

OpenGL[®] ES Native Platform Graphics Interface
(Version 1.0)

Editor: Jon Leech

Copyright © 2002-2003 Promoters of the Khronos Group (3Dlabs, ARM Ltd., ATI Technologies, Inc., Discreet, Ericsson Mobile, Imagination Technologies Group plc, Motorola, Inc., Nokia, Silicon Graphics, Inc., SK Telecom, and Sun Microsystems).

This document is protected by copyright, and contains information proprietary to The Khronos Group. Any copying, adaptation, distribution, public performance, or public display of this document without the express written consent of the copyright holders is strictly prohibited. The receipt or possession of this document does not convey any rights to reproduce, disclose, or distribute its contents, or to manufacture, use, or sell anything that it may describe, in whole or in part.

This document is a derivative work of "OpenGL[®] Graphics with the X Window System (Version 1.4)". Silicon Graphics, Inc. owns, and reserves all rights in, the latter document.

OpenGL is a registered trademark, and OpenGL ES is a trademark, of Silicon Graphics, Inc.

Contents

1	Overview	1
2	EGL Operation	2
2.1	Native Window System and Rendering APIs	2
2.1.1	Scalar Types	2
2.1.2	Displays	3
2.2	Rendering Contexts and Drawing Surfaces	3
2.2.1	Using Rendering Contexts	4
2.2.2	Rendering Models	4
2.2.3	Interaction With Native Rendering	4
2.3	Direct Rendering and Address Spaces	5
2.4	Shared State	5
2.4.1	Texture Objects	6
2.5	Multiple Threads	6
2.6	Power Management	7
3	EGL Functions and Errors	8
3.1	Errors	8
3.2	Initialization	10
3.3	EGL Versioning	11
3.4	Configuration Management	12
3.4.1	Querying Configurations	15
3.4.2	Lifetime of Configurations	18
3.4.3	Querying Configuration Attributes	19
3.5	Rendering Surfaces	19
3.5.1	Creating On-Screen Rendering Surfaces	19
3.5.2	Creating Off-Screen Rendering Surfaces	20
3.5.3	Creating Native Pixmap Rendering Surfaces	21
3.5.4	Destroying Rendering Surfaces	22

3.5.5	Querying Surface Attributes	22
3.6	Rendering Contexts	23
3.6.1	Creating Rendering Contexts	23
3.6.2	Destroying Rendering Contexts	24
3.6.3	Binding Contexts and Drawables	24
3.7	Synchronization Primitives	26
3.8	Posting the Color Buffer	27
3.8.1	Posting to a Window	27
3.8.2	Copying to a Native Pixmap	28
3.8.3	Posting Semantics	28
3.8.4	Posting Errors	29
3.9	Obtaining Extension Function Pointers	29
4	Extending EGL	31
5	EGL Versions and Enumerants	32
5.1	Compile-Time Version Detection	32
5.2	Enumerant Values	32
6	Glossary	33
A	Version 1.0	35
A.1	Acknowledgements	35

List of Tables

3.1	EGLConfig attributes.	13
3.2	Types of surfaces supported by an EGLConfig	13
3.3	Default values and match criteria for EGLConfig attributes.	17

Chapter 1

Overview

This document describes EGL, the interface between OpenGL ES and the underlying native platform window system. It refers to concepts discussed in the OpenGL ES specification, and may be viewed as an appendix to that document. EGL uses OpenGL ES conventions for naming entry points and macros.

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

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 OpenGL ES 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.

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.

2.2 Rendering Contexts and Drawing Surfaces

The OpenGL ES specification is intentionally vague on how a *rendering context* (an abstract OpenGL ES state machine) is created. One of the purposes of EGL is to provide a means to create an OpenGL ES context and associate it with a surface.

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 *pixmap*s, 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 rendering context. If several rendering contexts are all writing to the same window, 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.

A rendering 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

- have color buffers and ancillary buffers of the same depth.
- 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 rendering contexts. It is also possible to use a single context to render into multiple `EGLSurfaces`.

2.2.1 Using Rendering Contexts

OpenGL ES defines both client state and server state. Thus a rendering context consists of two parts: one to hold the client state and one to hold the server state.

Each thread can have at most one current rendering context. In addition, a rendering context can be current for only one thread at a time. The client is responsible for creating a rendering context and a surface.

2.2.2 Rendering Models

EGL and OpenGL ES supports 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 OpenGL ES is 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.

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

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 3.8.3).

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 OpenGL ES rendering is desirable, since there is no need to move data between the back buffer visible to OpenGL ES 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 OpenGL ES back buffer.

When both native rendering APIs and OpenGL ES 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 3.7.

Some state is shared between OpenGL ES and the underlying native window system and rendering APIs, including pixel values in the visible frame buffer and, in the case of pixmaps, color buffer values.

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 OpenGL ES client and server 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.

Both the client context state and the server context state of a rendering context exist 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 OpenGL ES 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

rendering contexts to share such state rather than replicating it in each context.

EGL provides for sharing certain types of server state among contexts existing in a single address space. At present such state includes only *texture objects*; additional types of state may be shared in future revisions of OpenGL ES where such types of state (for example, display lists) are defined and where such sharing makes sense.

2.4.1 Texture Objects

OpenGL ES texture state can be encapsulated in a named texture object. A texture object is created by binding an unused name to the texture target `GL_TEXTURE_2D` of a rendering context. When a texture object is bound, OpenGL ES 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 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 rendering context.

2.5 Multiple Threads

The EGL and OpenGL ES client side libraries must be threadsafe. Interrupt routines may not share a rendering context with their main thread.

EGL guarantees sequentiality within a command stream for OpenGL ES, but not between OpenGL ES and other rendering APIs which may 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.

OpenGL ES commands are not guaranteed to be atomic. Some OpenGL ES rendering 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 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 the **`glFinish`**, **`eglWaitGL`**, and **`eglWaitNative`**

commands, as well as (if they exist) synchronization commands present in native rendering APIs. OpenGL ES 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 OpenGL ES and native rendering is done.

2.6 Power Management

EGL 1.0 does not address power management issues. Although this is an important area for developing robust applications on mobile devices, we instead encourage implementations to provide platform notes documenting interaction of EGL and OpenGL ES with platform-specific power management issues, including event detection, scope and nature of resource loss, behavior of EGL and OpenGL ES calls under resource loss, and recommended techniques for recovering from events.

Implementations are expected to develop EGL extensions to assist with power management. Future versions of EGL are expected to develop crossplatform power management support based on these extensions.

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, in the form of an error code, can be obtained by calling

```
EGLint eglGetError();
```

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.

`EGL_BAD_ACCESS`

EGL cannot access a requested resource (for example, a context is bound in another thread).

`EGL_BAD_ALLOC`

EGL failed to allocate resources for the requested operation.

EGL_BAD_ATTRIBUTE

An unrecognized attribute or attribute value was passed in an attribute list.

EGL_BAD_CONTEXT

An `EGLContext` argument does not name a valid `EGLContext`.

EGL_BAD_CONFIG

An `EGLConfig` argument does not name a valid `EGLConfig`.

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`; or, `EGL` is not initialized on the specified `EGLDisplay`.

EGL_BAD_SURFACE

An `EGLSurface` argument does not name a valid surface (window, pbuffer, or pixmap) configured for OpenGL ES rendering.

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.

EGL_BAD_NATIVE_PIXMAP

A `NativePixmapType` argument does not refer to a valid native pixmap.

EGL_BAD_NATIVE_WINDOW

A `NativeWindowType` argument does not refer to a valid native window.

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.

EGL normally checks the validity of objects passed into it, but detecting invalid native objects (pixmap, windows, and displays) may not always be possible. Specifying such invalid handles may result in undefined behavior, although implemen-

tations should generate `EGL_BAD_NATIVE_PIXMAP` and `EGL_BAD_NATIVE_WINDOW` errors if possible.

3.2 Initialization

Initialization must be performed once for each display prior to calling most other EGL functions. A display can be obtained by calling

```
EGLDisplay eglGetDisplay(NativeDisplayType
    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 would require *display_id* to be an `X Display`, while an implementation under Microsoft Windows would require *display_id* to be a Windows Device Context. If *display_id* is `EGL_DEFAULT_DISPLAY`, a *default display* is returned.

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. *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 OpenGL ES on a display, call

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

Termination marks **all** EGL-specific resources associated with the specified display for deletion. If contexts or surfaces created with respect to *dpy* are *current* (see section 3.6.3) to any thread, then they are not actually released while they remain current. Such contexts and surfaces will be destroyed, and all future references to them will become invalid, as soon as any otherwise valid **eglMakeCurrent** call is made from the thread they are bound to.

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 references to them are not valid.

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. *name* may be EGL_VENDOR, EGL_VERSION, or EGL_EXTENSIONS. The format and contents of the EGL_VENDOR string is implementation dependent. 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 EGL_VERSION string is laid out as follows:

```
<major_version.minor_version><space><vendor-specific info>
```

Both the major and minor portions of the version number are of arbitrary length. 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.

`EGL_BUFFER_SIZE` gives the total depth of the color buffer in bits; this is the sum of `EGL_RED_SIZE`, `EGL_GREEN_SIZE`, `EGL_BLUE_SIZE`, and `EGL_ALPHA_SIZE`.

`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 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.

`EGL_SURFACE_TYPE` is a mask indicating the surface types 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.

`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.

Attribute	Type	Notes
EGL_BUFFER_SIZE	integer	depth of 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_ALPHA_SIZE	integer	bits of Alpha in the color buffer
EGL_CONFIG_CAVEAT	enum	any caveats for the configuration
EGL_CONFIG_ID	integer	unique EGLConfig identifier
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_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_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.

EGL Token Name	Description
EGL_WINDOW_BIT	EGLConfig supports windows
EGL_PIXMAP_BIT	EGLConfig supports pixmaps
EGL_PBUFFER_BIT	EGLConfig supports pbuffers

Table 3.2: Types of surfaces supported by an EGLConfig

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 `XVisual`.

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.

OpenGL ES conformance requires that a set of `EGLConfig`s 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 specification.

`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 $(2^{**}EGL_RED_SIZE)-1$.

`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 `EGL_MAX_PBUFFER_PIXELS`.

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 3.3) 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`. 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 3.3 and 3.1, as described below under **Selection and Sorting**.

Selection of `EGLConfigs`

Attributes are matched in an attribute-specific manner, as shown in Table 3.3. The match criteria listed in the table have the following meanings¹:

Smaller `EGLConfigs` with an attribute value that meets or exceeds the specified value are matched.

Larger `EGLConfigs` with an attribute value that meets or exceeds the specified value are matched.

Exact `EGLConfigs` whose attribute value equals the requested value are matched.

Mask `EGLConfigs` for which the set bits of attribute include all the bits that are set in the requested value are matched. (Additional bits might be set in the attribute).

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.

¹ The distinction between Smaller and Larger, which affects only sorting, not selection, has proven confusing. We will update table 3.3 with separate selection criteria and sort order columns in the next EGL revision.

Attribute	Default	Selection and Sorting Criteria	Sort Priority
EGL_BUFFER_SIZE	0	<i>Smaller</i>	3
EGL_RED_SIZE	0	<i>Larger</i>	2
EGL_GREEN_SIZE	0	<i>Larger</i>	2
EGL_BLUE_SIZE	0	<i>Larger</i>	2
EGL_ALPHA_SIZE	0	<i>Larger</i>	2
EGL_CONFIG_CAVEAT	EGL_DONT_CARE	<i>Exact</i>	1
EGL_CONFIG_ID	EGL_DONT_CARE	<i>Exact</i>	9 (last)
EGL_DEPTH_SIZE	0	<i>Smaller</i>	6
EGL_LEVEL	0	<i>Exact</i>	
EGL_NATIVE_RENDERABLE	EGL_DONT_CARE	<i>Exact</i>	
EGL_NATIVE_VISUAL_TYPE	EGL_DONT_CARE	<i>Exact</i>	8
EGL_SAMPLE_BUFFERS	0	<i>Smaller</i>	4
EGL_SAMPLES	0	<i>Smaller</i>	5
EGL_STENCIL_SIZE	0	<i>Smaller</i>	7
EGL_SURFACE_TYPE	EGL_WINDOW_BIT	<i>Mask</i>	
EGL_TRANSPARENT_TYPE	EGL_NONE	<i>Exact</i>	
EGL_TRANSPARENT_RED_VALUE	EGL_DONT_CARE	<i>Exact</i>	
EGL_TRANSPARENT_GREEN_VALUE	EGL_DONT_CARE	<i>Exact</i>	
EGL_TRANSPARENT_BLUE_VALUE	EGL_DONT_CARE	<i>Exact</i>	

Table 3.3: Default values and match criteria for EGLConfig attributes.

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

Sorting of `EGLConfigs`

If more than one matching `EGLConfig` is found, then a list of `EGLConfigs`, sorted according to the *best* match criteria, is returned. The list is sorted according to the following precedence rules that are applied in ascending order (i.e., configurations that are considered equal by lower numbered rule are sorted by the higher numbered rule):

1. By `EGL_CONFIG_CAVEAT` where the precedence is `EGL_NONE`, `EGL_SLOW_CONFIG`, `EGL_NON_CONFORMANT_CONFIG`.
2. Larger total number of `RGBA` color bits (`EGL_RED_SIZE`, `EGL_GREEN_SIZE`, `EGL_BLUE_SIZE`, plus `EGL_ALPHA_SIZE`). 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.
3. Smaller `EGL_BUFFER_SIZE`.
4. Smaller `EGL_SAMPLE_BUFFERS`.
5. Smaller `EGL_SAMPLES`.
6. Smaller `EGL_DEPTH_SIZE`.
7. Smaller `EGL_STENCIL_SIZE`.
8. By `EGL_NATIVE_VISUAL_TYPE` (the actual sort order is implementation-defined, depending on the meaning of native visual types).
9. Smaller `EGL_CONFIG_ID` (this is always the last sorting rule, and guarantees a unique ordering).

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 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.

Refer to Table 3.1 and Table 3.3 for a list of valid EGL attributes.

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 with attributes corresponding to the desired `EGLConfig` (e.g. with the same color depth, with other constraints specific to the platform). Using a platform-specific type (here called `NativeWindowType`) referring to a handle to that native window, then call:

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

eglCreateWindowSurface creates an onscreen `EGLSurface` and returns a handle to it. Any EGL rendering 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**. Currently no attributes are recognized, so *attrib_list* will normally be `NULL` or empty (first attribute of `EGL_NONE`). However, it is possible that some platforms will define attributes specific to those environments, as an EGL extension.

On failure **eglCreateWindowSurface** returns `EGL_NO_SURFACE`. If the attributes of *win* do not correspond to *config*, 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* 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 `EGLConfig` 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 APIs other than OpenGL ES and EGL.

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`. Currently only three attributes can be specified in *attrib_list*: `EGL_WIDTH`, `EGL_HEIGHT`, and `EGL_LARGEST_PBUFFER`. 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 of `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. The default values for `EGL_WIDTH` and `EGL_HEIGHT` are zero.

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. Use `eglQuerySurface` to retrieve the dimensions of the allocated pbuffer. By default, `EGL_LARGEST_PBUFFER` is `EGL_FALSE`.

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

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 *config* does not support pbuffers, an `EGL_BAD_MATCH` error is generated.

3.5.3 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 OpenGL ES and EGL.

To create a pixmap rendering surface, first create a native platform pixmap with attributes corresponding to the desired `EGLConfig` (e.g. with the same color depth, with other constraints specific to the platform). Using a platform-specific type (here called `NativePixmapType`) referring to a handle to that native pixmap, then call:

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

eglCreatePixmapSurface creates an offscreen `EGLSurface` and returns a handle to it. Any EGL rendering 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**. Currently no attributes are recognized, so *attrib_list* will normally be `NULL` or empty (first attribute of `EGL_NONE`). However, it is possible that some platforms will define attributes specific to those environments, as an EGL extension.

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* 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.

3.5.4 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* are marked for deletion as soon as possible. If *surface* is current to any thread (see section 3.6.3), resources are not actually released while the surface remains current. Future references to *surface* remain valid only so long as it is current; it will be destroyed, and all future references to it will become invalid, as soon as any otherwise valid **eglMakeCurrent** call is made from the thread it is bound to.

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

3.5.5 Querying Surface Attributes

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 `EGL_WIDTH`, `EGL_HEIGHT`, `EGL_LARGEST_PBUFFER`, or `EGL_CONFIG_ID`.

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 3.8.1). 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`).

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 Contexts

3.6.1 Creating Rendering Contexts

To create an OpenGL ES rendering context, call

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

If **eglCreateContext** succeeds, it initializes the rendering context to the initial OpenGL ES state and returns a handle to it. The handle can be used to render to any compatible `EGLSurface`.

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

Currently no attributes are recognized, so *attrib_list* will normally be `NULL` or empty (first attribute of `EGL_NONE`). However, it is possible that some platforms will define attributes specific to those environments, as an EGL extension.

On failure **eglCreateContext** returns `EGL_NO_CONTEXT`. If *share_context* is neither zero nor a valid EGL rendering context, then an `EGL_BAD_CONTEXT` error is generated. If *config* is not a valid `EGLConfig`, then an `EGL_BAD_CONFIG` error is generated. If the 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.

3.6.2 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. If *ctx* is current to any thread (see section 3.6.3), resources are not actually released while the context remains current. Future references to *ctx* remain valid only so long as it is current; it will be destroyed, and all future references to it will become invalid, as soon as any otherwise valid **eglMakeCurrent** call is made from the thread it is bound to).

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

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

If the calling thread already has a current rendering context, then that context is flushed and marked as no longer current. *ctx* is made the current context for the calling thread.

eglMakeCurrent returns `EGL_FALSE` on failure. 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 *ctx* is not a valid EGL rendering 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 will be generated.

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 frame buffer state becomes 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`. 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 *ctx* 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.

Only one rendering context may be in use, or *current*, for a particular thread at a given time, and only one context may be bound to a particular surface at a given time.

The minimum number of current rendering contexts that must be supported by an EGL implementation is one.

To get the current context, call

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

If there is no current context, `EGL_NO_CONTEXT` is returned (this is not an error).

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

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

readdraw is either `EGL_READ` or `EGL_DRAW` to respectively return the read or draw surfaces. If there is no corresponding surface, `EGL_NO_SURFACE` is returned (this is

not an error) If *redraw* is neither `EGL_READ` nor `EGL_DRAW`, `EGL_NO_SURFACE` is returned and an `EGL_BAD_PARAMETER` error is generated.

To get the display associated with the current context, call

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

If there is no current context, `EGL_NO_DISPLAY` is returned.

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`.

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

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.

3.7 Synchronization Primitives

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

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

OpenGL ES calls made prior to **eglWaitGL** are guaranteed to be executed before native rendering calls made after **eglWaitGL** which affect the surface associated with the calling thread's current context. The same result can be achieved using **glFinish**. Clients rendering to single buffered surfaces (e.g. pixmap surfaces) should call **eglWaitGL** before accessing the native pixmap from the client.

eglWaitGL returns `EGL_TRUE` on success. If there is no current rendering context, 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.

To prevent the OpenGL ES 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 surface associated with the calling thread's current context, are guaranteed to be executed before OpenGL ES 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, 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 than OpenGL ES itself.

eglWaitNative returns `EGL_TRUE` on success. If there is no current rendering context, the function has no effect but still returns `EGL_TRUE`. If the 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.

3.8 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.

3.8.1 Posting to a Window

To post the color buffer to a window, call

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

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

The color buffer of *surface* is left in an undefined state after calling **eglSwapBuffers**.

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.

3.8.2 Copying to a Native Pixmap

To copy the color buffer to a native pixmap, call

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

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

The target pixmap should have the same number of components and component sizes as the color buffer it's being copied from. Implementations may choose to relax this restriction by converting data to the native pixmap formats. If they do so, they should define an EGL extension specifying which pixmap formats are supported, and specifying the conversion arithmetic used.

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 OpenGL ES .

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

3.8.3 Posting Semantics

In EGL 1.0, *surface* must be bound to the current context. This restriction is expected to 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 **glFlush**. Subsequent OpenGL ES commands can be issued immediately, but will not be executed until posting is completed (for **eglSwapBuffers**, this is typically during vertical retrace of the display).

3.8.4 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 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, an `EGL_BAD_MATCH` 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.

3.9 Obtaining Extension Function Pointers

The GL 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 extension functions may be queried at runtime. The function

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

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 that 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 query **glGetString(GL_EXTENSIONS)** (for OpenGL ES extensions) or **eglQueryString(dpy, EGL_EXTENSIONS)** (for EGL extensions) to determine if an extension is supported by a particular context.

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

eglGetProcAddress may be queried for all of the following functions:

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

eglGetProcAddress may not be queried for core (non-extension) functions in GL and EGL. For functions that are queryable with **eglGetProcAddress**, implementations may choose to also export those functions statically from the OpenGL ES link library. However, portable clients cannot rely on this behavior.

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 such as the Khronos SIG and the OpenGL ARB use similar identifiers (OML and OES for Khronos, and ARB for the ARB).

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

<http://oss.sgi.com/projects/ogl-sample/registry/>

for more information on defining extensions.

Chapter 5

EGL Versions and Enumerants

Each version of EGL supports a specified OpenGL ES version, and all prior versions of OpenGL ES up to that version. EGL 1.0 supports OpenGL ES 1.0, including both Common and Common-Lite profiles.

5.1 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. The base EGL specification defines an ISO C language binding, and in that environment, the EGL header file `<GLES/egl.h>` must define a C preprocessor symbol:

```
#define EGL_VERSION_1_0 1
```

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

5.2 Enumerant Values

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/>

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 processes 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.

Compatible an 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 ES rendering context. This is a virtual OpenGL ES machine. All 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.

EGLContext a handle to a rendering context. Rendering contexts consist of client side state and server side 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 processes 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 Strom, 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

Index of EGL Commands

EGL_*_SIZE, 12
EGL_ALPHA_SIZE, 12, 13, 17, 18
EGL_BAD_ACCESS, 8, 24
EGL_BAD_ALLOC, 8, 20–22, 24, 25
EGL_BAD_ATTRIBUTE, 9, 15, 19, 23, 26
EGL_BAD_CONFIG, 9, 19, 21, 23
EGL_BAD_CONTEXT, 9, 23, 24, 26
EGL_BAD_CURRENT_SURFACE, 9, 25–27
EGL_BAD_DISPLAY, 9–11
EGL_BAD_MATCH, 9, 19, 21, 23–25, 29
EGL_BAD_NATIVE_PIXMAP, 9, 10, 21, 29
EGL_BAD_NATIVE_WINDOW, 9, 10, 19, 25, 29
EGL_BAD_PARAMETER, 9, 11, 15, 26, 27
EGL_BAD_SURFACE, 9, 22–24, 29
EGL_BLUE_SIZE, 12, 13, 17, 18
EGL_BUFFER_SIZE, 12, 13, 17, 18
EGL_CONFIG_CAVEAT, 13, 14, 17, 18
EGL_CONFIG_ID, 12, 13, 16–18, 22, 26
EGL_CORE_NATIVE_ENGINE, 27
EGL_DEPTH_SIZE, 13, 17, 18
EGL_DONT_CARE, 15–18
EGL_DRAW, 25, 26
EGL_EXTENSIONS, 11, 29
EGL_FALSE, 2, 8, 10, 15, 19, 20, 22–24, 26, 27, 29
EGL_GREEN_SIZE, 12, 13, 17, 18
EGL_HEIGHT, 20, 22
EGL_LARGEST_PBUFFER, 20, 22, 23
EGL_LEVEL, 13, 16, 17
EGL_MAX_PBUFFER_HEIGHT, 13, 14, 16
EGL_MAX_PBUFFER_PIXELS, 13–16
EGL_MAX_PBUFFER_WIDTH, 13, 14, 16
EGL_NATIVE_RENDERABLE, 12, 13, 17
EGL_NATIVE_VISUAL_ID, 13, 14, 16
EGL_NATIVE_VISUAL_TYPE, 13, 14, 16–18
EGL_NEW_DEFINITION_SGI, 31
EGL_NO_CONTEXT, 8, 23, 25
EGL_NO_DISPLAY, 10, 26
EGL_NO_SURFACE, 19, 21, 25, 26
EGL_NON_CONFORMANT_CONFIG, 14, 18
EGL_NONE, 14–21, 23
EGL_NOT_INITIALIZED, 8–11, 15
EGL_PBUFFER_BIT, 12, 13
EGL_PIXMAP_BIT, 12, 13, 21
EGL_READ, 25, 26
EGL_RED_SIZE, 12–14, 16–18
EGL_SAMPLE_BUFFERS, 12, 13, 17, 18
EGL_SAMPLES, 12, 13, 17, 18
EGL_SLOW_CONFIG, 14, 18
EGL_STENCIL_SIZE, 13, 17, 18
EGL_SUCCESS, 8
EGL_SURFACE_TYPE, 12, 13, 16, 17, 19, 21
EGL_TRANSPARENT_BLUE_VALUE, 13, 14, 16, 17
EGL_TRANSPARENT_GREEN_VALUE, 13, 14, 16, 17
EGL_TRANSPARENT_RED_VALUE, 13, 14, 16, 17
EGL_TRANSPARENT_RGB, 14

EGL_TRANSPARENT_TYPE, 13, 14, 16, 17
EGL_TRUE, 2, 8, 10, 11, 13, 15, 19, 23, 26, 27
EGL_VENDOR, 11
EGL_VERSION, 11
EGL_WIDTH, 20, 22
EGL_WINDOW_BIT, 12, 13, 16, 17, 19
EGLBoolean, 2, 8, 12
eglChooseConfig, 12, 15, 18–21
EGLConfig, 3, 9, 12–23, 26
EGLContext, 9, 23
eglCopyBuffers, 4, 28, 29
eglCreateContext, 23
eglCreatePbufferSurface, 14, 20–22
eglCreatePixmapSurface, 21, 22
eglCreateWindowSurface, 19, 20
eglDestroyContext, 24
eglDestroySurface, 22
EGLDisplay, 3, 9–11, 18
eglGetConfigAttrib, 19
eglGetConfigs, 15, 18
eglGetCurrentContext, 25
eglGetCurrentDisplay, 26
eglGetCurrentSurface, 25
eglGetDisplay, 10
eglGetError, 8
eglGetProcAddress, 29, 30
eglInitialize, 10, 11
EGLint, 2
eglMakeCurrent, 11, 22, 24, 25
eglNewCommandSGI, 31
eglQueryContext, 26
eglQueryString, 11, 29
eglQuerySurface, 20, 22, 23
EGLSurface, 3, 9, 12, 19, 21–24
eglSwapBuffers, 4, 27–29
eglTerminate, 10, 11
eglWaitGL, 6, 26
eglWaitNative, 6, 26, 27

GL_EXTENSIONS, 29
GL_TEXTURE_2D, 6
glBindTexture, 6
glFinish, 6, 26
glFlush, 28
glGetString, 29
glReadPixels, 25
glScissor, 25
glViewport, 25