented to the user, and they remain visible during the entire time it takes to render a new frame. The result is a flicker-free animation.

Chapter 2. Goals

This extension should enable clients to:

- Allocate and deallocate double-buffering for a window.
- Draw to and read from the front and back buffers associated with a window.
- Swap the front and back buffers associated with a window.
- Specify a wide range of actions to be taken when a window is swapped. This includes explicit, simple swap actions (defined below), and more complex actions (for example, clearing ancillary buffers) that can be put together within explicit "begin" and "end" requests (defined below).
- Request that the front and back buffers associated with multiple double-buffered windows be swapped simultaneously.

In addition, the extension should:

- Allow multiple clients to use double-buffering on the same window.
- Support a range of implementation methods that can capitalize on existing hardware features.
- Add no new event types.
- Be reasonably easy to integrate with a variety of direct graphics hardware access (DGHA) architectures.

Chapter 3. Concepts

Normal windows are created using the core CreateWindow request, which allocates a set of window attributes and, for InputOutput windows, a front buffer, into which an image can be drawn. The contents of this buffer will be displayed when the window is visible.

This extension enables applications to use double-buffering with a window. This involves creating a second buffer, called a back buffer, and associating one or more back buffer names (XIDs) with the window for use when referring to (that is, drawing to or reading from) the window's back buffer. The back buffer name is a DRAWABLE of type BACKBUFFER.

DBE provides a relative double-buffering model. One XID, the window, always refers to the front buffer. One or more other XIDs, the back buffer names, always refer to the back buffer. After a buffer swap, the window continues to refer to the (new) front buffer, and the back buffer name continues to refer to the (new) back buffer. Thus, applications and toolkits that want to just render to the back buffer always use the back buffer name for all drawing requests to the window. Portions of an application that want to render to the front buffer always use the window XID for all drawing requests to the window.

Multiple clients and toolkits can all use double-buffering on the same window. DBE does not provide a request for querying whether a window has double-buffering support, and if so, what the back buffer name is. Given the asynchronous nature of the X Window System, this would cause race conditions. Instead, DBE allows multiple back buffer names to exist for the same window; they all refer to the same physical back buffer. The first time a back buffer name is allocated for a window, the window becomes double-buffered and the back buffer name is associated with the window. Subsequently, the window already is a double-buffered window, and nothing about the window changes when a new back buffer name is allocated, except that the new back buffer name is associated with the window. The window remains double-buffered until either the window is destroyed or until all of the back buffer names for the window are deallocated.

In general, both the front and back buffers are treated the same. particular, here are some important characteristics:

- Only one buffer per window can be visible at a time (the front buffer).
- Both buffers associated with a window have the same visual type, depth, width, height, and shape as the window.
- Both buffers associated with a window are "visible" (or "obscured") in the same way. When an Expose event is generated for a window, both buffers should be considered to be damaged in the exposed area. Damage that occurs to either buffer will result in an Expose event on the window. When a double-buffered window is exposed, both buffers are tiled with the window background, exactly as stated by the core protocol. Even though the back buffer is not visible, terms such as obscure apply to the back buffer as well as to the front buffer.
- It is acceptable at any time to pass a BACKBUFFER in any request, notably any core or extension drawing request, that expects a DRAWABLE. This enables an application to draw directly into BACKBUFFERs in the same fashion as it would draw into any other DRAWABLE.

- It is an error (Window) to pass a BACKBUFFER in a core request that expects a Window.
- A BACKBUFFER will never be sent by core X in a reply, event, or error where a Window is specified.
- If core X11 backing-store and save-under applies to a double-buffered window, it applies to both buffers equally.
- If the core ClearArea request is executed on a double-buffered window, the same area in both the front and back buffers is cleared.

The effect of passing a window to a request that accepts a `DRAWABLE` is unchanged by this extension. The window and front buffer are synonomous with each other. This includes obeying the `GetImage` semantics and the subwindow-mode semantics if a core graphics context is involved. Regardless of whether the window was explicitly passed in a `GetImage` request, or implicitly referenced (that is, one of the window's ancestors was passed in the request), the front (that is, visible) buffer is always referenced. Thus, DBE-naive screen dump clients will always get the front buffer. `GetImage` on a back buffer returns undefined image contents for any obscured regions of the back buffer that fall within the image.

Drawing to a back buffer always uses the clip region that would be used to draw to the front buffer with a GC subwindow-mode of `ClipByChildren`. If an ancestor of a double-buffered window is drawn to with a core GC having a subwindow-mode of IncludeInferiors, the effect on the double-buffered window's back buffer depends on the depth of the double-buffered window and the ancestor. If the depths are the same, the contents of the back buffer of the double-buffered window are not changed. If the depths are different, the contents of the back buffer of the double-buffered window are undefined for the pixels that the `IncludeInferiors` drawing touched.

DBE adds no new events. DBE does not extend the semantics of any existing events with the exception of adding a new DRAWABLE type called BACKBUFFER. If events, replies, or errors that contain a DRAWABLE (for example, `GraphicsExpose`) are generated in response to a request, the DRAWABLE returned will be the one specified in the request.

DBE advertises which visuals support double-buffering.

DBE does not include any timing or synchronization facilities. Applications that need such facilities (for example, to maintain a constant frame rate) should investigate the Synchronization Extension, an X Consortium standard.

Window Management Operations

The basic philosophy of DBE is that both buffers are treated the same by core X window management operations.

When the core `DestroyWindow` is executed on a double-buffered window, both buffers associated with the window are destroyed, and all back buffer names associated with the window are freed.

If the core `ConfigureWindow` request changes the size of a window, both buffers assume the new size. If the window's size increases, the effect on the buffers depends on whether the implementation honors bit gravity for buffers. If bit gravity is implemented, then the contents of both buffers are moved in accordance with the window’s bit gravity (see the core `ConfigureWindow` request), and the remaining areas are tiled with the window background. If bit gravity is not implemented, then the entire unobscured region of both buffers is tiled with the window background. In either case, `Expose` events are generated for the region that is tiled with the window background. If the core GetGeometry request is executed on a BACKBUFFER, the returned x, y, and border-width will be zero.

If the Shape extension `ShapeRectangles`, `ShapeMask`, `ShapeCombine`, or `ShapeOffset` request is executed on a double-buffered window, both buffers are reshaped to match the new window shape. The region difference is the following:

D = newshape − oldshape

It is tiled with the window background in both buffers, and `Expose` events are generated for D.

Complex Swap Actions

DBE has no explicit knowledge of ancillary buffers (for example, depth buffers or alpha buffers), and only has a limited set of defined swap actions. Some applications may need a richer set of swap actions than DBE provides. Some DBE implementations have knowledge of ancillary buffers, and/or can provide a rich set of swap actions. Instead of continually extending DBE to increase its set of swap actions, DBE provides a flexible "idiom" mechanism. If an application’s needs are served by the defined swap actions, it should use them; otherwise, it should use the following method of expressing a complex swap action as an idiom. Following this policy will ensure the best possible performance across a wide variety of implementations.

As suggested by the term "idiom," a complex swap action should be expressed as a group/series of requests. Taken together, this group of requests may be combined into an atomic operation by the implementation, in order to maximize performance. The set of idioms actually recognized for optimization is implementation dependent. To help with idiom expression and interpretation, an idiom must be surrounded by two protocol requests: `DBEBeginIdiom` and `DBEEndIdiom`. Unless this begin-end pair surrounds the idiom, it may not be recognized by a given implementation, and performance will suffer.

For example, if an application wants to swap buffers for two windows, and use core X to clear only certain planes of the back buffers, the application would issue the following protocol requests as a group, and in the following order:

- `DBEBeginIdiom` request.
- `DBESwapBuffers` request with XIDs for two windows, each of which uses a swap action of Untouched.
- Core X `PolyFillRectangle` request to the back buffer of one window.
- Core X `PolyFillRectangle` request to the back buffer of the other window.
- `DBEEndIdiom` request.

The `DBEBeginIdiom` and `DBEEndIdiom` requests do not perform any actions themselves. They are treated as markers by implementations that can combine certain groups/series of requests as idioms, and are ignored by other implementations or for nonrecognized groups/series of requests. If these requests are sent out of order, or are mismatched, no errors are sent, and the requests are executed as usual, though performance may suffer.

An idiom need not include a `DBESwapBuffers` request. For example, if a swap action of `Copied` is desired, but only some of the planes should be copied, a core X `CopyArea` request may be used instead of `DBESwapBuffers`. If `DBESwapBuffers` is included in an idiom, it should immediately follow the `DBEBeginIdiom` request. Also, when the `DBESwapBuffers` is included in an idiom, that request's swap action will still be valid, and if the swap action might overlap with another request, then the final result of the idiom must be as if the separate requests were executed serially. For example, if the specified swap action is `Untouched`, and if a `PolyFillRectangle` using a client clip rectangle is done to the window's back buffer after the `DBESwapBuffers` request, then the contents of the new back buffer (after the idiom) will be the same as if the idiom was not recognized by the implementation.

It is highly recommended that Application Programming Interface (API) providers define, and application developers use, "convenience" functions that allow client applications to call one procedure that encapsulates common idioms. These functions will generate the `DBEBeginIdiom` request, the idiom requests, and `DBEEndIdiom` request. Usage of these functions will ensure best possible performance across a wide variety of implementations.

Chapter 4. C Language Binding

All identifier The header for this extension is <X11/extensions/Xdbe.h>. names provided by this header begin with Xdbe.

Types

The type `XdbeBackBuffer` is a `Drawable`.

The type `XdbeSwapAction` can be one of the constants `XdbeUndefined`, `XdbeBackground`, `XdbeUntouched`, or `XdbeCopied`.

C Functions

The C functions provide direct access to the protocol and add no additional semantics. For complete details on the effects of these functions, refer to the appropriate protocol request, which can be derived by replacing Xdbe at the start of the function name with DBE. All functions that have return type `Status` will return nonzero for success and zero for failure.

`Status` **`XdbeQueryExtension`**`(` *`*dpy`*`,` *`*major_version_return`*`,` *`*minor_version_return`*`);`

`XdbeQueryExtension` sets major version return and minor version return to the major and minor DBE protocol version supported by the server. If the DBE library is compatible with the version returned by the server, it returns nonzero. If dpy does not support the DBE extension, or if there was an error during communication with the server, or if the server and library protocol versions are incompatible, it returns zero. No other Xdbe functions may be called before this function. If a client violates this rule, the effects of all subsequent Xdbe calls that it makes are undefined.

`XdbeScreenVisualInfo *`**`XdbeGetVisualInfo`**`(` *`*dpy`*`,` *`*screen_specifiers`*`,` *`*num_screens`*`);`

`XdbeGetVisualInfo` returns information about which visuals support double buffering. The argument num_screens specifies how many elements there are in the screen_specifiers list. Each drawable in screen_specifiers designates a screen for which the supported visuals are being requested. If num_screens is zero, information for all screens is requested. In this case, upon return from this function, num_screens will be set to the number of screens that were found. If an error occurs, this function returns NULL; otherwise, it returns a pointer to a list of `XdbeScreenVisualInfo` structures of length num_screens. The nth element in the returned list corresponds to the nth drawable in the screen_specifiers list, unless element in the returned list corresponds to the nth screen of the server, starting with screen zero.

The XdbeScreenVisualInfo structure has the following fields:

int	count	number of items in visinfo
XdbeVisualInfo*	visinfo	list of visuals and depths for this screen

The `XdbeVisualInfo` structure has the following fields:

VisualID	visual	one visual ID that supports double-buffering
int	depth	depth of visual in bits
int	perflevel	performance level of visual

`void XdbeFreeVisualInfo` **`XdbeGetVisualInfo`**`( *visual_info);`

`XdbeFreeVisualInfo` frees the list of `XdbeScreenVisualInfo` returned by `XdbeGetVisualInfo`.

`XdbeBackBuffer` **`XdbeAllocateBackBufferName`**`( *dpy, *window, swap_action);`

`XdbeAllocateBackBufferName` returns a drawable ID used to refer to the back buffer of the specified window. The swap_action is a hint to indicate the swap_action that will likely be used in subsequent calls to `XdbeSwapBuffers`. The actual swap_action used in calls to `XdbeSwapBuffers` does not have to be the same as the swap_action passed to this function, though clients are encouraged to provide accurate information whenever possible.

`Status` **`XdbeDeallocateBackBufferName`**`( *dpy, buffer);`

`XdbeDeallocateBackBufferName` frees the specified drawable ID, buffer, that was obtained via `XdbeAllocateBackBufferName`. The buffer must be a valid name for the back buffer of a window, or an `XdbeBadBuffer` error results.

`Status` **`XdbeSwapBuffers`**`( *dpy, *swap_info, num_windows);`

`XdbeSwapBuffers` swaps the front and back buffers for a list of windows. The argument num_windows specifies how many windows are to have their buffers swapped; it is the number of elements in the swap_info array. The argument swap_info specifies the information needed per window to do the swap.

The XdbeSwapInfo structure has the following fields:

Window	swap_window	window for which to swap buffers
XdbeSwapAction	swap_action	swap action to use for this swap window

`Status` **`XdbeBeginIdiom`**`( *dpy);`

`XdbeBeginIdiom` marks the beginning of an idiom sequence. See the section called “Complex Swap Actions” for a complete discussion of idioms.

`Status` **`XdbeEndIdiom`**`( *dpy);`

`XdbeEndIdiom` marks the end of an idiom sequence.

`XdbeBackBufferAttributes *`**`XdbeGetBackBufferAttributes`**`( *dpy, buffer);`

`XdbeGetBackBufferAttributes` returns the attributes associated with the specified buffer.

The XdbeBackBufferAttributes structure has the following fields:

Window	window	window that buffer belongs to

If buffer is not a valid `XdbeBackBuffer`, window is set to None.

The returned `XdbeBackBufferAttributes` structure can be freed with the Xlib function `XFree`.

Errors

The `XdbeBufferError` structure has the following fields:

int	type	
Display *	display	Display the event was read from
XdbeBackBuffer	buffer	resource id
unsigned long	serial	serial number of failed request
unsigned char	error code	error base + `XdbeBadBuffer`
unsigned char	request code	Major op-code of failed request
unsigned char	minor code	Minor op-code of failed request

Chapter 5. Acknowledgements

We wish to thank the following individuals who have contributed their time and talent toward shaping the DBE specification:

T. Alex Chen, IBM; Peter Daifuku, Silicon Graphics, Inc.; Ian Elliott, Hewlett-Packard Company; Stephen Gildea, X Consortium, Inc.; Jim Graham, Sun; Larry Hare, AGE Logic; Jay Hersh, X Consortium, Inc.; Daryl Huff, Sun; Deron Dann Johnson, Sun; Louis Khouw, Sun; Mark Kilgard, Silicon Graphics, Inc.; Rob Lembree, Digital Equipment Corporation; Alan Ricker, Metheus; Michael Rosenblum, Digital Equipment Corporation; Bob Scheifler, X Consortium, Inc.; Larry Seiler, Digital Equipment Corporation; Jeanne Sparlin Smith, IBM; Jeff Stevenson, Hewlett-Packard Company; Walter Strand, Metheus; Ken Tidwell, Hewlett-Packard Company; and David P. Wiggins, X Consortium, Inc.

Mark provided the impetus to start the DBE project. Ian wrote the first draft of the specification. David served as architect.

Chapter 6. References

Jeffrey Friedberg, Larry Seiler, and Jeff Vroom, "Multi-buffering Extension Specification Version 3.3."

Tim Glauert, Dave Carver, Jim Gettys, and David P. Wiggins, "X Synchronization Extension Version 3.0."