Khronos Native Platform Graphics Interface
(EGL Version 1.4 - December 4, 2013)

Editor: Jon Leech

Contents

List of Tables

Chapter 1

Overview

This document describes EGL, an interface between rendering APIs such as OpenGL , OpenGL ES or OpenVG (referred to collectively as *client APIs*) and an underlying native platform window system. It refers to concepts discussed in the OpenGL , OpenGL ES and OpenVG specifications, and should be read together with those documents. EGL uses OpenGL ES conventions for naming entry points and macros.

EGL provides mechanisms for creating rendering surfaces onto which client APIs can draw, creating graphics contexts for client APIs , and synchronizing drawing by client APIs as well as native platform rendering APIs. EGL does not explicitly support remote or *indirect* rendering, unlike the similar GLX API.

1.1 Comments on edits to the EGL Specification

Changes in the most recent revision of the spec are typeset in magenta.
Older changes are typeset in purple.

Chapter 2

EGL Operation

2.1 Native Window System and Rendering APIs

EGL is intended to be implementable on multiple operating systems (such as Symbian, embedded Linux, Unix, and Windows) and *native window systems* (such as X and Microsoft Windows). Implementations may also choose to allow rendering into specific types of EGL *surfaces* via other supported *native rendering APIs*, such as Xlib or GDI. Native rendering is described in more detail in section 2.2.3.

To the extent possible, EGL itself is independent of definitions and concepts specific to any native window system or rendering API. However, there are a few places where native concepts must be mapped into EGL-specific concepts, including the definition of the *display* on which graphics are drawn, and the definition of native windows and pixmaps which can also support client API rendering.

2.1.1 Scalar Types

`EGLBoolean` is an integral type representing a boolean value, and should only take on the values `EGL_TRUE` (1) and `EGL_FALSE` (0). If boolean parameters passed to EGL take on other values, behavior is undefined, although typically any non-zero value will be interpreted as `EGL_TRUE`.

`EGLint` is an integral type used because EGL may need to represent scalar values larger than the native platform "int" type. All legal attribute names and values, whether their type is boolean, bitmask, enumerant (symbolic constant), integer, handle, or other, may be converted to and from `EGLint` without loss of information.

Starting with the November, 2013 update of EGL 1.4, `EGLint` is defined to be at least the same size as the native platform `int` type. This change means that handle and pointer attribute values may not be representable in attribute lists

on platforms where sizeof(pointer) > sizeof(EGLint). Existing extensions which assume that pointers can always be represented in EGLint attributes are being replaced with new extensions specifying new entry points and attribute types, to work around this issue[1].

2.1.2 Displays

Most EGL calls include an EGLDisplay parameter. This represents the abstract display on which graphics are drawn. In most environments a display corresponds to a single physical screen. The initialization routines described in section 3.2 include a method for querying a *default display*, and platform-specific EGL extensions may be defined to obtain other displays.

All EGL objects are associated with an EGLDisplay, and exist in a namespace defined by that display. Objects are always specified by the combination of an EGLDisplay parameter with a parameter representing the handle of the object.

2.2 Rendering Contexts and Drawing Surfaces

The client API specifications are intentionally vague on how a *rendering context* (e.g. the state machine defined by a client API) is created. One of the purposes of EGL is to provide a means to create client API rendering contexts (henceforth simply referred to as *contexts*), and associate them with drawing surfaces.

EGL defines several types of drawing surfaces collectively referred to as EGLSurfaces. These include *windows*, used for onscreen rendering; *pbuffers*, used for offscreen rendering; and *pixmaps*, used for offscreen rendering into buffers that may be accessed through native APIs. EGL windows and pixmaps are tied to native window system windows and pixmaps.

EGLSurfaces are created with respect to an EGLConfig. The EGLConfig describes the depth of the color buffer components and the types, quantities and sizes of the *ancillary buffers* (i.e., the depth, multisample, and stencil buffers).

Ancillary buffers are associated with an EGLSurface, not with a context. If several contexts are all writing to the same surface, they will share those buffers. Rendering operations to one window never affect the unobscured pixels of another window, or the corresponding pixels of ancillary buffers of that window.

[1] This functionality regression has been adopted because EGL implementations on some 64-bit platforms chose their EGLint type to be a 32-bit integer type, and changing the definition would break their ABIs in a way considered to be too disruptive to their application base. The EGL_KHR_cl_event2 and EGL_KHR_lock_surface3 extensions replace similar earlier extensions allowing pointers in attribute lists, and work around this regression by providing new interfaces using attribute types which **are** guaranteed to be sufficiently large.

Contexts for different client APIs all share the color buffer of a surface, but ancillary buffers are not necessarily meaningful for every client API . In particular, depth, multisample, and stencil buffers are currently used only by OpenGL and OpenGL ES .

A context can be used with any `EGLSurface` that it is *compatible* with (subject to the restrictions discussed in the section on address space). A surface and context are compatible if

- They support the same type of color buffer (RGB or luminance).
- They have color buffers and ancillary buffers of the same depth.

  Depth is measured per-component. For example, color buffers in RGB565 and RGBA4444 formats have the same aggregate depth of 16 bits/pixel, but are not compatible because their per-component depths are different.

  Ancillary buffers not meaningful to a client API do not affect compatibility; for example, a surface with both color and stencil buffers will be compatible with an OpenVG context so long as the color buffers associated with the contexts are of the same depth. The stencil buffer is irrelevant because OpenVG does not use it.
- The surface was created with respect to an `EGLConfig` supporting client API rendering of the same type as the API type of the context (in environments supporting multiple client APIs).
- They were created with respect to the same `EGLDisplay` (in environments supporting multiple displays).

As long as the compatibility constraint and the address space requirement are satisfied, clients can render into the same `EGLSurface` using different contexts. It is also possible to use a single context to render into multiple `EGLSurfaces`.

2.2.1 Using Rendering Contexts

OpenGL and OpenGL ES define both client state and server state. Thus an OpenGL or OpenGL ES context consists of two parts: one to hold the client state and one to hold the server state. OpenVG does not separate client and server state.

The OpenGL , OpenGL ES , and OpenVG client APIs rely on an *implicit* context used by all entry points, rather than passing an explicit context parameter. The implicit context for each API is set with EGL calls (see section 3.7.3). The implicit contexts used by these APIs are called *current contexts*.

Each thread can have at most one current rendering context for each supported client API ; for example, there may be both a current OpenGL ES context and a current OpenVG context in an implementation supporting both of these APIs. In addition, a context can be current to only one thread at a time. The client is responsible for creating contexts and surfaces. Because OpenGL and OpenGL ES contexts share many entry points, additional restrictions on current contexts exists for these client APIs when both are supported (see section 3.7).

2.2.2 Rendering Models

EGL, OpenGL , and OpenGL ES support two rendering models: back buffered and single buffered.

Back buffered rendering is used by window and pbuffer surfaces. Memory for the color buffer used during rendering is allocated and owned by EGL. When the client is finished drawing a frame, the back buffer may be copied to a visible window using **eglSwapBuffers**. Pbuffer surfaces have a back buffer but no associated window, so the back buffer need not be copied.

Single buffered rendering is used by pixmap surfaces. Memory for the color buffer is specified at surface creation time in the form of a native pixmap, and client APIs are required to use that memory during rendering. When the client is finished drawing a frame, the native pixmap contains the final image. Pixmap surfaces typically do not support multisampling, since the native pixmap used as the color buffer is unlikely to provide space to store multisample information.

Some client APIs , such as OpenGL and OpenVG , also support single buffered rendering to window surfaces. This behavior can be selected when creating the window surface, as defined in section 3.5.1. When mixing use of client APIs which do not support single buffered rendering into windows, like OpenGL ES , with client APIs which do support it, back color buffers and visible window contents must be kept consistent when binding window surfaces to contexts for each API type (see section 3.7.3).

Both back and single buffered surfaces may also be copied to a specified native pixmap using **eglCopyBuffers**.

2.2.2.1 Native Surface Coordinate Systems

The coordinate system for native windows and pixmaps in most window systems is inverted relative to the OpenGL , OpenGL ES , and OpenVG client API coordinate systems. In such systems, native windows and pixmaps have $(0, 0)$ in the upper left of the pixmap, while the client APIs have $(0, 0)$ in the lower left. To accomodate this, client API rendering to window and pixmap surfaces must invert their own

y coordinate when accessing the color buffer in the underlying native window or pixmap, so that the resulting images appear as intended by the application when the final image is displayed by **eglSwapBuffers** or copied from a pixmap to a visible window using native rendering APIs.

2.2.2.2 Window Resizing

EGL window surfaces need to be resized when their corresponding native window is resized. Implementations typically use hooks into the OS and native window system to perform this resizing on demand, transparently to the client. Some implementations may instead define an EGL extension giving explicit control of surface resizing.

Implementations which cannot resize EGL window surfaces on demand must instead respond to native window size changes in **eglSwapBuffers** (see section 3.9.3).

2.2.3 Interaction With Native Rendering

Native rendering will always be supported by pixmap surfaces (to the extent that native rendering APIs can draw to native pixmaps). Pixmap surfaces are typically used when mixing native and client API rendering is desirable, since there is no need to move data between the back buffer visible to the client APIs and the native pixmap visible to native rendering APIs. However, pixmap surfaces may, for the same reason, have restricted capabilities and performance relative to window and pbuffer surfaces.

Native rendering will not be supported by pbuffer surfaces, since the color buffers of pbuffers are allocated internally by EGL and are not accessible through any other means.

Native rendering may be supported by window surfaces, but only if the native window system has a compatible rendering model allowing it to share the back color buffer, or if single buffered rendering to the window surface is being done.

When both native rendering APIs and client APIs are drawing into the same underlying surface, no guarantees are placed on the relative order of completion of operations in the different rendering streams other than those provided by the synchronization primitives discussed in section 3.8.

Some state is shared between client APIs and the underlying native window system and rendering APIs, including color buffer values in window and pixmap surfaces.

2.3 Direct Rendering and Address Spaces

EGL is assumed to support only *direct* rendering, unlike similar APIs such as GLX. EGL objects and related context state cannot be used outside of the *address space* in which they are created. In a single-threaded environment, each process has its own address space. In a multi-threaded environment, all threads may share the same virtual address space; however, this capability is not required, and implementations may choose to restrict their address space to be per-thread even in an environment supporting multiple application threads.

Context state, including both the client and server state of OpenGL and OpenGL ES contexts, exists in the client's address space; this state cannot be shared by a client in another process.

Support of indirect rendering (in those environments where this concept makes sense) may have the effect of relaxing these limits on sharing. However, such support is beyond the scope of this document.

2.4 Shared State

Most context state is small. However, some types of state are potentially large and/or expensive to copy, in which case it may be desirable for multiple contexts to share such state rather than replicating it in each context. Such state may only be shared between different contexts of the same API type (e.g. two OpenGL contexts, two OpenGL ES contexts, or two OpenVG contexts, but not a mixture).

EGL provides for sharing certain types of context state among contexts existing in a single address space. The types of client API objects that are shareable are defined by the corresponding client API specifications. ~~OpenGL contexts may share *texture objects*, *shader objects*, *program objects*, *display list objects*, and pixel and vertex *buffer objects*. OpenGL ES contexts may share texture objects, shader and program objects, and vertex buffer objects. OpenVG contexts may share *images*, *paint objects*, and *paths*. Additional types of state may be shared in future revisions of client APIs where such types of state (for example, display lists) are defined and where such sharing makes sense.~~

2.4.1 OpenGL and OpenGL ES Texture Objects

Texture state can be encapsulated in a named texture object. A texture object is created by binding an unused name to one of the supported texture targets (`GL_TEXTURE_2D`, `GL_TEXTURE_3D`, or `GL_TEXTURE_CUBE_MAP`) of OpenGL or OpenGL ES context. When a texture object is bound, operations on the target to

which it is bound affect the bound texture object, and queries of the target to which it is bound return state from the bound texture object.

OpenGL and OpenGL ES makes no attempt to synchronize access to texture objects. If a texture object is bound to more than one context, then it is up to the programmer to ensure that the contents of the object are not being changed via one context while another context is using the texture object for rendering. The results of changing a texture object while another context is using it are undefined.

All modifications to shared context state as a result of executing **glBindTexture** are atomic. Also, a texture object will not be deleted while it is still bound to any context.

2.4.2 OpenGL and OpenGL ES Buffer Objects

If a OpenGL or OpenGL ES buffer object is bound to more than one context, then it is up to the programmer to ensure that the contents of the object are not being changed via one context while another context is using the buffer object for rendering. The results of changing a buffer object while another context is using it are undefined.

All modifications to shared context state as a result of executing **glBindBuffer** are atomic. Also, a buffer object will not be deleted while it is still bound to any context.

2.5 Multiple Threads

EGL and its client APIs must be threadsafe. Interrupt routines may not share a context with their main thread.

EGL guarantees sequentiality within a command stream for each of its client APIs , ~~such as OpenGL ES and OpenVG ,~~ but not between these APIs and native APIs which may also be rendering into the same surface. It is possible, for example, that a native drawing command issued by a single threaded client after an OpenGL ES command might be executed before that OpenGL ES command.

Client API commands are not guaranteed to be atomic. Some such commands might otherwise impair interactive use of the windowing system by the user. For instance, rendering a large texture mapped polygon on a system with no graphics hardware, or drawing a large OpenGL ES vertex array, could prevent a user from popping up a menu soon enough to be usable.

Synchronization is in the hands of the client. It can be maintained at moderate cost with the judicious use of commands such as **glFinish**, **vgFinish**, **eglWaitClient**, and **eglWaitNative**, as well as (if they exist) synchronization commands

present in native rendering APIs. Client API and native rendering can be done in parallel so long as the client does not preclude it with explicit synchronization calls.

Some performance degradation may be experienced if needless switching between client APIs and native rendering is done.

2.6 Power Management

Power management events can occur asynchronously while an application is running. When the system returns from the power management event the `EGLContext` will be invalidated, and all subsequent client API calls will have no effect (as if no context is bound).

Following a power management event, calls to **eglSwapBuffers**, **eglCopyBuffers**, or **eglMakeCurrent** will indicate failure by returning `EGL_FALSE`. The error `EGL_CONTEXT_LOST` will be returned if a power management event has occurred.

On detection of this error, the application must destroy all contexts (by calling **eglDestroyContext** for each context). To continue rendering the application must recreate any contexts it requires, and subsequently restore any client API state and objects it wishes to use.

Any `EGLSurfaces` that the application has created need not be destroyed following a power management event, but their contents will be invalid.

Note that not all implementations can be made to generate power management events, and developers should continue to refer to platform-specific documentation in this area. We expected continued work in platform-specific extensions to enable more control over power management issues, including event detection, scope and nature of resource loss, behavior of EGL and client API calls under resource loss, and recommended techniques for recovering from events. Future versions of EGL may incorporate additional functionality in this area.

Chapter 3

EGL Functions and Errors

3.1 Errors

Where possible, when an EGL function fails it has no side effects.

EGL functions usually return an indicator of success or failure; either an `EGLBoolean` `EGL_TRUE` or `EGL_FALSE` value, or in the form of an out-of-band return value indicating failure, such as returning `EGL_NO_CONTEXT` instead of a requested context handle. Additional information about the success or failure of the **most recent** EGL function called in a specific thread[1], in the form of an error code, can be obtained by calling

```
EGLint eglGetError(void);
```

The error codes that may be returned from **eglGetError**, and their meanings, are:

`EGL_SUCCESS`
: Function succeeded.

`EGL_NOT_INITIALIZED`
: EGL is not initialized, or could not be initialized, for the specified display. Any command may generate this error.

`EGL_BAD_ACCESS`
: EGL cannot access a requested resource (for example, a context is bound in

[1] Note that calling **eglGetError** twice without any other intervening EGL calls will always return `EGL_SUCCESS` on the second call, since **eglGetError** is itself an EGL function, and the second call is reporting the success or failure of the first call. In other words, error checking must always be performed immediately after an EGL function fails.

another thread). Any command accessing a named resource may generate this error.

`EGL_BAD_ALLOC`
: EGL failed to allocate resources for the requested operation. Any command allocating resources may generate this error.

`EGL_BAD_ATTRIBUTE`
: An unrecognized attribute or attribute value was passed in an attribute list. Any command taking an attribute parameter or attribute list may generate this error.

`EGL_BAD_CONTEXT`
: An `EGLContext` argument does not name a valid `EGLContext`. Any command taking an `EGLContext` parameter may generate this error.

`EGL_BAD_CONFIG`
: An `EGLConfig` argument does not name a valid `EGLConfig`. Any command taking an `EGLConfig` parameter may generate this error.

`EGL_BAD_CURRENT_SURFACE`
: The current surface of the calling thread is a window, pbuffer, or pixmap that is no longer valid.

`EGL_BAD_DISPLAY`
: An `EGLDisplay` argument does not name a valid `EGLDisplay`. Any command taking an `EGLDisplay` parameter may generate this error.

`EGL_BAD_SURFACE`
: An `EGLSurface` argument does not name a valid surface (window, pbuffer, or pixmap) configured for rendering. Any command taking an `EGLSurface` parameter may generate this error.

`EGL_BAD_MATCH`
: Arguments are inconsistent; for example, an otherwise valid context requires buffers (e.g. depth or stencil) not allocated by an otherwise valid surface.

`EGL_BAD_PARAMETER`
: One or more argument values are invalid. Any command taking parameters may generate this error.

`EGL_BAD_NATIVE_PIXMAP`
: An `EGLNativePixmapType` argument does not refer to a valid native

pixmap. Any command taking an EGLNativePixmapType parameter may generate this error.

EGL_BAD_NATIVE_WINDOW
An EGLNativeWindowType argument does not refer to a valid native window. Any command taking an EGLNativeWindowType parameter may generate this error.

EGL_CONTEXT_LOST
A power management event has occurred. The application must destroy all contexts and reinitialise client API state and objects to continue rendering, as described in section 2.6. Any command may generate this error.

When there is no status to return (in other words, when **eglGetError** is called as the first EGL call in a thread, or immediately after calling **eglReleaseThread**), EGL_SUCCESS will be returned.

3.1.0.1 Generic Errors Are Not Described Repeatedly

Some specific error codes that may be generated by a failed EGL function, and their meanings, are described together with each function. However, not all possible errors are described with each function. Errors whose meanings are identical across many functions (such as returning EGL_BAD_DISPLAY or EGL_NOT_INITIALIZED for an unsuitable EGLDisplay argument) may not be described repeatedly. Some of the error codes above describe a class of commands which may generate them. Such errors are not necessarily described repeatedly together with each such command in the class.

3.1.0.2 Parameter Validation

EGL normally checks the validity of objects passed into it, but detecting invalid native objects (pixmaps, windows, and displays) may not always be possible. Specifying such invalid handles may result in undefined behavior, although implementations should generate EGL_BAD_NATIVE_PIXMAP and EGL_BAD_NATIVE_WINDOW errors if possible.

3.2 Initialization

A display can be obtained by calling

```
EGLDisplay eglGetDisplay(EGLNativeDisplayType
  display_id);
```

The type and format of *display_id* are implementation-specific, and it describes a specific display provided by the system EGL is running on. For example, an EGL implementation under X windows could define *display_id* to be an X `Display`, while an implementation under Microsoft Windows could define *display_id* to be a Windows Device Context. If *display_id* is `EGL_DEFAULT_DISPLAY`, a *default display* is returned. Multiple calls made to **eglGetDisplay** with the same *display_id* will all return the same `EGLDisplay` handle.

If no display matching *display_id* is available, `EGL_NO_DISPLAY` is returned; no error condition is raised in this case.

EGL may be initialized on a display by calling

```
EGLBoolean eglInitialize(EGLDisplay dpy, EGLint
    *major, EGLint *minor);
```

`EGL_TRUE` is returned on success, and *major* and *minor* are updated with the major and minor version numbers of the EGL implementation (for example, in an EGL 1.2 implementation, the values of **major* and **minor* would be 1 and 2, respectively). *major* and *minor* are not updated if they are specified as `NULL`.

`EGL_FALSE` is returned on failure and *major* and *minor* are not updated. An `EGL_BAD_DISPLAY` error is generated if the *dpy* argument does not refer to a valid `EGLDisplay`. An `EGL_NOT_INITIALIZED` error is generated if EGL cannot be initialized for an otherwise valid *dpy*.

Initializing an already-initialized display is allowed, but the only effect of such a call is to return `EGL_TRUE` and update the EGL version numbers. An initialized display may be used from other threads in the same address space without being initialized again in those threads.

To release resources associated with use of EGL and client APIs on a display, call

```
EGLBoolean eglTerminate(EGLDisplay dpy);
```

Termination marks **all** EGL-specific resources, such as contexts and surfaces, associated with the specified display for deletion. Handles to all such resources are invalid as soon as **eglTerminate** returns, but the *dpy* handle itself remains valid. Passing such handles to any other EGL command will generate `EGL_BAD_SURFACE` or `EGL_BAD_CONTEXT` errors. Applications should not try to perform useful work with such resources following **eglTerminate**; only **eglMakeCurrent** or **eglReleaseThread** should be called, to complete deletion of these resources. [2]

[2] Immediately invalidating handles is a subtle behavior change. Prior to the January 13, 2009 release of the EGL 1.4 Specification, handles remained valid so long as the underlying surface was current. In the September 18, 2010 release, handle invalidation was explicitly extended to all EGL resources associated with *dpy*, not just contexts and surfaces.

If contexts or surfaces created with respect to *dpy* are *current* (see section 3.7.3) to any thread, then they are not actually destroyed while they remain current. Such contexts and surfaces will be destroyed as soon as **eglReleaseThread** is called from the thread they are bound to, or **eglMakeCurrent** is called from that thread with the current rendering API (see section 3.7) set such that the current context is affected. Use of bound contexts and surfaces (that is, continuing to issue commands to a bound client API context) will not result in interruption or termination of applications, but rendering results are undefined, and client APIs may generate errors.

eglTerminate returns `EGL_TRUE` on success.

If the *dpy* argument does not refer to a valid `EGLDisplay`, `EGL_FALSE` is returned, and an `EGL_BAD_DISPLAY` error is generated.

Termination of a display that has already been terminated, or has not yet been initialized, is allowed, but the only effect of such a call is to return `EGL_TRUE`, since there are no EGL resources associated with the display to release. A terminated display may be re-initialized by calling **eglInitialize** again. When re-initializing a terminated display, resources which were marked for deletion as a result of the earlier termination remain so marked, and handles which previously referred to them remain invalid

At any point a display may either be initialized or uninitialized. All displays start out uninitialized. A display becomes initialized after **eglInitialize** is successfully called on it. A display becomes uninitialized after **eglTerminate** is successfully called on it. An uninitialized display may be passed to the functions **eglInitialize**, **eglTerminate**, and in some cases **eglMakeCurrent**. All other EGL functions which take a *display* argument will fail and generate an `EGL_NOT_INITIALIZED` error when passed a valid but uninitialized display. [3]

3.3 EGL Versioning

```
const char *eglQueryString(EGLDisplay dpy, EGLint
    name);
```

eglQueryString returns a pointer to a static, zero-terminated string describing some aspect of the EGL implementation running on the specified display. *name* may be one of `EGL_CLIENT_APIS`, `EGL_EXTENSIONS`, `EGL_VENDOR`, or `EGL_VERSION`.

[3] Note that once an `EGLDisplay` is created, the handle will necessarily remain valid for the lifetime of the application, although the corresponding display may be repeatedly initialized and terminated.

The EGL_CLIENT_APIS string describes which client rendering APIs are supported. It is zero-terminated and contains a space-separated list of API names, which must include at least one of ``OpenGL'', ``OpenGL_ES'' or ``OpenVG''.

The EGL_EXTENSIONS string describes which EGL extensions are supported by the EGL implementation running on the specified display. The string is zero-terminated and contains a space-separated list of extension names; extension names themselves do not contain spaces. If there are no extensions to EGL, then the empty string is returned.

The format and contents of the EGL_VENDOR string is implementation dependent.

The format of the EGL_VERSION string is:

<major_version.minor_version><space><vendor_specific_info>

Both the major and minor portions of the version number are numeric. Their values must match the *major* and *minor* values returned by **eglInitialize** (see section 3.2). The vendor-specific information is optional; if present, its format and contents are implementation specific.

On failure, NULL is returned. An EGL_NOT_INITIALIZED error is generated if EGL is not initialized for *dpy*. An EGL_BAD_PARAMETER error is generated if *name* is not one of the values described above.

3.4 Configuration Management

An EGLConfig describes the format, type and size of the color buffers and ancillary buffers for an EGLSurface. If the EGLSurface is a window, then the EGLConfig describing it may have an associated native *visual type*.

Names of EGLConfig attributes are shown in Table 3.1. These names may be passed to **eglChooseConfig** to specify required attribute properties.

EGL_CONFIG_ID is a unique integer identifying different EGLConfigs. Configuration IDs must be small positive integers starting at 1 and ID assignment should be compact; that is, if there are N EGLConfigs defined by the EGL implementation, their configuration IDs should be in the range $[1, N]$. Small gaps in the sequence are allowed, but should only occur when removing configurations defined in previous revisions of an EGL implementation.

Buffer Descriptions and Attributes

The various buffers that may be contained by an EGLSurface, and the EGLConfig attributes controlling their creation, are described below. Attribute

Attribute	Type	Notes
EGL_BUFFER_SIZE	integer	total color component bits in the color buffer
EGL_RED_SIZE	integer	bits of Red in the color buffer
EGL_GREEN_SIZE	integer	bits of Green in the color buffer
EGL_BLUE_SIZE	integer	bits of Blue in the color buffer
EGL_LUMINANCE_SIZE	integer	bits of Luminance in the color buffer
EGL_ALPHA_SIZE	integer	bits of Alpha in the color buffer
EGL_ALPHA_MASK_SIZE	integer	bits of Alpha Mask in the mask buffer
EGL_BIND_TO_TEXTURE_RGB	boolean	True if bindable to RGB textures.
EGL_BIND_TO_TEXTURE_RGBA	boolean	True if bindable to RGBA textures.
EGL_COLOR_BUFFER_TYPE	enum	color buffer type
EGL_CONFIG_CAVEAT	enum	any caveats for the configuration
EGL_CONFIG_ID	integer	unique EGLConfig identifier
EGL_CONFORMANT	bitmask	whether contexts created with this config are conformant
EGL_DEPTH_SIZE	integer	bits of Z in the depth buffer
EGL_LEVEL	integer	frame buffer level
EGL_MAX_PBUFFER_WIDTH	integer	maximum width of pbuffer
EGL_MAX_PBUFFER_HEIGHT	integer	maximum height of pbuffer
EGL_MAX_PBUFFER_PIXELS	integer	maximum size of pbuffer
EGL_MAX_SWAP_INTERVAL	integer	maximum swap interval
EGL_MIN_SWAP_INTERVAL	integer	minimum swap interval
EGL_NATIVE_RENDERABLE	boolean	EGL_TRUE if native rendering APIs can render to surface
EGL_NATIVE_VISUAL_ID	integer	handle of corresponding native visual
EGL_NATIVE_VISUAL_TYPE	integer	native visual type of the associated visual
EGL_RENDERABLE_TYPE	bitmask	which client APIs are supported
EGL_SAMPLE_BUFFERS	integer	number of multisample buffers
EGL_SAMPLES	integer	number of samples per pixel
EGL_STENCIL_SIZE	integer	bits of Stencil in the stencil buffer
EGL_SURFACE_TYPE	bitmask	which types of EGL surfaces are supported.
EGL_TRANSPARENT_TYPE	enum	type of transparency supported
EGL_TRANSPARENT_RED_VALUE	integer	transparent red value
EGL_TRANSPARENT_GREEN_VALUE	integer	transparent green value
EGL_TRANSPARENT_BLUE_VALUE	integer	transparent blue value

Table 3.1: EGLConfig attributes.

values include the *depth* of these buffers, expressed in bits/pixel component. If the depth of a buffer in an EGLConfig is zero, then an EGLSurface created with respect to that EGLConfig will not contain the corresponding buffer.

Not all buffers are used or required by all client APIs . To conserve resources, implementations may delay creation of buffers until they are needed by EGL or a client API . For example, if an EGLConfig describes an alpha mask buffer with depth greater than zero, that buffer need not be allocated by a surface until an OpenVG context is bound to that surface.

The Color Buffer

The *color buffer* contains pixel color values, and is shared by all client APIs rendering to a surface.

EGL_COLOR_BUFFER_TYPE indicates the color buffer type, and must be either EGL_RGB_BUFFER for an RGB color buffer, or EGL_LUMINANCE_BUFFER for a luminance color buffer. For an RGB buffer, EGL_RED_SIZE, EGL_GREEN_SIZE, EGL_BLUE_SIZE must be non-zero, and EGL_LUMINANCE_SIZE must be zero. For a luminance buffer, EGL_RED_SIZE, EGL_GREEN_SIZE, EGL_BLUE_SIZE must be zero, and EGL_LUMINANCE_SIZE must be non-zero. For both RGB and luminance color buffers, EGL_ALPHA_SIZE may be zero or non-zero (the latter indicates the existence of a *destination alpha* buffer).

If OpenGL or OpenGL ES rendering is supported for a luminance color buffer (as described by the value of the EGL_RENDERABLE_TYPE attribute, described below), it is treated as RGB rendering with the value of GL_RED_BITS equal to EGL_LUMINANCE_SIZE and the values of GL_GREEN_BITS and GL_BLUE_BITS equal to zero. The red component of fragments is written to the luminance channel of the color buffer, the green and blue components are discarded, and the alpha component is written to the alpha channel of the color buffer (if present).

EGL_BUFFER_SIZE gives the total of the color component bits of the color buffer[4] For an RGB color buffer, the total is the sum of EGL_RED_SIZE, EGL_GREEN_SIZE, EGL_BLUE_SIZE, and EGL_ALPHA_SIZE. For a luminance color buffer, the total is the sum of EGL_LUMINANCE_SIZE and EGL_ALPHA_SIZE.

The Alpha Mask Buffer

The *alpha mask buffer* is used only by OpenVG . EGL_ALPHA_MASK_SIZE indicates the depth of this buffer.

[4]The value of EGL_BUFFER_SIZE does not include any padding bits that may be present in the pixel format, nor does it account for any alignment or padding constraints of surfaces, so it cannot be reliably used to compute the memory consumed by a surface. No such query exists in EGL 1.4.

The Depth Buffer

The *depth buffer* is shared by OpenGL and OpenGL ES . It contains fragment depth (Z) information generated during rasterization. `EGL_DEPTH_SIZE` indicates the depth of this buffer in bits.

The Stencil Buffer

The *stencil buffer* is shared by OpenGL and OpenGL ES . It contains fragment stencil information generated during rasterization. `EGL_STENCIL_SIZE` indicates the depth of this buffer in bits.

The Multisample Buffer

The *multisample buffer* may be shared by OpenGL , OpenGL ES and OpenVG , although such sharing cannot be guaranteed (see comments at the end of section 3.7.3.1 for more information about sharing the multisample buffer). It contains multisample information (color values, and possibly stencil and depth values) generated by multisample rasterization. The format of the multisample buffer is not specified, and its contents are not directly accessible. Only the existence of the multisample buffer, together with the number of samples it contains, are exposed by EGL.

Operations such as posting a surface with **eglSwapBuffers** (see section 3.9.1, copying a surface with **eglCopyBuffers** (see section 3.9.2), reading from the color buffer using client API commands, and binding a client API context to a surface (see section 3.7.3), may cause *resolution* of the multisample buffer to the color buffer.

Multisample resolution combines and filters per-sample information in the multisample buffer to create per-pixel colors stored in the color buffer. The details of this filtering process are normally chosen by the implementation, but under some circumstances may be controlled on a per-surface basis using **eglSurfaceAttrib** (see section 3.5.6).

`EGL_SAMPLE_BUFFERS` indicates the number of multisample buffers, which must be zero or one. `EGL_SAMPLES` gives the number of samples per pixel; if `EGL_SAMPLE_BUFFERS` is zero, then `EGL_SAMPLES` will also be zero. If `EGL_SAMPLE_BUFFERS` is one, then the number of color, depth, and stencil bits for each sample in the multisample buffer are as specified by the `EGL_*_SIZE` attributes.

There are no single-sample depth or stencil buffers for a multisample `EGLConfig`; the only depth and stencil buffers are those in the multisample buffer. If the color samples in the multisample buffer store fewer bits than are

EGL Token Name	Description
EGL_WINDOW_BIT	EGLConfig supports windows
EGL_PIXMAP_BIT	EGLConfig supports pixmaps
EGL_PBUFFER_BIT	EGLConfig supports pbuffers
EGL_MULTISAMPLE_RESOLVE_BOX_BIT	EGLConfig supports box filtered multisample resolve
EGL_SWAP_BEHAVIOR_PRESERVED_BIT	EGLConfig supports setting swap behavior for color buffers
EGL_VG_COLORSPACE_LINEAR_BIT	EGLConfig supports OpenVG rendering in linear colorspace
EGL_VG_ALPHA_FORMAT_PRE_BIT	EGLConfig supports OpenVG rendering with premultiplied alpha

Table 3.2: Types of surfaces supported by an EGLConfig

stored in the color buffers, this fact will not be reported accurately. Presumably a compression scheme is being employed, and is expected to maintain an aggregate resolution equal to that of the color buffers.

Other EGLConfig Attribute Descriptions

EGL_SURFACE_TYPE is a mask indicating capabilities of surfaces that can be created with the corresponding EGLConfig (the config is said to *support* these surface types). The valid bit settings are shown in Table 3.2.

For example, an EGLConfig for which the value of the EGL_SURFACE_TYPE attribute is

EGL_WINDOW_BIT | EGL_PIXMAP_BIT | EGL_PBUFFER_BIT

can be used to create any type of EGL surface, while an EGLConfig for which this attribute value is EGL_WINDOW_BIT cannot be used to create a pbuffer or pixmap.

If EGL_MULTISAMPLE_RESOLVE_BOX_BIT is set in EGL_SURFACE_TYPE, then the EGL_MULTISAMPLE_RESOLVE attribute of a surface can be specified as a box filter with **eglSurfaceAttrib** (see section 3.5.6).

If EGL_SWAP_BEHAVIOR_PRESERVED_BIT is set in EGL_SURFACE_TYPE, then the EGL_SWAP_BEHAVIOR attribute of a surface can be specified to preserve color buffer contents using **eglSurfaceAttrib** (see section 3.5.6).

If EGL_VG_COLORSPACE_LINEAR_BIT is set in EGL_SURFACE_TYPE, then the EGL_VG_COLORSPACE attribute may be set to EGL_VG_COLORSPACE_LINEAR when creating a window, pixmap, or pbuffer surface (see section 3.5).

EGL Token Name	Client API and Version Supported
EGL_OPENGL_BIT	OpenGL
EGL_OPENGL_ES_BIT	OpenGL ES 1.x
EGL_OPENGL_ES2_BIT	OpenGL ES 2.x
EGL_OPENVG_BIT	OpenVG 1.x

Table 3.3: Types of client APIs supported by an EGLConfig

If EGL_VG_ALPHA_FORMAT_PRE_BIT is set in EGL_SURFACE_TYPE, then the EGL_VG_ALPHA_FORMAT attribute may be set to EGL_VG_ALPHA_FORMAT_PRE when creating a window, pixmap, or pbuffer surface (see section 3.5).

EGL_RENDERABLE_TYPE is a mask indicating which client APIs can render into a surface created with respect to an EGLConfig. The valid bit settings are shown in Table 3.3.

Creation of a client API context based on an EGLConfig will fail unless the EGLConfig's EGL_RENDERABLE_TYPE attribute include the bit corresponding to that API and version.

EGL_NATIVE_RENDERABLE is an EGLBoolean indicating whether the native window system can be used to render into a surface created with the EGLConfig. Constraints on native rendering are discussed in more detail in sections 2.2.2 and 2.2.3.

If an EGLConfig supports windows then it may have an associated native visual. EGL_NATIVE_VISUAL_ID specifies an identifier for this visual, and EGL_NATIVE_VISUAL_TYPE specifies its type. If an EGLConfig does not support windows, or if there is no associated native visual type, then querying EGL_NATIVE_VISUAL_ID will return 0 and querying EGL_NATIVE_VISUAL_TYPE will return EGL_NONE.

The interpretation of the native visual identifier and type is platform-dependent. For example, if the native window system is X, then the identifier will be the XID of an X Visual.

The EGL_CONFIG_CAVEAT attribute may be set to one of the following values: EGL_NONE, EGL_SLOW_CONFIG or EGL_NON_CONFORMANT_CONFIG. If the attribute is set to EGL_NONE then the configuration has no caveats; if it is set to EGL_SLOW_CONFIG then rendering to a surface with this configuration may run at reduced performance (for example, the hardware may not support the color buffer depths described by the configuration); if it is set to EGL_NON_CONFORMANT_CONFIG then rendering to a surface with this configuration will not pass the required OpenGL ES conformance tests (note that EGL_NON_CONFORMANT_CONFIG

is obsolete, and the same information can be obtained from the EGL_CONFORMANT attribute on a per-client-API basis, not just for OpenGL ES).

API conformance requires that a set of EGLConfigs supporting certain defined minimum attributes (such as the number, type, and depth of supported buffers) be supplied by any conformant implementation. Those requirements are documented only in the conformance specifications for client APIs .

EGL_CONFORMANT is a mask indicating if a client API context created with respect to the corresponding EGLConfig will pass the required conformance tests for that API. The valid bit settings are the same as for EGL_RENDERABLE_TYPE, as defined in table 3.3, but the presence or absence of each client API bit determines whether the corresponding context will be conformant or non-conformant. [5]

EGL_LEVEL is the framebuffer overlay or underlay level in which an EGLSurface created with **eglCreateWindowSurface** will be placed. Level zero indicates the default layer. The behavior of windows placed in overlay and underlay levels depends on the underlying native window system.

EGL_TRANSPARENT_TYPE indicates whether or not a configuration supports transparency. If the attribute is set to EGL_NONE then windows created with the EGLConfig will not have any transparent pixels. If the attribute is EGL_TRANSPARENT_RGB, then the EGLConfig supports transparency; a transparent pixel will be drawn when the red, green and blue values which are read from the framebuffer are equal to EGL_TRANSPARENT_RED_VALUE, EGL_TRANSPARENT_GREEN_VALUE and EGL_TRANSPARENT_BLUE_VALUE, respectively.

If EGL_TRANSPARENT_TYPE is EGL_NONE, then the values for EGL_TRANSPARENT_RED_VALUE, EGL_TRANSPARENT_GREEN_VALUE, and EGL_TRANSPARENT_BLUE_VALUE are undefined. Otherwise, they are interpreted as integer framebuffer values between 0 and the maximum framebuffer value for the component. For example, EGL_TRANSPARENT_RED_VALUE will range between 0 and $2^{\texttt{EGL\_RED\_SIZE}} - 1$.

EGL_MAX_PBUFFER_WIDTH and EGL_MAX_PBUFFER_HEIGHT indicate the maximum width and height that can be passed into **eglCreatePbufferSurface**, and EGL_MAX_PBUFFER_PIXELS indicates the maximum number of pixels (width times height) for a pbuffer surface. Note that an implementation may return a value for EGL_MAX_PBUFFER_PIXELS that is less than the maximum width times the maximum height. The value for EGL_MAX_PBUFFER_PIXELS is static and assumes that no other pbuffers or native resources are contending for the framebuffer memory. Thus it may not be possible to allocate a pbuffer of the size given by

[5] Most EGLConfigs should be conformant for all supported client APIs . Conformance requirements limit the number of non-conformant configs that an implementation can define.

EGL_MAX_PBUFFER_PIXELS.

EGL_MAX_SWAP_INTERVAL is the maximum value that can be passed to **eglSwapInterval**, and indicates the number of swap intervals that will elapse before a buffer swap takes place after calling **eglSwapBuffers**. Larger values will be silently clamped to this value.

EGL_MIN_SWAP_INTERVAL is the minimum value that can be passed to **eglSwapInterval**, and indicates the number of swap intervals that will elapse before a buffer swap takes place after calling **eglSwapBuffers**. Smaller values will be silently clamped to this value.

EGL_BIND_TO_TEXTURE_RGB and EGL_BIND_TO_TEXTURE_RGBA are booleans indicating whether the color buffers of a pbuffer created with the EGLConfig can be bound to a OpenGL ES RGB or RGBA texture respectively. Currently only pbuffers can be bound as textures, so these attributes may only be EGL_TRUE if the value of the EGL_SURFACE_TYPE attribute includes EGL_PBUFFER_BIT. It is possible to bind a RGBA visual to a RGB texture, in which case the values in the alpha component of the visual are ignored when the color buffer is used as a RGB texture.

Implementations may choose not to support EGL_BIND_TO_TEXTURE_RGB for RGBA visuals.

Texture binding to OpenGL textures is not supported.

3.4.1 Querying Configurations

Use

```
EGLBoolean eglGetConfigs(EGLDisplay dpy,
    EGLConfig *configs, EGLint config_size,
    EGLint *num_config);
```

to get the list of all EGLConfigs that are available on the specified display. *configs* is a pointer to a buffer containing *config_size* elements. On success, EGL_TRUE is returned. The number of configurations is returned in *num_config*, and elements 0 through $num\_config - 1$ of *configs* are filled in with the valid EGLConfigs. No more than *config_size* EGLConfigs will be returned even if more are available on the specified display. However, if **eglGetConfigs** is called with *configs* = NULL, then no configurations are returned, but the total number of configurations available will be returned in *num_config*.

On failure, EGL_FALSE is returned. An EGL_NOT_INITIALIZED error is generated if EGL is not initialized on *dpy*. An EGL_BAD_PARAMETER error is generated if *num_config* is NULL.

Use

```
EGLBoolean eglChooseConfig(EGLDisplay dpy, const
    EGLint *attrib_list, EGLConfig *configs,
    EGLint config_size, EGLint *num_config);
```

to get EGLConfigs that match a list of attributes. The return value and the meaning of *configs*, *config_size*, and *num_config* are the same as for **eglGetConfigs**. However, only configurations matching *attrib_list*, as discussed below, will be returned.

On failure, EGL_FALSE is returned. An EGL_BAD_ATTRIBUTE error is generated if *attrib_list* contains an undefined EGL attribute or an attribute value that is unrecognized or out of range.

All attribute names in *attrib_list* are immediately followed by the corresponding desired value. The list is terminated with EGL_NONE. If an attribute is not specified in *attrib_list*, then the default value (listed in Table 3.4) is used (it is said to be specified implicitly). If EGL_DONT_CARE is specified as an attribute value, then the attribute will not be checked. EGL_DONT_CARE may be specified for all attributes except EGL_LEVEL and EGL_MATCH_NATIVE_PIXMAP. If *attrib_list* is NULL or empty (first attribute is EGL_NONE), then selection and sorting of EGLConfigs is done according to the default criteria in Tables 3.4 and 3.1, as described below in sections 3.4.1.1 and 3.4.1.2.

3.4.1.1 Selection of EGLConfigs

Attributes are matched in an attribute-specific manner, as shown in the "Selection Critera" column of table 3.4. The criteria listed in the table have the following meanings:

AtLeast Only EGLConfigs with an attribute value that meets or exceeds the specified value are selected.

Exact Only EGLConfigs whose attribute value equals the specified value are matched.

Mask Only EGLConfigs for which the bits set in the attribute value include all the bits that are set in the specified value are selected (additional bits might be set in the attribute value)[6].

Special As described for the specific attribute.

[6] Some readers have found this phrasing confusing. Another way to think of it to say that any bits present in the mask attribute must also be present in the EGLConfig attribute. Thus, setting a mask attribute value of zero means that all configs will match that value.

Some of the attributes must match the specified value exactly; others, such as EGL_RED_SIZE, must meet or exceed the specified minimum values.

To retrieve an EGLConfig given its unique integer ID, use the EGL_CONFIG_ID attribute. When EGL_CONFIG_ID is specified, all other attributes are ignored, and only the EGLConfig with the given ID is returned.

If EGL_MAX_PBUFFER_WIDTH, EGL_MAX_PBUFFER_HEIGHT, EGL_MAX_PBUFFER_PIXELS, or EGL_NATIVE_VISUAL_ID are specified in *attrib_list*, then they are ignored (however, if present, these attributes must still be followed by an attribute value in *attrib_list*). If EGL_SURFACE_TYPE is specified in *attrib_list* and the mask that follows does not have EGL_WINDOW_BIT set, or if there are no native visual types, then the EGL_NATIVE_VISUAL_TYPE attribute is ignored.

If EGL_TRANSPARENT_TYPE is set to EGL_NONE in *attrib_list*, then the EGL_TRANSPARENT_RED_VALUE, EGL_TRANSPARENT_GREEN_VALUE, and EGL_TRANSPARENT_BLUE_VALUE attributes are ignored.

If EGL_MATCH_NATIVE_PIXMAP is specified in *attrib_list*, it must be followed by an attribute value which is the handle of a valid native pixmap. Only EGLConfigs which support rendering to that pixmap will match this attribute [7].

If no EGLConfig matching the attribute list exists, then the call succeeds, but *num_config* is set to 0.

3.4.1.2 Sorting of EGLConfigs

If more than one matching EGLConfig is found, then a list of EGLConfigs is returned. The list is sorted by proceeding in ascending order of the "Sort Priority" column of table 3.4. That is, configurations that are not ordered by a lower numbered rule are sorted by the next higher numbered rule.

Sorting for each rule is either numerically *Smaller* or *Larger* as described in the "Sort Order" column, or a *Special* sort order as described for each sort rule below:

1. *Special*: by EGL_CONFIG_CAVEAT where the precedence is EGL_NONE, EGL_SLOW_CONFIG, EGL_NON_CONFORMANT_CONFIG.

2. *Special*: by EGL_COLOR_BUFFER_TYPE where the precedence is EGL_RGB_BUFFER, EGL_LUMINANCE_BUFFER.

3. *Special*: by larger *total* number of color bits (for an RGB color buffer, this is the sum of EGL_RED_SIZE, EGL_GREEN_SIZE, EGL_BLUE_SIZE,

[7] The special match criteria for EGL_MATCH_NATIVE_PIXMAP was introduced due to the difficulty of determining an EGLConfig equivalent to a native pixmap using only color component depths.

Attribute	Default	Selection Criteria	Sort Order	Sort Priority
EGL_BUFFER_SIZE	0	*AtLeast*	*Smaller*	4
EGL_RED_SIZE	0	*AtLeast*	*Special*	3
EGL_GREEN_SIZE	0	*AtLeast*	*Special*	3
EGL_BLUE_SIZE	0	*AtLeast*	*Special*	3
EGL_LUMINANCE_SIZE	0	*AtLeast*	*Special*	3
EGL_ALPHA_SIZE	0	*AtLeast*	*Special*	3
EGL_ALPHA_MASK_SIZE	0	*AtLeast*	*Smaller*	9
EGL_BIND_TO_TEXTURE_RGB	EGL_DONT_CARE	*Exact*	*None*	
EGL_BIND_TO_TEXTURE_RGBA	EGL_DONT_CARE	*Exact*	*None*	
EGL_COLOR_BUFFER_TYPE	EGL_RGB_BUFFER	*Exact*	*Special*	2
EGL_CONFIG_CAVEAT	EGL_DONT_CARE	*Exact*	*Special*	1
EGL_CONFIG_ID	EGL_DONT_CARE	*Special*	*Smaller*	11 (last)
EGL_CONFORMANT	0	*Mask*	*None*	
EGL_DEPTH_SIZE	0	*AtLeast*	*Smaller*	7
EGL_LEVEL	0	*Exact*	*None*	
EGL_MATCH_NATIVE_PIXMAP	EGL_NONE	*Special*	*None*	
EGL_MAX_SWAP_INTERVAL	EGL_DONT_CARE	*Exact*	*None*	
EGL_MIN_SWAP_INTERVAL	EGL_DONT_CARE	*Exact*	*None*	
EGL_NATIVE_RENDERABLE	EGL_DONT_CARE	*Exact*	*None*	
EGL_NATIVE_VISUAL_TYPE	EGL_DONT_CARE	*Exact*	*Special*	10
EGL_RENDERABLE_TYPE	EGL_OPENGL_ES_BIT	*Mask*	*None*	
EGL_SAMPLE_BUFFERS	0	*AtLeast*	*Smaller*	5
EGL_SAMPLES	0	*AtLeast*	*Smaller*	6
EGL_STENCIL_SIZE	0	*AtLeast*	*Smaller*	8
EGL_SURFACE_TYPE	EGL_WINDOW_BIT	*Mask*	*None*	
EGL_TRANSPARENT_TYPE	EGL_NONE	*Exact*	*None*	
EGL_TRANSPARENT_RED_VALUE	EGL_DONT_CARE	*Exact*	*None*	
EGL_TRANSPARENT_GREEN_VALUE	EGL_DONT_CARE	*Exact*	*None*	
EGL_TRANSPARENT_BLUE_VALUE	EGL_DONT_CARE	*Exact*	*None*	

Table 3.4: Default values and match criteria for EGLConfig attributes.

and EGL_ALPHA_SIZE; for a luminance color buffer, the sum of EGL_LUMINANCE_SIZE and EGL_ALPHA_SIZE) [8] If the requested number of bits in *attrib_list* for a particular color component is 0 or EGL_DONT_CARE, then the number of bits for that component is not considered.

4. *Smaller* EGL_BUFFER_SIZE.
5. *Smaller* EGL_SAMPLE_BUFFERS.
6. *Smaller* EGL_SAMPLES.
7. *Smaller* EGL_DEPTH_SIZE.
8. *Smaller* EGL_STENCIL_SIZE.
9. *Smaller* EGL_ALPHA_MASK_SIZE.
10. *Special*: by EGL_NATIVE_VISUAL_TYPE (the actual sort order is implementation-defined, depending on the meaning of native visual types).
11. *Smaller* EGL_CONFIG_ID (this is always the last sorting rule, and guarantees a unique ordering).

EGLConfigs are not sorted with respect to the parameters EGL_BIND_TO_TEXTURE_RGB, EGL_BIND_TO_TEXTURE_RGBA, EGL_CONFORMANT, EGL_LEVEL, EGL_NATIVE_RENDERABLE, EGL_MAX_SWAP_INTERVAL, EGL_MIN_SWAP_INTERVAL, EGL_RENDERABLE_TYPE, EGL_SURFACE_TYPE, EGL_TRANSPARENT_TYPE, EGL_TRANSPARENT_RED_VALUE, EGL_TRANSPARENT_GREEN_VALUE, and EGL_TRANSPARENT_BLUE_VALUE.

3.4.2 Lifetime of Configurations

Configuration handles (EGLConfigs) returned by **eglGetConfigs** and **eglChooseConfig** remain valid so long as the EGLDisplay from which the handles were obtained is not terminated. Implementations supporting a large number of different configurations, where it might be burdensome to instantiate data structures for each configuration so queried (but never used), may choose to return handles

[8] This rule places configs with deeper color buffers first in the list returned by **eglChooseConfig**. Applications may find this counterintuitive if they expect configs with smaller buffer sizes to be returned first. For example, if an implementation has two configs with RGBA depths of 8888 and 5650, and the application specifies RGBA sizes of 1110, the 8888 config will be returned first. To avoid this rule altogether, specify 0 or EGL_DONT_CARE for each component size. In this case this rule will be ignored, and rule 4, which prefers configs with a smaller EGL_BUFFER_SIZE, will apply.

encoding sufficient information to instantiate the corresponding configurations dynamically, when needed to create EGL resources or query configuration attributes.

3.4.3 Querying Configuration Attributes

To get the value of an EGLConfig attribute, use

```
EGLBoolean eglGetConfigAttrib(EGLDisplay dpy,
    EGLConfig config, EGLint attribute, EGLint
    *value);
```

If **eglGetConfigAttrib** succeeds then it returns EGL_TRUE and the value for the specified attribute is returned in *value*. Otherwise it returns EGL_FALSE. If *attribute* is not a valid attribute then EGL_BAD_ATTRIBUTE is generated.

attribute may be any of the EGL attributes listed in tables 3.1 and 3.4, with the exception of EGL_MATCH_NATIVE_PIXMAP.

3.5 Rendering Surfaces

3.5.1 Creating On-Screen Rendering Surfaces

To create an on-screen rendering surface, first create a native platform window whose pixel format corresponds to the format, type, and size of the color buffers required by *config*. On some implementations, the pixel format of the native window must match that of the EGLConfig[9]. Other implementations may allow any *win* and *config* to correspond, even if their formats differ[10].

Using the platform-specific type EGLNativeWindowType, which is the type of a handle to that native window, then call:

```
EGLSurface eglCreateWindowSurface(EGLDisplay dpy,
    EGLConfig config, EGLNativeWindowType win,
    const EGLint *attrib_list);
```

[9] The exact definition of matching formats is implementation-dependent, but usually means the color format (which of R, G, B, and A components are present), type (EGL expects unsigned integer color components), and size (number of bits/component) are the same. For example, X11-based EGL implementations often require *win* to have an X visual ID whose format matches *config* in this fashion.

[10] It may still be desirable for *win* and *config* to have matching formats, even if the implementation does not require this. Otherwise potentially costly operations such as format conversion during **eglSwapBuffers** may be required.

eglCreateWindowSurface creates an onscreen EGLSurface and returns a handle to it. Any EGL context created with a compatible EGLConfig can be used to render into this surface.

attrib_list specifies a list of attributes for the window. The list has the same structure as described for **eglChooseConfig**. Attributes that can be specified in *attrib_list* include EGL_RENDER_BUFFER, EGL_VG_COLORSPACE, and EGL_VG_ALPHA_FORMAT.

It is possible that some platforms will define additional attributes specific to those environments, as an EGL extension.

attrib_list may be NULL or empty (first attribute is EGL_NONE), in which case all attributes assumes their default value as described below.

EGL_RENDER_BUFFER specifies which buffer should be used by default for client API rendering to the window, as described in section 2.2.2. If its value is EGL_SINGLE_BUFFER, then client APIs should render directly into the visible window. If its value is EGL_BACK_BUFFER, then all client APIs should render into the back buffer. The default value of EGL_RENDER_BUFFER is EGL_BACK_BUFFER.

Client APIs may not be able to respect the requested rendering buffer. To determine the actual buffer that a context will render to by default, call **eglQueryContext** with *attribute* EGL_RENDER_BUFFER (see section 3.7.4).

Some client APIs expose the ability to switch between rendering to the front or the back buffer. In this case **eglQueryContext** does not reflect such changes, and will report the buffer used as a render target when the context was first created, which may not be the same as the current render target for that buffer.

Some window systems may not allow rendering directly to the front buffer of a window surface. When such windows are made current to a context, the context will always have an EGL_RENDER_BUFFER attribute value of EGL_BACK_BUFFER. From the client API point of view these surfaces have only a back buffer and no front buffer, similar to pbuffer rendering (see section 2.2.2). Client APIs which generally have the ability to switch render target from back to front will not be able to do so when the window system does not allow this; from the point of view of the client API the front buffer for such windows does not exist.

EGL_VG_COLORSPACE specifies the *color space* used by OpenVG when rendering to the surface. If its value is EGL_VG_COLORSPACE_sRGB, then a non-linear, perceptually uniform color space is assumed, with a corresponding VGImageFormat of form VG_s*. If its value is EGL_VG_COLORSPACE_LINEAR, then a linear color space is assumed, with a corresponding VGImageFormat of form VG_l*. The default value of EGL_VG_COLORSPACE is EGL_VG_COLORSPACE_sRGB.

EGL_VG_ALPHA_FORMAT specifies how alpha values are interpreted by

OpenVG when rendering to the surface. If its value is EGL_VG_ALPHA_FORMAT_NONPRE, then alpha values are not premultipled. If its value is EGL_VG_ALPHA_FORMAT_PRE, then alpha values are premultiplied. The default value of EGL_VG_ALPHA_FORMAT is EGL_VG_ALPHA_FORMAT_NONPRE.

Note that the EGL_VG_COLORSPACE and EGL_VG_ALPHA_FORMAT attributes are used only by OpenVG . EGL itself, and other client APIs such as OpenGL and OpenGL ES , do not distinguish multiple colorspace models. Refer to section 11.2 of the OpenVG 1.0 specification for more information.

Similarly, the EGL_VG_ALPHA_FORMAT attribute does **not** necessarily control or affect the window system's interpretation of alpha values, even when the window system makes use of alpha to composite surfaces at display time. The window system's use and interpretation of alpha values is outside the scope of EGL. However, the preferred behavior is for window systems to ignore the value of EGL_VG_ALPHA_FORMAT when compositing window surfaces.

On failure **eglCreateWindowSurface** returns EGL_NO_SURFACE. If the pixel format of *win* does not correspond to the format, type, and size of the color buffers required by *config*, as discussed above, then an EGL_BAD_MATCH error is generated. If *config* does not support rendering to windows (the EGL_SURFACE_TYPE attribute does not contain EGL_WINDOW_BIT), an EGL_BAD_MATCH error is generated. If *config* does not support the colorspace or alpha format attributes specified in *attrib_list* (as defined for **eglCreateWindowSurface**), an EGL_BAD_MATCH error is generated. If *config* is not a valid EGLConfig, an EGL_BAD_CONFIG error is generated. If *win* is not a valid native window handle, then an EGL_BAD_NATIVE_WINDOW error should be generated. If there is already an EGLSurface associated with *win* (as a result of a previous **eglCreateWindowSurface** call), then an EGL_BAD_ALLOC error is generated. Finally, if the implementation cannot allocate resources for the new EGL window, an EGL_BAD_ALLOC error is generated.

3.5.2 Creating Off-Screen Rendering Surfaces

EGL supports off-screen rendering surfaces in pbuffers. Pbuffers differ from windows in the following ways:

1. Pbuffers are typically allocated in offscreen (non-visible) graphics memory and are intended only for accelerated offscreen rendering. Allocation can fail if there are insufficient graphics resources (implementations are not required to virtualize framebuffer memory). Clients should deallocate pbuffers when they are no longer in use, since graphics memory is often a scarce resource.

2. Pbuffers are EGL resources and have no associated native window or native window type. It may not be possible to render to pbuffers using native

rendering APIs.

To create a pbuffer, call

```
EGLSurface eglCreatePbufferSurface(EGLDisplay dpy,
   EGLConfig config, const EGLint
   *attrib_list);
```

This creates a single pbuffer surface and returns a handle to it.

attrib_list specifies a list of attributes for the pbuffer. The list has the same structure as described for **eglChooseConfig**. Attributes that can be specified in *attrib_list* include EGL_WIDTH, EGL_HEIGHT, EGL_LARGEST_PBUFFER, EGL_TEXTURE_FORMAT, EGL_TEXTURE_TARGET, EGL_MIPMAP_TEXTURE, EGL_VG_COLORSPACE, and EGL_VG_ALPHA_FORMAT.

It is possible that some platforms will define additional attributes specific to those environments, as an EGL extension.

attrib_list may be NULL or empty (first attribute is EGL_NONE), in which case all the attributes assume their default values as described below.

EGL_WIDTH and EGL_HEIGHT specify the pixel width and height of the rectangular pbuffer. If the value of EGLConfig attribute EGL_TEXTURE_FORMAT is not EGL_NO_TEXTURE, then the pbuffer width and height specify the size of the level zero texture image. The default values for EGL_WIDTH and EGL_HEIGHT are zero.

EGL_TEXTURE_FORMAT specifies the format of the OpenGL ES texture that will be created when a pbuffer is bound to a texture map. It can be set to EGL_TEXTURE_RGB, EGL_TEXTURE_RGBA, or EGL_NO_TEXTURE. The default value of EGL_TEXTURE_FORMAT is EGL_NO_TEXTURE.

EGL_TEXTURE_TARGET specifies the target for the OpenGL ES texture that will be created when the pbuffer is created with a texture format of EGL_TEXTURE_RGB or EGL_TEXTURE_RGBA. The target can be set to EGL_NO_TEXTURE or EGL_TEXTURE_2D. The default value of EGL_TEXTURE_TARGET is EGL_NO_TEXTURE.

EGL_MIPMAP_TEXTURE indicates whether storage for OpenGL ES mipmaps should be allocated. Space for mipmaps will be set aside if the attribute value is EGL_TRUE and EGL_TEXTURE_FORMAT is not EGL_NO_TEXTURE. The default value for EGL_MIPMAP_TEXTURE is EGL_FALSE.

Use EGL_LARGEST_PBUFFER to get the largest available pbuffer when the allocation of the pbuffer would otherwise fail. The width and height of the allocated pbuffer will never exceed the values of EGL_WIDTH and EGL_HEIGHT, respectively. If the pbuffer will be used as a OpenGL ES texture (i.e., the value of

EGL_TEXTURE_TARGET is EGL_TEXTURE_2D, and the value of EGL_TEXTURE_FORMAT is EGL_TEXTURE_RGB or EGL_TEXTURE_RGBA), then the aspect ratio will be preserved and the new width and height will be valid sizes for the texture target (e.g. if the underlying OpenGL ES implementation does not support non-power-of-two textures, both the width and height will be a power of 2). Use **eglQuerySurface** to retrieve the dimensions of the allocated pbuffer. The default value of EGL_LARGEST_PBUFFER is EGL_FALSE.

EGL_VG_COLORSPACE and EGL_VG_ALPHA_FORMAT have the same meaning and default values as when used with **eglCreateWindowSurface**.

The resulting pbuffer will contain color buffers and ancillary buffers as specified by *config*.

The contents of the depth and stencil buffers may not be preserved when rendering an OpenGL ES texture to the pbuffer and switching which image of the texture is rendered to (e.g., switching from rendering one mipmap level to rendering another).

On failure **eglCreatePbufferSurface** returns EGL_NO_SURFACE. If the pbuffer could not be created due to insufficient resources, then an EGL_BAD_ALLOC error is generated. If *config* is not a valid EGLConfig, an EGL_BAD_CONFIG error is generated. If the value specified for either EGL_WIDTH or EGL_HEIGHT is less than zero, an EGL_BAD_PARAMETER error is generated. If *config* does not support pbuffers, an EGL_BAD_MATCH error is generated. In addition, an EGL_BAD_MATCH error is generated if any of the following conditions are true:

- The EGL_TEXTURE_FORMAT attribute is not EGL_NO_TEXTURE, and EGL_WIDTH and/or EGL_HEIGHT specify an invalid size (e.g., the texture size is not a power of two, and the underlying OpenGL ES implementation does not support non-power-of-two textures).

- The EGL_TEXTURE_FORMAT attribute is EGL_NO_TEXTURE, and EGL_TEXTURE_TARGET is something other than EGL_NO_TEXTURE; or, EGL_TEXTURE_FORMAT is something other than EGL_NO_TEXTURE, and EGL_TEXTURE_TARGET is EGL_NO_TEXTURE.

Finally, an EGL_BAD_ATTRIBUTE error is generated if any of the EGL_TEXTURE_FORMAT, EGL_TEXTURE_TARGET, or EGL_MIPMAP_TEXTURE attributes are specified, but *config* does not support OpenGL ES rendering (e.g. the EGL_RENDERABLE_TYPE attribute does not include at least one of EGL_OPENGL_ES_BIT or EGL_OPENGL_ES2_BIT.

3.5.3 Binding Off-Screen Rendering Surfaces To Client Buffers

Pbuffers may also be created by binding renderable buffers created in client APIs to EGL. Currently, the only client API resources which may be bound in this fashion are OpenVG `VGImage` objects.

To bind a client API renderable buffer to a pbuffer, call

```
EGLSurface eglCreatePbufferFromClient-
    Buffer(EGLDisplay dpy, EGLenum buftype,
    EGLClientBuffer buffer, EGLConfig config,
    const EGLint *attrib_list);
```

This creates a single pbuffer surface bound to the specified *buffer* for part or all of its buffer storage, and returns a handle to it. The width and height of the pbuffer are determined by the width and height of *buffer*.

buftype specifies the type of buffer to be bound. The only allowed value of *buftype* is `EGL_OPENVG_IMAGE`.

buffer is a client API reference to the buffer to be bound. When *buftype* is `EGL_OPENVG_IMAGE`, *buffer* must be a valid `VGImage` handle, cast into the type `EGLClientBuffer`.

attrib_list specifies a list of attributes for the pbuffer. The list has the same structure as described for **eglChooseConfig**. Attributes that can be specified in *attrib_list* include `EGL_TEXTURE_FORMAT`, `EGL_TEXTURE_TARGET`, and `EGL_MIPMAP_TEXTURE`. The meaning of these attributes is as described above for **eglCreatePbufferSurface**. The `EGL_VG_COLORSPACE` and `EGL_VG_ALPHA_FORMAT` attributes of the surface are determined by the `VGImageFormat` of *buffer*.

attrib_list may be `NULL` or empty (first attribute is `EGL_NONE`), in which case all the attributes assume their default values as described above for **eglCreatePbufferSurface**.

The resulting pbuffer will contain color and ancillary buffers as specified by *config*. Buffers which are present in *buffer* (normally, just the color buffer) will be bound to EGL. Buffers which are not present in *buffer* (such as depth and stencil, if *config* includes those buffers) will be allocated by EGL in the same fashion as for a surface created with **eglCreatePbufferSurface**

On failure **eglCreatePbufferFromClientBuffer** returns `EGL_NO_SURFACE`. In addition to the errors described above for **eglCreatePbufferSurface**, **eglCreatePbufferFromClientBuffer** may fail and generate errors for the following reasons:

- If *buftype* is not a recognized client API resource type (e.g. is not `EGL_OPENVG_IMAGE`), an `EGL_BAD_PARAMETER` error is generated.

- If *buffer* is not a valid handle or name of a client API resource of the specified *buftype* in the currently bound context corresponding to that type, an EGL_BAD_PARAMETER error is generated.
- If the buffers contained in *buffer* do not correspond to a proper subset of the buffers described by *config*, and match the bit depths for those buffers specified in *config*, then an EGL_BAD_MATCH error is generated. For example, a VGImage with pixel format VG_lRGBA_8888 corresponds to an EGLConfig with EGL_RED_SIZE, EGL_GREEN_SIZE, EGL_BLUE_SIZE, and EGL_ALPHA_SIZE values of 8.
- If no context corresponding to the specified *buftype* is current, an EGL_BAD_ACCESS error is generated.
- There may be additional constraints on which types of buffers may be bound to EGL surfaces, as described in client API specifications. If those constraints are violated, then an EGL_BAD_MATCH error is generated [11].
- If *buffer* is already bound to another pbuffer, or is in use by a client API as discussed below, an EGL_BAD_ACCESS error is generated.

3.5.3.1 Lifetime and Usage of Bound Buffers

Binding client API buffers to EGL pbuffers create the possibility of race conditions, and of buffers being deleted through one API while still in use in another API. To avoid these problems, a number of constraints apply to bound client API buffers:

- Bound buffers may be used exclusively by either EGL, or the client API that originally created them.

  For example, if a VGImage is bound to a pbuffer, and that pbuffer is bound to any client API rendering context, then the VGImage may not be used as the explicit source or destination of any OpenVG operation. Errors resulting from such use are described in client API specifications.

  Similarly, while a VGImage is in use by OpenVG , the pbuffer it is bound to may not be made current to any client API context, as described in section 3.7.3.

[11] An example of such an additional constraint is an implementation which cannot support an OpenVG VGImage being bound to a pbuffer which will be used as a mipmapped OpenGL ES texture (e.g. whose EGL_MIPMAP_TEXTURE attribute is EGL_TRUE).

- Binding a buffer creates an additional reference to it, and implementations must respect outstanding references when destroying objects.

  For example, if a VGImage is bound to a pbuffer, destroying the image with **vgDestroyImage** will not free the underlying buffer, because it is still in use by EGL. However, following **vgDestroyImage** the buffer may only be referred to via the EGL pbuffer handle, since the OpenVG handle to that buffer no longer exists.

  Similarly, destroying the pbuffer with **eglDestroySurface** will not free the underlying buffer, because it is still in use by OpenVG . However, following **eglDestroySurface** the buffer may only be referred to via the OpenVG VGImage handle, since the EGL pbuffer handle no longer exists.

3.5.4 Creating Native Pixmap Rendering Surfaces

EGL also supports rendering surfaces whose color buffers are stored in native pixmaps. Pixmaps differ from windows in that they are typically allocated in off-screen (non-visible) graphics or CPU memory. Pixmaps differ from pbuffers in that they do have an associated native pixmap and native pixmap type, and it may be possible to render to pixmaps using APIs other than client APIs .

To create a pixmap rendering surface, first create a native platform pixmap, then select an EGLConfig matching the pixel format of that pixmap (calling **eglChooseConfig** with an attribute list including EGL_MATCH_NATIVE_PIXMAP returns only EGLConfigs matching the pixmap specified in the attribute list - see section 3.4.1).

Using the platform-specific type EGLNativePixmapType, which is the type of a handle to that native pixmap, then call:

```
EGLSurface eglCreatePixmapSurface(EGLDisplay dpy,
    EGLConfig config, EGLNativePixmapType
    pixmap, const EGLint *attrib_list);
```

eglCreatePixmapSurface creates an offscreen EGLSurface and returns a handle to it. Any EGL context created with a compatible EGLConfig can be used to render into this surface.

attrib_list specifies a list of attributes for the pixmap. The list has the same structure as described for **eglChooseConfig**. Attributes that can be specified in *attrib_list* include EGL_VG_COLORSPACE and EGL_VG_ALPHA_FORMAT.

It is possible that some platforms will define additional attributes specific to those environments, as an EGL extension.

attrib_list may be NULL or empty (first attribute is EGL_NONE), in which case all attributes assumes their default value.

EGL_VG_COLORSPACE and EGL_VG_ALPHA_FORMAT have the same meaning and default values as when used with **eglCreateWindowSurface**.

The resulting pixmap surface will contain color and ancillary buffers as specified by *config*. Buffers which are present in *pixmap* (normally, just the color buffer) will be bound to EGL. Buffers which are not present in *pixmap* (such as depth and stencil, if *config* includes those buffers) will be allocated by EGL in the same fashion as for a surface created with **eglCreatePbufferSurface**.

On failure **eglCreatePixmapSurface** returns EGL_NO_SURFACE. If the attributes of *pixmap* do not correspond to *config*, then an EGL_BAD_MATCH error is generated. If *config* does not support rendering to pixmaps (the EGL_SURFACE_TYPE attribute does not contain EGL_PIXMAP_BIT), an EGL_BAD_MATCH error is generated. If *config* does not support the colorspace or alpha format attributes specified in *attrib_list* (as defined for **eglCreateWindowSurface**), an EGL_BAD_MATCH error is generated. If *config* is not a valid EGLConfig, an EGL_BAD_CONFIG error is generated. If *pixmap* is not a valid native pixmap handle, then an EGL_BAD_NATIVE_PIXMAP error should be generated. If there is already an EGLSurface associated with *pixmap* (as a result of a previous **eglCreatePixmapSurface** call), then a EGL_BAD_ALLOC error is generated. Finally, if the implementation cannot allocate resources for the new EGL pixmap, an EGL_BAD_ALLOC error is generated.

3.5.5 Destroying Rendering Surfaces

An EGLSurface of any type (window, pbuffer, or pixmap) is destroyed by calling

```
EGLBoolean eglDestroySurface(EGLDisplay dpy,
    EGLSurface surface);
```

All resources associated with *surface* which were allocated by EGL are marked for deletion as soon as possible. Following **eglDestroySurface**, the surface and the handle referring to it are treated in the same fashion as a surface destroyed by **eglTerminate** (see section 3.2).

Resources associated with *surface* but not allocated by EGL, such as native windows, native pixmaps, or client API buffers, are not affected when the surface is destroyed. Only storage actually allocated by EGL is marked for deletion.

Furthermore, resources associated with a pbuffer surface are not released until all color buffers of that pbuffer bound to a OpenGL ES texture object have been released.

eglDestroySurface returns EGL_FALSE on failure. An EGL_BAD_SURFACE error is generated if *surface* is not a valid rendering surface.

3.5.6 Surface Attributes

To set an attribute for an EGLSurface, call

```
EGLBoolean eglSurfaceAttrib(EGLDisplay dpy,
    EGLSurface surface, EGLint attribute,
    EGLint value);
```

The specified *attribute* of *surface* is set to *value*. Attributes that can be specified are EGL_MIPMAP_LEVEL, EGL_MULTISAMPLE_RESOLVE, and EGL_SWAP_BEHAVIOR.

If *attribute* is EGL_MIPMAP_LEVEL, then *value* indicates which level of the OpenGL ES mipmap texture should be rendered. If the value of this attribute is outside the range of supported mipmap levels, the closest valid mipmap level is selected for rendering. The initial value of this attribute is 0.

If the value of pbuffer attribute EGL_TEXTURE_FORMAT is EGL_NO_TEXTURE, if the value of attribute EGL_TEXTURE_TARGET is EGL_NO_TEXTURE, or if *surface* is not a pbuffer, then attribute EGL_MIPMAP_LEVEL may be set, but has no effect.

If OpenGL ES rendering is not supported by *surface*, then trying to set EGL_MIPMAP_LEVEL will cause an EGL_BAD_PARAMETER error.

If *attribute* is EGL_MULTISAMPLE_RESOLVE, then *value* specifies the filter to use when resolving the multisample buffer. A *value* of EGL_MULTISAMPLE_RESOLVE_DEFAULT chooses the default implementation-defined filtering method, while EGL_MULTISAMPLE_RESOLVE_BOX chooses a one-pixel wide box filter placing equal weighting on all multisample values.

If *value* is EGL_MULTISAMPLE_RESOLVE_BOX, and the EGL_SURFACE_TYPE attribute of the EGLConfig used to create *surface* does not contain EGL_MULTISAMPLE_RESOLVE_BOX_BIT, a EGL_BAD_MATCH error is generated.

The initial value of EGL_MULTISAMPLE_RESOLVE is EGL_MULTISAMPLE_RESOLVE_DEFAULT.

If *attribute* is EGL_SWAP_BEHAVIOR, then *value* specifies the effect on the color buffer of posting a surface with **eglSwapBuffers** (see section 3.9). A *value* of EGL_BUFFER_PRESERVED indicates that color buffer contents are unaffected, while EGL_BUFFER_DESTROYED indicates that color buffer contents may be destroyed or changed by the operation.

If *value* is EGL_BUFFER_PRESERVED, and the EGL_SURFACE_TYPE attribute of the EGLConfig used to create *surface* does not contain EGL_SWAP_BEHAVIOR_PRESERVED_BIT, a EGL_BAD_MATCH error is generated.

Attribute	Type	Description
EGL_VG_ALPHA_FORMAT	enum	Alpha format for OpenVG
EGL_VG_COLORSPACE	enum	Color space for OpenVG
EGL_CONFIG_ID	integer	ID of EGLConfig surface was created with
EGL_HEIGHT	integer	Height of surface
EGL_HORIZONTAL_RESOLUTION	integer	Horizontal dot pitch
EGL_LARGEST_PBUFFER	boolean	If true, create largest pbuffer possible
EGL_MIPMAP_TEXTURE	boolean	True if texture has mipmaps
EGL_MIPMAP_LEVEL	integer	Mipmap level to render to
EGL_MULTISAMPLE_RESOLVE	enum	Multisample resolve behavior
EGL_PIXEL_ASPECT_RATIO	integer	Display aspect ratio
EGL_RENDER_BUFFER	enum	Render buffer
EGL_SWAP_BEHAVIOR	enum	Buffer swap behavior
EGL_TEXTURE_FORMAT	enum	Format of texture: RGB, RGBA, or no texture
EGL_TEXTURE_TARGET	enum	Type of texture: 2D or no texture
EGL_VERTICAL_RESOLUTION	integer	Vertical dot pitch
EGL_WIDTH	integer	Width of surface

Table 3.5: Queryable surface attributes and types.

The initial value of EGL_SWAP_BEHAVIOR is chosen by the implementation.
To query an attribute associated with an EGLSurface call:

```
EGLBoolean eglQuerySurface(EGLDisplay dpy,
    EGLSurface surface, EGLint attribute,
    EGLint *value);
```

eglQuerySurface returns in *value* the value of *attribute* for *surface*. *attribute* must be set to one of the attributes in table 3.5.

Querying EGL_CONFIG_ID returns the ID of the EGLConfig with respect to which the surface was created.

Querying EGL_LARGEST_PBUFFER for a pbuffer surface returns the same attribute value specified when the surface was created with **eglCreatePbufferSurface**. For a window or pixmap surface, the contents of *value* are not modified.

Querying EGL_WIDTH and EGL_HEIGHT returns respectively the width and height, in pixels, of the surface. For a window or pixmap surface, these values are initially equal to the width and height of the native window or pixmap with respect

to which the surface was created. If a native window is resized, the corresponding window surface will eventually be resized by the implementation to match (as discussed in section 3.9.1). If there is a discrepancy because EGL has not yet resized the window surface, the size returned by **eglQuerySurface** will always be that of the EGL surface, not the corresponding native window.

For a pbuffer, they will be the actual allocated size of the pbuffer (which may be less than the requested size if EGL_LARGEST_PBUFFER is EGL_TRUE).

Querying EGL_HORIZONTAL_RESOLUTION and EGL_VERTICAL_RESOLUTION returns respectively the horizontal and vertical dot pitch of the display on which a window surface is visible. The values returned are equal to the actual dot pitch, in pixels/meter, multiplied by the constant value EGL_DISPLAY_SCALING (10000)[12].

Querying EGL_PIXEL_ASPECT_RATIO returns the aspect ratio of an individual pixel (the ratio of a pixel's width to its height), multiplied by EGL_DISPLAY_SCALING. For almost all displays, the returned value will be EGL_DISPLAY_SCALING, indicating an aspect ratio of one (square pixels).

For an offscreen (pbuffer or pixmap) surface, or a surface whose pixel dot pitch or aspect ratio are unknown, querying EGL_HORIZONTAL_RESOLUTION, EGL_VERTICAL_RESOLUTION, and EGL_PIXEL_ASPECT_RATIO will return the constant value EGL_UNKNOWN (-1).

Querying EGL_RENDER_BUFFER returns the buffer which client API rendering is requested to use. For a window surface, this is the same attribute value specified when the surface was created. For a pbuffer surface, it is always EGL_BACK_BUFFER. For a pixmap surface, it is always EGL_SINGLE_BUFFER. To determine the actual buffer being rendered to by a context, call **eglQueryContext** (see section 3.7.4).

Querying EGL_MULTISAMPLE_RESOLVE returns the filtering method used when performing multisammple buffer resolution. The filter may be either EGL_MULTISAMPLE_RESOLVE_DEFAULT or EGL_MULTISAMPLE_RESOLVE_BOX, as described above for **eglSurfaceAttrib**.

Querying EGL_SWAP_BEHAVIOR describes the effect on the color buffer when posting a surface with **eglSwapBuffers** (see section 3.9). Swap behavior may be either EGL_BUFFER_PRESERVED or EGL_BUFFER_DESTROYED, as described above for **eglSurfaceAttrib**.

Querying EGL_TEXTURE_FORMAT, EGL_TEXTURE_TARGET, EGL_MIPMAP_TEXTURE, or EGL_MIPMAP_LEVEL for a non-pbuffer surface is not an error, but

[12] EGL_DISPLAY_SCALING is used where EGL needs to return floating-point attribute values, which would normally be smaller than 1, as integers while still retaining sufficient precision to be meaningful.

value is not modified.

eglQuerySurface returns EGL_FALSE on failure and *value* is not updated. If *attribute* is not a valid EGL surface attribute, then an EGL_BAD_ATTRIBUTE error is generated. If *surface* is not a valid EGLSurface then an EGL_BAD_SURFACE error is generated.

3.6 Rendering to Textures

This section describes how to render to an OpenGL ES texture using a pbuffer surface configured for this operation. If a pbuffer surface does not support OpenGL ES rendering, or if OpenGL ES is not implemented on a platform, then calling **eglBindTexImage** or **eglReleaseTexImage** will always generate EGL_BAD_SURFACE errors.

3.6.1 Binding a Surface to a OpenGL ES Texture

The command

```
EGLBoolean eglBindTexImage(EGLDisplay dpy,
  EGLSurface surface, EGLint buffer);
```

defines a two-dimensional texture image. The texture image consists of the image data in *buffer* for the specified surface, and need not be copied. Currently the only value accepted for *buffer* is EGL_BACK_BUFFER, which indicates the buffer into which OpenGL ES rendering is taking place (this is true even when using a single-buffered surface, such as a pixmap). In future versions of EGL, additional *buffer* values may be allowed to bind textures to other buffers in an EGLSurface.

The texture target, the texture format and the size of the texture components are derived from attributes of the specified *surface*, which must be a pbuffer supporting one of the EGL_BIND_TO_TEXTURE_RGB or EGL_BIND_TO_TEXTURE_RGBA attributes.

Note that any existing images associated with the different mipmap levels of the texture object are freed (it is as if **glTexImage** was called with an image of zero width).

The pbuffer attribute EGL_TEXTURE_FORMAT determines the base internal format of the texture. The component sizes are also determined by pbuffer attributes as shown in table 3.6:

The texture target is derived from the EGL_TEXTURE_TARGET attribute of *surface*. If the attribute value is EGL_TEXTURE_2D, then *buffer* defines a texture for

Texture Component	Size
R	EGL_RED_SIZE
G	EGL_GREEN_SIZE
B	EGL_BLUE_SIZE
A	EGL_ALPHA_SIZE

Table 3.6: Size of texture components

the two-dimensional texture object which is bound to the current context (hereafter referred to as the current texture object).

If *dpy* and *surface* are the display and surface for the calling thread's current context, **eglBindTexImage** performs an implicit **glFlush**. For other *surface*s, **eglBindTexImage** waits for all effects from previously issued client API commands drawing to the surface to complete before defining the texture image, as though **glFinish** were called on the last context to which that surface were bound.

After **eglBindTexImage** is called, the specified *surface* is no longer available for reading or writing. Any read operation, such as **glReadPixels** or **eglCopyBuffers**, which reads values from any of the surface's color buffers or ancillary buffers will produce indeterminate results. In addition, draw operations that are done to the surface before its color buffer is released from the texture produce indeterminate results. Specifically, if the surface is current to a context and thread then rendering commands will be processed and the context state will be updated, but the surface may or may not be written. **eglSwapBuffers** has no effect if it is called on a bound surface.

Client APIs other than OpenGL ES may be used to render into a surface later bound as a texture. The effects of binding a surface as an OpenGL ES texture when the surface is current to a client API context other than OpenGL ES are generally similar those described above, but there may be additional restrictions. Applications using mixed-mode render-to-texture in this fashion should unbind surfaces from all client API contexts before binding those surfaces as OpenGL ES textures.

Note that the color buffer is bound to a texture object. If the texture object is shared between contexts, then the color buffer is also shared. If a texture object is deleted before **eglReleaseTexImage** is called, then the color buffer is released and the surface is made available for reading and writing.

Texture mipmap levels are automatically generated when all of the following conditions are met while calling **eglBindTexImage**:

- The EGL_MIPMAP_TEXTURE attribute of the pbuffer being bound is EGL_TRUE.

- The OpenGL ES texture parameter GL_GENERATE_MIPMAP is GL_TRUE for the currently bound texture.
- The value of the EGL_MIPMAP_LEVEL attribute of the pbuffer being bound is equal to the value of the texture parameter GL_TEXTURE_BASE_LEVEL.

In this case, additional mipmap levels are generated as described in section 3.8 of the OpenGL ES 1.1 Specification.

It is not an error to call **glTexImage2D** or **glCopyTexImage2D** to replace an image of a texture object that has a color buffer bound to it. However, these calls will cause the color buffer to be released back to the surface and new memory will be allocated for the texture. Note that the color buffer is released even if the image that is being defined is a mipmap level that was not defined by the color buffer.

If **eglBindTexImage** is called and the surface attribute EGL_TEXTURE_FORMAT is set to EGL_NO_TEXTURE, then an EGL_BAD_MATCH error is returned. If *buffer* is already bound to a texture then an EGL_BAD_ACCESS error is returned. If *buffer* is not a valid buffer, then an EGL_BAD_PARAMETER error is generated. If *surface* is not a valid EGLSurface, or is not a pbuffer surface supporting texture binding, then an EGL_BAD_SURFACE error is generated.

eglBindTexImage is ignored if there is no current rendering context.

3.6.2 Releasing a Surface from an OpenGL ES Texture

To release a color buffer that is being used as a texture, call

```
EGLBoolean eglReleaseTexImage(EGLDisplay dpy,
  EGLSurface surface, EGLint buffer);
```

The specified color buffer is released back to the surface. The surface is made available for reading and writing when it no longer has any color buffers bound as textures.

The contents of the color buffer are undefined when it is first released. In particular, there is no guarantee that the texture image is still present. However, the contents of other color buffers are unaffected by this call. Also, the contents of the depth and stencil buffers are not affected by **eglBindTexImage** and **eglReleaseTexImage**.

If the specified color buffer is no longer bound to a texture (e.g., because the texture object was deleted) then **eglReleaseTexImage** has no effect. No error is generated.

After a color buffer is released from a texture (either explicitly by calling **eglReleaseTexImage** or implicitly by calling a routine such as **glTexImage2D**),

all texture images that were defined by the color buffer become NULL (it is as if **glTexImage** was called with an image of zero width).

If **eglReleaseTexImage** is called and the value of surface attribute EGL_TEXTURE_FORMAT is EGL_NO_TEXTURE, then an EGL_BAD_MATCH error is returned. If *buffer* is not a valid buffer (currently only EGL_BACK_BUFFER may be specified), then an EGL_BAD_PARAMETER error is generated. If *surface* is not a valid EGLSurface, or is not a bound pbuffer surface, then an EGL_BAD_SURFACE error is returned.

3.6.3 Implementation Caveats

Developers should note that conformant OpenGL ES implementations are not required to support render to texture; that is, there may be no EGLConfigs supporting the EGL_BIND_TO_TEXTURE_RGB or EGL_BIND_TO_TEXTURE_RGBA attributes. Render to texture is functionally subsumed by the newer framebuffer object extension to OpenGL ES , and may eventually be deprecated. Render to texture is not supported for OpenGL contexts.

3.7 Rendering Contexts

EGL provides functions to create and destroy rendering contexts for each supported client API ; to query information about rendering contexts; and to bind rendering contexts to surfaces, making them *current*.

At most one context for each supported client API may be current to a particular thread at a given time, and at most one context may be bound to a particular surface at a given time. [13] The minimum number of current contexts that must be supported by an EGL implementation is one for each supported client API . [14]

Only one OpenGL or OpenGL ES context may be current to a particular thread, even if the implementation supports OpenGL and both OpenGL ES 1.x and OpenGL ES 2.x in the same runtime[15]. This restriction is enforced by **eglMakeCurrent** as described in section 3.7.3.

[13] Note that this implies that implementations must allow (for example) both an OpenGL ES and an OpenVG context to be current to the **same** thread, so long as they are drawing to **different** surfaces.

[14] This constraint allows valid implementations which are restricted to supporting only one active rendering thread in a thread group. Such implementations may generate errors in **eglMakeCurrent**.

[15] This restriction is necessary because many entry points are shared by OpenGL and both versions of OpenGL ES . Determining which library version to call into is based on properties of the current OpenGL or OpenGL ES context.

Some of the functions described in this section make use of the *current rendering API*, which is set on a per-thread basis [16] by calling

```
EGLBoolean eglBindAPI(EGLenum api);
```

api must specify one of the supported client APIs , either EGL_OPENGL_API, EGL_OPENGL_ES_API, or EGL_OPENVG_API.

Applications using multiple client APIs are responsible for ensuring the current rendering API is correct before calling the functions **eglCreateContext**, **eglGetCurrentContext**, **eglGetCurrentDisplay**, **eglGetCurrentSurface**, **eglCopyBuffers**, **eglSwapBuffers**, **eglMakeCurrent** (when its *ctx* parameter is EGL_NO_CONTEXT), **eglWaitClient**, or **eglWaitNative**.

EGL_OPENGL_API and EGL_OPENGL_ES_API are interchangeable for all purposes except **eglCreateContext**[17].

eglBindAPI returns EGL_FALSE on failure. If *api* is not one of the values specified above, or if the client API specified by *api* is not supported by the implementation, an EGL_BAD_PARAMETER error is generated.

To obtain the value of the current rendering API, call

```
EGLenum eglQueryAPI(void);
```

The value returned will be one of the valid *api* parameters to **eglBindAPI**, or EGL_NONE.

The initial value of the current rendering API is EGL_OPENGL_ES_API, unless OpenGL ES is not supported by an implementation, in which case the initial value is EGL_NONE.

3.7.1 Creating Rendering Contexts

To create a rendering context for the current rendering API, call

[16] Note that the current rendering API is set on a per-thread basis, but not on a per-EGLDisplay basis. This is because current contexts are bound in the same manner.

[17] This is a behavior change introduced in the February, 2013 EGL 1.4 specification update. Prior to this change, operations such as (for example) calling **eglGetCurrentContext** when an OpenGL ES context is current but the current rendering API is EGL_OPENGL_API would return EGL_NO_CONTEXT instead of the OpenGL ES context.

The change is subtle, unlikely to affect any existing applications, and intended as a convenience to the programmer. It is based on the restriction described above (that only one OpenGL or OpenGL ES context may be current to a particular thread). It is still necessary to distinguish between the two current rendering APIs when creating a context on an implementation which supports both OpenGL and OpenGL ES .

```
EGLContext eglCreateContext(EGLDisplay dpy,
   EGLConfig config, EGLContext share_context,
   const EGLint *attrib_list);
```

If **eglCreateContext** succeeds, it initializes the context to the initial state defined for the current rendering API, and returns a handle to it. The context can be used to render to any compatible EGLSurface.

Although contexts are specific to a single client API , all contexts created in EGL exist in a single namespace. This allows many EGL calls which manage contexts to avoid use of the current rendering API.

If *share_context* is not EGL_NO_CONTEXT, then all shareable data, as defined by the client API (note that for OpenGL and OpenGL ES , shareable data excludes texture objects named 0) will be shared by *share_context*, all other contexts *share_context* already shares with, and the newly created context. An arbitrary number of EGLContexts can share data in this fashion. The OpenGL and OpenGL ES server context state for all sharing contexts must exist in a single address space or an EGL_BAD_MATCH error is generated.

attrib_list specifies a list of attributes for the context. The list has the same structure as described for **eglChooseConfig**. The only attribute that can be specified in *attrib_list* is EGL_CONTEXT_CLIENT_VERSION, and this attribute may only be specified when creating a OpenGL ES context (e.g. when the current rendering API is EGL_OPENGL_ES_API).

attrib_list may be NULL or empty (first attribute is EGL_NONE), in which case attributes assume their default values as described below.

EGL_CONTEXT_CLIENT_VERSION determines which version of an OpenGL ES context to create. An attribute value of 1 specifies creation of an OpenGL ES 1.x context. An attribute value of 2 specifies creation of an OpenGL ES 2.x context. The default value for EGL_CONTEXT_CLIENT_VERSION is 1.

On failure **eglCreateContext** returns EGL_NO_CONTEXT. If the current rendering api is EGL_NONE, then an EGL_BAD_MATCH error is generated (this situation can only arise in an implementation which does not support OpenGL ES , and prior to the first call to **eglBindAPI**). If *share_context* is neither zero nor a valid context of the same client API type as the newly created context, then an EGL_BAD_CONTEXT error is generated.

If *config* is not a valid EGLConfig, or does not support the requested client API , then an EGL_BAD_CONFIG error is generated (this includes requesting creation of an OpenGL ES 1.x context when the EGL_RENDERABLE_TYPE attribute of *config* does not contain EGL_OPENGL_ES_BIT, or creation of an OpenGL ES 2.x context when the attribute does not contain EGL_OPENGL_ES2_BIT).

If the OpenGL or OpenGL ES server context state for *share_context* exists in an address space that cannot be shared with the newly created context, if *share_context* was created on a different display than the one referenced by *config*, or if the contexts are otherwise incompatible (for example, one context being associated with a hardware device driver and the other with a software renderer), then an `EGL_BAD_MATCH` error is generated. If the server does not have enough resources to allocate the new context, then an `EGL_BAD_ALLOC` error is generated.

3.7.2 Destroying Rendering Contexts

A rendering context is destroyed by calling

```
EGLBoolean eglDestroyContext(EGLDisplay dpy,
    EGLContext ctx);
```

All resources associated with *ctx* are marked for deletion as soon as possible. When multiple contexts share objects (see **eglCreateContext**), such shared objects are not deleted until after all contexts on the share list are destroyed, unless the objects are first explicitly deleted by the application. Following **eglDestroyContext**, the context and the handle referring to it are treated in the same fashion as a context destroyed by **eglTerminate** (see section 3.2).

eglDestroyContext returns `EGL_FALSE` on failure. An `EGL_BAD_CONTEXT` error is generated if *ctx* is not a valid context.

3.7.3 Binding Contexts and Drawables

To make a context current, call

```
EGLBoolean eglMakeCurrent(EGLDisplay dpy,
    EGLSurface draw, EGLSurface read,
    EGLContext ctx);
```

eglMakeCurrent binds *ctx* to the current rendering thread and to the *draw* and *read* surfaces.

For an OpenGL or OpenGL ES context, *draw* is used for all operations except for any pixel data read back or copied, which is taken from the frame buffer values of *read*. Note that the same `EGLSurface` may be specified for both *draw* and *read*.

For an OpenVG context, the same `EGLSurface` **must** be specified for both *draw* and *read*.

If the calling thread already has a current context of the same client API type as *ctx*, then that context is flushed and marked as no longer current. *ctx* is then made the current context for the calling thread. For purposes of **eglMakeCurrent**, the client API type of all OpenGL ES and OpenGL contexts is considered the same. In other words, if any OpenGL ES context is currently bound and *ctx* is an OpenGL context, or if any OpenGL context is currently bound and *ctx* is an OpenGL ES context, the currently bound context will be made no longer current and *ctx* will be made current.

OpenGL and OpenGL ES buffer mappings created by e.g. **glMapBuffer** are not affected by **eglMakeCurrent**; they persist whether the context owning the buffer is current or not.

When an OpenGL or OpenGL ES context is made no longer current, buffer mappings created by e.g.

eglMakeCurrent returns `EGL_FALSE` on failure. Errors generated may include:

- If *draw* or *read* are not compatible with *ctx*, then an `EGL_BAD_MATCH` error is generated.
- If *ctx* is current to some other thread, or if either *draw* or *read* are bound to contexts in another thread, an `EGL_BAD_ACCESS` error is generated.
- If binding *ctx* would exceed the number of current contexts of that client API type supported by the implementation, an `EGL_BAD_ACCESS` error is generated.
- If either *draw* or *read* are pbuffers created with **eglCreatePbufferFromClientBuffer**, and the underlying bound client API buffers are in use by the client API that created them, an `EGL_BAD_ACCESS` error is generated.
- If *ctx* is not a valid context, an `EGL_BAD_CONTEXT` error is generated.
- If either *draw* or *read* are not valid EGL surfaces, an `EGL_BAD_SURFACE` error is generated.
- If a native window underlying either *draw* or *read* is no longer valid, an `EGL_BAD_NATIVE_WINDOW` error is generated.
- If *draw* and *read* cannot fit into graphics memory simultaneously, an `EGL_BAD_MATCH` error is generated.
- If the previous context of the calling thread has unflushed commands, and the previous surface is no longer valid, an `EGL_BAD_CURRENT_SURFACE` error is generated.

- If the ancillary buffers for *draw* and *read* cannot be allocated, an EGL_BAD_ALLOC error is generated.
- If a power management event has occurred, an EGL_CONTEXT_LOST error is generated.
- As with other commands taking EGLDisplay parameters, if *dpy* is not a valid EGLDisplay handle, an EGL_BAD_DISPLAY error is generated[18].

Other errors may arise when the context state is inconsistent with the surface state, as described in the following paragraphs.

If *draw* is destroyed after **eglMakeCurrent** is called, then subsequent rendering commands will be processed and the context state will be updated, but the surface contents become undefined. If *read* is destroyed after **eglMakeCurrent** then pixel values read from the framebuffer (e.g., as result of calling **glReadPixels**) are undefined. If a native window or pixmap underlying the *draw* or *read* surfaces is destroyed, rendering and readback are handled as above.

To release the current context without assigning a new one, set *ctx* to EGL_NO_CONTEXT and set *draw* and *read* to EGL_NO_SURFACE. The currently bound context for the client API specified by the current rendering API is flushed and marked as no longer current, and there will be no current context for that client API after **eglMakeCurrent** returns. This is the only case in which **eglMakeCurrent** respects the current rendering API. In all other cases, the client API affected is determined by *ctx*. This is the only case where an uninitialized display may be passed to **eglMakeCurrent**.

If *ctx* is not EGL_NO_CONTEXT, *read* is not EGL_NO_SURFACE, or *draw* is not EGL_NO_SURFACE, then an EGL_NOT_INITIALIZED error is generated if *dpy* is a valid but uninitialized display.

If *ctx* is EGL_NO_CONTEXT and *draw* and *read* are not EGL_NO_SURFACE, or if *draw* or *read* are set to EGL_NO_SURFACE and *ctx* is not EGL_NO_CONTEXT, then an EGL_BAD_MATCH error will be generated.

The first time a OpenGL or OpenGL ES context is made current the viewport and scissor dimensions are set to the size of the *draw* surface (as though **glViewport**(0, 0, w, h) and **glScissor**(0, 0, w, h) were called, where w and h are the width and height of the surface, respectively). However, the viewport and scissor dimensions are not modified when *ctx* is subsequently made current. The client is responsible for resetting the viewport and scissor in this case.

[18] Some implementations have chosen to allow EGL_NO_DISPLAY as a valid *dpy* parameter for **eglMakeCurrent**. This behavior is not portable to all EGL implementations, and should be considered as an undocumented vendor extension.

Implementations may delay allocation of auxiliary buffers for a surface until they are required by a context (which may result in the EGL_BAD_ALLOC error described above). Once allocated, however, auxiliary buffers and their contents persist until a surface is deleted.

3.7.3.1 Multisample Buffers and Multiple Rendering Streams

When rendering to a surface containing multisample buffers (created with respect to an EGLConfig whose EGL_SAMPLE_BUFFERS attribute has a value of one), switching rendering between client APIs may force resolution of the multisample buffer into the color buffer. This can occur for many reasons, such as client APIs which do not share the same interpretation of the multisample information (for example, using different sample locations or weightings); client APIs which do not support multisample rendering; or applications which enable multisample rendering in one client API and disable it in another.

Repeated resolution of the multisample buffer may result in lower quality images. For this reason, applications mixing rendering by multiple client APIs onto the same surface should minimize switching between client APIs . Ideally, each client API rendering to a surface should be made current only once for each frame being rendered.

3.7.3.2 Order of Rendering Operations Between Contexts

EGL makes no guarantees on the rendering order between contexts, even within the same thread. For example, rendering operations performed by a thread while one context is current do not necessarily complete before rendering operations performed later in the same thread but with a different context current. It is the responsibility of the application to employ the correct synchronization when the drawing result of one context needs to be complete before another context accesses that result. Otherwise the result is undefined.

To achieve synchronization, an application can use client API -specific commands such as **glFinish** to wait for rendering operations to complete in one context before making the next current. Alternatively, synchronization objects can be used to order rendering operations between contexts, if supported by the underlying implementation. Synchronization objects are defined by the EGL_KHR_fence_sync and EGL_KHR_wait_sync EGL extensions, or alternatively, they may be available as a feature of the underlying client API . Use of synchronization objects may allow asynchronous execution of the rendering operations, achieving better performance than synchronous wait functions like **glFinish**.

3.7.4 Context Queries

Several queries exist to return information about contexts.

To get the current context for the current rendering API, call

```
EGLContext eglGetCurrentContext(void);
```

If there is no current context for the current rendering API, or if the current rendering API is EGL_NONE, then EGL_NO_CONTEXT is returned (this is not an error). If the current context has been marked for deletion as a result of calling **eglTerminate** or **eglDestroyContext**, the handle returned by **eglGetCurrentContext** is not valid, and cannot be passed successfully to any other EGL function, as discussed in section 3.2.

To get the surfaces used for rendering by a current context, call

```
EGLSurface eglGetCurrentSurface(EGLint readdraw);
```

readdraw is either EGL_READ or EGL_DRAW, to return respectively the read or draw surfaces bound to the current context in the calling thread, for the current rendering API.

If there is no current context for the current rendering API, then EGL_NO_SURFACE is returned (this is not an error). If *readdraw* is neither EGL_READ nor EGL_DRAW, EGL_NO_SURFACE is returned and an EGL_BAD_PARAMETER error is generated. If a current surface has been marked for deletion as a result of calling **eglTerminate** or **eglDestroySurface**, the handle returned by **eglGetCurrentSurface** is not valid, and cannot be passed successfully to any other EGL function, as discussed in section 3.2.

To get the display associated with a current context, call

```
EGLDisplay eglGetCurrentDisplay(void);
```

The display for the current context in the calling thread, for the current rendering API, is returned. If there is no current context for the current rendering API, EGL_NO_DISPLAY is returned (this is not an error).

Note that EGL_NO_DISPLAY is used solely to represent an error condition, and is not a valid EGLDisplay handle. Passing EGL_NO_DISPLAY to any command taking an EGLDisplay parameter will generate either an EGL_BAD_DISPLAY error if the EGL implementation validates EGLDisplay handles, or undefined behavior as described at the end of section 3.1.

To obtain the value of context attributes, use

```
EGLBoolean eglQueryContext(EGLDisplay dpy,
   EGLContext ctx, EGLint attribute, EGLint
   *value);
```

eglQueryContext returns in *value* the value of *attribute* for *ctx*. *attribute* must be set to EGL_CONFIG_ID, EGL_CONTEXT_CLIENT_TYPE, EGL_CONTEXT_CLIENT_VERSION, or EGL_RENDER_BUFFER.

Querying EGL_CONFIG_ID returns the ID of the EGLConfig with respect to which the context was created.

Querying EGL_CONTEXT_CLIENT_TYPE returns the type of client API this context supports (the value of the *api* parameter to **eglBindAPI**).

Querying EGL_CONTEXT_CLIENT_VERSION returns the version of the client API this context actually supports (which may differ from the version specified at context creation time). The resulting value is only meaningful for an OpenGL ES context.

Querying EGL_RENDER_BUFFER returns the buffer which client API rendering via this context will use. The value returned depends on properties of both the context, and the draw surface to which the context is bound:

- If the context is bound to a pixmap surface, then EGL_SINGLE_BUFFER will be returned.
- If the context is bound to a pbuffer surface, then EGL_BACK_BUFFER will be returned.
- If the context is bound to a window surface, then either EGL_BACK_BUFFER or EGL_SINGLE_BUFFER may be returned. The value returned depends on both the buffer *requested* by the setting of the EGL_RENDER_BUFFER property of the *surface* (which may be queried by calling **eglQuerySurface** - see section 3.5.6), and on the client API (not all client APIs support single-buffer rendering to window surfaces). Some client APIs allow control of whether rendering goes to the front or back buffer. This client API -specific choice is not reflected in the returned value, which only describes the buffer that will be rendered to by default if not overridden by the client API .
- If the context is not bound to a surface, then EGL_NONE will be returned.

eglQueryContext returns EGL_FALSE on failure and *value* is not updated. If *attribute* is not a valid EGL context attribute, then an EGL_BAD_ATTRIBUTE error is generated. If *ctx* is invalid, an EGL_BAD_CONTEXT error is generated.

3.8 Synchronization Primitives

To prevent native rendering API functions from executing until any outstanding client API rendering affecting the same surface is complete, call

```
EGLBoolean eglWaitClient(void);
```

All rendering calls for the currently bound context, for the current rendering API, made prior to **eglWaitClient**, are guaranteed to be executed before native rendering calls made after **eglWaitClient** which affect the read or draw surfaces associated with that context.

The same result can be achieved using client API -specific calls such as **glFinish** or **vgFinish**.

Clients rendering to single buffered surfaces (e.g. pixmap surfaces) should call **eglWaitClient** before accessing the native pixmap from the client.

eglWaitClient returns `EGL_TRUE` on success. If there is no current context for the current rendering API, the function has no effect but still returns `EGL_TRUE`. If a surface associated with the calling thread's current context is no longer valid, `EGL_FALSE` is returned and an `EGL_BAD_CURRENT_SURFACE` error is generated.

For backwards compatibility, the function

```
EGLBoolean eglWaitGL(void);
```

is equivalent to

```
EGLenum api = eglQueryAPI();
eglBindAPI(EGL_OPENGL_ES_API);
eglWaitClient();
eglBindAPI(api);
```

To prevent a client API command sequence from executing until any outstanding native rendering affecting the same surface is complete, call

```
EGLBoolean eglWaitNative(EGLint engine);
```

Native rendering calls made with the specified marking *engine*, and which affect the read or draw surfaces associated with the calling thread's current context, for the current rendering API, are guaranteed to be executed before client API rendering calls made after **eglWaitNative**. The same result may be (but is not necessarily) achievable using native synchronization calls.

engine denotes a particular *marking engine* (another drawing API, such as GDI or Xlib) to be waited on. Valid values of *engine* are defined by EGL extensions specific to implementations, but implementations will always recognize the symbolic constant EGL_CORE_NATIVE_ENGINE, which denotes the most commonly used marking engine other then a client API .

eglWaitNative returns EGL_TRUE on success. If there is no current context, the function has no effect but still returns EGL_TRUE. If a surface does not support native rendering (e.g. pbuffer and in most cases window surfaces), the function has no effect but still returns EGL_TRUE. If the surface associated with the calling thread's current context is no longer valid, EGL_FALSE is returned and an EGL_BAD_CURRENT_SURFACE error is generated. If *engine* does not denote a recognized marking engine, EGL_FALSE is returned and an EGL_BAD_PARAMETER error is generated.

3.9 Posting the Color Buffer

After completing rendering, the contents of the color buffer can be made visible in a native window, or copied to a native pixmap.

3.9.1 Posting to a Window

To post the color buffer to a window, call

```
EGLBoolean eglSwapBuffers(EGLDisplay dpy,
    EGLSurface surface);
```

If *surface* is a back-buffered window surface, then the color buffer is copied to the native window associated with that surface. If *surface* is a single-buffered window, pixmap, or pbuffer surface, **eglSwapBuffers** has no effect.

The contents of ancillary buffers are always undefined after calling **eglSwapBuffers**. The contents of the color buffer are undefined if the value of the EGL_SWAP_BEHAVIOR attribute of *surface* is not EGL_BUFFER_PRESERVED. The value of EGL_SWAP_BEHAVIOR can be set for some surfaces using **eglSurfaceAttrib**, as described in section 3.5.6[19].

[19] The EGL 1.4 specification has been updated to acknowledge that ancillary buffers are not necessarily preserved after a swap, and that the EGL_SWAP_BEHAVIOR attribute applies only to the color buffer. This is a change in the specification acknowledging the behavior of many shipping implementations, and is not intended to result in behavior changes in any existing implementation. Applications which require preservation of ancillary buffers across a swap should be aware that not all implementations can preserve them, and that EGL 1.4 has no way to query whether or not they are preserved. Khronos is developing an extension which will allow explicit control over ancillary buffer preservation. This extension will be published in the EGL Registry when available.

3.9.1.1 Native Window Resizing

If the native window corresponding to *surface* has been resized prior to the swap, *surface* must be resized to match. *surface* will normally be resized by the EGL implementation at the time the native window is resized. If the implementation cannot do this transparently to the client, then **eglSwapBuffers** must detect the change and resize *surface* prior to copying its pixels to the native window.

If *surface* shrinks as a result of resizing, some rendered pixels are lost. If *surface* grows, the newly allocated buffer contents are undefined. The resizing behavior described here only maintains consistency of EGL surfaces and native windows; clients are still responsible for detecting window size changes (using platform-specific means) and changing their viewport and scissor regions accordingly.

3.9.2 Copying to a Native Pixmap

To copy the color buffer to a native pixmap, call

```
EGLBoolean eglCopyBuffers(EGLDisplay dpy,
  EGLSurface surface, EGLNativePixmapType
  target);
```

The color buffer is copied to the specified *target*, which must be a valid native pixmap handle.

The mapping of pixels in the color buffer to pixels in the pixmap is platform-dependent, since the native platform pixel coordinate system may differ from that of client APIs .

The color buffer of *surface* is left unchanged after calling **eglCopyBuffers**.

3.9.3 Posting Semantics

surface must be bound to the draw surface of the calling thread's current context, for the current rendering API. This restriction may be lifted in future EGL revisions.

~~If *dpy* and *surface* are the display and surface for the calling thread's current context,~~ **eglSwapBuffers** and **eglCopyBuffers** perform an implicit flush operation on the context (**glFlush** for an OpenGL or OpenGL ES context, **vgFlush** for an OpenVG context). Subsequent client API commands can be issued immediately, but will not be executed until posting is completed.

The destination of a posting operation (a visible window, for **eglSwapBuffers**, or a native pixmap, for **eglCopyBuffers**) should have the same number of components and component sizes as the color buffer it's being copied from.

In the specific case of a luminance color buffer being posted to an RGB destination, the luminance component value will normally be replicated in each of the red, green, and blue components of the destination. Some implementations may use alternate color-space conversion algorithms to map luminance to red, green, and blue values, so long as the perceptual result is unchanged. Such alternate conversions should be documented by the implementation.

In other cases where this compatibility constraint is not met by the surface and posting destination, implementations may choose to relax the constraint by converting data to the destination format. If they do so, they should define an EGL extension specifying which destination formats are supported, and specifying the conversion arithmetic used.

The function

```
EGLBoolean eglSwapInterval(EGLDisplay dpy, EGLint
    interval);
```

specifies the minimum number of video frame periods per buffer swap for the draw surface of the current context, for the current rendering API. The interval takes effect when **eglSwapBuffers** is first called subsequent to the **eglSwapInterval** call. The swap interval has no effect on **eglCopyBuffers**.

The parameter *interval* specifies the minimum number of video frames that are displayed before a buffer swap will occur. The *interval* specified by the function applies to the draw surface bound to the context that is current on the calling thread.

If *interval* is set to a value of 0, buffer swaps are not synchronized to a video frame, and the swap happens as soon as all rendering commands outstanding for the current context are complete. *interval* is silently clamped to minimum and maximum implementation dependent values before being stored; these values are defined by `EGLConfig` attributes `EGL_MIN_SWAP_INTERVAL` and `EGL_MAX_SWAP_INTERVAL` respectively.

The default swap interval is 1.

3.9.4 Posting Errors

eglSwapBuffers and **eglCopyBuffers** return `EGL_FALSE` on failure. If *surface* is not a valid EGL surface, an `EGL_BAD_SURFACE` error is generated. If *surface* is not bound to the draw surface of the calling thread's current context, an `EGL_BAD_SURFACE` error is generated. If *target* is not a valid native pixmap handle, an `EGL_BAD_NATIVE_PIXMAP` error should be generated. If the format of *target* is not compatible with the color buffer, or if the size of *target* is not the same as the size of the color buffer, and there is no defined conversion between the

source and target formats, an EGL_BAD_MATCH error is generated. If called after a power management event has occurred, a EGL_CONTEXT_LOST error is generated. If **eglSwapBuffers** is called and the native window associated with *surface* is no longer valid, an EGL_BAD_NATIVE_WINDOW error is generated. If **eglCopyBuffers** is called and the implementation does not support native pixmaps, an EGL_BAD_NATIVE_PIXMAP error is generated.

eglSwapInterval returns EGL_FALSE on failure. If there is no current context on the calling thread, a EGL_BAD_CONTEXT error is generated. If there is no surface bound to the current context, a EGL_BAD_SURFACE error is generated.

3.10 Obtaining Extension Function Pointers

The client API and EGL extensions which are available to a client may vary at runtime, depending on factors such as the rendering path being used (hardware or software), resources available to the implementation, or updated device drivers. Therefore, the address of client API and EGL extension functions may be queried at runtime. The function

```
void (*eglGetProcAddress(const char
   *procname))(void);
```

returns the address of the ~~extension~~ function named by *procName*. *procName* must be a NULL-terminated string. The pointer returned should be cast to a function pointer type matching the ~~extension~~ function's definition in the corresponding extension specification. A return value of NULL indicates that the specified function does not exist for the implementation.

A non-NULL return value for **eglGetProcAddress** does not guarantee that an extension function is actually supported at runtime. The client must also make a corresponding query, such as **glGetString**(GL_EXTENSIONS) for OpenGL and OpenGL ES extensions; **vgGetString**(VG_EXTENSIONS) for OpenVG extensions; or **eglQueryString**(*dpy*, EGL_EXTENSIONS) for EGL extensions, to determine if an extension is supported by a particular client API context[20]

Client API function pointers returned by **eglGetProcAddress** are independent of the display and the currently bound client API context, and may be used by any client API context which supports the extension.

eglGetProcAddress may be queried for all of the following functions:

[20] If an extension is not supported by the current client API context, preferred behavior of calling through the function pointer is to generate an error, such as GL_INVALID_OPERATION for OpenGL and OpenGL ES contexts. However, undefined behavior up to and including program termination is possible.

- All EGL and client API extension functions supported by the implementation (whether those extensions are supported by the current client API context or not). This includes any mandatory OpenGL ES extensions.

eglGetProcAddress may not be queried for core (non-extension) functions in EGL or client APIs [21].

For functions that are queryable with **eglGetProcAddress**, implementations may choose to also export those functions statically from the object libraries implementing those functions. However, portable clients cannot rely on this behavior.

3.11 Releasing Thread State

EGL maintains a small amount of per-thread state, including the error status returned by **eglGetError**, the currently bound rendering API defined by **eglBindAPI**, and the current contexts for each supported client API . The overhead of maintaining this state may be objectionable in applications which create and destroy many threads, but only call EGL or client APIs in a few of those threads at any given time.

To return EGL to its state at thread initialization, call

```
EGLBoolean eglReleaseThread(void);
```

`EGL_TRUE` is returned on success, and the following actions are taken:

- For each client API supported by EGL, if there is a currently bound context, that context is released. This is equivalent to calling **eglMakeCurrent** with *ctx* set to `EGL_NO_CONTEXT` and both *draw* and *read* set to `EGL_NO_SURFACE` (see section 3.7.3).
- The current rendering API is reset to its value at thread initialization (see section 3.7).
- Any additional implementation-dependent per-thread state maintained by EGL is marked for deletion as soon as possible.

eglReleaseThread may be called in any thread at any time, and may be called more than once in a single thread. The initialization status of EGL (see section 3.2) is **not** affected by releasing the thread; only per-thread state is affected.

[21] In practice, some implementations have chosen to relax this restriction.

Resources explicitly allocated by calls to EGL, such as contexts, surfaces, and configuration lists, are not affected by **eglReleaseThread**. Such resources belong not to the thread, but to the EGL implementation as a whole.

Applications may call other EGL routines from a thread following **eglReleaseThread**, but any such call may reallocate the EGL state previously released. In particular, calling **eglGetError** immediately following a successful call to **eglReleaseThread** should not be done. Such a call will return `EGL_SUCCESS` - but will also result in reallocating per-thread state.

eglReleaseThread returns `EGL_FALSE` on failure. There are no defined conditions under which failure will occur. Even if EGL is not initialized on any `EGLDisplay`, **eglReleaseThread** should succeed. However, platform-dependent failures may be signaled through the value returned from **eglGetError**. Unless the platform-dependent behavior is known, a failed call to **eglReleaseThread** should be assumed to leave the current rendering API, and the currently bound contexts for each supported client API , in an unknown state.

Chapter 4

Extending EGL

EGL implementors may extend EGL by adding new commands or additional enumerated values for existing EGL commands.

New names for EGL functions and enumerated types must clearly indicate whether some particular feature is in the core EGL or is vendor specific. To make a vendor-specific name, append a company identifier (in upper case) and any additional vendor-specific tags (e.g. machine names). For instance, SGI might add new commands and manifest constants of the form **eglNewCommandSGI** and `EGL_NEW_DEFINITION_SGI`. If two or more vendors agree in good faith to implement the same extension, and to make the specification of that extension publicly available, the procedures and tokens that are defined by the extension can be suffixed by `EXT`. Extensions approved by supra-vendor organizations use similar identifiers, such as KHR for extensions approved by the Khronos Group).

It is critically important for interoperability that enumerants and entry point names be unique across vendors. The Khronos API Registrar maintains a registry of enumerants, and all shipping enumerant values must be determined by requesting blocks of enumerants from the registry. See

http://www.opengl.org/registry/

for more information on defining extensions.

Chapter 5

EGL Versions, Header Files, and Enumerants

Each version of EGL supports specified client API versions, and all prior versions of those APIs up to that version. For OpenGL ES , such support includes both Common and Common-Lite profiles. EGL 1.0 supports OpenGL ES 1.0, EGL 1.1 supports OpenGL ES 1.1, EGL 1.2 supports OpenGL ES 2.0 and OpenVG 1.0, and EGL 1.4 supports OpenGL ES 2.0, OpenVG 1.0, and all versions of OpenGL .

Whether a particular client API is actually available at runtime may depend on additional factors. In most cases, EGL and each client API are provided in separate libraries, and applications must link to the EGL library and to each of the client APIs used by the application. However, details of this procedure vary, and developers must refer to platform-specific documentation.

5.1 Header Files

The EGL specification defines an ISO C language binding. This binding may also be used from C++ code. In these environments, the EGL header file `<EGL/egl.h>` provides prototypes for all the EGL entry points, and C preprocessor symbols for all the EGL tokens. C and C++ source code should `#include <EGL/egl.h>` before using any EGL entry points or symbols. [1]

Languages other than C and C++ will define the EGL interfaces using other methods, not described in this specification.

[1] For backwards compatibility, implementations supporting OpenGL ES 1.x must also support the EGL header on the path `<GLES/egl.h>`.

The Khronos Implementers Guidelines describe recommended practice, outline platform-specific issues, and provide other recommendations to people writing EGL implementations. For more details refer to the developer area at:

http://www.khronos.org/

5.2 Compile-Time Version Detection

To allow code to be written portably against future EGL versions, the compile-time environment must make it possible to determine which EGL version interfaces are available. The details of such detection are language-specific and should be specified in the language binding documents for each language. For C and C++ code, the <EGL/egl.h> header defines C preprocessor symbols corresponding to all versions of EGL supported by the implementation:

```
#define EGL_VERSION_1_0 1
#define EGL_VERSION_1_1 1
#define EGL_VERSION_1_2 1
#define EGL_VERSION_1_3 1
#define EGL_VERSION_1_4 1
```

Future versions of EGL will define additional preprocessor symbols corresponding to the major and minor numbers of those versions.

5.3 Enumerant Values and Header Portability

Enumerant values for EGL tokens are required to be common across all implementations. A reference version of the egl.h header file, including defined values for all EGL enumerants, accompanies this specification and can be downloaded from

http://www.khronos.org/

All platform-specific types, values, and macros used in egl.h are partitioned into a platform header, eglplatform.h, which is automatically included by egl.h. A copy of eglplatform.h providing definitions suitable for many platforms is included along with egl.h. Implementers should need to modify only eglplatform.h, never egl.h. [2]

[2] Please submit any additions to eglplatform.h made to support new platforms for inclusion in the reference copy.

Chapter 6

Glossary

Address Space the set of objects or memory locations accessible through a single name space. In other words, it is a data region that one or more threads may share through pointers.

Client an application, which communicates with the underlying EGL implementation and underlying native window system by some path. The application program is referred to as a client of the window system server. To the server, the client is the communication path itself. A program with multiple connections is viewed as multiple clients to the server. The resource lifetimes are controlled by the connection lifetimes, not the application program lifetimes.

Client API one of the rendering APIs supported by EGL. At present client APIs include OpenGL , OpenGL ES and OpenVG , but other clients are expected to be added in future versions of EGL. Context creation / management, rendering semantics, and interaction between client APIs are all well-defined by EGL. There is (considerably more limited) support for rendering to EGL surfaces by non-client (native) rendering APIs, and the semantics of such support are more implementation-dependent.

Compatible an OpenGL or OpenGL ES rendering context is compatible with (may be used to render into) a surface if they meet the constraints specified in section 2.2.

Connection a bidirectional byte stream that carries the X (and EGL) protocol between the client and the server. A client typically has only one connection to a server.

(Rendering) Context an OpenGL or OpenGL ES rendering context. This is a virtual machine. All OpenGL or OpenGL ES rendering is done with respect

to a context. The state maintained by one rendering context is not affected by another except in case of state that may be explicitly shared at context creation time, such as textures.

Current Context an implicit context used by OpenGL , OpenGL ES and OpenVG , rather than passing a context parameter to each API entry point. The current OpenGL , OpenGL ES and OpenVG contexts are set as defined in section 3.7.3.

EGLContext a handle to a rendering context. OpenGL and OpenGL ES rendering contexts consist of client side state and server side state. Other client APIs do not distinguish between the two types of state.

(Drawing) Surface an onscreen or offscreen buffer where pixel values resulting from rendering through OpenGL ES or other APIs are written.

Thread one of a group of execution units all sharing the same address space. Typically, each thread will have its own program counter and stack pointer, but the text and data spaces are visible to each of the threads. A thread that is the only member of its group is equivalent to a process.

Appendix A

Version 1.0

EGL version 1.0, approved on July 23, 2003, is the original version of EGL. EGL was loosely based on GLX 1.3, generalized to be implementable on many different operating systems and window systems and simplified to reflect the needs of embedded devices running OpenGL ES .

A.1 Acknowledgements

EGL 1.0 is the result of the contributions of many people, representing a cross section of the desktop, hand-held, and embedded computer industry. Following is a partial list of contributors, including the company that they represented at the time of their contribution:

Aaftab Munshi, ATI
Andy Methley, Panasonic
Carl Korobkin, 3d4W
Chris Hall, Seaweed Systems
Claude Knaus, Silicon Graphics
David Blythe, 3d4W
Ed Plowman, ARM
Graham Connor, Imagination Technologies
Harri Holopainen, Hybrid Graphics
Jacob Ström, Ericsson
Jani Vaarala, Nokia
Jon Leech, Silicon Graphics
Justin Couch, Yumetech
Kari Pulli, Nokia
Lane Roberts, Symbian

Mark Callow, HI
Mark Tarlton, Motorola
Mike Olivarez, Motorola
Neil Trevett, 3Dlabs
Phil Huxley, Tao Group
Tom Olson, Texas Instruments
Ville Miettinen, Hybrid Graphics

Appendix B

Version 1.1

EGL version 1.1, approved on August 5, 2004, is the second release of EGL. It adds power management and swap control functionality based on vendor extensions from Imagination Technologies, and optional render-to-texture functionality based on the `WGL_ARB_render_texture` extension defined by the OpenGL ARB for desktop OpenGL.

B.1 Revision 1.1.2

EGL version 1.1.2 (revision 2 of EGL 1.1), approved on November 10, 2004, clarified that vertex buffer objects are shared among contexts in the same fashion as texture objects.

B.2 Acknowledgements

EGL 1.1 is the result of the contributions of many people, representing a cross section of the desktop, hand-held, and embedded computer industry. Following is a partial list of contributors, including the company that they represented at the time of their contribution:

Aaftab Munshi, ATI
Andy Methley, Panasonic
Axel Mamode, Sony
Barthold Lichtenbelt, 3Dlabs
Benji Bowman, Imagination Technologies
Borgar Ljosland, Falanx
Brian Murray, Motorola
Bryce Johnstone, Texas Instruments

Carlos Sarria, Imagination Technologies
Chris Tremblay, Motorola
Claude Knaus, Esmertec
Clay Montgomery, Nokia
Dan Petersen, Sun
Dan Rice, Sun
David Blythe, HI
David Yoder, Motorola
Doug Twilleager, Sun
Ed Plowman, ARM
Graham Connor, Imagination Technologies
Greg Stoner, Motorola
Hannu Napari, Hybrid
Harri Holopainen, Hybrid
Jacob Ström, Ericsson
Jani Vaarala, Nokia
Jerry Evans, Sun
John Metcalfe, Imagination Technologies
Jon Leech, Silicon Graphics
Kari Pulli, Nokia
Lane Roberts, Symbian
Madhukar Budagavi, Texas Instruments
Mathias Agopian, PalmSource
Mark Callow, HI
Mark Tarlton, Motorola
Mike Olivarez, Motorola
Neil Trevett, 3Dlabs
Nick Triantos, Nvidia
Petri Kero, Hybrid
Petri Nordlund, Bitboys
Phil Huxley, Tao Group
Remi Arnaud, Sony
Robert Simpson, Bitboys
Tero Sarkkinen, Futuremark
Timo Suoranta, Futuremark
Thomas Tannert, Silicon Graphics
Tomi Aarnio, Nokia
Tom McReynolds, Nvidia
Tom Olson, Texas Instruments
Ville Miettinen, Hybrid Graphics

Appendix C

Version 1.2

EGL version 1.2, approved on July 8, 2005, is the third release of EGL. It adds support for the OpenVG 2D client API , in addition to support for OpenGL ES , and generalizes EGL concepts to enable supporting other client APIs in the future.

C.1 Acknowledgements

EGL 1.2 is the result of the contributions of many people, representing a cross section of the desktop, hand-held, and embedded computer industry. Following is a partial list of contributors, including the company that they represented at the time of their contribution:

Aaftab Munshi, ATI
Anu Ramanathan, TI
Daniel Rice, Sun Microsystems
Espen Aamodt, Falanx
Jani Vaarala, Nokia
Jon Leech, SGI
Jussi Räsänen, Hybrid Graphics
Koichi Mori, Nokia
Mark Callow, HI Corporation
Members of the Khronos OpenGL ES Working Group
Members of the Khronos OpenVG Working Group
Michael.Nonweiler, ARM
Neil Trevett, 3Dlabs / NVIDIA
Petri Kero, Hybrid Graphics
Robert Simpson, Bitboys
Simon Fenney, PowerVR

Tom Olson, TI

Appendix D

Version 1.3

EGL version 1.3 was voted out of the OpenKODE Working Group on December 4, 2006, and formally approval by the Khronos Board of Promoters on February 8, 2007. EGL 1.3 is the fourth release of EGL. It adds support for separate OpenGL ES 1.x and 2.x contexts with the `EGL_CONTEXT_CLIENT_VERSION` attribute to **eglCreateContext** and the `EGL_OPENGL_ES2_BIT` in the `EGL_RENDERABLE_TYPE` attribute, and adds the `EGL_MATCH_NATIVE_PIXMAP` pseudo-attribute to **eglChooseConfig**, to allow selecting configs matching specific native pixmaps. The `EGL_CONFORMANT` attribute was added to indicate if client API contexts will pass the required conformance tests, and the `EGL_SURFACE_TYPE` attribute was extended with the `EGL_VG_COLORSPACE_LINEAR_BIT` and `EGL_VG_ALPHA_FORMAT_PRE_BIT` bitfields to define whether or not linear colorspace and premultiplied alpha format are supported by the OpenVG implementation. For naming consistency, some tokens from EGL 1.2 have been renamed as shown in table D.1. The old names are also retained for backwards compatibility. The specification adds a number of clarifications (but not behavior changes) regarding config sorting, surface resource ownership, multiple client API context versions, and SDK issues.

Finally, the `eglplatform.h` header is defined to accompany the reference `egl.h` header provided by Khronos.

D.1 Acknowledgements

EGL 1.3 is the result of the contributions of many people, representing a cross section of the desktop, hand-held, and embedded computer industry. Following is a list of contributors, including the company that they represented at the time of their contribution:

EGL 1.2 Token Name	EGL 1.3 Token Name
EGL_COLORSPACE	EGL_VG_COLORSPACE
EGL_COLORSPACE_LINEAR	EGL_VG_COLORSPACE_LINEAR
EGL_COLORSPACE_sRGB	EGL_VG_COLORSPACE_sRGB
EGL_ALPHA_FORMAT	EGL_VG_ALPHA_FORMAT
EGL_ALPHA_FORMAT_PRE	EGL_VG_ALPHA_FORMAT_PRE
EGL_ALPHA_FORMAT_NONPRE	EGL_VG_ALPHA_FORMAT_NONPRE
NativeDisplayType	EGLNativeDisplayType
NativePixmapType	EGLNativePixmapType
NativeWindowType	EGLNativeWindowType

Table D.1: Renamed tokens

Aaftab Munshi, ATI
Daniel Rice, Sun Microsystems
Espen Aamodt, Falanx
Gary King, NVIDIA
Jani Vaarala, Nokia
Jasin Bushnaief, Hybrid Graphics
Jay Abbott, TAO
Jon Kennedy, 3Dlabs
Jon Leech
Jussi Räsänen, Hybrid Graphics
Kalle Raita, Hybrid Graphics
Kari Pulli, Nokia
Koichi Mori, Nokia
Leonardo Estevez, TI
Mark Callow, HI Corporation
Members of the Khronos OpenGL ES Working Group
Members of the Khronos OpenKODE Working Group
Members of the Khronos OpenVG Working Group
Neil Trevett, NVIDIA
Petri Kero, Hybrid Graphics
Remi Arnaud, Sony Computer Entertainment
Robert J. Simpson, Bitboys
Robert Palmer, Symbian
Sampo Lappalainen, Hybrid Graphics
Simon Fenney, Imagination Technologies

Sven Gothel, ATI
Teemu Rantalaiho, Hybrid Graphics
Tom Olson, TI

Appendix E

Version 1.4

EGL version 1.4 was voted out of the Khronos Technical Working Group on March 25, 2008, and formally approved by the Khronos Board of Promoters on May 29, 2008.

EGL 1.4 is the fifth release of EGL. It introduces the following new features:

- Allow multisampled configurations for OpenVG , by relaxing OpenGL ES -specific language and documenting that multisample buffer resolution may be performed when switching which client API is rendering to a surface.
- Allow control of multisample resolution behavior (use of a box filter) using the `EGL_MULTISAMPLE_RESOLVE` `EGLSurface` attribute.
- Allow control of swap behavior (preserving back buffer contents) using the `EGL_SWAP_BEHAVIOR` bit in the `EGL_SURFACE_TYPE` `EGLSurface` attribute.
- Enable support for OpenGL (in addition to, or instead of OpenGL ES) as a client API.
- Relax definition of `EGLNativeDisplayType` to allow a variety of mappings to X and Microsoft Windows data structures.
- Document the meaning of the `EGL_LEVEL` `EGLConfig` attribute.
- Document that **eglMakeCurrent** can raise an `EGL_BAD_ACCESS` error when binding more contexts in the current thread group than are supported by the implementation.
- Add a specific example of how **eglCreatePbufferFromClientBuffer** can fail due to implementation constraints.

- Fix prototypes of functions with empty argument lists.

E.1 Updates to EGL 1.4

After the initial version of EGL 1.4 was released, minor changes and corrections were made in later revisions as described below.

Changes in the revision approved on January 20, 2009:

- Change object destruction behavior such that object handles become invalid immediately after an object is deleted, although the underlying object may remain valid if it's current to a context. This affects **eglTerminate** (section 3.2), **eglDestroySurface** (section 3.5.5), **eglDestroyContext** (section 3.7.2), and **eglGetCurrentContext** and **eglGetCurrentSurface** (section 3.7.4).
- Clarify initialization and termination behavior of `EGLDisplays`, and behavior of EGL functions when passed an uninitialized display, in sections 3.2 and 3.7.3.

Changes in the revision approved on April 15, 2009:

- Specified in section 2.1.2 that all objects exist in the namespace of an `EGLDisplay` (bug 4303).
- Clarified meaning of `EGL_PIXEL_ASPECT_RATIO` and the purpose of `EGL_DISPLAY_SCALING` in section 3.5.6 (bug 3594).

Changes in the revision approved on June 23, 2009:

- Expanded description of "generic" errors applying to multiple commands in section 3.1 (bug 4993).
- Noted in sections 3.7.4 and 3.7.3 that `EGL_NO_DISPLAY` is not a valid `EGLDisplay`, and passing it as a display parameter should generate errors (bug 4993).
- Added clarification of meaning of config masks in section 3.4.1 (bug 5276).

Changes in the revision approved on September 25, 2009:

- Updated language in section 3.5.1 to make clear that the window system (as well as EGL and client APIs other than OpenVG) is not necessarily affected by the value of the EGL_VG_ALPHA_FORMAT attribute, and that preferred window system behavior is to ignore EGL_VG_ALPHA_FORMAT (bug 5526).

- Clarified error conditions for **eglCreatePbufferFromClientBuffer** in section 3.5.3 (bug 5473).

Changes in the revision approved on March 3, 2010:

- Change descriptions of EGL_SWAP_BEHAVIOR_PRESERVED_BIT in table 3.2 and EGL_SWAP_BEHAVIOR in section 3.4 to specify that they apply only to the color buffer. Relax language in section 3.9.1 to allow ancillary buffer contents to be undefined after swap, regardless of the value of EGL_SWAP_BEHAVIOR; clarify how EGL_SWAP_BEHAVIOR controls color buffer preservation; and add a footnote describing this subtle behavior change relative to older versions of EGL 1.4 (bug 5970).

Changes in the revision approved on April 7, 2010:

- Update table 3.1 and the description of EGL_BUFFER_SIZE in section 3.4 to clarify that this attribute is simply the sum of the RGBA or LA component sizes, and does not include any padding or alignment bits that may be present in the underlying pixel format (bug 6143).

Changes in the revision approved on May 21, 2010:

- Note that EGL_MATCH_NATIVE_PIXMAP is not a valid *attribute* to **eglGetConfigAttrib** in section 3.4.3 (bug 6285).

Changes in the revision approved on July 21, 2010:

- Clarify lifetime of shared objects when contexts on the share list are destroyed in section 3.7.2 (Bug 6582).

Changes in the revision approved on October 6, 2010:

- Fix typo in section 2.4 (public Bug 340).

- Refine **eglTerminate** language in section 3.2 to specify that handles to all types of EGL resources owned by the terminated display are invalidated, although the display handle itself remains valid (Bug 6776).

- Fix error condition for **eglCreateWindowSurface** in section 3.5.1 to be generated if there is already an **EGLSurface** associated with the native window, rather than an `EGLConfig` (Bug 6667).
- Expand footnote describing counterintuitive behavior of `EGLConfig` sort rule 3 in section 3.4.1 (public Bug 327).
- Add Tero Pihlajakoski to the Acknowledgements.

Changes in the revision approved on April 20, 2011:

- Note that `EGL_DONT_CARE` is not a valid attribute value for `EGL_MATCH_NATIVE_PIXMAP` in section 3.4.1 (Bug 7456).
- Correct sort order of `EGL_COLOR_BUFFER_TYPE` in table 3.4 (Bug 7431).

Changes in the revision approved on February 13, 2013:

- Clarify support for OpenGL as well as OpenGL ES in sections 1, 2.2, 2.2, 2.2.2, 2.3, 2.4, 2.5, and 5 (Bug 9864).
- Added new section 2.2.2.1, clarifying that the y coordinate used when rendering to native window or pixmap surfaces is inverted relative to the client API coordinate system, so that images appear as expected. N.b. this is not a behavior change (Bug 9701).
- Note in section 3.1 that since **eglGetError** always returns error information about the most recently called EGL function, calling **eglGetError** twice in a row will return `EGL_SUCCESS` on the second call.
- Add language to the description of **eglBindAPI** in section 3.7 making `EGL_OPENGL_API` and `EGL_OPENGL_ES_API` equivalent for all purposes other than **eglCreateContext**, and added **eglCopyBuffers** and **eglSwapBuffers** to the list of commands affected by the current rendering API (Bug 9118).
- Minor language fixes to description of **eglGetProcAddress** in section 3.10 (Bug 9865).
- Clarify support for OpenGL as well as OpenGL ES in chapter 6 (Bug 9864).

Changes in the revision approved on December 4, 2013:

- Modified the definition of `EGLint` in section 2.1.1 so that it may not be large enough to hold a native pointer, and described why the regression is being adopted (Bug 11027).

- Updated section 2.2.2.1 to not mandate that all window systems invert the coordinate system relative to client APIs (Bug 9701).
- Change selection type of EGL_CONFIG_ID from *Exact* to *Special* in table 3.4 (Bug 10567).
- Added new section 3.7.3.2 specifying that EGL does not provide ordering guarantees across **eglMakeCurrent** (Bug 10664).
- Change description of **eglQueryContext** query in section 3.7.4 so EGL_CONTEXT_CLIENT_VERSION returns the version of the context actually created, not the version requested (Bug 10906).
- Clarify that querying EGL_RENDER_BUFFER returns values depending on the draw surface in section 3.7.4; that **eglWaitClient** and **eglWaitNative** guarantee synchronization to both read and draw surfaces in section 3.8; and that **eglSwapInterval** affects, and **eglSwapBuffers** and **eglCopyBuffers** are restricted to the currently bound draw surface in sections 3.9.3 and 3.9.4 (Bug 10200).
- Add a footnote to the description of **eglGetProcAddress** in section 3.10 clarifying that calling through an extension function pointer to an extension not implemented by a client API results in undefined behavior (Bug 10147).

E.2 Acknowledgements

EGL 1.4 is the result of the contributions of many people, representing a cross section of the desktop, hand-held, and embedded computer industry. Following is a list of contributors, including the company that they represented at the time of their contribution:

Acorn Pooley, NVIDIA
Andrzej Mamona, AMD
Barthold Lichtenbelt, NVIDIA
Benj Lipchak, AMD
Benji Bowman, Imagination Technologies
Bill Licea-Kane, AMD
Dongkyun Jeong, Samsung
Ed Plowman, ARM
Gabriele Svelto, ST Microelectronics

Gary King, NVIDIA
Georg Kolling, Imagination Technologies
Graham Connor, Imagination Technologies
Ian Romanick, Intel
Jim Van Welzen, NVIDIA
Jon Leech
Kari Pulli, Nokia
Leonardo Estevez, TI
Mark Callow, HI Corporation
Marko Lukat, Antix Labs
Matti Paavola, Nokia
Maurice Ribble, AMD
Members of the Khronos OpenGL ES, OpenKODE, OpenMAX, OpenVG, and OpenWF Working Groups
Michael Giovinco, Seaweed Systems
Neil Trevett, NVIDIA
Pasi Keranen, Nokia
Pierre Boudier, AMD
Richard Sahlin, Ericsson
Robert Palmer, Symbian
Robert Simpson, AMD
Roger Nixon, Broadcom
Sami Kyostila, Nokia
Steven Fischer, Motorola
Tero Pihlajakoski, Symbio
Tim Renouf, Antix Labs
Tom Olson, TI
Yeshwant Muthusamy, Nokia
Zhifang Long, Marvell

Index