NAME

DBE - Double Buffer Extension

SYNOPSIS

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, with no visual artifacts. Thus, only completely rendered images are presented to the user, and remain visible during the entire time it takes to render a new frame. The result is a flicker-free animation.

DESCRIPTION

Concepts

Normal windows are created using **XCreateWindow()** or **XCreateSimpleWindow(),** which allocate 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 (i.e., drawing to or reading from) the window's back buffer. The back buffer name is a drawable of type *XdbeBackBuffer.*

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 ae treated the same. In 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, this event is considered to apply to both buffers equally. When a double-buffered window is exposed, both buffers are tiled with the window background. 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 an *XdbeBackBuffer* in any function that expects a drawable. This enables an application to draw directly into *XdbeBackBuffer* in the same fashion as it would draw into any other drawable.

It is an error (Window) to pass an *XdbeBackBuffer* in a function that expects a Window.

An *XdbeBackBuffer* will never be sent in a reply, event, or error where a Window is specified.

If backing-store and save-under applies to a double-buffered window, it applies to both buffers equally.

If the **XClearArea()** or **XClearWindow()** function 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 function that accepts a drawable is unchanged by this extension. The window and front buffer are synonymous with each other. This includes obeying the **XGetImage()** and **XGetSubImage()** semantics and the subwindow-mode semantics if a graphics context is involved. Regardless of whether the window was explicitly passed in an **XGetImage()** or **XGetSubImage()** call, or implicitly referenced (i.e., one of the window's ancestors was passed in the function), the front (i.e. visible) buffer is always referenced. Thus, DBE-naive screen dump clients will always get the front buffer. **XGetImage()** and **XGetSubImage()** on a back buffer return 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 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 *XdbeBackBuffer.*

If events, replies, or errors that contain a drawable (e.g., 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 (e.g., 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 X window management operations.

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

If the size of a double-buffered window changes, 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, 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 **XGetGeometry()** function is executed on an *XdbeBackBuffer,* 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 D = new shape - old shape 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 (e.g. 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 function calls: **XdbeBeginIdiom()** and **XdbeEndIdiom().** 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 X to clear only certain planes of the back buffers, the application would make the following calls as a group, and in the following order:

XdbeBeginIdiom().

XdbeSwapBuffers() with XIDs for two windows, each of which uses a swap action of Untouched.

XFillRectangle() to the back buffer of one window.

XFillRectangle() to the back buffer of the other window.

XdbeEndIdiom().

The **XdbeBeginIdiom()** and **XdbeEndIdiom()** functions 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 non-recognized groups/series of requests. If these function calls are made out of order, or are mismatched, no errors are sent, and the functions are executed as usual, though performance may suffer.

XdbeSwapBuffers() need not be included in an idiom. For example, if a swap action of Copied is desired, but only some of the planes should be copied, **XCopyArea()** may be used instead of **XdbeSwapBuffers().** If **XdbeSwapBuffers()** is included in an idiom, it should immediately follow the **XdbeBeginIdiom()** call. Also, when the **XdbeSwapBuffers()** 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 **XFillRectangle()** using a client clip rectangle is done to the window's back buffer after the **XdbeSwapBuffers()** call, 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 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 **XdbeBeginIdiom(),** idiom, and **XdbeEndIdiom()** calls. Usage of these functions will ensure best possible performance across a wide variety of implementations.

SEE ALSO

XdbeAllocateBackBufferName(), XdbeBeginIdiom(), XdbeDeallocateBackBufferName(), XdbeEndIdiom(), XdbeFreeVisualInfo(), XdbeGetBackBufferAttributes(), XdbeGetVisualInfo(), XdbeQueryExtension(), XdbeSwapBuffers().