10 The Game Kit

Introduction 3

BWindowScreen 5

Overview 5

Hook Functions 6

Constructor and Destructor 6

Member Functions 7

10 The Game Kit

The Game Kit is a collection of software that's especially useful for developing games. Currently, the collection consists of just one class, BWindowScreen, but it will grow in future releases. A BWindowScreen object gives an application direct access to the screen—that is, direct access to the driver for the graphics card so it can bypass the Application Server, customize the card for the game, call graphics functions the driver implements, and draw directly into the frame buffer.

Although designed with games in mind, nothing in the Game Kit is restricted to game applications. Other kinds of applications can profitably take advantage of this Kit.

BWindowScreen

Derived from: public BWindow

Declared in: <game/WindowScreen.h>

Overview

A BWindowScreen object has the dual nature its name implies: It's both a window and an object that provides direct access to the screen, bypassing the window system. When a BWindowScreen object becomes the active window—which it does when constructed—it establishes a direct connection to the graphics card driver for the screen, independent of the Application Server. This permits the application to set up a game-specific graphics environment on the card, call driver-implemented drawing functions, and directly manipulate the frame buffer.

The Application Server's graphic operations are suspended until the BWindowScreen object gives up active window status. While it's active, normal drawing operations have no effect; application code can move windows and call upon BView objects to draw, but nothing is rendered on-screen. Only the BWindowScreen object can provide access to the frame buffer.

By constructing a BWindowScreen object, an application takes over the whole screen. The object's frame rectangle is as large as the screen, so that the Application Server will automatically erase every pixel when the window becomes active and refresh everything when it ceases to be the active window. While the BWindowScreen is active, nothing except what the application draws will be visible to the user—no dock and no other windows. The entire screen is the application's canvas.

A BWindowScreen object remains a window while it has control of the screen; it stays attached to the Application Server and its message loop continues to function. It gets messages reporting the user's actions on the keyboard and mouse, just like any other active window. Because it covers the whole screen, it's notified of all mouse and keyboard events. < Messages that report mouse events are currently unreliable; the cursor is reported at a static location, inhibiting mouse-moved messages and making mouse-down and mouse-up messages inaccurate. >

This class respects workspaces. A BWindowScreen object releases its grip on the screen when the user turns to another workspace and reestablishes its control when the user returns to the workspace and it again becomes the active window. Short of quitting the application, changing workspaces is the only way that the user can move in and out of the game. Because other windows and applications aren't visible while the BWindowScreen

object is connected to the screen, the usual methods of selecting another application (picking it from the application list or clicking in one of its windows) are not available.

Hook Functions

ScreenConnected() Can be implemented to do whatever is necessary when the BWindowScreen object obtains direct access to the frame buffer for the screen, and when it loses that access.

Constructor and Destructor

BWindowScreen()

BWindowScreen(const char ***title*, ulong *space***)**

Initializes the BWindowScreen object by assigning the window a *title* and specifying a *space* configuration for the screen. The window won't have a visible border or a tab in which to display the title to the user. However, others—such as the Workspaces application—can use the title to identify the window.

The window is constructed to fill the screen; its frame rectangle contains every screen pixel when the screen is configured according to the *space* argument. That argument describes the pixel dimensions and bits-per-pixel depth of the screen that the BWindowScreen object should establish when it first obtains direct access to the frame buffer. It should be one of the following constants:

B_8_BIT_640x480	**B_16_BIT_640x480**	**B_32_BIT_640x480**
B_8_BIT_800x600	**B_16_BIT_800x600**	**B_32_BIT_800x600**
B_8_BIT_1024x768	**B_16_BIT_1024x768**	**B_32_BIT_1024x768**
B_8_BIT_1152x900	**B_16_BIT_1152x900**	**B_32_BIT_1152x900**
B_8_BIT_1280x1024	**B_16_BIT_1280x1024**	**B_32_BIT_1280x1024**
B_8_BIT_1600x1200	**B_16_BIT_1600x1200**	**B_32_BIT_1600x1200**

These are the same constants that can be passed to **set_screen_space()**, the Interface Kit function that preference applications call to configure the screen. < Sixteen-bit depths are not currently supported. >

The constructor assigns the window to the active workspace (**B_CURRENT_WORKSPACE**) and calls **Show()** to immediately place it on-screen, make it the active window, and have it take direct charge of the workspace screen. It fails if another BWindowScreen object in any application already has established a direct screen connection for the same workspace.

To be sure there wasn't an error in constructing the object, call the **Error()** function. If there is an error, it's likely to occur in this constructor, not the inherited BWindow constructor.

Since the object will probably have spawned a thread and will be running a message loop, you'll need to instruct it to quit. For example:

```
MyWindowScreen *screen =
                new MyWindowScreen("Glacier", B_8_BIT_1024x768);
if ( Error() != B_NO_ERROR )
    screen->PostMessage(B_QUIT_REQUESTED);
```

If all goes well, **Error()** will return **B_NO_ERROR** (0).

See also: **Error()**, **get_screen_info()** in the Interface Kit

~BWindowScreen()

virtual **~BWindowScreen(**void**)**

Closes the clone of the graphics card driver (through which the BWindowScreen object established its connection to the screen), unloads it from the application, and cleans up after it.

Member Functions

CanControlFrameBuffer()

bool **CanControlFrameBuffer(**void**)**

Returns **TRUE** if the graphics card driver permits applications to control the configuration of the frame buffer, and **FALSE** if not. Control is exercised through four functions:

ProposeFrameBuffer()
SetFrameBuffer()
SetDisplayArea()
MoveDisplayArea()

A return of **TRUE** means that all of these functions can communicate with the graphics card driver and at least the first two of them will do something useful. A return of **FALSE** means that none of them will work.

See also: **ProposeFrameBuffer()**, **SetDisplayArea()**

CardHookAt()

inline graphics_card_hook **CardHookAt(**long *index***)**

Returns a pointer to the function implemented by the graphics card driver and located at *index* in its list of hook functions, or **NULL** if the graphics card driver doesn't implement a function at that index or the index is out-of-range.

The hook functions are documented under "Hook Functions" on page 87 in *The Device Kit* chapter and are summarized briefly below. Currently, 12 functions are defined, from index 0 through index 11. However, the first three, which set and manipulate the cursor, are unavailable through the Game Kit; if you pass an index of 0, 1, or 2 to **CardHookAt()**, it will return **NULL**.

The other hook functions are summarized by index in the chart below:

Index:	What the function does:	What arguments it takes:
3	Draws a line (8-bit depth)	(long *startX*, long *startY*, long *endX*, long *endY*, uchar *colorIndex*, bool *clipToRect*, short *clipLeft*, short *clipTop*, short *clipRight*, short *clipBottom*)
4	Draws a line (32-bit depth)	(long *startX*, long *startY*, long *endX*, long *endY*, ulong *color*, bool *clipToRect*, short *clipLeft*, short *clipTop*, short *clipRight*, short *clipBottom*)
5	Draws a rectangle (8-bit depth)	(long *left*, long *top*, long *right*, long *bottom*, uchar *colorIndex*)
6	Draws a rectangle (32-bit depth)	(long *left*, long *top*, long *right*, long *bottom*, ulong *color*)
7	Copies pixel data (blits)	(long *sourceX*, long *sourceY*, long *destinationX*, long *destinationY*, long *width*, long *height*)
8	Draws a line array (8-bit depth)	(indexed_color_line **array*, long *numItems*, bool *clipToRect*, short *clipLeft*, short *clipTop*, short *clipRight*, short *clipBottom*)
9	Draws a line array (32-bit depth)	(rgb_color_line **array*, long *numItems*, bool *clipToRect*, short *clipLeft*, short *clipTop*, short *clipRight*, short *clipBottom*)
10	Waits for drawing to finish	*none*
11	Inverts the colors in a rectangle	(long *left*, long *top*, long *right*, long *bottom*)

You must ensure that all coordinate values passed to these functions lie somewhere in the frame buffer; the function will not do the checking for you. (An *x* coordinate value is a left-to-right index to a pixel column in the frame buffer and a *y* coordinate value is a top-to-bottom index to a pixel row.)

For example, before calling the function at index 7, which blits a rectangle *width* pixels wide and *height* pixels high from (*sourceX*, *sourceY*) to (*destinationX*, *destinationY*), you

should be sure that the source and destination rectangles both lie entirely within the area defined by the frame buffer.

CardInfo()

inline graphics_card_info ***CardInfo(**void**)**

Returns a description of the current configuration of the graphics card, as kept by the driver for the card. The returned **graphics_card_info** structure is defined in **device/GraphicsCard.h** and contains the following fields:

short **version**	The version of the Be architecture for graphics cards; the current version is 2.
short **id**	An identifier for the driver.
void ***frame_buffer**	A pointer to the first byte of the frame buffer. Applications can use this pointer to draw directly to the frame buffer.
char **rgba_order**[4]	The characters 'r' (red), 'g' (green), 'b' (blue), and 'a' (alpha) ordered as those components are intermeshed for each pixel in the frame buffer. This field is valid only for screen depths of 32 bits per pixel.
short **flags**	A mask formed from three flags (**B_CRT_CONTROL**, **B_FRAME_BUFFER_CONTROL**, and **B_GAMMA_CONTROL**) that describe the ability of the graphics card driver to perform particular tasks. **B_FRAME_BUFFER_CONTROL** matches the **CanControlFrameBuffer()** function; the other two flags aren't important to the control exercised through the BWindowScreen object.
short **bits_per_pixel**	The depth of the screen in bits per pixel.
long **bytes_per_row**	The offset, in bytes, between two adjacent rows of pixel data in the frame buffer (the number of bytes assigned to each row).
short **width**	The width of the frame buffer in pixels (the number of pixel columns it defines).
short **height**	The height of the frame buffer measured in lines of pixels (the number of pixel rows the frame buffer defines).

The returned structure belongs to the BWindowScreen object and is provided for information only; you should not modify any of its fields.

See **"B_GET_GRAPHICS_CARD_INFO"** on page 80 in *The Device Kit* chapter for a fuller description of the **graphics_card_info** structure.

ColorList() *see* SetColorList()

Disconnect() *see* Quit()

Error()

long **Error(**void**)**

Returns the error code for the last BWindowScreen operation—including constructing the BWindowScreen object. The code will be **B_NO_ERROR** if the operation was successful and some other value (currently just **B_ERROR**) if not. Most functions also return the error code directly.

See also: the BWindowScreen constructor

FrameBufferInfo()

inline frame_buffer_info ***FrameBufferInfo(**void**)**

Returns a pointer to the **frame_buffer_info** structure that holds the application's current conception of the frame buffer. This may or may not capture the actual configuration of the frame buffer. If the application has proposed a configuration (**ProposeFrameBuffer()**) but not yet set it (**SetFrameBuffer()**), the returned structure will reflect the proposal, not the reality.

The **frame_buffer_info** structure is defined in **device/GraphicsCard.h** and contains the following fields:

short **bits_per_pixel**	The depth of the frame buffer; the number of bits assigned to a pixel.
short **bytes_per_row**	The number of bytes required to store one row of pixel data in the frame buffer.
short **width**	The width of the frame buffer in pixels (the number of columns).
short **height**	The height of the frame buffer in pixels (the number of rows).
short **display_width**	The width of the screen display in pixels (the number of pixel columns shown on-screen).
short **display_height**	The height of the screen display in pixels (the number of pixel rows shown on-screen).

short **display_x**	The horizontal position of the left top pixel shown on-screen, where 0 is the leftmost column of pixels in the frame buffer.
short **display_y**	The vertical position of the left top pixel shown on-screen, where 0 is the topmost row of pixels in the frame buffer.

Note that the first four fields of this structure are identical to the last four of **graphics_card_info**.

The returned structure belongs to the BWindowScreen object. Call functions like **ProposeFrameBuffer()** to modify its fields; don't modify them directly.

See "Control Operations for Manipulating the Frame Buffer" on page 85 in *The Device Kit* chapter for a fuller description of the **frame_buffer_info** structure.

See also: **ProposeFrameBuffer()**, **SetDisplayArea()**, **CardInfo()**

IOBase()

inline void ***IOBase(**void**)**

Returns a pointer to the base address for the input/output registers on the graphics card. Registers are addressed by 16-bit offsets from this base address.

MoveDisplayArea() *see* SetDisplayArea()

ProposeFrameBuffer(), SetFrameBuffer()

long **ProposeFrameBuffer(**short *depth*, short *width*,
short **height*, short **bytesPerRow* = NULL**)**

long **SetFrameBuffer(**short *height***)**

long **SetFrameBuffer(**short *height*, short *displayWidth*, short *displayHeight*,
short *displayX* = 0, short *displayY* = 0**)**

These functions, in a two-step process, configure the frame buffer on the graphics card. They work only if the driver for the graphics card allows custom configurations (as reported by **CanControlFrameBuffer()**).

The first function proposes a possible configuration for the frame buffer and in return receives information on how accommodating the graphics card driver will be. Based on that information, the second function can set the dimensions and depth of the frame buffer.

ProposeFrameBuffer() proposes two parameters for the frame buffer—*depth*, the number of bits assigned to each pixel, and *width*, the number of pixels in one row of data (the total number of pixel columns). If the driver can accommodate those two parameters, this

function returns **B_NO_ERROR** and reports on the two other parameters that define the configuration. In the integer referred to by *height*, it writes the maximum number of pixel rows (the maximum number of pixels in one column) that the graphics card can provide at the proposed depth and width. If a *bytesPerRow* argument is provided, it reports the minimum number of bytes the driver must dedicate to one row of pixel data at the proposed width.

If the driver can't accommodate the proposed *depth* and *width*, **ProposeFrameBuffer()** returns **B_ERROR** and puts no useful information in the integers that *height* and *bytesPerRow* refer to.

An application can call **ProposeFrameBuffer()** any number of times to test possible configurations. This function doesn't make any changes in the frame buffer (though it does set values in the structure that **FrameBufferInfo()** returns). When the application finds a configuration that it wants to use, it can call **SetFrameBuffer()** to set the desired *height* of the frame buffer—the actual number of rows—plus the most recently proposed *depth* and *width*. The height set should not be greater than the maximum height reported by **ProposeFrameBuffer()**.

By default, **SetFrameBuffer()** sets the display area—the part of the frame buffer that's mapped to the screen—to be the same size as the frame buffer. In other words, it maps the entire frame buffer to the screen.

If you want a display area that's smaller than the frame buffer, you must set it explicitly by passing **SetFrameBuffer()** a *displayWidth* and *displayHeight* in pixels. The left top corner of the display area will be located at pixel (0, 0), the first pixel in the first row of data. If you want to locate it somewhere else, you must pass this function different *displayX* and *displayY* values.

The display area can subsequently be moved and resized through the **MoveDisplayArea()** and **SetDisplayArea()** functions.

To locate the display area, all these functions assume a coordinate system in which an *x* coordinate value is a left-to-right index to a pixel column in the frame buffer and a *y* coordinate value is a top-to-bottom index to a pixel row.

See also: **SetDisplayArea()**

Quit(), Disconnect()

virtual void **Quit(**void**)**

void **Disconnect(**void**)**

Quit() overrides the BWindow version of the same function to force the BWindowScreen object to disconnect itself from the screen, so that it doesn't quit while in control of the frame buffer.

Disconnect() similarly causes the BWindowScreen object to give up its authority over the graphics card driver, allowing the Application Server to reassert control. It doesn't force the application to quit.

Although **Quit()** disconnects the object before quitting, this may not be soon enough for your application. For example, if you need to destroy some drawing threads before the BWindowScreen object is itself destroyed, you should get rid of them after the screen connection is severed. You can force the object to disconnect itself by calling **Disconnect()**. For example:

```
void MyWindowScreen::Quit()
{
    Disconnect();
    kill_thread(drawing_thread_a);
    kill_thread(drawing_thread_b);
    BWindowScreen::Quit();
}
```

Before breaking the screen connection, both **Quit()** and **Disconnect()** cause the BWindowScreen object to receive a **ScreenConnected()** notification with a flag of **FALSE**. Neither function returns until **ScreenConnected()** returns and the connection is broken.

See also: **ScreenConnected()**

ScreenChanged()

virtual void **ScreenChanged(**BRect *frame*, color_space *mode***)**

Overrides the BWindow version of **ScreenChanged()** so that it does nothing. This function is called automatically when the screen configuration changes. It's not one that you should call (or override) in application code.

See also: **BWindow::ScreenChanged()**

ScreenConnected()

virtual void **ScreenConnected(**bool *connected***)**

Implemented by derived classes to take action when the application gains direct access to the screen and when it's about to lose that access.

This function is called with the *connected* flag set to **TRUE** immediately after the BWindowScreen object becomes the active window and establishes a direct connection to the graphics card driver for the screen. At that time, the Application Server's connection to the screen is suspended; drawing can only be accomplished through the screen access that the BWindowScreen object provides.

It's called with a flag of **FALSE** just before the BWindowScreen object is scheduled to lose its control over the screen and the Application Server's control is reasserted. The BWindowScreen's connection to the screen is not broken until **ScreenConnected()**

returns. It should delay returning until the application has finished all current drawing and no longer needs direct screen access.

Note that whenever **ScreenConnected()** is called, the BWindowScreen object is guaranteed to be connected to the screen; if *connected* is **TRUE**, it just became connected, if *connected* is **FALSE**, it's still connected but will be disconnected when the function returns.

Derived classes typically use this function to regulate access to the screen. For example, they may acquire a semaphore when the *connected* flag is **FALSE**, so that application threads won't attempt direct drawing when the connection isn't in place, and release the semaphore for drawing threads to acquire when the flag is **TRUE**. For example:

```
void MyWindowScreen::ScreenConnected(bool connected)
{
    if ( connected == FALSE )
        acquire_sem(directDrawingSemaphore);
    else
        release_sem(directDrawingSemaphore);
}
```

SetColorList(), ColorList()

void **SetColorList(**rgb_color **colors*, long *first* = 0, long *last* = 255**)**

inline rgb_color ***ColorList(**void**)**

These functions set and return the list of 256 colors that can be displayed when the frame buffer has a depth of 8 bits per pixel (the **B_COLOR_8_BIT** color space). **SetColorList()** passes an array of one or more colors to replace *colors* currently in the list. The first color in the array replaces the color at the specified *first* index in the list; the last color that's passed replaces the color at the *last* index. **ColorList()** returns a pointer to the entire list of 256 colors.

SetColorList() alters the list of colors kept on the graphics card. **ColorList()** doesn't return a pointer to that list, but to a local copy. This list belongs to the BWindowScreen object; it should be altered only by calling **SetColorList()**.

See also: **system_colors()** in the Interface Kit

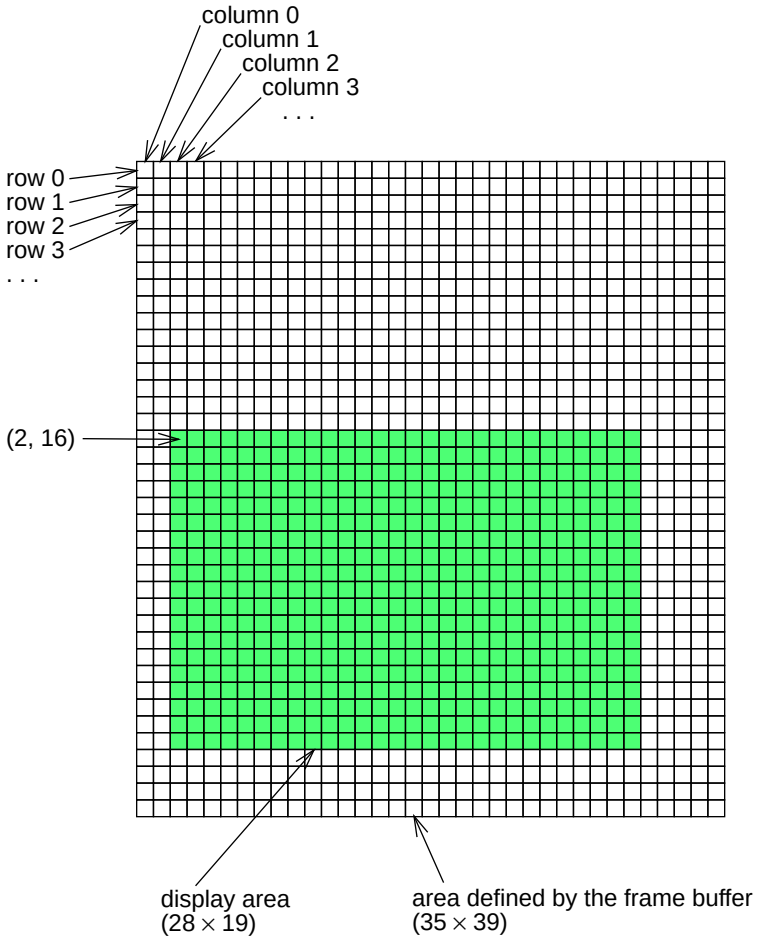
SetDisplayArea(), MoveDisplayArea()

long **SetDisplayArea(**short *width*, short *height*, short *x* = 0, short *y* = 0**)**

long **MoveDisplayArea(**short *x*, short *y***)**

These functions resize and move the display area, the portion of the frame buffer that's mapped to the screen. The area is defined by a rectangle *width* pixels wide and *height* pixels high located entirely within the frame buffer, as illustrated in miniature below. The left top pixel in the rectangle is located at (*x*, *y*), where *x* coordinate values are left-to-right

indices to a pixel column defined by the frame buffer and *y* coordinate values are top-to-bottom indices to a pixel row.



For example, the frame buffer might define twice as many pixel rows as the screen displays, so the display area can alternate between the top and bottom halves of the frame buffer for a smooth transition between images. Or the dimensions of the display area can be incrementally reduced to simulate a zoom effect as the size of on-screen pixels becomes bigger.

Like **ProposeFrameBuffer()** and **SetFrameBuffer()**, these functions work only if the graphics card driver permits application control over the frame buffer. It must also permit a display area that's smaller than the total area the frame buffer defines. If successful in moving or resizing the display area, they return **B_NO_ERROR**; if not, they return **B_ERROR**.

See also: **ProposeFrameBuffer()**, **CanControlFrameBuffer()**

SetFrameBuffer() *see* ProposeFrameBuffer()

SetSpace()

long **SetSpace(**ulong *space***)**

Configures the screen space to one of the standard combinations of width, height, and depth. The configuration is first set by the class constructor—permitted *space* constants are documented there—and it may be altered by the **SetFrameBuffer()** function in addition to this one.

If the requested configuration is refused by the graphics card driver, this function returns **B_ERROR**. If all goes well, it returns **B_NO_ERROR**.

See also: the BWindowScreen constructor, **ProposeFrameBuffer()**

WindowActivated()

virtual void **WindowActivated(**bool *active***)**

Overrides the BWindow version of **WindowActivated()** to connect the BWindowScreen object to the screen (give it control over the graphics card driver) when the *active* flag is **TRUE**.

This function doesn't disconnect the BWindowScreen when the flag is **FALSE**, because there's no way for the window to cease being the active window without the connection already having been lost.

Don't reimplement this function in your application, even if you call the inherited version; rely instead on **ScreenConnected()** for accurate notifications of when the BWindowScreen gains and loses control over the screen.

See also: **BWindow::WindowActivated()**, **ScreenConnected()**

WorkspaceActivated()

virtual void **WorkspaceActivated(**long *workspace*, bool *active***)**

Overrides the BWindow version of **WorkspaceActivated()** to connect the BWindowScreen object to the screen when the *active* flag is **TRUE** and to disconnect it when the flag is **FALSE**. User's typically activate the game by activating the workspace in which it's running, and deactivate it by moving to another workspace.

Don't override this function in your application; implement **ScreenConnected()** instead.

See also: **BWindow::WorkspaceActivated()**, **ScreenConnected()**