DrawSprocket API v1.1
External ERS

December 12, 1996

Game Technology Group, Apple Computer
sprockets@adr.apple.com

Table Of Contents

Errors and Warnings

kDSpNotInitializedErr

DSpStartup() was not called before issuing other calls to DrawSprocket.

kDSpSystemSWTooOldErr

The system software installed is too old to be used with DrawSprocket. DrawSprocket is supported on systems 7.1.2 and above.

kDSpInvalidContextErr

The context referenced is not a valid DrawSprocket context.

kDSpInvalidAttributesErr

The context attributes referenced contain one or more invalid fields.

kDSpContextAlreadyReservedErr

The context being reserved has already been reserved.

kDSpContextNotReservedErr

The context being referenced has not been reserved.

kDSpContextNotFoundErr

Unable to find the context referenced.

kDSpFrameRateNotReadyErr

DrawSprocket cannot estimate the frequency of the display because 3 seconds have not passed since the context was activated. This error will only occur if DSpContext_SetMaxFrameRate() or DSpContext_GetMaxFrameRate is called, and the frequency is unknown to DrawSprocket (and must be estimated); DSpContext_GetMonitorFrequency() will not return an error if too little time has elapsed to estimate the frequency, but the results will be more accurate as more time passes.

If a call to DSpContext_GetAttributes() returns 0 in the frequency field, DrawSprocket doesn't know the frequency of the context.

kDSpConfirmSwitchWarning

The context uses a video mode that may be unstable. If you get this error, you must display a dialog confirming that the mode is visible to the user, asking them to click OK if it is. If the user does not press OK within about 5 seconds, you should deactivate the context and let the user know that the mode could not be used. You can determine if this will occur ahead of time if you get the context attributes and check the gameMustConfirmSwitch flag.

kDSpInternalErr

If you get this error, send a bug report to sprockets@adr.apple.com and let us know how to reproduce it.

kDSpStereoContextErr

An error occurred while attempting to process a stereo context. This can happen in multiple situations, so you should examine your function inputs to determine a likely cause.

Constants

```
typedef UInt32 DSpDepthMask;
enum {
      kDSpDepthMask_1         = 1U<<0,
      kDSpDepthMask_2         = 1U<<1,
      kDSpDepthMask_4         = 1U<<2,
      kDSpDepthMask_8         = 1U<<3,
      kDSpDepthMask_16        = 1U<<4,
      kDSpDepthMask_32        = 1U<<5,
      kDSpDepthMask_All       = ~0U
};
```

DSpDepthMask is used to indicate choices among the supported DrawSprocket depths. The values are bit masks that can be combined and used at the same time.

```
typedef UInt32 DSpColorNeeds;
enum {
      kDSpColorNeeds_DontCare = 0L,
      kDSpColorNeeds_Request  = 1L,
      kDSpColorNeeds_Require  = 2L
};
```

DSpColorNeeds indicate the color needs of your game. Most games use the “require” setting.

```
typedef UInt32 DSpContextState;
enum {
      kDSpContextState_Active       = 0L,
      kDSpContextState_Paused       = 1L,
      kDSpContextState_Inactive     = 2L
};
```

DSpContextState is used to set the current state of a DSpContextReference. See DSpContext_SetState() for detailed information.

```
/* kDSpContextOption_QD3DAccel not yet implemented */
typedef UInt32 DSpContextOption;
enum {
/*    kDSpContextOption_QD3DAccel         = 1U<<0,*/
      kDSpContextOption_PageFlip          = 1U<<1,
      kDSpContextOption_DontSyncVBL       = 1U<<2,
      kDSpContextOption_Stereoscopic      = 1U<<3
};
```

DSpContextOption indicate special features supported by a context. The options are used as both inputs and outputs; when used as inputs, they specify desired features that the game requires, when used as outputs they specify actual features supported by the context.

kDSpContextOption_PageFlip indicates hardware page flipping capability.

kDSpContextOption_DontSyncVBL indicates that the context updates are not synchronized with the vertical retrace of the display.

kDSpContextOption_Stereoscopic indicates that the context supports stereoscopic imaging

with the currently active GoggleSprocket device (see the GoggleSprocket ERS for additional information).

```
typedef UInt32 DSpAltBufferOption;
enum {
      kDSpAltBufferOption_RowBytesEqualsWidth         = 1U<<0
};
```

DSpAltBufferOption is used to indicate special characteristics of an alt buffer.

kDSpAltBufferOption_RowBytesEqualsWidth forces the rowBytes and width fields of an alt buffer to be identical (in pixel units; if a context is using 16 bit color then there will be twice as many bytes in the rowBytes as there are pixels in the width, because it takes two bytes to represent one pixel).

```
typedef UInt32 DSpBufferKind;
enum {
      kDSpBufferKind_Normal   = 0U,
      kDSpBufferKind_LeftEye  = 0U,
      kDSpBufferKind_RightEye = 1U
};
```

DSpBufferKind indicates the type of buffer you wish to retrieve when calling one of the "get buffer" calls. The left and right eye buffer kinds are only applicable when using stereoscopic contexts.

```
typedef UInt32 DSpBlitMode;
enum
{
      kDSpBlitMode_SrcKey                 = 1U << 0,
      kDSpBlitMode_DstKey                 = 1U << 1,
      kDSpBlitMode_Interpolation          = 1U << 2
};
```

DSpBlitMode indicates the type of blitter operation you wish to perform. The modes can be used in combination with each other.

kDSpBlitMode_SrcKey copies image data where the source data ***is not*** the same color as the source key.

kDSpBlitMode_DstKey copies image data where the destination data ***is*** the same color as the destination key.

kDSpBlitMode_Interpolation uses interpolation between color values when scaling pixels.

```
#define kDSpEveryContext ((DSpContextReference)NULL)
```

An identifier that matches all contexts.

Data Types

```
typedef struct DSpAltBufferPrivate *DSpAltBufferReference;
```

Opaque reference to the alt buffer data type.

```
typedef struct DSpContextPrivate *DSpContextReference;
```

Opaque reference to the context reference data type.

```
typedef Boolean (*DSpEventProcPtr)( EventRecord *inEvent );
```

An event processing callback routine.

```
typedef Boolean (*DSpCallbackProcPtr)( DSpContextReference inContext,
                  void *inRefCon );
```

A general callback routine.

```
struct DSpContextAttributes {
      Fixed                   frequency;
      UInt32                  displayWidth;
      UInt32                  displayHeight;
      UInt32                  reserved1;
      UInt32                  reserved2;
      UInt32                  colorNeeds;
      CTabHandle              colorTable;
      OptionBits              contextOptions;
      OptionBits              backBufferDepthMask;
      OptionBits              displayDepthMask;
      UInt32                  backBufferBestDepth;
      UInt32                  displayBestDepth;
      UInt32                  pageCount;
      Boolean                 gameMustConfirmSwitch;
      UInt32                  reserved3[4];
};
typedef struct DSpContextAttributes DSpContextAttributes;
typedef struct DSpContextAttributes *DSpContextAttributesPtr;
```

Specification of a context's capabilities. Can be used as an input parameter or as an output parameter.

frequency (input) indicates the desired refresh rate of the context.
frequency (output) indicates the refresh rate of the context display. If the refresh rate is not known to DrawSprocket, this field will be zero.

displayWidth (input) is the desired width of the context, in pixels.
displayWidth (output) is the actual width of a context, in pixels. If this value is larger than displayWidth (input), then your context is being centered on a larger display.

displayHeight (input) is the desired height of the context, in pixels.
displayHeight (output) is the actual height of the context, in pixels. If this value is larger than displayHeight (input), then your context is being centered on a larger display.

reserved1 and reserved2 must be set to zero.

colorNeeds (input) indicate the color requirements your game has for a context.
colorNeeds (output) indicate the supported color options of the context.

colorTable (input) is the initial color table to be used by the context on input.
colorTable (output) is undefined.

contextOptions (input) are your desired context options.
contextOptions (output) are the actual context options.

backBufferDepthMask (input) are your choices for back buffer depths.
backBufferDepthMask (output) is the actual back buffer depth for a context (only one flag will be set).

displayDepthMask (input) are your choices for display depths.
displayDepthMask (output) is the actual display depth for a context (only one flag will be set).

backBufferBestDepth (input) indicates the best back buffer depth for your game. This is a numeric field, not a bit flag. The depth mask values just happen to equal their numeric equivalents, but new values may not (a mask for 24 bit depth wouldn't work if it is ever supported) so be careful when assigning values to this field.
backBufferBestDepth (output) is the best back buffer depth for a context (only one flag will be set).

displayBestDepth (input) indicates the best display depth for your game. This is a numeric field, not a bit flag. The depth mask values just happen to equal their numeric equivalents, but new values may not (a mask for 24 bit depth wouldn't work if it is ever supported) so be careful when assigning values to this field.
displayBestDepth (output) is the best display depth for a context (only one flag will be set).

pageCount (input) is the desired number of pages to be used with a context. Legal values are 1, 2, and 3. 1 will give you a single buffered context where the back buffer is always visible on the display. 2 will give you double buffering, and 3 will give you triple buffering.
pageCount (output) is the maximum number of pages that could be supported by the context.

gameMustConfirmSwitch (input) is ignored.
gameMustConfirmSwitch (output) is TRUE if the game must confirm that the video mode just entered is stable and can be seen by the user. This flag will be FALSE if the mode is a known safe mode.

reserved3[] must be set to zero.

```
struct DSpAltBufferAttributes {
      UInt32                        width;
      UInt32                        height;
      DSpAltBufferOption            options;
      UInt32                        reserved[4];
```



```
};
typedef struct DSpAltBufferAttributes DSpAltBufferAttributes;
```

Optional specification of alt buffer attributes. If this data structure is provided when allocating an alt buffer, the attributes of the alt buffer will be tailored to match the specified attributes. This will result in an alt buffer that cannot be used as an underlay, but may be used for sprite storage or other purposes.

width indicates the alt buffer width, in pixels.

height indicates the alt buffer height, in pixels.

options indicates the alt buffer options to apply.

reserved[] must be set to zero.

```
typedef void (*DSpBlitDoneProc)(struct DSpBlitInfo *);
```

A callback procedure that will be executed when an asynchronous blitter operation is completed.

```
typedef struct DSpBlitInfo
{
      Boolean                 completionFlag;
      DSpBlitDoneProc         completionProc;
      DSpContextReference     srcContext;
      CGrafPtr                srcBuffer;
      Rect                    srcRect;
      UInt32                  srcKey;

      DSpContextReference     dstContext;
      CGrafPtr                dstBuffer;
      Rect                    dstRect;
      UInt32                  dstKey;

      DSpBlitMode             mode;
      UInt32                  reserved[4];
} DSpBlitInfo, *DSpBlitInfoPtr;
```

Information specifying a blitter operation to perform. If an asynchronous blitter operation is to be performed, do not release or modify the memory occupied by this structure until the operation completes.

completionFlag (input) must be set to zero.
completionFlag (output) is set to TRUE when the blitter operation has been completed.

completionProc is a pointer to a DSpBlitDoneProc routine, or NULL no callback routine is provided.

srcContext is a pointer to the source DrawSprocket context, or NULL if the source buffer does not belong to a context.

srcBuffer is the CGrafPtr that contains the source image data.

srcRect is the rectangle containing the source image data.

srcKey is the source color key. The format of the key is the same as the pixel data format of the srcBuffer. Only pixels in the source that do NOT equal this color value will be transferred to the destination. This field is ignored if source color keying is not used.

dstContext is a pointer to the destination DrawSprocket context, or NULL if the destination buffer does not belong to a context.

dstBuffer is the destination CGrafPtr.

dstRect is the rectangle in which to place the data. If you wish to scale the data, you may specify a different dstRect than the srcRect.

dstKey is the destination color key. The format of the key is the same as the pixel data format of the srcBuffer. Only destination pixels that DO contain this color value will be replaced with source pixels. This field is ignored if destination color keying is not used.

mode is the blit mode to use.

reserved[] must be set to zero.

API Reference

The DrawSprocket API naming convention is that each function is of the form DSpVerb(), or DSpNoun_Verb().

Input parameters are prefixed with “in”, output parameters are prefixed with “out”, and input/output parameters are prefixed with “io”.

All of the DrawSprocket calls return an OSStatus function result. Check this value, don’t just assume that the call will succeed.

In the debugging version of DrawSprocket, each call performs a number of parameter validation checks. If a check fails, it will drop you into the debugger with an explanation of what went wrong. These checks result in a few frames per second speed difference when compared against the non-debug library, but they go a long way towards reducing mis-use of DrawSprocket by your game.


```
OSStatus
DSpAltBuffer_Dispose(
        DSpAltBufferReference   inAltBuffer );
```

Disposes the specified alternate buffer. If the alt buffer is currently being used as an underlay, the context will have its underlay removed.

```
OSStatus
DSpAltBuffer_GetCGrafPtr(
        DSpAltBufferReference   inAltBuffer,
        DSpBufferKind           inBufferKind,
        CGrafPtr                *outCGrafPtr,
        GDHandle                *outGDevice );
```

Returns the CGrafPtr and GDHandle associated with the alternate buffer, which may then be used to image into the alternate buffer. If the alt buffer is an underlay, remember to invalidate the rects that you have worked in.

```
OSStatus
DSpAltBuffer_InvalRect(
        DSpAltBufferReference   inAltBuffer,
        DSpBufferKind           inBufferKind,
        const Rect              *inInvalidRect );
```

Invalidates the contents of the rectangle in the alternate buffer.

```
OSStatus
DSpAltBuffer_New(
        DSpContextReference inContext,
        Boolean inVRAMBuffer,
        DSpAltBufferAttributes *inAttributes,
        DSpAltBufferReference *outAltBuffer );
```

Creates a new alternate buffer.

If the 'inVRAMBuffer' parameter is TRUE, then DrawSprocket will attempt to allocate the buffer in VRAM. This is useful for hardware accelerated blits where the alt buffer is used to store image data in VRAM. If there is no space left in VRAM, or if there is no hardware acceleration to take advantage of the VRAM buffer, the buffer will be allocated in system RAM and no error will be returned.

If the 'inAttributes' parameter is NULL the alternate buffer will have the same characteristics as the specified context, otherwise it will have the specified characteristics and may ***not*** be used as an underlay buffer.

```
OSStatus
DSpBlit_Faster(
        DSpBlitInfoPtr inBlitInfo,
```



```
        Boolean inAsyncFlag );
```

Performs the specified blit operation. The blit will be clipped to the destination.

If 'inAsyncFlag' is TRUE, the blitter operation will be performed asynchronously if possible.

```
OSStatus
DSpBlit_Fastest(
        DSpBlitInfoPtr inBlitInfo,
        Boolean inAsyncFlag );
```

Performs the specified blit operation, without scaling. The blit will be clipped to the destination.

If 'inAsyncFlag' is TRUE, the blitter operation will be performed asynchronously if possible.

```
OSStatus
DSpCanUserSelectContext(
        DSpContextAttributes    *inDesiredAttributes,
        Boolean                 *outUserCanSelectContext );
```

Given the attributes for a context, this call will determine if you should allow the user to select a display using DSpUserSelectContext(). Even on a system with multiple displays, your game may have context attributes that only one device can support. If the 'outUserCanSelectContext' field is FALSE on output, you should not enable any user interface items that allow display selection.

If 'outUserCanSelectContext' is FALSE on output, then a call to DSpUserSelectContext() will return immediately without presenting a dialog to the user (because only one choice was possible).

```
OSStatus
DSpContext_FadeGamma(
        DSpContextReference     inContext,
        SInt32                  inPercentOfOriginalIntensity,
        RGBColor                *inZeroIntensityColor );
```

Manually fade the gamma for the specified context. If you specify kDSpEveryContext as the context, all display devices will be affected. Gamma fading works on direct and indexed devices, and avoids the problems with indexed fading where brighter colors will fade slower than darker colors.

An intensity of 100 will set the gamma to 100% of the normal values (i.e. it will look exactly the same as if you hadn't changed the intensity). A value of 50 will set the intensity to 50%, a value of 150 will set the value to 150% of normal -- values above 100% begin to converge on white.

If you specify a zero intensity color, instead of converging on black as the display fades to zero intensity it will fade to the color you specify (i.e. fade to red). When fading to a non-black color, negative intensity values begin to converge from the intensity color towards black, so you could fade to red by

fading to 0% and specifying a red intensity color, and then from red to black by specifying the same color and negative intensities until you hit -100%.

```
OSStatus
DSpContext_FadeGammaIn(
        DSpContextReference     inContext,
        RGBColor                *inZeroIntensityColor );
```

Automatically fade the gamma in from from 0% to 100% intensity over a period of one second. If 'inContext' is kDSpEveryContext, then all contexts will be faded back in.

```
OSStatus
DSpContext_FadeGammaOut(
        DSpContextReference     inContext,
        RGBColor                *inZeroIntensityColor );
```

Automatically fade the gamma out from 100% to 0% intensity over a period of one second. If 'inContext' is kDSpEveryContext, then all contexts will be faded out. Fading all contexts at once is highly recommended before activating your first context or inactivating your last context, as it will fade all displays on a multiple monitor system.

```
OSStatus
DSpContext_Flatten(
        DSpContextReference     inContext,
        void                    *outFlatContext );
```

Saves, in the provided buffer, a representation of the context suitable for saving to disk. The buffer must be large enough to hold the flattened context; you can find out the correct size by calling DSpContext_GetFlattenedSize().

```
OSStatus
DSpContext_GetAttributes(
        DSpContextReference     inContext,
        DSpContextAttributesPtr outAttributes );
```

Return the actual attributes of the specified context. The attributes indicate the maximum settings for a context, not the attributes in use by a context that has been activated. To determine the attributes that are in use by an active context you must save the attribute settings you provide to DSpContext_Reserve() for your own reference.

The monitor frequency may not be known until a context is actually activated, so it may be returned as zero. If this is the case, then you can call DSpGetMonitorFrequency() after the context has been active for at least 3 seconds (DrawSprocket must get a running average of the display rate, and calling the function sooner will result in an error).

```
OSStatus
DSpContext_GetBackBuffer(
```



```
    DSpContextReference     inContext,
    DSpBufferKind           inBufferKind,
    CGrafPtr                *outBackBuffer );
```

Return the back buffer for the context, which is where the game should image to. The back buffer is the next buffer that will be displayed upon a call to DSpContext_SwapBuffers().

The pointer to the back buffer may change after a call to DSpContext_SwapBuffers(), so you must call this function before rendering every frame.

If you have specified an underlay for the context, the back buffer will have the underlay image restored before this call returns.

If there are no available back buffers, i.e. they are all queued up for display, then this function will block until one is available. To avoid blocking, call DSpContext_IsBusy() until it returns FALSE.

For stereoscopic contexts, call this function twice -- once with kDSpBufferKind_LeftEye and once with kDSpBufferKind_RightEye, and render the appropriate images into each buffer before calling DSpContext_SwapBuffers() to display the stereoscopic image.

```
OSStatus
DSpContext_GetCLUTEntries(
    DSpContextReference     inContext,
    ColorSpec               *outEntries,
    UInt16                  inStartingEntry,
    UInt16                  inLastEntry );
```

Return the current color palette for the context. inStartingEntry and inLastEntry are index values -- they must be between 0 and 255. inStartingEntry + inLastEntry must be < = 255.

```
OSStatus
DSpContext_GetDirtyRectGridSize(
    DSpContextReference     inContext,
    UInt32                  *outCellPixelWidth,
    UInt32                  *outCellPixelHeight );
```

Return the current pixel size of the grid units. The current pixel size may be different from the values specified in DSpContext_SetDirtyRectGridSize() because the grid units must be aligned to meet certain boundary conditions. To find out the dimensions of what the basic cell unit will be, and what all actual grid sizes must be multiples of, you can use the DSpContext_GetDirtyRectGridUnits() call.

```
OSStatus
DSpContext_GetDirtyRectGridUnits(
    DSpContextReference     inContext,
    UInt32                  *outCellPixelWidth,
    UInt32                  *outCellPixelHeight );
```


Return the pixel size of the grid units that are always used. The grid unit sizes are based upon a number of machine characteristics such as the bus width and L1 cache size, and so when you specify a different grid size it will always be rounded up to the closest grid unit size.

```
OSStatus
DSpContext_GetDisplayID(
        DSpContextReference     inContext,
        DisplayIDType           *outFlatContextSize );
```

Return the Display Manager DisplayID for the context. The DisplayID can be used to determine the GDevice associated with a context by calling the Display Manager function DMGetGDeviceByDisplayID().

```
OSStatus
DSpContext_GetFlattenedSize(
        DSpContextReference     inContext,
        UInt32                  *outFlatContextSize );
```

Return the size required for a flattened version of the context. You should allocate a buffer of this size and pass it in to DSpContext_Flatten().

```
OSStatus
DSpContext_GetMaxFrameRate(
        DSpContextReference     inContext,
        UInt32                  *outMaxFPS );
```

Return the current maximum frame rate for your game. If 0 is returned as the maximum FPS, there are no frame rate restrictions in place. The maximum frame rate may be slightly different than that set by DSpContext_SetMaxFrameRate(), as DrawSprocket will use the requested maximum frame rate to determine a number of frames to skip before drawing a frame.

```
OSStatus
DSpContext_GetMonitorFrequency(
        DSpContextReference     inContext,
        Fixed                   *outFrequency );
```

Return the frequency for the monitor associated with the context. If DrawSprocket can determine the frequency from the DisplayManger, the DSpContextAttributes.frequency field will be non-zero after a call to DSpContext_GetAttributes(). If the frequency is not known, this call will estimate the frequency of the display using a slot VBL task. The context must have been active for a reasonable amount of time (three seconds, at least) in order to receive a correct value because the value returned is calculated by timing the frame rate of the context in its active state; no error will be returned if that is not the case, but the result will be inaccurate.

```
OSStatus
DSpContext_GetState(
```



```
        DSpContextReference     inContext,
        UInt32                  *outState );
```

Return the current play state of the context.

```
OSStatus
DSpContext_GetUnderlayAltBuffer(
        DSpContextReference     inContext,
        DSpAltBufferReference   *outUnderlay );
```

Return the current underlay buffer.

```
OSStatus
DSpContext_GlobalToLocal(
        DSpContextReference     inContext,
        Point                   *ioPoint );
```

Convert global mouse coordinates to local context coordinates.

```
OSStatus
DSpContext_InvalBackBufferRect(
        DSpContextReference     inContext,
        const Rect              *inRect );
```

Invalidate a rectangle in the back buffer. Multiple invalid rectangles may be set. If not called, then the entire back buffer is assumed to be dirty and will be transferred to the screen.

The invalid rectangles must be set prior to each call to DSpContext_SwapBuffers(); the dirty rectangles are cleared during DSpContext_GetBackBuffer() when it returns the back buffer for re-use.

```
OSStatus
DSpContext_IsBusy(
        DSpContextReference     inContext,
        Boolean                 *outBusyFlag );
```

This function will return TRUE in the busy flag if there are no back buffers available. This is useful if you would like to know if calling DSpContext_GetBackBuffer() will block.

```
OSStatus
DSpContext_LocalToGlobal(
        DSpContextReference     inContext,
        Point                   *ioPoint );
```

Convert local context coordinates to global coordinates.

```
OSStatus
DSpContext_Release(
```



```
    DSpContextReference inContext );
```

When you are finished using a context, you must release it with this function. The display device related to this context will remain covered by the blanking window as long as there is at least one active or paused context.

```
OSStatus
DSpContext_Reserve(
    DSpContextReference         inContext,
    DSpContextAttributesPtr     inDesiredAttributes );
```

This function reserves the specified context so that you may begin using it in your game. The contextwill become visible once the context has been placed into the active or paused states. The input attributes specify the configuration information for the display when it is in the active or paused states.

The context is reserved in the inactive state. There will be no visible indication that the context has been reserved at this point. To enable your context, call DSpContext_SetState().

If you would like to override the attributes of the context, you may specify attributes in the input attributes structure. For example if you ask for a 320x240x16 display but the closest match is a context that is 640x480x32, then passing in your requested attributes when you reserve the context will cause DrawSprocket to setup the world so that you believe you are working in a 320x240x16 environment.

Along the same lines, you should turn off features that you are not interested in when you reserve the context. For example if the context supports page flipping (and you know this because you requested the actual capabilities of the context using DSpContext_GetAttributes()), you can turn off the page flipping bit in your desired attributes so that you will be insured of using software buffering.

You should only specify a back buffer depth different from the front buffer depth when you absolutely must, as it is the worst case scenario for DrawSprocket and will result in a synchronous call to CopyBits() to bring your back buffer to the front. When the back buffer and front buffer depths are the same, DrawSprocket uses highly optimized custom blitters to transfer the back buffer to the display.

```
OSStatus
DSpContext_Restore(
    void                    *inFlatContext,
    DSpContextReference     *outRestoredContext );
```

Restores a flattened context. Typically the flat context would have been saved out to disk and reloaded on a later execution of the game before calling this function.

This function has a high probability of failing, so your game should not rely on being able to restore the context, but should attempt to do so if you are saving the user's preferences for example.

```
OSStatus
DSpContext_SetCLUTEntries(
```



```
    DSpContextReference     inContext,
    const ColorSpec         *inEntries,
    UInt16                  inStartingEntry,
    UInt16                  inEndingEntry );
```

Sets the specified CLUT entries for an indexed display. inStartingEntry and inLastEntry are index values -- they must be between 0 and 255. inStartingEntry + inLastEntry must be < = 255.

```
OSStatus
DSpContext_SetDirtyRectGridSize(
    DSpContextReference     inContext,
    UInt32                  inCellPixelWidth,
    UInt32                  inCellPixelHeight );
```

You can suggest a grid cell size for the context dirty rect, and DrawSprocket will attempt to match your suggested size as closely as possible. The actual size that is set is dependent on factors such as the L1 cache size and the CPU bus width, so your suggested values may not be the actual values used. If you are interested in seeing what DrawSprocket is using, then you can examine the actual grid size as a result of this call by using the DSpContext_GetDirtyRectGridSize() call, and you can examine the basic grid units with the DSpContext_GetDirtyRectGridUnits() call.

```
OSStatus
DSpContext_SetMaxFrameRate(
    DSpContextReference     inContext,
    UInt32                  inMaxFPS );
```

This call allows you to set a MAXIMUM frame rate for your game. This does not guarantee that your game will achieve that rate, but if your game attempts to exceed that frame rate DrawSprocket will slow the buffer swapping down to the correct speed.

The actual frame rate that is set will not necessarily be the frame rate you have specified, because DrawSprocket internally will convert your max frame rate into a value that can be used to skip a number of frames for each frame that is drawn.

For example, if the monitor refresh rate is 66.7hz, and you request a frame rate of 30fps, then internally DrawSprocket will skip every other frame, and your resulting frame rate will be about 33.3hz.

```
OSStatus
DSpContext_SetState(
    DSpContextReference     inContext,
    UInt32                  inState );
```

Set the state of a context. A context may exist in one of three states:

> Inactive, as it is initially. When no displays are active then the system looks exactly as it does when the user is using their Macintosh normally: the monitor resolutions are set to the default,

the menu bar is available, etc.

Active, in which the characteristics of the context are used to change the display resolution, remove the menu bar, and so on. When at least one context is active, all of the display devices in the system are covered by a blanking window. When a context is in the active state, the display is completely owned by the game.

Paused, in which the characteristics of the context are in place, but the system items such as the menu bar and floating windows are available to the user. This allows your user the opportunity to use the menus and switch to other applications. While your context is in the paused state it is very important to call DSpProcessEvent() to allow DrawSprocket to correctly handle events such as suspend/resume. Page flipping and double buffering are inactive in this state, and the context will be placed back at page 0 if page flipping was being used.

```
OSStatus
DSpContext_SetUnderlayAltBuffer(
        DSpContextReference     inContext,
        DSpAltBufferReference   inNewUnderlay );
```

This call is how a game can specify that a buffer is to be used as an underlay buffer. Underlay buffers are used to "clean" a back buffer when DSpContext_GetBackBuffer() is called. When a back buffer is retrieved and there is an underlay buffer, then the invalid areas in the back buffer will be restored from the underlay buffer. This is most useful in sprite games, or in games where the background is static (at least for a while).

To remove an underlay, make this call with inNewUnderlay set to NULL.

```
OSStatus
DSpContext_SetVBLProc(
        DSpContextReference     inContext,
        DSpCallbackProcPtr      inProcPtr,
        void                    *inRefCon );
```

Because DrawSprocket needs to setup VBL tasks of its own, you can piggyback your own VBL task to a particular context easily with this call, instead of digging down through the system to find the correct slot id and installing your own.

The return value from inProcPtr is ignored by DrawSprocket.

```
OSStatus
DSpContext_SwapBuffers (
        DSpContextReference     inContext,
        DSpCallbackProcPtr      inBusyProc,
        void                    *inUserRefCon );
```

Displays somewhere between just the dirty pixels and all the pixels from the back buffer. If DSpContext_InvalBackBufferRect() is not called, then the entire back buffer is assumed to be dirty.

This routine will return immediately, even though the display swap may not yet have occured. To determine when the swap has actually occured and a back buffer is again available, call DSpContext_IsBusy() until it returns a value of false.

Before performing the swap, DrawSprocket will call inBusyProc to insure that any constraints imposed by the calling program are satisfied before peforming the swap. A sample function of inBusyProc would be to check if any QuickDraw 3D acceleration hardware has completed rendering. The inBusyProc will be repeatedly called until it returns FALSE.

```
OSStatus
DSpFindBestContext(
        DSpContextAttributesPtr         inDesiredAttributes,
        DSpContextReference             *outContext );
```

DrawSprocket will attempt to find the context that best approximates the context you describe in the context attributes. The context attributes should specify as best as possible the details of a context you would like to use.

All display contexts are considered that meet or exceed the requirements in the attributes. If no such contexts exist, then an error is returned.

The game should make sure that the actual attributes for the context are acceptable by calling DSpContext_GetAttributes(). For example, it is possible that the game will request a mode such as 320x200x8 but the best match is a 640x480x8 display; the game may wish to adapt to a full screen mode once it is aware of the situation.

The context attributes are treated as requirements. If any portion of the attributes can not be adapted to a context (hardware page flipping, for example), the context will not be considered a match.

```
OSStatus
DSpFindContextFromPoint(
        Point                   inGlobalPoint,
        DSpContextReference     *outContext );
```

Given a point in global coordinates, this function will return the context that contains the point.

```
OSStatus
DSpGetFirstContext(
        DisplayIDType           inDisplayID,
        DSpContextReference     *outContext );
```

Returns the first context reference for the specified display. The display ID can be obtained by using the Display Manager.

It is important to note that you cannot use this context with any call other than DSpContext_GetAttributes() until you reserve it with DSpContext_Reserve(). This call is provided so

that you may iterate over the available contexts and choose one that best suits your needs, should you decide not to let DrawSprocket find one for you with DSpFindBestContext() or let the user select one by calling DSpUserSelectContext().

```
OSStatus
DSpGetMouse(
        Point *outGlobalPoint );
```

Returns the mouse position in global coordinates.

```
OSStatus
DSpGetNextContext(
        DSpContextReference     inCurrentContext,
        DSpContextReference     *outContext );
```

Returns the next context in the list of available contexts for a display.

A sample use would be:

```
DSpContextReference theContext;

DSpGetFirstContext( theDisplayID, &theContext );
while( theContext )
{
      /* process the context */

      /* get the next context */
      DSpGetNextContext( theContext, &theContext );
}
```

```
OSStatus
DSpProcessEvent(
        EventRecord     *inEvent,
        Boolean *outEventWasProcessed );
```

When your context is in the inactive or paused states, you must pass system events through DrawSprocket so that it may correctly handle items such as suspend/resume events and updating the blanking window.

If you don't call this function when you are using the Macintosh Event Manager, then your game will run into some serious problems if some events pass without DrawSprocket's knowledge.

```
OSStatus
DSpSetBlankingColor(
        const RGBColor  *inBlankingColor );
```

When at least one context is in the active state, all display devices in the system are obscured with a blanking window. You can set the color of the window content area with this call, the

default blanking window color is black.

If you are using an indexed color mode in your context and one of the colors doesn't approximate the blanking color, the blanking window will have an undefined color.

```
OSStatus
DSpSetDebugMode(
        Boolean inDebugMode );
```

During development of your game it can become very awkward to view your debugger when the gamma has been faded to zero intensity, or the blanking window is covering all of the displays. By setting the input value to TRUE for this call, DrawSprocket will not do things that make your life hard during the debug cycle. These things include:

Gamma Fading: when a gamma fade occurs, the affected contexts will be "flashed" to half intensity for a brief period and then restored to normal. This lets you know that a gamma fade would have happened at that point in time on the context.

Blanking Window: the blanking window is not created. This means that some display devices may still be visible, and you may see some artifacts here and there. You may experience some strange interactions with the mouse if you click in the window of another application (such as the Finder), but at least you'll be able to see your debugging window.

This call has no effect in the non-debug version of DrawSprocket.

An alternate way to enable debug mode is to create a folder named "DSpSetDebugMode" in the same folder as your game. This allows you to enter/exit debug mode at any time in your program without rebuilding.

```
OSStatus
DSpShutdown( void );
```

Shutdown DrawSprocket. Call this function after you are finished using DrawSprocket.

```
OSStatus
DSpStartup( void );
```

Startup DrawSprocket. Call this function before you use DrawSprocket.

```
OSStatus
DSpUserSelectContext(
        const DSpContextAttributesPtr   inDesiredAttributes,
        DisplayIDType                   inDialogDisplayLocation,
        DSpEventProcPtr                 inEventProc,
        DSpContextReference             *outContext );
```

DrawSprocket will display a movable modal dialog to the user that allows them to choose the

display that they wish to run your game on. Only displays that are capable of matching or exceeding your specified attributes will be selectable by the user.

The event proc you provide will allow your game to process update events that are generated when the user moves the dialog around the screen. If you process an event, you must return TRUE to from your event proc.

You should only make this call if DSpCanUserSelectContext() indicates that there is more than one display for the user to choose from. If there is only one display choice, this call will not present the dialog and will return immediately with the context that matches your attributes. This can be confusing for the user.

