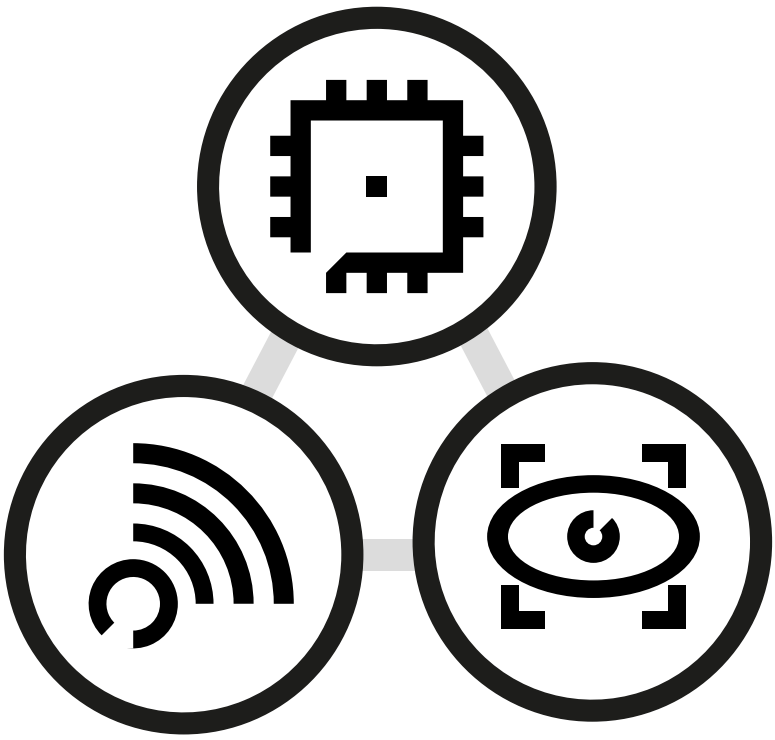
PowerVR Supported Extensions for OpenGL® ES and EGL

Revision: 1.0
11/02/2020
Public



Published: 11/02/2020-09:04

Contents

1. Document Overview

This document provides a reference for all of the extensions supported by the PowerVR reference driver implementation

What are OpenGL ES and EGL extensions?

OpenGL ES extensions provide new or expanded functionality that is not supported by the core of OpenGL ES, by default. Using extensions should not be looked on as something to be avoided, instead it should be a standard, accepted practice for the OpenGL ES user.

Document Overview

The purpose of this document is to serve as a reference for all of the extensions supported by the PowerVR reference driver implementation. Each page in the next two section contains information about a particular extension.

Whilst the information about each extension will vary, each extension page will generally include:

- **Valid APIs** - The API versions of OpenGL ES or EGL which are compatible with the extension
- **Description** - A short description of the functionality of the extension
- **Registry Link** - A link to the Khronos registry entry about this extension
- **Example** - An example of how the extension can be used in an application

Note: While the PowerVR reference driver implementation supports all of the extensions mentioned in this document, it is the decision of our licensees as to which are exposed in their own implementations.

2. EGL Extensions

EGL_ANDROID_blob_cache

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

EGL 1.4

Descriptions

Shader compilation and optimization has been a troublesome aspect of OpenGL programming for a long time. It can consume seconds of CPU cycles during application start-up. Additionally, state-based re-compiles done internally by the drivers add an unpredictable element to application performance tuning, often leading to occasional pauses in otherwise smooth animations.

This extension provides a mechanism through which client API implementations may cache shader binaries after they are compiled. It may then retrieve those cached shaders during subsequent executions of the same program. The management of the cache is handled by the application (or middleware), allowing it to be tuned to a particular platform or environment.

While the focus of this extension is on providing a persistent cache for shader binaries, it may also be useful for caching other data. This is perfectly acceptable, but the guarantees provided (or lack thereof) were designed around the shader use case.

Note

Although this extension is written as if the application implements the caching functionality, on the Android OS it is implemented as part of the Android EGL module. This extension is not exposed to applications on Android but will be used automatically in every application that uses EGL, if it is supported by the underlying device-specific EGL implementation.

On the Android OS, the extension doesn’t provide a mechanism to query the cache size and that the cache size is platform specific. If developers have lots of shaders and want to make sure they are cached, they should cache them manually with GL_OES_get_program_binary in ES 2.0 or glGetProgramBinary() in ES 3.0+.

Registry Link

https://www.khronos.org/registry/egl/extensions/ANDROID/EGL_ANDROID_blob_cache.txt

Example

```
void setFunction(const void* key, EGLsizeiANDROID keySize,
           const void* value, EGLsizeiANDROID valueSize)
{
  //set key-value pair into cache
}
EGLsizeiANDROID getFunction(const void* key, EGLsizeiANDROID keySize,
    void* value, EGLsizeiANDROID valueSize)
{
    //get value from cache
}
eglSetBlobCacheFuncsANDROID(eglDisplay, setFunction, getFunction);
```

EGL_ANDROID_framebuffer_target

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

EGL 1.4

Descriptions

Android supports a number of different ANativeWindow implementations that can be used to create an EGLSurface. One implementation, which is used to send the result of performing window composition to a display, may have some device-specific restrictions. Because of this, some EGLConfigs may be incompatible with these ANativeWindows. This extension introduces a new boolean EGLConfig attribute that indicates whether the EGLConfig supports rendering to an ANativeWindow for which the buffers are passed to the HWComposer HAL as a framebuffer target layer.

Note

This extension is written against the wording of the EGL 1.4 Specification

Registry Link

https://www.khronos.org/registry/egl/extensions/ANDROID/EGL_ANDROID_framebuffer_target.txt

Example

```
// Set up the config attributes, specifying that we want a config which
// supports rendering to an ANativeWindow for which the buffers are passed
// to the HWComposer HAL as a framebuffer target layer.
EGLint configAttribs[] =
{
 EGL_SURFACE_TYPE, EGL_WINDOW_BIT,
 EGL_RENDERABLE_TYPE , EGL_OPENGL_ES2_BIT,
 EGL_FRAMEBUFFER_TARGET_ANDROID, EGL_TRUE,
 EGL_NONE
};
// Choose an appropriate configuration - just get the first available one that matches in this
 case
EGLint iConfigs;
EGLConfig eglConfig;
eglChooseConfig(eglDisplay, configAttribs, &eglConfig, 1, &iConfigs);
//the buffer will be passed to the HWComposer HAL as a framebuffer target layer
```

```
eglSurface = eglCreateWindowSurface(eglDisplay, eglConfig, (EGLNativeWindowType)nativeWindow,
 NULL);
```

EGL_ANDROID_front_buffer_auto_refresh

Valid APIs

EGL 1.2+

Description

This extension is intended for latency-sensitive applications that are doing front-buffer rendering. It allows them to indicate to the Android compositor that it should perform composition every time the display refreshes. This removes the overhead of having to notify the compositor that the window surface has been updated, but it comes at the cost of doing potentially unneeded composition work if the window surface has not been updated.

Registry Link

https://www.khronos.org/registry/EGL/extensions/ANDROID/EGL_ANDROID_front_buffer_auto_refresh.txt

EGL_ANDROID_image_native_buffer

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

EGL 1.2

Description

This extension enables using an Android window buffer (struct ANativeWindowBuffer) as an EGLImage source.

Note

EGL 1.2 is required.

EGL_KHR_image_base is required.

This extension is written against the wording of the EGL 1.2 Specification.

Registry Link

https://www.khronos.org/registry/egl/extensions/ANDROID/EGL_ANDROID_image_native_buffer.txt

Example

```
ANativeWindowBuffer* sSrcBuffer = graphicBuffer->getNativeBuffer();
EGLint attrs[] = {
    EGL_IMAGE_PRESERVED_KHR,    EGL_TRUE,
    EGL_NONE,
};
EGLImageKHR eglSrcImage =eglCreateImageKHR(eglDisplay, EGL_NO_CONTEXT,
 EGL_NATIVE_BUFFER_ANDROID,
(EGLClientBuffer)&sSrcBuffer, attrs);
```

EGL_ANDROID_native_fence_sync

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

EGL 1.1

Description

This extension enables the creation of EGL fence sync objects that are associated with a native synchronization fence object that is referenced using a file descriptor. These EGL fence sync objects have nearly identical semantics to those defined by the KHR_fence_sync extension, except that they have an additional attribute storing the file descriptor referring to the native fence object.

This extension assumes the existence of a native fence synchronization object that behaves similarly to an EGL fence sync object. These native objects must have a signal status like that of an EGLSyncKHR object that indicates whether the fence has ever been signaled. Once signaled the native object's signal status may not change again.

Note

Requires EGL 1.1

This extension is written against the wording of the EGL 1.2 Specification.

EGL_KHR_fence_sync is required.

Registry Link

https://www.khronos.org/registry/egl/extensions/ANDROID/EGL_ANDROID_native_fence_sync.txt

Example

```
//===============
// Process A:
//===============
EGLSyncKHR eglSyncKHR = eglCreateSyncKHR(eglDisplay, EGL_SYNC_REUSABLE_KHR, NULL);
EGLint FileDescriptor = eglDupNativeFenceFDANDROID(eglDisplay, eglSyncKHR);
//===============
// Process B:
//===============
```

```
EGLint attrs[] = {
  EGL_SYNC_NATIVE_FENCE_FD_ANDROID, FileDescriptor,
  EGL_SYNC_CONDITION_KHR, EGL_SYNC_NATIVE_FENCE_SIGNALED_ANDROID
  EGL_NONE,
};
EGLSyncKHR eglSyncKHR = eglCreateSyncKHR(eglDisplay, EGL_SYNC_NATIVE_FENCE_ANDROID, attrs);
eglClientWaitSyncKHR(eglDisplay, eglSyncKHR, 0, EGL_FOREVER_KHR);
```

EGL_ANDROID_presentation_time

Valid APIs

EGL 1.1+

Description

Often when rendering a sequence of images, there is some time at which each image is intended to be presented to the viewer. This extension allows this desired presentation time to be specified for each frame rendered to an EGLSurface, allowing the native window system to use it.

Registry Link

https://www.khronos.org/registry/EGL/extensions/ANDROID/EGL_ANDROID_presentation_time.txt

Example

```
EGLBoolean result = eglPresentationTimeANDROID(
                        display,
                        surface,
                        time);
```

EGL_ANDROID_recordable

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

EGL 1.4

Description

Android supports a number of different ANativeWindow implementations that can be used to create an EGLSurface. One implementation, which records the rendered image as a video each time eglSwapBuffers gets called, may have some device-specific restrictions. Because of this, some EGLConfigs may be incompatible with these ANativeWindows.

This extension introduces a new boolean EGLConfig attribute that indicates whether the EGLConfig supports rendering to an ANativeWindow that records images to a video.

Note

Requires EGL 1.0

This extension is written against the wording of the EGL 1.4 Specification

Registry Link

https://www.khronos.org/registry/egl/extensions/ANDROID/EGL_ANDROID_recordable.txt

Example

```
// Set up the config attributes, specifying that we want a config which
// support recording the rendered image as a video.
EGLint configAttribs[] =
{
 EGL_SURFACE_TYPE, EGL_WINDOW_BIT,
 EGL_RENDERABLE_TYPE , EGL_OPENGL_ES2_BIT,
 EGL_RECORDABLE_ANDROID, EGL_TRUE,
 EGL_NONE
};
// Choose an appropriate configuration - just get the first available one that matches in this
 case
EGLint iConfigs;
EGLConfig eglConfig;
eglChooseConfig(eglDisplay, configAttribs, &eglConfig, 1, &iConfigs);
```

EGL_EXT_buffer_age

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

EGL 1.4

Description

This extension lets applications query the age of the back buffer contents for an EGL surface as the number of frames elapsed since the contents were most recently defined. The back buffer can either be reported as invalid (has an age of 0) or it may be reported to contain the contents from n frames prior to the current frame.

Once the application has queried the buffer age, the age of contents remains valid until the end of the frame for all pixels that continue to pass the pixel ownership test.

For many use-cases this extension can provide an efficient alternative to using the EGL_BUFFER_PRESERVED swap behaviour. The EGL_BUFFER_PRESERVED swap behaviour adds a direct dependency for any frame n on frame n - 1 which can affect the pipelining of multiple frames but also implies a costly copy-back of data to initialize the back-buffer at the start of each frame.

Note

Requires EGL 1.4

This extension is written against the wording of the EGL 1.4 Specification.

Registry Link

https://www.khronos.org/registry/egl/extensions/EXT/EGL_EXT_buffer_age.txt

Example

```
EGLint iBufferAge;
eglQuerySurface(eglDisplay, eglSurface, EGL_BUFFER_AGE_EXT, &iBufferAge);
```

EGL_EXT_create_context_robustness

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

EGL 1.4

Description

Several recent trends in how OpenGL integrates into modern computer systems have created new requirements for robustness and security for OpenGL rendering contexts. This extension introduces the concept of robust contexts for OpenGL ES, by providing context reset strategies on hardware resets, and concessions for robust buffer access.

The extension adds two new attributes for context creation:

- EGL_CONTEXT_OPENGL_ROBUST_ACCESS_EXT: This guarantees a level of safety for all buffer accesses in the context, so that attempting to access data outside of a buffer's bounds will result in undefined values being returned but must not result in program termination. The exact behaviour of this is defined in the GL_EXT_robustness extension.
- EGL_CONTEXT_OPENGL_RESET_NOTIFICATION_STRATEGY_EXT: This allows users to force a context to be deleted in the event of it causing a hardware reset, notifying the application of such an event.

More information on what happens with these strategies is contained within the GL_EXT_robustness specification.

Note

This functionality is core to EGL 1.5, so the extension is no longer needed.

Registry Link

http://www.khronos.org/registry/egl/extensions/EXT/EGL_EXT_create_context_robustness.txt

Example

```
// Create a robust context with hardware notifications
EGLint contextAttribs[] =
{
    EGL_CONTEXT_OPENGL_ROBUST_ACCESS_EXT, EGL_TRUE,
    EGL_CONTEXT_OPENGL_RESET_NOTIFICATION_STRATEGY_EXT, EGL_LOSE_CONTEXT_ON_RESET_EXT,
    EGL_NONE
};
EGLContext context = eglCreateContext(display, config, NULL, contextAttribs);
```

EGL_IMG_context_priority

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

EGL 1.0, 1.1, 1.2, 1.3, 1.4, 1.5

Description

This extension allows a developer to set the execution priority of an EGL context, by setting a new attribute when creating the context: EGL_CONTEXT_PRIORITY_LEVEL_IMG. This can be useful in a multi-process environment, where an API-utilising task can be considered lower or higher priority.

Three priority levels are described in this extension, EGL_CONTEXT_PRIORITY_HIGH_IMG, EGL_CONTEXT_PRIORITY_MEDIUM_IMG and EGL_CONTEXT_PRIORITY_LOW_IMG, meaning high, medium and low priority, respectively. By default, all contexts are created with medium priority, and so are on equal execution footing. The specification of the priority is only a hint though, and so can be ignored by the implementation - this is typical when, for example, high priority contexts are reserved for system processes.

A query is provided to obtain the real priority of the client after it has been created.

Note

For Series5 and Series5XT implementations of this extension, developer can only set a priority per process, no matter how many contexts are in that process. To set multiple priorities, multiple processes must be used, and EGLImages typically used for sharing. For Series6, Series6XE and Series6XT implementations of this extension, developer can set priority at a context granularity rather than process granularity.

Registry Link

http://www.khronos.org/registry/egl/extensions/IMG/EGL_IMG_context_priority.txt

Example

```
// Set up the context attributes, specifying OpenGL ES 3.0 support and a low priority context.
EGLint contextAttribs[] =
{
 EGL_CONTEXT_CLIENT_VERSION, 3,
 EGL_CONTEXT_PRIORITY_LEVEL_IMG, EGL_CONTEXT_PRIORITY_LOW_IMG,
```

```
 EGL_NONE
};
// Create the context with the context attributes supplied
eglContext = eglCreateContext(eglDisplay, eglConfig, NULL, contextAttribs);
```

EGL_KHR_create_context

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

EGL 1.4

Description

This extension enables a number of new context attributes to be specified in EGL to allow greater control of the returned context. The new functionality consists of four aspects; version control, core vs compatibility layer control for OpenGL, debugging (see extension GL_KHR_debug), and robustness in OpenGL (see desktop extension "GL_ARB_robustness"). This extension mirrors the desktop WGL_ARB_create_context and GLX_ARB_create_context extensions and provides similar functionality through EGL.

Version Control

This part of the extension enables developers to explicitly create a context with a given level of API support. An explicitly defined EGL_OPENGL_ES3_BIT is provided to query the new API via the original mechanism to make sure a compatible context is available, and then this extension allows developers to explicitly state which version of OpenGL ES they are actually going to be using. The two additional attributes to specify this are EGL_CONTEXT_MAJOR_VERSION_KHR (which is an alias for EGL_CONTEXT_CLIENT_VERSION) and EGL_CONTEXT_MINOR_VERSION_KHR. Each then accepts an integer value specifying what version you require.

OpenGL Core vs Compatibility

EGL_OPENGL_PROFILE_MASK_KHR allows a developer to specify EGL_CONTEXT_OPENGL_CORE_PROFILE_BIT_KHR or EGL_CONTEXT_OPENGL_COMPATIBILITY_PROFILE_BIT_KHR to explicitly choose a core or compatibility context when using the OpenGL client API.

There is also a bit that can be set for EGL_CONTEXT_FLAGS_KHR which tells the implementation to return only OpenGL contexts which do not support functionality marked as deprecated by any version of OpenGL after OpenGL 3.0. Further information on this is available in the OpenGL specification.

Debug Contexts

When the GL_KHR_debug extension exists in a client OpenGL/ES API, this bit flag tells the implementation to enable debug functionality for that API. This is necessary

to inform the underlying implementation that debug information should be tracked. This bit should only be set for purposes of debugging, and not in shipping code.

OpenGL Robustness

When setting EGL_CONTEXT_FLAGS_KHR, the EGL_CONTEXT_OPENGL_ROBUST_ACCESS_BIT_KHR bit is provided to allow a user to require buffer access to remain robust as defined in the GL_ARB_robustness extension.

Various problems in either hardware or software can occasionally cause a hardware reset in a GPU. Unlike a CPU, these can typically be recovered from without having to restart the device - though setting up the underlying context again is usually a necessity. Setting EGL_CONTEXT_OPENGL_RESET_NOTIFICATION_STRATEGY_KHR to either EGL_NO_RESET_NOTIFICATION_KHR or EGL_LOSE_CONTEXT_ON_RESET_KHR will set the behaviour to either notify the user or ignore it, as defined by GL_ARB_robustness using GL_NO_RESET_NOTIFICATION_ARB and GL_LOSE_CONTEXT_ON_RESET_ARB respectively.

Note: This only affects **OpenGL** contexts, **not OpenGL ES** contexts, which are handled by EGL_KHR_create_context_robustness.

Note

This functionality is core to EGL 1.5, so the extension is no longer needed.

Registry Link

http://www.khronos.org/registry/egl/extensions/KHR/EGL_KHR_create_context.txt

Example

```
// Set up the config attributes, specifying that we want a config with Window support and ES3
// compatibility.
EGLint configAttribs[] =
{
 EGL_SURFACE_TYPE, EGL_WINDOW_BIT,
 EGL_RENDERABLE_TYPE , EGL_OPENGL_ES3_BIT_KHR,
 EGL_NONE
};
// Choose an appropriate configuration - just get the first available one that matches here
EGLint iConfigs;
EGLConfig eglConfig;
eglChooseConfig(eglDisplay, configAttribs, &eglConfig, 1, &iConfigs);
// Set up the context attributes, specifying OpenGL ES 3.0 support and a debug context.
EGLint contextAttribs[] =
{
 EGL_CONTEXT_MAJOR_CLIENT_VERSION, 3,
 EGL_CONTEXT_MINOR_CLIENT_VERSION, 0,
 EGL_CONTEXT_FLAGS_KHR, EGL_CONTEXT_OPENGL_DEBUG_BIT_KHR,
 EGL_NONE
};
// Create the context with the context attributes supplied
eglContext = eglCreateContext(eglDisplay, eglConfig, NULL, contextAttribs);
```

EGL_KHR_fence_sync

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

EGL 1.1, 1.2, 1.3, 1.4

Description

This extension introduces sync objects to EGL and provides application developers with a mechanism of notifying the CPU when a GPU operation has completed. Fence sync objects are inserted into the GL command stream immediately after the call which the user wants to be notified about or wait on. Any thread can then make a call to eglClientWaitSync() to wait for the GPU to finish what it's doing, with a user specified timeout. A timeout of 0 equates to simply querying whether it's complete yet, similar to how queries work in later versions of OpenGL ES. A common use case for this is to be loading resources on one thread and informing the CPU when it can start submitting calls on another thread that can actually draw with them. The benefits of such multi-threaded rendering can allow developers to create games with seamless loading. To actually interact with client APIs such as OpenGL ES, the client API needs to have an extension that specifies it will be inserted into the same command stream. For OpenGL ES this is GL_OES_EGL_sync.

Note

This functionality is core to EGL 1.5, so the extension is no longer needed.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/EGL_KHR_fence_sync.txt

Example

```
// Create a fence sync object - there are no valid attributes that can be passed in so pass
// NULL or an attribute list containing only EGL_NONE.
EGLSyncKHR eglFenceSync = eglCreateSyncKHR(eglDisplay, EGL_SYNC_FENCE_KHR, NULL);
/*
    Wait for the sync object, flushing the context so it definitely completes in finite time.
    This call will wait forever until the context finishes what it's doing.
    This also works as a more flexible and guaranteed version of glFlush/glFinish, as you can
    specify a timeout.
*/
EGLint waitResult = eglClientWaitSyncKHR(eglDisplay, eglFenceSync,
 EGL_SYNCH_FLUSH_COMMANDS_BIT_KHR, EGL_FOREVER_KHR);
// Destroy the fence sync object once we're done with it.
EGLBoolean success = eglDestroySyncKHR(eglDisplay, eglFenceSync);
```

EGL_KHR_get_all_proc_addresses

Valid APIs

EGL 1.2+

Description

eglGetProcAddress is currently defined to not support the querying of non-extension EGL or client API functions. Non-extension functions are expected to be exposed as library symbols that can be resolved statically at link time, or dynamically at run time using OS-specific runtime linking mechanisms.

To avoid requiring applications to fall back to OS-specific dynamic linking mechanisms, this extension drops the requirement that eglGetProcAddress return only non-extension functions. If the extension string is present, applications can query all EGL and client API functions using eglGetProcAddress.

Registry Link

https://www.khronos.org/registry/EGL/extensions/KHR/EGL_KHR_get_all_proc_addresses.txt

EGL_KHR_gl_renderbuffer_image

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

EGL 1.2, 1.3, 1.4

Description

This extension extends EGL_KHR_image_base and enables EGLImages to be created from a source renderbuffer from OpenGL. This particular extension string denotes that an EGLImage can be created from a renderbuffer. This extension is also required to create an OpenGL renderbuffer from an EGLImage when GL_KHR_image is supported in the implementation.

Note

This functionality is core to EGL 1.5, so the extension is no longer needed.

Registry Link

http://www.khronos.org/registry/egl/extensions/KHR/EGL_KHR_gl_image.txt

Example

```
// Create an EGLImageKHR from an OpenGL ES Renderbuffer
EGLImageKHR eglImage = eglCreateImageKHR(eglDisplay, eglOpenGLESContext,
 EGL_GL_RENDERBUFFER_KHR, (EGLClientBuffer)anOpenGLRenderbuffer, NULL);
```

EGL_KHR_gl_texture_2D_image

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

EGL 1.2, 1.3, 1.4

Description

This extension extends EGL_KHR_image_base and enables EGLImages to be created from a source texture from OpenGL. This particular extension string denotes that an EGLImage can be created from a basic 2D texture. This extension is also required to create an OpenGL texture from an EGLImage when GL_KHR_image is supported in the implementation.

Note

This functionality is core to EGL 1.5, so the extension is no longer needed.

Registry Link

http://www.khronos.org/registry/egl/extensions/KHR/EGL_KHR_gl_image.txt

Example

```
// Attributes that tell the CreateImage command to use the first level of the texture
// (NB: This is the default and doesn't actually need to be specified, it's just for
// illustration.)
EGLint imageAttributes[] =
{
 EGL_GL_TEXTURE_LEVEL, 0,
 EGL_NONE
};
// Create an EGLImageKHR from an OpenGL ES Texture. The context which contains the OpenGL ES
// texture must be specified, as OpenGL ES is supported by an EGLContext.
EGLImageKHR eglImage = eglCreateImageKHR(eglDisplay, eglOpenGLESContext, EGL_GL_TEXTURE_2D_KHR,
  (EGLClientBuffer)anOpenGLTexture, imageAttributes);
```

EGL_KHR_gl_texture_cubemap_image

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

EGL 1.2, 1.3, 1.4

Description

This extension extends EGL_KHR_image_base and enables EGLImages to be created from a source texture from OpenGL. This particular extension string denotes that an EGLImage can be created from a cubemap texture. This extension is also required to

create an OpenGL texture from an EGLImage when GL_KHR_image is supported in the implementation.

Note

This functionality is core to EGL 1.5, so the extension is no longer needed.

Registry Link

http://www.khronos.org/registry/egl/extensions/KHR/EGL_KHR_gl_image.txt

Example

```
// Attributes that tell the CreateImage command to use the first level of the texture
// (NB: This is the default and doesn't actually need to be specified, it's just for
// illustration.)
EGLint imageAttributes[] =
{
 EGL_GL_TEXTURE_LEVEL, 0,
 EGL_NONE
};
// Create an EGLImageKHR from an OpenGL ES Cubemap Texture. The target in this case specifies
// that the image is created from the positive X face of the cubemap, as EGLImages are 2D only.
EGLImageKHR eglImage = eglCreateImageKHR(eglDisplay, eglOpenGLESContext,
 EGL_GL_TEXTURE_CUBE_MAP_POSITIVE_X_KHR , (EGLClientBuffer)anOpenGLCubeMap, imageAttributes);
```

EGL_KHR_image

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

EGL 1.2, 1.3, 1.4

Description

This was originally the extension which defined EGLImages, but pixmaps were core to this extension's operation. Subsequent implementations have wanted to support EGLImages without having pixmap support, and so this has been split into two extensions. EGL_KHR_image_base and EGL_KHR_image_pixmap are the two child extensions which when supported together are the equivalent of EGL_KHR_image.

Note

This functionality is core to EGL 1.5, so the extension is no longer needed.

Registry Link

http://www.khronos.org/registry/egl/extensions/KHR/EGL_KHR_image.txt

Example

```
// Create an EGLImageKHR from an EGL Pixmap
EGLImageKHR eglImage = eglCreateImageKHR(eglDisplay, eglOpenGLESContext, EGL_NATIVE_PIXMAP_KHR,
 (EGLClientBuffer)anEGLPixmap, NULL);
```

EGL_KHR_image_base

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

EGL 1.2, 1.3, 1.4

Description

This extension introduces image objects to EGL, which are a way of defining a somewhat generic object which stores something that may be considered an array of image data; textures, renderbuffers, pixmaps and the like. This extension defines what an EGLImage is to a developer and makes no assumptions about the underlying data - internally data is stored in a format that is friendly to whatever client APIs are supported by a given implementation. This extension also does not define any source targets for creating an EGLImage, this extension on its own is of little use beyond defining the object itself and the interface. Extensions which add functionality to create EGLImages from various sources are listed below: EGLEGL_KHR_image_pixmap OpenGL/ ESEGL_KHR_gl_texture_2D_imageEGL_KHR_gl_texture_cubemap_imageEGL_KHR_gl_texture_ OpenCLEGL_IMG_cl_image AndroidEGL_ANDROID_image_native_buffer LinuxEGL_EXT_image_dma_buf_import OpenVGEGL_KHR_vg_parent_image Other various client extensions enable the creation of a client object from an EGLImage, such as GL_KHR_EGL_image.

Note

This functionality is core to EGL 1.5, so the extension is no longer needed.

Registry Link

http://www.khronos.org/registry/egl/extensions/KHR/EGL_KHR_image_base.txt

Example

```
// Create an EGLImageKHR from an EGL Pixmap
EGLImageKHR eglImage = eglCreateImageKHR(eglDisplay, eglOpenGLESContext, EGL_NATIVE_PIXMAP_KHR,
 (EGLClientBuffer)anEGLPixmap, NULL);
```

EGL_KHR_image_pixmap

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

EGL 1.2, 1.3, 1.4, 1.5

Description

This extension extends EGL_KHR_image_base and enables the creation of an EGLImage an EGLNativePixmapType.

Note: This extension is not explicitly exposed on PowerVR hardware but is implied as supported whenever EGL_KHR_image is present, as it is a compound extension which includes both this and EGL_KHR_image_base.

Registry Link

http://www.khronos.org/registry/egl/extensions/KHR/EGL_KHR_image_pixmap.txt

Example

```
// Create an EGLImageKHR from an EGL Pixmap
EGLImageKHR eglImage = eglCreateImageKHR(eglDisplay, eglOpenGLESContext, EGL_NATIVE_PIXMAP_KHR,
 (EGLClientBuffer)anEGLPixmap, NULL);
```

EGL_KHR_mutable_render_buffer

Valid APIs

EGL 1.2+

Description

The aim of this extension is to allow toggling of front-buffer rendering for window surfaces after their initial creation. This allows for implementations to switch between back-buffered and single-buffered rendering without requiring re-creation of the surface. It is not expected for toggling to be a frequent event.

Registry Link

https://www.khronos.org/registry/EGL/extensions/KHR/EGL_KHR_mutable_render_buffer.txt

EGL_KHR_partial_update

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

EGL 1.4 or later is required.

Written based on the EGL 1.5 specification (March 12, 2014).

Description

The aim of this extension is to allow efficient partial updates for postable surfaces. It allows implementations to completely avoid processing areas of the surface which have not changed between frames, allowing increased efficiency.

It does so by providing information and guarantees about the content of the current back buffer which allow the application to "repair" only areas that have become out of date since the particular back buffer was last used.

The information provided is in the form of the "age" of the buffer, that is, how many frames ago it was last used as the back buffer for the surface. If the application tracks what changes it has made to the surface since this back buffer was last used, it can bring the entire back buffer up to date by only re-rendering the areas it knows to be out of date.

Use of this extension provides a more efficient alternative to EGL_BUFFER_PRESERVED swap behaviour. EGL_BUFFER_PRESERVED typically implies an expensive full-frame copy at the beginning of the frame, as well as a dependency on the previous frame. Usage of this extension avoids both and requires only the necessary updates to a back buffer to be made.

Note

The behavior of part of this extension is different depending on whether the EGL_EXT_buffer_age extension is also present. This extension trivially interacts with EGL_KHR_swap_buffers_with_damage and EGL_EXT_swap_buffers_with_damage. This extension is worded against the KHR version, but the interactions with the EXT version are identical.

Different with EGL_KHR_swap_buffers_with_damage. EGL_KHR_partial_update

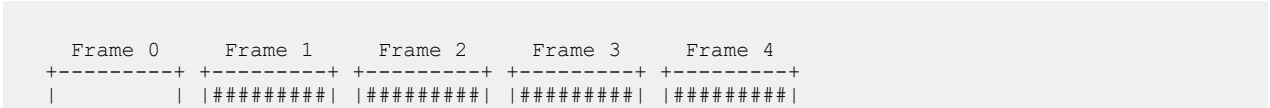
concerns the area of a particular buffer that has changed since that same buffer was last used. As it only concerns changes to a single buffer, there is no dependency on the next or previous frames or any other buffer. It therefore cannot be used to infer anything about changes to the surface, which requires linking one frame or buffer to another. Buffer damage is therefore only useful to the producer.

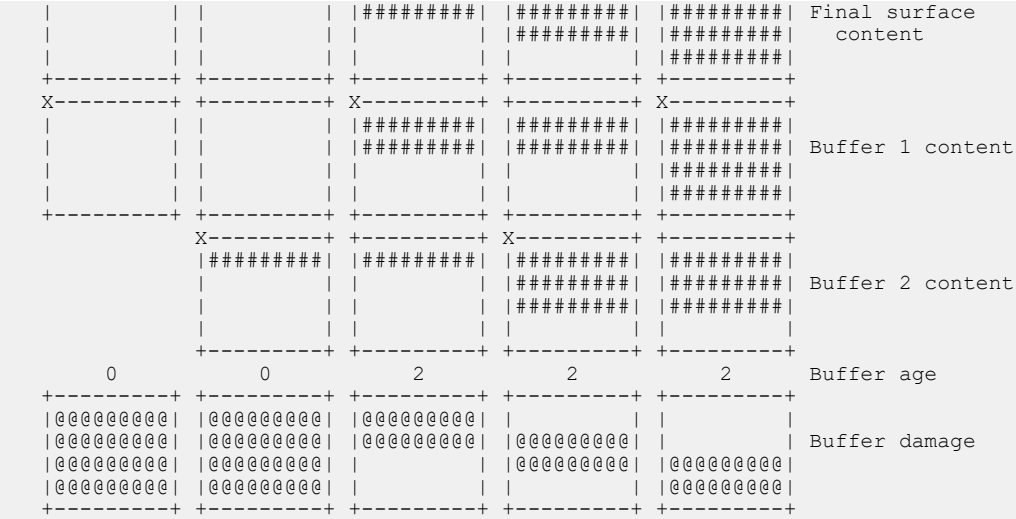
Buffer damage example

The buffer damage for a frame is the area changed since that same buffer was last used. If the buffer has not been used before, the buffer damage is the entire area of the buffer.

The buffer marked with an 'X' in the top left corner is the buffer that is being used for that frame. This is the buffer to which the buffer age and the buffer damage relate.

Note: This example shows a double buffered surface - the actual number of buffers could be different and variable throughout the lifetime of the surface. The age **must** therefore be queried for every frame.





Registry Link

https://www.khronos.org/registry/egl/extensions/KHR/EGL_KHR_partial_update.txt

Example

```
//* The surface's swap behavior is EGL_BUFFER_DESTROYED
eglSurfaceAttrib(eglDisplay, eglSurface, EGL_SWAP_BEHAVIOR, EGL_BUFFER_DESTROYED);
while(render_loop)
{
 EGLint rects[] = {100,  \\start x position
    100, \\start y position
    100, \\width
    100} \\height
 //The damage region for <surface> is set to the area described by <n_rects> and
 //<rects> if all of the following conditions are met:
 //* <surface> is the current draw surface of the calling thread
 //* <surface> is a postable surface
 //* There have been no client API commands which result with rendering to
   //<surface> since eglSwapBuffers was last called with <surface>, or since
   //<surface> was created in case eglSwapBuffers has not yet been called with
   //<surface>.
 eglSetDamageRegionKHR(eglDisplay, eglSurface, rects, 1); //before any render commands in this
 frame.
 //begin render
 //end render
 eglSwapBuffers(eglDisplay, eglSurface);
}
```

EGL_KHR_surfaceless_context

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

EGL 1.0, 1.1, 1.2, 1.3, 1.4

Description

Applications that do not want to use any sort of main framebuffer, currently still need to create an EGLSurface to make a context current. EGL_KHR_surfaceless_context

adds the ability to create a context without a surface, primarily for applications outside of OpenGL ES where interop is required. It is also useful for OpenGL ES applications that wish to perform purely off-screen rendering into framebuffer objects, but the OpenGL ES context has to support GL_OES_surfaceless_context as well for it to be compatible.

Note

This functionality is core to EGL 1.5, so the extension is no longer needed.

Registry Link

https://www.khronos.org/registry/egl/extensions/KHR/EGL_KHR_surfaceless_context.txt

Example

```
// Make a context current without any surfaces
eglMakeCurrent(display, EGL_NO_SURFACE, EGL_NO_SURFACE, context);
```

EGL_KHR_swap_buffers_with_damage

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

EGL 1.4

Description

This extension provides a means to issue a swap buffers request to display the contents of the current back buffer and also specify a list of damage rectangles that can be passed to a system compositor so it can minimize how much it has to recompose.

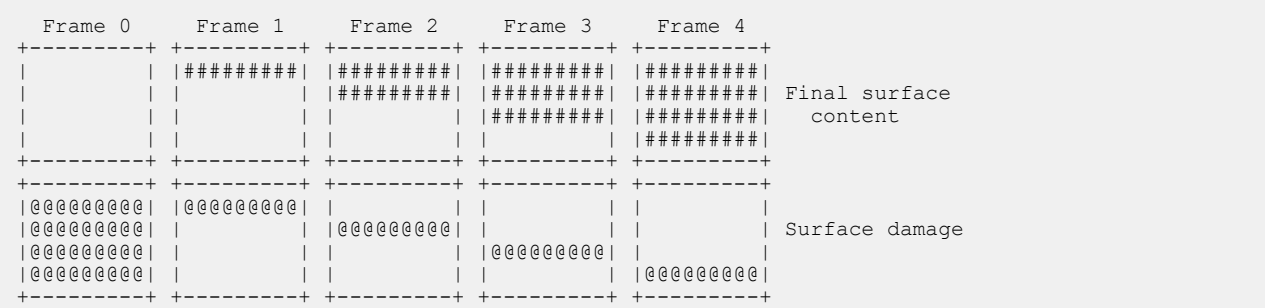
This should be used in situations where an application is only animating a small portion of a surface since it enables the compositor to avoid wasting time recomposing parts of the surface that haven't changed.

Note

Requires EGL 1.4 This extension is written against the wording of the EGL 1.4 Specification. Different with EGL_KHR_partial_update. EGL_KHR_swap_buffers_with_damage concerns the area of the surface that changes between frames for that surface. It concerns the differences between two buffers - the current back buffer and the current front buffer. It is useful only to the consumer.

Surface Damage Example

The surface damage for frame n is the difference between frame n and frame (n-1) and represents the area that a compositor must recompose.



Registry Link

https://www.khronos.org/registry/egl/extensions/KHR/EGL_KHR_swap_buffers_with_damage.txt

Example

```
//As an alternative to eglSwapBuffers
eglSwapBuffersWithDamageKHR(eglDisplay, eglSurface, rects, n_rects);
```

EGL_KHR_wait_sync

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

EGL 1.4

Description

This extension adds the ability to wait for signalling of sync objects in the server for a client API context, rather than in the application thread bound to that context. This form of wait does not necessarily block the application thread which issued the wait (unlike eglClientWaitSyncKHR), so the application may continue to issue commands to the client API context or perform other work in parallel, leading to increased performance. The best performance is likely to be achieved by implementations which can perform this new wait operation in GPU hardware, although this is not required.

Note

Requires EGL 1.4

This extension is written against the wording of the EGL 1.4 Specification.

Registry Link

https://www.khronos.org/registry/egl/extensions/KHR/EGL_KHR_wait_sync.txt

Example

```
while(!eglWaitSyncKHR(eglDisplay, eglSyncKHR, 0))
{
 //render the waiting animation.
}
```

3. OpenGL ES Extensions

GL_ANDROID_extension_pack_es31a

Supported Hardware

Series6XT

Valid APIs

OpenGL ES 3.1

Descriptions

This extension adds no functionality by itself, but rather is a guarantee of a set of other functionality being supported via other extensions - and thus don't need to be checked for individually. If this extension is supported, the following set of extensions is guaranteed to be supported:

- KHR_debug
- KHR_texture_compression_astc_ldr
- KHR_blend_equation_advanced
- OES_sample_shading
- OES_sample_variables
- OES_shader_image_atomic
- OES_shader_multisample_interpolation
- OES_texture_stencil8
- OES_texture_storage_multisample_2d_array
- EXT_copy_image
- EXT_draw_buffers_indexed
- EXT_geometry_shader
- EXT_gpu_shader5
- EXT_primitive_bounding_box
- EXT_shader_io_blocks
- EXT_tessellation_shader
- EXT_texture_border_clamp
- EXT_texture_buffer
- EXT_texture_cube_map_array
- EXT_texture_sRGB_decode

GL_APPLE_texture_2D_limited_npot

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x

Description

This extension adds limited support for non-power-of-two (POT) textures in OpenGL ES 1.x. specifically; users are now able to create 2D textures with NPOT dimensions. This relaxation does not apply to cubemaps or 3D textures. There are also several limitations:

- Texture Wrap Modes
  - Only GL_CLAMP_TO_EDGE may be used as a wrap mode when sampling from an NPOT texture.
- MIP Mapping
  - MIP Mapping is not supported.
  - Only minification filters GL_NEAREST or GL_LINEAR are allowed.
  - glGenerateMIPMap does not work.

Most of these restrictions are lifted if GL_OES_texture_npot is present.

Note

This functionality is core to OpenGL ES 2.0 and 3.0. All limitations on non-power of two textures are lifted for OpenGL ES 3.0

Example

```
// Upload a texture with dimensions of 15 by 47. Typically, this isn't supported without
// extension support.
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 15, 47, 0, GL_RGBA, GL_UNSIGNED_BYTE, pixelData);
// Only the following parameters are specifiable, anything else is invalid (including the
// default texture parameters!).
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

GL_EXT_blend_minmax

Supported Hardware

Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x, 2.0

Description

This extension adds two additional blending modes to OpenGL ES; GL_MIN_EXT and GL_MAX_EXT. These two blend modes compare the source and destination colours, and the result of the comparison will be the minimum/maximum colour that was written. Each component is operated on separately, so that it is entirely possible to have an output colour that has channel values from multiple different polygons.

Note

This functionality is core to OpenGL ES 3.0.

Registry Link

http://www.khronos.org/registry/gles/extensions/EXT/blend_minmax.txt

Example

```
// Set the blend equation to use additive blending
glBlendEquation(GL_FUNC_ADD_EXT);
// Set the blend equation to use minimum value blending
glBlendEquation(GL_MIN_EXT);
// Set the blend equation to use maximum value blending
glBlendEquation(GL_MAX_EXT);
```

GL_EXT_buffer_storage

Valid APIs

OpenGL ES 3.1+

Description

This extension applies the immutable texture storage concept to buffer objects. If an implementation is aware of a buffer's immutability, it may be able to make certain assumptions or apply particular optimizations in order to increase performance or reliability.

Furthermore, this extension allows applications to pass additional information about a requested allocation to the implementation which it may use to select memory heaps, caching behaviour or allocation strategies.

Registry Link

https://www.khronos.org/registry/OpenGL/extensions/EXT/EXT_buffer_storage.txt

Example

```
//generate texture buffer object
glGenBuffers(1, &tbo);
//bind texture buffer object
glBindBuffer(GL_TEXTURE_BUFFER, tbo);
```

```
//allocate immutable storage to texture buffer object
glBufferStorageEXT(GL_TEXTURE_BUFFER, sizeof(data), &data, GL_MAP_WRITE_BIT_EXT |
 GL_DYNAMIC_STORAGE_BIT_EXT );
//rebind the default texture buffer object
glBindBuffer(GL_TEXTURE_BUFFER, 0);
```

GL_EXT_clear_texture

Valid APIs

OpenGL ES 3.1+

Description

This extension enables developers to initialize textures directly to a desired value.

Registry Link

https://www.khronos.org/registry/OpenGL/extensions/EXT/EXT_clear_texture.txt

Example

```
float data[] = {0.0f, 0.0f, 0.0f, 1.0f};
glClearTexImageEXT(textureHandle, 0, GL_RGBA, GL_BYTE, &data);
```

GL_EXT_color_buffer_float

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 3.x

Description

Rendering to a floating-point buffer has been a core part of the desktop GL specification for a long time now but was not included in the OpenGL ES 3.0 specification. This extension adds that functionality to ES3.0 class hardware, allowing users to output floating point values from fragments, enabling things like HDR rendering and advanced post-processing techniques that need a higher range or precision than fixed point values provide.

Registry Link

http://www.khronos.org/registry/gles/extensions/EXT/EXT_color_buffer_float.txt

Example

```
// Create a floating point colour buffer (texture)
GLint floatingPointColourBuffer = glTexStorage(GL_TEXTURE_2D, 1, GL_RGBA32F, 1024, 1024);
// Attach it to a framebuffer. This would usually result in an incomplete framebuffer without
// this extension.
```

```
glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
 floatingPointColourBuffer, 0);
```

GL_EXT_conservative_depth

Valid APIs

OpenGL ES 3.0+

Description

This extension allows developers to pass information about fragment shader depth modifications to the GL implementation so that early depth testing can still be performed safely.

Registry Link

https://www.khronos.org/registry/OpenGL/extensions/EXT/EXT_conservative_depth.txt

Example

```
//depth will only be increased, so if depth test is GL_LESS early depth testing can still be
 performed
  layout (depth_greater) out float gl_FragDepth;
```

GL_EXT_copy_image

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 3.x

Description

This extension adds the ability to do a copy of data between two images, without any need to bind or involve the rendering pipeline. Functionally this is roughly equivalent to a memcpy in C/C++.

Example

```
// Copy all of renderBuffer's data into the top MIP level of texture
glCopyImageSubDataEXT(renderBuffer, GL_RENDERBUFFER, 0, 0, 0, 0,
                      texture, GL_TEXTURE_2D, 0, 0, 0, 0,
    renderBufferWidth, renderBufferHeight, 1);
// Copy data from the bottom left of texture to the top right of the same texture
glCopyImageSubDataEXT(texture, GL_TEXTURE_2D, 0, 0, 0, 0,
                      texture, GL_TEXTURE_2D, 0, textureWidth/2, textureHeight/2, 0,
                      textureWidth/2, textureHeight/2, 1);
```

GL_EXT_debug_marker

Supported Hardware

Series5, Series5XT

Valid APIs

OpenGL ES 1.x, 2.0, 3.x

Description

This extension enables developers to insert debug markers into the command stream, which are essentially a way of annotating code in GL tracing tools such as PVRTrace. There are no other benefits to this extension beyond this, as runtime markers could be introduced by an application developer themselves if they needed a way to label parts of their code.

Note

This extension's functionality has been absorbed into *GL_KHR_debug*, which adds additional functionality and should be used preferentially. As a result, this extension is now considered deprecated.

Registry Link

http://www.khronos.org/registry/gles/extensions/EXT/EXT_debug_marker.txt

Example

```
// Insert a regular event marker, used effectively as a simple string annotation to your code.
glInsertEventMarkerEXT(0, "This is a debug marker annotation");
// Signal that any following functions belong to a nested functionality group. For instance,
 this might denote the start of a function which renders a specific object.
glPushGroupMarkerEXT(0, "This is a group of functionality.");
// End the previously defined functionality group
glPopGroupMarkerEXT();
```

GL_EXT_discard_framebuffer

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT (ES2/3 only)

Valid APIs

OpenGL ES 1.x, 2.0

Description

Generally, OpenGL ES is expected to have the result of a render stored into memory so that it can be used for later computation. If the storage memory is also the memory that is being rendered to then this isn't a huge problem. However, in cases where the result of a render is stored in a temporary buffer to be written outer

later, bandwidth is wasted in copying this data out if it isn't going to be used. This extension provides a mechanism for users to explicitly state that they don't need to store part of the render output, allowing the underlying implementation to avoid the costly data copy, and potentially improving performance.

Note

This functionality is core to OpenGL ES 3.0 however a different function, "glInvalidateFramebuffer", is used instead. This function works identically to glDiscardFramebufferEXT but was renamed and re-specified to be brought in line with other OpenGL functionality.

Registry Link

http://www.khronos.org/registry/gles/extensions/EXT/EXT_discard_framebuffer.txt

Example

```
// Finish rendering to a particular framebuffer
glDraw(...);
// Specify attachments to discard, typically depth and stencil are discarded in this way.
GLenum discardAttachments[] =
{
 GL_DEPTH_ATTACHMENT,
 GL_STENCIL_ATTACHMENT
};
// Discard the framebuffer's contents which we aren't interested in
glDiscardFramebufferEXT(GL_FRAMEBUFFER, sizeof(discardAttachments)/sizeof(GLenum),
 discardAttachments);
// The next call should always be a framebuffer change, nothing should occur between this and
// the discard.
glBindFramebuffer(0);
// Clearing will prevent any data being written back to on-chip memory as well.
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
```

GL_EXT_draw_buffers

Supported Hardware

Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 2.0

Description

Typically, only one colour output is enabled by default in OpenGL ES 2.0, and whilst querying for additional colour buffers has exists, the base OpenGL ES 2.0 specification does not allow for additional colour buffers. This extension adds language and functionality that enable up to a maximum of 16 total colour buffers per framebuffer object, though the actual available number varies by platform. The main benefit of this extension is to reduce the number of draw passes in a scene, as a number of techniques need more than 4 channels of data to output for a given scene, usually for further calculation. Without additional colour buffers, developers are forced to render the same data multiple times or sacrifice precision by packing data.

This functionality is also more commonly known as "Multiple Render Targets" or MRTs.

Note

This functionality is core to OpenGL ES 3.0.

Example

```
// Attach a texture to a framebuffer, using a colour attachment beyond 0.
glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT1, GL_TEXTURE_2D,
 aTextureColourBuffer, 0);
// The writable draw buffers also need to be set separately. In this instance there are two
// colour buffers. Create a list and allow them to be rendered to.
GLenum buffers[2] = {GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1};
glDrawBuffers(2, buffers);
```

GL_EXT_draw_buffers_indexed

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 3.x

This extension is also compatible with OpenGL ES 2.0 drivers that support EXT_draw_buffers.

Description

This extension builds upon the EXT_draw_buffers extension. In EXT_draw_buffers (part of OpenGL ES 3.0), separate values could be written to each colour buffer, but the blend enable, blend functions, blend equations and colour write masks are global and apply to all colour outputs.

This extension provides the ability to independently:

- enable or disable blending,
- set the blend equations,
- set the blend functions, and
- set the colour write masks per colour output.

This extension introduces indexed versions of the enable, blend equation, blend function, and colour mask commands, as well as associated indexed queries in order to control and query these states independently on a per-colour output basis.

Registry Link

https://www.khronos.org/registry/gles/extensions/EXT/EXT_draw_buffers_indexed.txt

Example

```
EnableiEXT(BLEND, 0);
DisableiEXT(BLEND, 0);
IsEnablediEXT(BLEND, 0);
BlendEquationiEXT(0, FUNC_ADD);
BlendEquationSeparateiEXT(0, FUNC_ADD, FUNC_SUBTRACT);
BlendFunciEXT(0, ZERO, ONE);
BlendFuncSeparateiEXT(0, SRC_COLOR, DST_COLOR, CONSTANT_ALPHA, SRC_ALPHA_SATURATE);
ColorMaskiEXT(0, EGL_TRUE, EGL_TRUE, EGL_TRUE, EGL_TRUE);
```

GL_EXT_draw_elements_base_vertex

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

Requires OpenGL ES 2.0

Description

This extension provides a method to specify a "base vertex offset" value which is effectively added to every vertex index that is transferred through DrawElements. This mechanism can be used to decouple a set of indices from the actual vertex array that it is referencing. This is useful if an application stores multiple indexed models in a single vertex array. The same index array can be used to draw the model no matter where it ends up in a larger vertex array simply by changing the base vertex value. Without this functionality, it would be necessary to rebind all the vertex attributes every time geometry is switched which can have a larger performance penalty.

Registry Link

https://www.khronos.org/registry/gles/extensions/EXT/EXT_draw_elements_base_vertex.txt

Example

```
glDrawElementsBaseVertexEXT(TRIANGLES, 6, UNSIGNED_BYTE, &indices, 100)
//[[ If OpenGL ES 3.0 is supported: ]]
glDrawRangeElementsBaseVertexEXT(TRIANGLES, 0, 200,
                                 6, UNSIGNED_BYTE,
                                 &indices, 100);
glDrawElementsInstancedBaseVertexEXT(TRIANGLES, 6,
                                     UNSIGNED_BYTE, &indices,
                                     10, 100);
//[[ If EXT_multi_draw_arrays is supported: ]]
glMultiDrawElementsBaseVertexEXT(TRIANGLES,
                                 &count,
                                 UNSIGNED_BYTE,
                                 &indices,
                                 10,
                                 &basevertex);
```

GL_EXT_float_blend

Valid APIs

OpenGL ES 3.0+

Description

This extension allows blending with 32-bit floating-point colour buffers.

Registry Link

https://www.khronos.org/registry/OpenGL/extensions/EXT/EXT_float_blend.txt

GL_EXT_geometry_point_size

Supported Hardware

Series6XT

Valid APIs

OpenGL ES 3.1

Description

This extension adds programmable point size to the geometry shader and allows resizing of generated point sprites that come through as input. This is useful for things like particle effects.

Example

```
#extension GL_EXT_geometry_shader : require
#extension GL_EXT_geometry_point_size : require
#extension GL_EXT_shader_io_blocks : require
// Take in point primitives.
layout(points) in;
// Output up to 1 line primitive - needs at least two vertices (one line).
layout(points, max_vertices = 1) out;
void main()
{
 // Re-emit the point primitive - at twice the size
 gl_Position = gl_in[0].gl_Position + vec4(0.0,-0.1, 0.0, 0.0);
 gl_PointSize = gl_in[0].gl_PointSize * 2.0f;
 EmitVertex();
 // End the (point) primitive
 EndPrimitive();
}
```

GL_EXT_geometry_shader

Supported Hardware

Series6XT

Valid APIs

OpenGL ES 3.1

Description

Geometry Shaders are a new programmable pipeline step that sits in the current OpenGL ES pipeline directly after primitive assembly, and before clipping, culling etc. This stage allows access to the vertices in the primitive constructed by the earlier phases, in order to interpret them and re-emit new geometry. For example, the vertex shader and primitive assembly could output single point primitives, and the geometry shader could convert this into a set of triangles for further processing. Geometry shaders can be used for a form of tessellation, but this is considered superceded by EXT_tessellation_shader. Geometry shaders can also discard primitives.

Adjacency Primitives

As well as the standard GL primitives, this extension adds four new primitive types: LINES_ADJACENCY, LINE_STRIP_ADJACENCY, TRIANGLES_ADJACENCY and TRIANGLE_STRIP_ADJACENCY. These new types function the same as their non-adjacency counterparts everywhere except the geometry shader. In the geometry shader, these modes allow access to neighbouring vertices for each vertex. These neighbouring vertices can be useful as control points to bound any transformations done on input geometry.

Multiple Invocations

As well as allowing multiple outputs, geometry shaders can actually run over the same input geometry multiple times as a form of instancing - allowing the same inputs to generate multiple outputs with different properties. By default, only one invocation is executed, but multiple can be processed with the layout qualifier - "invocations = integer-constant".

Layered Rendering

For use cases such as stereoscopic (or other multi-view) rendering, it is often desirable to do some vertex shading only once, rather than repeating it over and over for each view. Geometry shader functionality described so far can allow this by using multiple invocations to transform geometry from different angles, but each view may need to be output to a separate render target. To handle this use case, this extension adds functionality called layered rendering, which allows framebuffers to be created with multiple layers. The geometry shader can then choose which layer its outputs will be sent to via the special gl_Layer output value.

Inputs

Geometry shader inputs come in the form of Shader IO blocks - including the built-in inputs. Built-in inputs match the built-in vertex shader outputs, but are passed through a built-in io block: gl_in. This is to distinguish them from output values.

Note: This block does not include point size unless EXT_geometry_point_size is supported and enabled.

Each io block is actually an array of blocks, and each is either implicitly (or explicitly) sized to the number of vertices available for each primitive type (e.g. 1 for a point, 2 for a line, 3 for a triangle, etc.).

As well as the built-in block for vertex shader outputs, two additional values are provided:

- gl_PrimitiveIDIn, which is a counter that describes how many primitives have been processed by the shader during this render.
- gl_InvocationID, which indicates how many times this shader has been invoked using Multiple Invocations.

Outputs

Geometry shader outputs are the same as inputs, except for the built-in values, which are not part of an io block - instead they are free variables in the same way as they are in a vertex shader. gl_PrimitiveID is also provided as a modifiable output value that is passed on towards the fragment shader.

Example

```
// Create a Geometry Shader
GLuint geometryShader = glCreateShader(GL_GEOMETRY_SHADER_EXT);
```

Example

```
#extension GL_EXT_geometry_shader : require
#extension GL_EXT_shader_io_blocks : require
// Take in point primitives.
layout(points) in;
// Output up to 1 line primitive - needs at least two vertices (one line).
layout(lines, max_vertices = 2) out;
void main()
{
 // Emit the first vertex for the output primitive - below the original point
 gl_Position = gl_in[0].gl_Position + vec4(0.0,-0.1, 0.0, 0.0);
 EmitVertex();
 // Emit the second vertex for the output primitive - above the original point
 gl_Position = gl_in[0].gl_Position + vec4( 0.0, 0.1, 0.0, 0.0);
 EmitVertex();
 // End the (line) primitive
 EndPrimitive();
}
```

GL_EXT_gpu_shader5

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 3.1

Description

This extension adds a number of useful, but miscellaneous features to the OpenGL ES shading language. These features are as follows:

- support for indexing into arrays of opaque types (samplers, and atomic counters) using dynamically uniform integer expressions;
- support for indexing into arrays of images and shader storage blocks using only constant integral expressions;
- extending the uniform block capability to allow shaders to index into an array of uniform blocks;
- a "precise" qualifier allowing computations to be carried out exactly as specified in the shader source to avoid optimization-induced invariance issues (which might cause cracking in tessellation);
- new built-in functions supporting:
  - fused floating-point multiply-add operations;
- extending the textureGather() built-in functions provided by OpenGL ES Shading Language 3.10:
  - allowing shaders to use arbitrary offsets computed at run-time to select a 2x2 footprint to gather from; and
  - allowing shaders to use separate independent offsets for each of the four texels returned, instead of requiring a fixed 2x2 footprint.

Example

```
#extension GL_EXT_gpu_shader5 : require
uniform lowp sampler2DArray arrayOfTextureArrays[8];
uniform lowp sampler2D gatherableTexture;
in highp vec2 textureCoords;
in highp float a, b, c;
void main()
{
 ...
 // Loop through and reference each member of an array of texture arrays.
 highp vec4 value = vec4(0.0);
 for (uint i = 0; i < 8; ++i)
 {
  for (uint j = 0; j < 8; ++j)
  {
   value += texture(arrayOfTextureArrays[i], vec3(textureCoords, j));
  }
 }
 ...
 // Perform an explicit fused multiply-add
 highp float result;
 result = fma(a,b,c);
 ...
 // Ensure that no optimisations are done on any calculations that affect the result of a value
 (result2);
 precise highp float result2;
 result2 = a * b + c;
 ...
 // Gather four texels from a texture that aren't necessarily neighbours, obtaining the red
 channel.
 ivec2 offsets[4] = {ivec2(0,0), ivec2(5, 0), ivec2(5, 5), ivec2(0, 5)};
 vec4 gatheredTexels = textureGatherOffsets(gatherableTexture, textureCoords, offsets, 0);
 ...
}
```

GL_EXT_multi_draw_arrays

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x, 2.0, 3.x

Description

This extension adds two new functions to OpenGL ES; glMultiDrawArrays and glMultiDrawElements. These functions perform the same tasks as glDrawArrays/Elements, but also allow the users to render multiple primitive groups in one function call. In other words, one "glMultiDrawArrays" is equivalent to multiple "glDrawArrays" with the same rendering state.

Registry Link

http://www.khronos.org/registry/gles/extensions/EXT/multi_draw_arrays.txt

Example

```
// Setup lists of buffer offsets and triangle counts
GLint firsts[] = {0, 32};
GLsizei counts[] = {32, 32};
// If multiple vertex objects share the same draw state (blending, depth testing, shaders,
// textures, etc.) then multi-draw can be called on them all at once, rather than calling
// glDraw multiple times.
glMultiDrawArrays(GL_TRIANGLES, firsts, counts, 2);
```

GL_EXT_multisampled_render_to_texture

Supported Hardware

Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x, 2.0, 3.x

Description

This extension adds multisampled rendering capabilities to OpenGLES, allowing a user to create multisampled render buffers, and attach textures to a framebuffer in a way that they can be used for multisampled renders. By enabling textures to be attached as multisampled but without using actual multisample textures, a significant amount of bandwidth can be saved. This mode of operation enables multisampled anti-aliasing to smooth lines at the edges of polygons and reduce the appearance of jagged lines. This is functionally equivalent to GL_IMG_multisampled_render_to_texture on PowerVR hardware.

Note

Part of this functionality is core to OpenGL ES 3.0. The renderbuffer functionality is present in core, however, the multisampled texture attachment is not.

Registry Link

http://www.khronos.org/registry/gles/extensions/EXT/EXT_multisampled_render_to_texture.txt

Example

```
// Create a multisampled depth renderbuffer
GLint depthBuffer = glRenderbufferStorageMultisampleEXT(GL_RENDERBUFFER, 4,
 GL_DEPTH_COMPONENT16, 1024, 1024);
// Attach the renderbuffer to the framebuffer as per usual. Note: If one attachment is
// multi-sampled, all attachments need to be. If there is a mix, the framebuffer will be
// incomplete.
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, depthBuffer);
// Create a regular colour buffer (texture). This extension doesn't introduce multi-sample
// textures, instead attaching regular textures that multi-sampling will resolve to before
// writing out.
GLint colourBuffer = glTexStorage(GL_TEXTURE_2D, 1, GL_RGBA8, 1024, 1024);
// Attach it to a framebuffer as multisampled, so that OpenGL ES knows that it should resolve
// a multisampled render to this texture.
glFramebufferTexture2DMultisampleEXT(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
 colourBuffer, 0);
```

GL_EXT_occlusion_query_boolean

Supported Hardware

Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 2.0

Description

This extension adds two new mechanisms to OpenGL ES 2.0 - Querying and Boolean Occlusion Queries.

Querying is a mechanism which allows the user to request information from the hardware directly, often allowing access to information that the hardware can gather with easily that would otherwise require expensive software checks. As modern GPUs work asynchronously to the CPU, querying does not instantly return a value; instead, a result is requested, and then at some point in the future the GL implementation will have a response ready. App developers can check when that query is ready, and then access that value.

Boolean occlusion queries use the querying mechanism to allow app developers to find out if an object they've been sending down the pipeline is actually visible or not. The result of the query can then be used in subsequent frames to determine whether or not to draw the object again. This allows a developer to better balance the load between software culling and hardware visibility tests.

Note

This functionality is core to OpenGL ES 3.0.

Registry Link

http://www.khronos.org/registry/gles/extensions/EXT/EXT_occlusion_query_boolean.txt

GL_EXT_polygon_offset_clamp

Valid APIs

OpenGL ES 3.1+

Description

This extension adds a new parameter to the polygon offset function that clamps the calculated offset to a minimum or maximum value. The clamping functionality is useful when polygons are nearly parallel to the view direction because their high slopes can result in arbitrarily large polygon offsets. In the particular case of shadow mapping, the lack of clamping can produce the appearance of unwanted holes when the shadow casting polygons are offset beyond the shadow receiving polygons, and this problem can be alleviated by enforcing a maximum offset value.

Registry Link

https://www.khronos.org/registry/OpenGL/extensions/EXT/EXT_polygon_offset_clamp.txt

Example

```
//set max offset
glPolygonOffsetClampEXT(1.0f, 1,0f, 1.0f);
```

GL_EXT_primitive_bounding_box

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 2.0

Description

On tile-based architectures, transformed primitives are generally written out to memory before rasterization, and then read back from memory for each tile they intersect. When geometry expands during transformation due to tessellation or one-to-many geometry shaders, the external bandwidth required grows proportionally. This extension provides a way for implementations to know which tiles incoming

geometry will intersect before fully transforming (and expanding) the geometry. This allows them to only store the unexpanded geometry in memory and perform expansion on-chip for each intersected tile.

New state is added to hold an axis-aligned bounding box which is assumed to contain any geometry submitted. An implementation is allowed to ignore any portions of primitives outside the bounding box, thus, if a primitive extends outside of a tile into a neighbouring tile that the bounding box didn't intersect, the implementation is not required to render those portions. The tessellation control shader is optionally able to specify a per-patch bounding box that overrides the bounding box state for primitives generated from that patch, in order to provide tighter bounds.

The typical usage is that an application will have an object-space bounding volume for a primitive or group of primitives, either calculated at runtime or provided with the mesh by the authoring tool or content pipeline. It will transform this volume to clip space, compute an axis-aligned bounding box that contains the transformed bounding volume, and provide that at either per-patch or per-draw granularity.

Note

OpenGL ES 3.1 and OpenGL ES Shading Language 3.10 are required.

This specification is written against the OpenGL ES 3.1 (March 17, 2014) and OpenGL ES 3.10 Shading Language (March 17, 2014) Specifications.

This extension interacts with EXT_tessellation_shader. EXT_geometry_shader trivially affects the definition of this extension.

Registry Link

https://www.khronos.org/registry/gles/extensions/EXT/EXT_primitive_bounding_box.txt

Example

```
//clip space
glPrimitiveBoundingBoxEXT(0.2f, 0.2f, 0.2f, 1.0f, //min value
      0.3f, 0.3f, 0.3f, 1.0f); //max value
//in tessellation control shader
gl_BoundingBoxEXT[0]= vec4(0.2f, 0.2f, 0.2f, 1.0f);//min value
gl_BoundingBoxEXT[1]= vec4(0.3f, 0.3f, 0.3f, 1.0f);//max value
```

GL_EXT_pvrtc_sRGB

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 2.0, OpenGL ES 3.x

Description

This extension allows PVRTC texture data (see IMG_texture_compression_pvrtc and IMG_texture_compression_pvrtc2)to be interpreted as sRGB values when sampled by a shader, and automatically convert those values to linear space.

Example

```
// Create a PVRTC1 4bpp sRGB without alpha texture
glTexStorage2D(GL_TEXTURE_2D, textureLevels, GL_COMPRESSED_SRGB_PVRTC_4BPPV1_EXT, textureWidth,
 textureHeight);
// Create a PVRTC2 2bpp sRGB texture
glTexStorage2D(GL_TEXTURE_2D, textureLevels, GL_COMPRESSED_SRGB_ALPHA_PVRTC_2BPPV2_IMG,
 textureWidth, textureHeight);
```

GL_EXT_read_format_bgra

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.1, OpenGL ES 2.0, OpenGL ES 3.x

Description

This extension adds additional formats that can be used when calling glReadPixels. Specifically it allows the format "BGRA_EXT" to be paired with UNSIGNED_BYTE, UNSIGNED_SHORT_4_4_4_4_REV_EXT or UNSIGNED_SHORT_5_5_5_1_REV_EXT, giving alternative byte/bit-ordering schemes for the returned pixels.

Example

```
// Read pixel data back as BGRA8888
GLubyte returnedData[width * height * 4];
glTexStorage2D(0, 0, width, height, GL_BGRA_EXT, GL_UNSIGNED_BYTE, (GLvoid*)returnedData);
```

GL_EXT_robustness

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.1, 2.0, 3.x

Description

Several recent trends in how OpenGL integrates into modern computer systems have created new requirements for robustness and security for OpenGL rendering contexts. This extension introduces a number of security features to OpenGL ES

which increase the safety of executing a program running untrusted content, such as in WebGL.

This extension adds two new features:

- User Notifications: it adds user notifications for when hardware resets occur and gives indications over whether a particular context caused it by indicating if it was guilty or not. This is achieved via calls to glGetGraphicsResetStatus, which returns if any hardware resets have occurred since the last call to the function. The reset status will either be GL_NO_ERROR if no reset occurred, GL_GUILTY_CONTEXT_RESET_EXT if the current context caused a reset, GL_INNOCENT_CONTEXT_RESET_EXT if a reset was caused by another context, and GL_UNKNOWN_CONTEXT_RESET_EXT if a reset was caused with no obvious reason. This feature is only active if a context has been created with hardware reset notifications enabled, as described in EGL_EXT_create_context_robustness.
- Robust API Calls: The other feature it adds is a number of safer entry points for the handful of functions which could potentially cause unintended buffer overruns. The new functions are glReadnPixelsEXT, glGetnUniformfvEXT and glGetnUniformivEXT, superseding glReadPixels, glGetUniformfv and glGetUniformiv respectively. These new functions add a bufSize parameter which allows the user to specify exactly how much storage space has been supplied for the result.

Registry Link

http://www.khronos.org/registry/gles/extensions/EXT/EXT_robustness.txt

Example

```
// Use the robust read pixels function to return a known amount of data
#define BUFFER_SIZE 72
struct
{
 GLubyte red;
 GLubyte green;
 GLubyte blue;
 GLubyte alpha;
} colourOutputBuffer[BUFFER_SIZE];
// Ask for 73 RGBA pixels, but only provide storage for 72 pixels. The last pixel will be
 discarded rather than overrunning the buffer.
glReadnPixelsEXT(0, 0, 73, 1, GL_RGBA, GL_UNSIGNED_BYTE, sizeof(colourOutputBuffer),
 (GLvoid*)colourOutputBuffer);
```

GL_EXT_separate_shader_objects

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 2.0, 3.0

Description

Conventional GLSL requires multiple shader stages (vertex and fragment) to be linked into a single monolithic program object to specify a GLSL shader for each stage. While this approach allows early optimisation opportunities, the hardware allows more flexibility, allowing developers to mix and match shaders at draw time instead.

This extension introduces program pipeline objects which act as containers for multiple program objects, each bound to different shader stages, to make up the equivalent of a monolithic program for fast switching, or users can bind individual shaders at run time. It also introduces a new way to update uniforms, without requiring a bind, reducing API overhead and reducing developer burden.

As well as API changes, this extension adds layout qualifiers to OpenGL ES Shading Language 1.0, so that shaders can have their inputs and outputs match up correctly even if their names do not.

Registry Link

http://www.khronos.org/registry/gles/extensions/EXT/EXT_separate_shader_objects.txt

Example

```
// Create a vertex program
GLuint vertexShaderProgram = glCreateShaderProgramvEXT(GL_VERTEX_SHADER, 1,
 &vertexShaderSource);
// Create a fragment program
GLuint fragmentShaderProgram = glCreateShaderProgramvEXT(GL_FRAGMENT_SHADER, 1,
 &fragmentShaderSource);
// Create and bind a program pipeline object
GLuint programPipeline = 0;
glGenProgramPipelinesEXT(1, &programPipeline);
glBindProgramPipelineEXT(programPipeline);
// Bind the shader programs to the program pipeline object
glUseProgramStagesEXT(programPipeline, GL_VERTEX_SHADER_BIT_EXT, vertexShaderProgram);
glUseProgramStagesEXT(programPipeline, GL_FRAGMENT_SHADER_BIT_EXT, fragmentShaderProgram);
// Set a uniform value to a program that isn't necessarily bound
glProgramUniform1fEXT(fragmentShaderProgram, 0, 1.0f);
```

Example (GLSL)

```
// Specify an attribute with a guaranteed location of zero
layout(location = 0) attribute highp vec3 normal;
// Specify a varying with a guaranteed location of 4
layout(location = 4) varying highp vec4 colour;
```

GL_EXT_shader_framebuffer_fetch

Supported Hardware

Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 2.0, 3.x

Description

This extension allows an application to access the previously written fragment colour from a fragment instance. The main use for this is to implement a custom blending technique, allowing developers to perform techniques beyond the standard fixed function blending modes available.

As well as being useful for programmable blending, it can be used in conjunction with multiple render targets to enable fully on-chip deferred shading. Typical deferred shading techniques involve computing a number of values and storing them out to multiple render targets. A second pass then reads these render targets back and computes a final colour.

By using shader framebuffer fetch on PowerVR hardware, values can be written to these fragment locations in one shader, then use these values directly in a full screen resolve pass - only writing out the final colour to main memory.

The steps to achieve this are as follows:

- Clear the framebuffer at the start of the frame
- Draw all your geometry as in a traditional deferred shading technique
- Draw a full screen quad to the same framebuffer
  - The shader used should read values by using this extension, rather than from textures
  - The shader otherwise does the same as it would in a normal deferred shading resolve.
- Discard all framebuffer attachments, other than the final colour, at the end of the frame

Registry Link

https://www.khronos.org/registry/gles/extensions/EXT/EXT_shader_framebuffer_fetch.txt

Example

```
void main()
{
 // This is a placeholder for doing the normal calculations in your shader.
 shaderColour = calculateColourResult();
 /*
 Blend by sampling the previous fragment colour value. This example is
 effectively equivalent to using glBlendFunc(GL_SRC_ALPHA,
 GL_ONE_MINUS_SRC_ALPHA) and glBlendEquation(GL_FUNC_ADD).
 */
 gl_FragColor = mix(gl_LastFragData[0], shaderColourResult, shaderColour.a);
}
```

Example (version 300 es)

```
// Specify a fragment output as an input also - note "inout" instead of "out"
layout(location = 0) inout lowp vec4 colourAttachmentZero;
void main()
{
 // This is a placeholder for doing the normal calculations in your shader.
 shaderColour = calculateColourResult();
 /*
```

```
 Blend by sampling the previous fragment colour value. This example is
 effectively equivalent to using glBlendFunc(GL_SRC_ALPHA,
 GL_ONE_MINUS_SRC_ALPHA) and glBlendEquation(GL_FUNC_ADD).
 */
 colourAttachmentZero = mix(colourAttachmentZero, shaderColour, shaderColour.a);
}
```

GL_EXT_shader_group_vote

Valid APIs

OpenGL ES 3.0+

Description

Implementations in some cases may group shader invocations and execute them in an SIMD fashion. When executing divergent conditional code paths, a SIMD implementation might stall one set of thread groups. This extension provides the ability to avoid divergent execution by evaluting a condition across an entire SIMD invocation group using code like:

```
if (allInvocationsEXT(condition)) {
                result = do_fast_path();
                } else {
                result = do_general_path();
                }
```

The built-in function `allInvocationsEXT()` will return the same value for all invocations in the group, so the group will either execute `do_fast_path()` or `do_general_path()`, but never both.

The built-in output values from the vertex shader stage are redefined when this extension exists, to the following:

Registry Link

https://www.khronos.org/registry/OpenGL/extensions/EXT/EXT_shader_group_vote.txt

Example GLSL

```
                //all SIMD threads execute one path, never both
                if (allInvocationsEXT(condition)) {
                result = do_fast_path();
                } else {
                result = do_general_path();
                }
```

GL_EXT_shader_io_blocks

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 3.1

Description

Typically, values passed between shader stages, such as between vertex and fragment, are passed as free, independent variables. Interface blocks add the ability to group these values together into a single packet of data, allowing structured data to flow between these stages.

The built-in output values from the vertex shader stage are redefined when this extension exists, to the following:

```
out gl_PerVertex {
    highp vec4  gl_Position;
    highp float gl_PointSize;
};
```

This has no real effect on the vertex shader but is an important distinction for other shader stages added by extensions, such as geometry and tessellation shaders.

Example

```
// Vertex Shader
#extension GL_EXT_shader_io_blocks : require
layout(location = 0) out SurfaceData
{
 highp vec3 normal;
 highp vec3 tangent;
} surfaceData;
// Fragment Shader
#extension GL_EXT_shader_io_blocks : require
layout(location = 0) in SurfaceData
{
 highp vec3 normal;
 highp vec3 tangent;
} surfaceData;
```

GL_EXT_shader_non_constant_global_initializers

Valid APIs

OpenGL ES 3.2

Description

This extension adds the ability to use non-constant initializers for global variables in the OpenGL ES Shading Language specifications. This functionality is already present in the OpenGL Shading language specification.

Registry Link

https://www.khronos.org/registry/OpenGL/extensions/EXT/EXT_shader_non_constant_global_initializers.txt

Example

```
uniform int i;
```

```
int a = i;
void foo();
void main() { foo(); }
```

GL_EXT_shader_pixel_local_storage

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 3.x

Description

Pixel local storage takes advantage of the fact that tile-based renderers use a small, on-chip cache to render images to before writing them to main memory. These registers are actually entirely independent from the final framebuffer, and there's no real need to reference them in any particular format or have any backing store behind them. This extension exposes these registers to the user and guarantees that they remain coherent between fragment shaders in a single render pass, as long as no operation causing a flush is performed. This allows users to store data in early fragment passes and then access these values in later fragment passes at the same pixel location. The principle behind it is thus similar to that of GL_EXT_framebuffer_fetch, but in a much more flexible way. A number of packing storage qualifiers are also added, to specify how data is written to these registers.

There are two primary expected use cases for this:

- Deferred Shading
  - Typically, deferred shading uses external textures to store GBuffer values. This consumes a significant amount of bandwidth due to the number of write and read operations performed. Instead, pixel local storage allows applications to store the GBuffer wholly on-chip and avoid this enormous bandwidth cost, only storing out the final evaluated colour.
- Order Independent Transparency
  - Pixel local storage can also be used to store data representing the colour of multiple pixels, stored in depth order rather than render order, allowing a bandwidth-free linked-list approach for generating order independent blending.

Example

```
#extension GL_EXT_shader_pixel_local_storage : require
__pixel_local_inEXT GBufferLocalStorage
{
 layout(rgba8)          lowp  vec4  albedo;
 layout(r11f_g11f_b10f) highp vec3  normal;
 layout(r32f)           highp float depth;
 layout(rgba8)          lowp  vec4  accumulatedColour;
};
uniform highp vec3  uLightColourIntensity;
in highp vec3 vLightDirection;
// Simple directional lighting evaluation
void main()
{
 // Calculate simple diffuse lighting for the current pixel.
```

```
 highp float n_dot_l = max(dot(-vLightDirection, normal), 0.0);
 lowp vec4 lightContribution = vec4(albedo.rgb * n_dot_l * uLightColourIntensity, 1.0);
 accumulatedColour += lightContribution;
}
```

GL_EXT_shader_pixel_local_storage2

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 3.x

Description

This extension builds on EXT_shader_pixel_local_storage by lifting the restriction that pixel local storage is not supported when rendering to multiple draw buffers.

Moreover, pixel local storage values are no longer lost when writing to user-defined fragment outputs, and, correspondingly, framebuffer pixel values do not always become undefined when the shader writes to pixel local storage.

This extension adds the following capabilities:

- support for pixel local storage in combination with multiple user-defined fragment outputs.
- support for clearing pixel local storage variables.
- support for multi-word pixel local storage variables.

Registry Link

https://www.khronos.org/registry/gles/extensions/EXT/EXT_shader_pixel_local_storage2.txt

Example

```
glClearPixelLocalStorageuiEXT( 0, //offset
       2, //number
       NULL); //set to 0
glFramebufferPixelLocalStorageSizeEXT(0, 10);
uint pixel_local_storage_size = glGetFramebufferPixelLocalStorageSizeEXT(0);
```

GL_EXT_shader_texture_lod

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 2.0

Description

This extension allows the user to manually select a level of detail from the fragment shader, rather than leaving it up to the implementation. Vanilla OpenGL ES 2.0 allows the user to add a level of detail bias to the MIP Map selection, but only allows explicit selection in the vertex shader. By exposing this in the fragment shader too, it means that custom level of details can be achieved - allowing for example a simple blurring effect that can be used to simulate focussing on different parts of the scene.

Note

This functionality is core to OpenGL ES 3.0.

Registry Link

http://www.khronos.org/registry/gles/extensions/EXT/EXT_shader_texture_lod.txt

Example

```
// Samples a texture explicitly from the 5th MIP Map level
lowp vec4 colour = texture2DLodEXT(aSampler, texCoords.xy, 4.0);
```

GL_EXT_shadow_samplers

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 2.0

Description

This extension adds shadow samplers to OpenGL ES 2, which are a type of sampler that has predicated comparisons built-in when texels are sampled. The typical use case for this is to sample a shadow map and determine whether the current pixel is in shadow or not.

Note: This functionality is core to OpenGL ES 3.0, so the extension is no longer needed.

Example

```
// Set the texture to perform comparison
glBindTexture(GL_TEXTURE_2D, texture)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE_EXT, GL_COMPARE_REF_TO_TEXTURE_EXT);
// Set the texture's comparison predicate function to "Greater Than"
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC_EXT, GL_GREATER);
[Example]
#extension GL_EXT_shadow_samplers : require
uniform sampler2Dshadow myShadowMap;
varying vec2 textureCoords;
void main()
{
 ...
 // Sample the shadow texture.
```

```
 // The value returned will be either 1.0 or 0.0 with nearest filtering, or a value in that
 range for linear filtering.
 float shadowAmount = shadow2DEXT(myShadowMap, vec3(textureCoords, referenceValue));
 ...
}
```

GL_EXT_sparse_texture

Valid APIs

OpenGL ES 3.1+

Description

Recent advances in application complexity and a desire for higher resolutions have pushed texture sizes up considerably. Often, the amount of physical memory available to a graphics processor is a limiting factor in the performance of texture-heavy applications. Once the available physical memory is exhausted, paging may occur bringing performance down considerably - or worse, the application may fail. Nevertheless, the amount of address space available to the graphics processor has increased to the point where many gigabytes - or even terabytes of address space may be usable even though that amount of physical memory is not present.

This extension allows the separation of the graphics processor's address space (reservation) from the requirement that all textures must be physically backed (commitment). This exposes a limited form of virtualisation for textures. Use cases include sparse (or partially resident) textures, texture paging, on-demand and delayed loading of texture assets and application-controlled level of detail.

Registry Link

https://www.khronos.org/registry/OpenGL/extensions/EXT/EXT_sparse_texture.txt

Example

```
//generate texture with sparse storage
glGenTextures(1, &tex);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D_ARRAY, tex);
glTexParameteri(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST);
glTexParameteri(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D_ARRAY, GL_TEXTURE_SPARSE_EXT, GL_TRUE);
glTexStorage3D(GL_TEXTURE_2D_ARRAY, levels, GL_RGBA8, size, size, 1);
ivec3 pageSize;
glGetInternalformativ(GL_TEXTURE_2D_ARRAY, GL_RGBA8, GL_VIRTUAL_PAGE_SIZE_X_EXT, 1,
 &pageSize.x);
glGetInternalformativ(GL_TEXTURE_2D_ARRAY, GL_RGBA8, GL_VIRTUAL_PAGE_SIZE_Y_EXT, 1,
 &pageSize.y);
glGetInternalformativ(GL_TEXTURE_2D_ARRAY, GL_RGBA8, GL_VIRTUAL_PAGE_SIZE_Z_EXT, 1,
 &pageSize.z);
//set page to commit
glTexPageCommitmentEXT(GL_TEXTURE_2D_ARRAY, level,
     pageSize.x * i, pageSize.y * j, 0, // xoffset, yoffset, zoffset
     pageSize.x, pageSize.y, 1, // width, height, depth
     GL_TRUE); //do commit
glTexSubImage3D(GL_TEXTURE_2D_ARRAY, level,
    pageSize.x * i, pageSize.y * j, 0, // xoffset, yoffset, zoffset
    pageSize.x, pageSize.y, 1, // width, height, depth
    GL_RGBA, GL_UNSIGNED_BYTE, // format, type
    &page[0][0]); //ptr
```

GL_EXT_tessellation_point_size

Supported Hardware

Series6XT

Valid APIs

OpenGL ES 3.1

Description

This extension adds programmable point size to the tessellation shaders and allows resizing of generated point sprites that come through as input - useful for things like particle effects.

Example

```
// Tessellation control shader
#extension GL_EXT_tessellation_shader : require
#extension GL_EXT_tessellation_point_size : require
#extension GL_EXT_shader_io_blocks : require
layout(vertices = 1) out;
void main(void)
{
 gl_TessLevelOuter[0] = 2.0;
 gl_TessLevelOuter[1] = 4.0;
 gl_TessLevelOuter[2] = 6.0;
 gl_TessLevelOuter[3] = 8.0;
 gl_TessLevelInner[0] = 8.0;
 gl_TessLevelInner[1] = 8.0;
 gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
 gl_out[gl_InvocationID].gl_PointSize = gl_in[gl_InvocationID].gl_PointSize * 1.1;
}
```

GL_EXT_tessellation_shader

Supported Hardware

Series6XT

Valid APIs

OpenGL ES 3.1

Description

This extension adds new tessellation stages to the OpenGL ES pipeline, and two new shader types - tessellation control and tessellation evaluation. When tessellation is active, a number of pipeline stages are affected. Tessellation operates after primitive assembly and before geometry shading (if supported).

Patch Primitives

Tessellation operates on a new primitive type - patches, which are a collection of vertices that are combined to form a patch. The actual number of vertices is application defined, allowing it to be fairly general-purpose.

Tessellation Control

The first step of tessellation is the control shader. This shader exists to allow further transformation of vertices and set the level of detail required by the fixed function tessellation stage. It may also add or subtract vertices from a patch before passing it on - though it cannot change the number of patches.

Fixed Function Tessellation

The fixed function stage turns patches into a number of primitives, which are then available to the tessellation evaluation shader. The algorithm is fairly flexible and has a number of determining factors depending on the control shader's outputs. Full details of this are available in the specification.

Note: This stage is also affected by the type of primitive specified in the tessellation evaluation shader.

Tessellation Evaluation

After the fixed function tessellation stage, this shader consumes those generated primitives, doing final evaluation of the vertices before passing them on to Geometry Shading. The values output by the fixed function tessellation are usually abstract and not suitable for rasterization directly, so are modified by this shader stage into a more final output, typically involving interpolation.

Example

```
// Create a Tessellation Control Shader
GLuint geometryShader = glCreateShader(GL_TESS_CONTROL_SHADER_EXT);
// Create a Tessellation Control Shader
GLuint geometryShader = glCreateShader(GL_TESS_EVALUATION_SHADER_EXT);
```

Example

```
// Tessellation control shader
#extension GL_EXT_tessellation_shader : require
#extension GL_EXT_shader_io_blocks : require
layout(vertices = 3) out;
void main(void)
{
 gl_TessLevelOuter[0] = 2.0;
 gl_TessLevelOuter[1] = 4.0;
 gl_TessLevelOuter[2] = 6.0;
 gl_TessLevelOuter[3] = 8.0;
 gl_TessLevelInner[0] = 8.0;
 gl_TessLevelInner[1] = 8.0;
 gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
}
// Tessellation evaluation shader
#extension GL_EXT_tessellation_shader : require
#extension GL_EXT_shader_io_blocks : require
layout(triangles, equal_spacing, ccw) in;
vec3 interpolate(vec3 v0, vec3 v1, vec3 v2)
{
    return (gl_TessCoord.x * v0) + (gl_TessCoord.y * v1) + (gl_TessCoord.z * v2);
```

```
}
void main(void)
{
 gl_Position = interpolate(gl_in[0].gl_Position, gl_in[1].gl_Position, gl_in[2].gl_Position);
}
```

GL_EXT_texture_border_clamp

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 2.0, OpenGL ES 3.x

Description

When sampling from a texture, if the requested UV coordinates reference a place outside of the texture, a wrapping behaviour is needed to specify what happens. OpenGL ES has three strategies depending on version:

- OpenGL ES 2.0 allows either clamping the UVs to the edge of the texture or wrapping around to the start of the valid range.
- OpenGL ES 3.0 added mirrored repeat, which acts as if the texture was flipped each time it wraps the coordinates.

This extension adds an additional behaviour - returning an application-defined border colour. This allows a shader to draw a texture to the screen with a dedicated border colour if so desired. It also gives the shader a cheap way to detect that a UV coordinate has gone out of range, without complex shader logic.

Example

```
// Set the texture's border colour to opaque red
//                        Red   Green Blue  Alpha
GLfloat borderColour[] = {1.0f, 0.0f, 0.0f, 1.0f};
glBindTexture(GL_TEXTURE_2D, texture)
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR_EXT, borderColour);
// Set the texture to use the border clamp wrapping mode.
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER_EXT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER_EXT);
```

GL_EXT_texture_buffer

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 3.1

Description

This extension adds a way to effectively treat a buffer as a 1-dimensional texture, and sample from that buffer in a shader. This is done by attaching a buffer object as the data store for a texture, in place of the TexStorage* or TexImage* functions.

This allows applications to have access to a linear texture in memory which can be mapped into CPU space and the data can be manipulated more readily than typically opaque textures.

Example

```
// Bind a texture object
GLuint textureObject;
glBindTexture(GL_TEXTURE_2D, textureObject);
// Set a buffer as the bound texture object's data store, with RGBA8 data
GLuint bufferObject;
glTexBufferEXT(GL_TEXTURE_2D, GL_RGBA8, bufferobject);
[Example]
#extension GL_EXT_texture_buffer : require
uniform samplerBuffer myTextureBuffer;
in float textureBufferCoords;
void main()
{
 ...
 // Sample the texture buffer.
 vec4 value = texture(myTextureBuffer, textureBufferCoords);
 ...
}
```

GL_EXT_texture_cube_map_array

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 3.1

Description

This extension adds the idea of a cube map array to the API, such that multiple cube maps can be stored in a single array texture. This allows things like skyboxes with multiple texture layers to be rendered, whilst only consuming a single texture unit.

Example

```
// Generate a cube map array texture
GLuint cubeArrayTexture;
glGenTextures(1, &cubeArrayTexture);
glBindTexture(GL_TEXTURE_CUBE_MAP_ARRAY_EXT, cubeArrayTexture);
// Create the texture storage with 8 layers.
// Number of layers allocated = floor(depth / 6), as there are 6 faces for every cube map. So
 simply multiply by 6 the number of layers needed.
glTexStorage3D(GL_TEXTURE_CUBE_MAP_ARRAY_EXT, 0, GL_RGBA8, textureWidth, textureHeight, 8 * 6);
[Example]
#extension GL_EXT_texture_cube_map_array : require
uniform lowp samplerCubeArray cubeArrayTexture;
in highp vec3 viewDirection;
in highp float layer;
void main()
{
```

```
  ...
  value += texture(cubeArrayTexture, vec4(viewDirection, layer);
  ...
}
```

GL_EXT_texture_filter_anisotropic

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 2.0, 3.x

Description

This extension enables anisotropic texture filtering, which is a method that can improve the quality of textures rendered at an oblique angle to the camera. Typically, texture filtering is performed isotropically, which means that the filtering is performed the same way across each axis (e.g. x and y). By effectively allowing the filter to vary with each axis independently, better results can be obtained when the scaling on each axis varies from its counterpart. Anisotropic filtering is a more computationally intensive method of filtering but can come with significant quality improvements in some cases and should be used accordingly.

Registry Link

http://www.khronos.org/registry/gles/extensions/EXT/texture_filter_anisotropic.txt

Example

```
// Enable anisotropic filtering with a maximum anisotropy of 4
glTexParameterf(GL_TEXTURE_2D, GL_MAX_ANISOTROPY, 4.0);
```

GL_EXT_texture_format_BGRA8888

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x, 2.0, 3.x

Description

Typically, textures in OpenGL ES run in RGBA order; this extension adds BGRA8888 as a new format so that textures in this byte order can be used instead. This can be effectively simulated in OpenGL ES 2.0 and 3.0 by using texture swizzling when it is not supported. Whilst this extension should be preferentially used, GL_IMG_texture_format_BGRA8888 mirrors this extension and can be used when this is not available. Apple hardware exposes a variation on this extension:

GL_APPLE_texture_format_BGRA888, which is identical except that when loading the texture, the internal format should be RGBA instead of being equivalent to the format.

Registry Link

http://www.khronos.org/registry/gles/extensions/EXT/EXT_texture_format_BGRA8888.txt

Example

```
// Create a BGRA8888 texture
glTexImage2D(GL_TEXTURE_2D, 0, GL_BGRA_EXT, 1024, 1024, 0, GL_BGRA_EXT, GL_UNSIGNED_BYTE,
 pixelData);
```

GL_EXT_texture_rg

Supported Hardware

Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 2.0

Description

OpenGL ES exposed luminance/alpha texture formats as the only single and double channel texture formats available, and these were not available as colour buffers when rendering. This extension adds the RG88 and R8 texture formats which can additionally be used as colour buffers. These also have the benefit of dropping the baggage previously associated with the LA formats, such as luminance being replicated across all channels - something that is no longer necessary with programmable shaders.

Note

This functionality is core to OpenGL ES 3.0.

Registry Link

http://www.khronos.org/registry/gles/extensions/EXT/EXT_texture_rg.txt

Example

```
// Create a two-channel byte texture
glTexImage2D(GL_TEXTURE_2D, 0, GL_RG_EXT, 1024, 1024, 0, GL_RG_EXT, GL_UNSIGNED_BYTE,
 pixelData);
```

GL_EXT_texture_sRGB_decode

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 2.0, 3.x

Description

Textures in the sRGB colour space are, by default, implicitly converted to the linear colour space when sampled in a shader. However, there are use cases where a shader may wish to sample the raw sRGB data instead. This function provides a texture parameter which allows individual textures/samplers to choose to not perform the implicit decode.

Registry Link

http://www.khronos.org/registry/gles/extensions/EXT/texture_sRGB_decode.txt

Example

```
// Turn off the implicit decode for a sampler
glSamplerParameteri(samplerName, GL_TEXTURE_SRGB_DECODE_EXT, GL_SKIP_DECODE_EXT);
```

GL_EXT_texture_sRGB_R8

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 2.0, 3.0, 3.1

Description

This extension introduces SR8_EXT as an acceptable internal format. This allows efficient sRGB sampling for source images stored as a separate texture per channel.

Registry Link

https://www.khronos.org/registry/gles/extensions/EXT/EXT_texture_sRGB_R8.txt

Example

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_SR8_EXT, 1024, 1024, 0, GL_SR8_EXT, GL_UNSIGNED_BYTE,
 pixelData);
```

GL_EXT_texture_sRGB_RG8

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 2.0, 3.0, 3.1

Description

This extension introduces SRG8_EXT as an acceptable internal format. This allows efficient sRGB sampling for source images stored with 2 channels.

Registry Link

https://www.khronos.org/registry/gles/extensions/EXT/EXT_texture_sRGB_RG8.txt

Example

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_SRG8_EXT, 1024, 1024, 0, GL_SRG8_EXT, GL_UNSIGNED_BYTE,
 pixelData);
```

GL_EXT_YUV_target

Valid APIs

OpenGL ES 3.1+

Description

This extension adds support for three new YUV related items: first rendering to YUV images, second sampling from YUV images while keeping the data in YUV space, third it defines a new built in function that does conversion from RGB to YUV with controls to choose ITU-R BT.601-7, ITU-R BT.601-7 Full range (JFIF images), or ITU-R BT.709-5 standard.

Registry Link

https://www.khronos.org/registry/OpenGL/extensions/EXT/EXT_YUV_target.txt

Example

```
uniform __samplerExternal2DY2YEXT   u_sTexture;
layout (yuv) out vec4 color;
void main()
{
 vec4 yuvTex = texture(u_sTexture, texcoord);
 //decode yuv texture according to standard
 vec3 rgbTex = yuv_2_rgb(yuvTex.xyz, itu_601); //yuv standards (could be itu_601_full_range,
 itu_709 as well)
 //do some colour operations in RGB
 //...
 //convert back to yuv
 yuvTex.xyz = rgb_2_yuv (rgbTex, itu_601);
```

```
 //write out in yuv format
 color = yuvTex;
}
```

GL_IMG_bindless_texture

Valid APIs

OpenGL ES 3.1+

Description

This extension allows OpenGL ES applications to access texture objects in shaders without first binding each texture to one of a limited number of texture image units. Using this extension, an application can query a 64-bit unsigned integer texture handle for each texture that it wants to access and then use that handle directly in GLSL ES. This extension significantly reduces the amount of API and internal GL driver overhead needed to manage resource bindings.

Registry Link

https://www.khronos.org/registry/OpenGL/extensions/IMG/IMG_bindless_texture.txt

Example

```
#define NUM_TEXTURES      256
GLuint   textures[NUM_TEXTURES];
GLuint64 texHandles[NUM_TEXTURES];
// Initialize the texture objects and handles.
glGenTextures(NUM_TEXTURES, textures);
for (int i = 0; i < NUM_TEXTURES; i++) {
 // Initialize the texture images with glTexStorage.
 // Initialize the texture parameters as required.
 //...
 // Get a handle for the texture.
 texHandles[i] = glGetTextureHandleIMG(textures[i]);
 // At this point, it's no longer possible to modify texture parameters
 // for "textures[i]". However, it's still possible to update texture
 // data via glTexSubImage.
}
// Render a little bit using each of the texture handles in turn.
for (int i = 0; i < NUM_TEXTURES; i++) {
 // Update the single sampler uniform <u> to point at "texHandles[i]".
 glUniformHandleui64IMG(location, texHandles[i]);
 drawStuff();
}
```

Example

```
uniform int whichSampler;
in highp vec2 texCoord;
out lowp vec4 finalColor;
uniform Samplers {
 sampler2D allTheSamplers[NUM_TEXTURES];
};
void main()
{
 finalColor = texture(allTheSamplers[whichSampler], texCoord);
}
```

GL_IMG_framebuffer_downsample

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 2.0, 3.0, 3.1

Description

This extension introduces the ability to attach colour buffers to a framebuffer that are at a lower resolution than the framebuffer itself, with the GPU automatically downsampling the colour attachment to fit.

This can be useful for various post-process rendering techniques where it is desirable to generate downsampled images in an efficient manner, or for a lower resolution post-process technique.

This extension exposes at least a 2 x 2 downscale. Other downsampling modes may be exposed on the system and this can be queried.

Registry Link

https://www.khronos.org/registry/gles/extensions/IMG/IMG_framebuffer_downsample.txt

Example

```
GLint xDownscale = 1;
GLint yDownscale = 1;
// Select a downscale amount if possible
if (extension_is_supported("GL_IMG_framebuffer_downsample")
{
 // Query the number of available scales
 GLint numScales;
 glGetIntegerv(GL_NUM_DOWNSAMPLE_SCALES_IMG, &numScales);
 // 2 scale modes are supported as minimum, so only need to check for
 // better than 2x2 if more modes are exposed.
 if (numScales > 2)
 {
  // Try to select most aggressive scaling.
  GLint bestScale = 1;
  GLint tempScale[2];
  GLint i;
  for (i = 0; i < numScales; ++i)
  {
   glGetIntegeri_v(GL_DOWNSAMPLE_SCALES_IMG, i, tempScale);
   // If the scaling is more aggressive, update our x/y scale values.
   if (tempScale[0] * tempScale[1] > bestScale)
   {
    xDownscale = tempScale[0];
    yDownscale = tempScale[1];
   }
  }
 }
 else
 {
  xDownscale = 2;
  yDownscale = 2;
 }
}
```

```
// Create depth texture. Depth and stencil buffers must be full size
GLuint depthTexture;
glGenTextures(1, &depthTexture);
glBindTexture(GL_TEXTURE_2D, depthTexture);
glTexStorage2D(GL_TEXTURE_2D, 1, GL_DEPTH_COMPONENT16, width, height);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glBindTexture(GL_TEXTURE_2D, 0);
// Create a full size RGBA texture with single mipmap level
GLuint texture;
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture);
glTexStorage2D(GL_TEXTURE_2D, 0, GL_RGBA4, width, height);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glBindTexture(GL_TEXTURE_2D, 0);
// Scale the width and height appropriately.
GLint scaledWidth = width / xDownscale;
GLint scaledHeight = height / yDownscale;
// Create a reduced size RGBA texture with single mipmap level
GLuint scaledTexture;
glGenTextures(1, &scaledTexture);
glBindTexture(GL_TEXTURE_2D, scaledTexture);
glTexStorage2D(GL_TEXTURE_2D, 0, GL_RGBA4, scaledWidth, scaledHeight);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glBindTexture(GL_TEXTURE_2D, 0);
// Create framebuffer object, attach textures
GLuint framebuffer;
glGenFramebuffers(1, &framebuffer);
glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
GL_TEXTURE_2D, depthTexture);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
GL_TEXTURE_2D, texture, 0);
glFramebufferTexture2DDownsampleIMG(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1,
GL_TEXTURE_2D, scaledTexture, 0, xDownscale, yDownscale);
// Handle unsupported cases
if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE)
{
 ...
}
// Draw to the texture
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
...
// Discard the depth renderbuffer contents if possible/available
if (extension_supported("GL_EXT_discard_framebuffer"))
{
 GLenum discard_attachments[] = { GL_DEPTH_ATTACHMENT };
 glDiscardFramebufferEXT(GL_FRAMEBUFFER, 1, discard_attachments);
}
/*
Draw to the default framebuffer using the textures with various post
processing effects.
*/
glBindFramebuffer(GL_FRAMEBUFFER, 0);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, scaledTexture);
```

GL_IMG_multisampled_render_to_texture

Supported Hardware

Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x, 2.0, 3.x

Description

This extension adds multisampled rendering capabilities to OpenGLES, allowing a user to create multisampled render buffers, and attach textures to a framebuffer in a way that they can be used for multisampled renders. By enabling textures to be attached as multisampled but without using actual multisample textures, a significant amount of bandwidth can be saved. This mode of operation enables multisampled anti-aliasing to smooth lines at the edges of polygons and reduce the appearance of jagged lines. This is functionally equivalent to GL_EXT_multisampled_render_to_texture on PowerVR hardware.

Note

Part of this functionality is core to OpenGL ES 3.0. The renderbuffer functionality is present in core, however, the multisampled texture attachment is not.

Registry Link

http://www.khronos.org/registry/gles/extensions/IMG/IMG_multisampled_render_to_texture.txt

Example

```
// Create a multisampled depth renderbuffer
GLint depthBuffer = glRenderbufferStorageMultisampleIMG(GL_RENDERBUFFER, 4,
 GL_DEPTH_COMPONENT16, 1024, 1024);
// Attach the renderbuffer to the framebuffer as per usual. Note: If one attachment is
// multi-sampled, all attachments need to be. If there is a mix, the framebuffer will be
// incomplete.
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, depthBuffer);
// Create a regular colour buffer (texture). This extension doesn't introduce multi-sample
// textures, instead attaching regular textures that multi-sampling will resolve to before
// writing out.
GLint colourBuffer = glTexStorage(GL_TEXTURE_2D, 1, GL_RGBA8, 1024, 1024);
// Attach it to a framebuffer as multisampled, so that OpenGL ES knows that it should resolve
// a multisampled render to this texture.
glFramebufferTexture2DMultisampleIMG(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
 colourBuffer, 0);
```

GL_IMG_polygon_offset_clamp

Supported Hardware

Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x, 2.0, 3.0

Description

This extension adds a new parameter to the polygon offset function that clamps the calculated offset to a minimum or maximum value. The clamping functionality is useful when polygons are nearly parallel to the view direction because their high slopes can result in arbitrarily large polygon offsets. In the particular case of shadow mapping, the lack of clamping can produce the appearance of unwanted holes when

the shadow casting polygons are offset beyond the shadow receiving polygons, and this problem can be alleviated by enforcing a maximum offset value.

Note

OpenGL ES 1.0 is required.

This extension is written against the OpenGL ES 3.1 Specification (October 29, 2014).

If the GL_EXT_polygon_offset_clamp is present, the EXT variant of this extension will be recommended to use.

Example

```
//<factor> scales the maximum depth slope of the polygon, and <units>
//scales an implementation-dependent constant that relates to the
//usable resolution of the depth buffer. The resulting values are
//summed to produce the polygon offset value, which may then be
//clamped to a minimum or maximum value specified by <clamp>. The
//values <factor>, <units>, and <clamp> may each be positive,
//negative, or zero. Calling the command PolygonOffset is equivalent
//to calling the command PolygonOffsetClampIMG with <clamp> equal to
//zero."
//   |   m x <factor> + r x <units>,          if <clamp> = 0 or NaN;
//   |
//o = <    min(m x <factor> + r x <units>, <clamp>),   if <clamp> > 0;
//   |
//   |_  max(m x <factor> + r x <units>, <clamp>),   if <clamp> < 0.
float factor = 0.2f;
float units = 0.2f;
float clamp = 0.2f;
glPolygonOffsetClampIMG(factor, units, clamp);
```

GL_IMG_program_binary

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 2.0, 3.x

Description

This extension enables PowerVR program binary formats to be queried and handled by the GL_OES_get_program_binary extension, which allows binary shader programs to be retrieved, stored and reloaded, avoiding compilation steps on subsequent runs of an application.

Registry Link

http://www.khronos.org/registry/gles/extensions/IMG/IMG_program_binary.txt

Example

```
// Create a shader program object
GLuint shaderProgram = glCreateProgram();
```

```
// If there isn't already a binary, create a program
if (!binaryExists)
{
 // Create shaders, compile them, attach them and link them as per normal, then...
 // Retrieve the program binary size
 GLint length=0;
 glGetProgramiv(shaderProgram, GL_PROGRAM_BINARY_LENGTH, &length);
 // Pointer to the binary shader program in memory, needs to be allocated with the
 // right size.
 GLvoid* programBinaryData = (GLvoid*)malloc(length);
 // The format that the binary is retrieved in.
 GLenum binaryFormat=0;
 // Error checking variable - this should be greater than 0 after
 // glGetProgramBinaryOES, otherwise there was an error.
 GLsizei lengthWritten=0;
 glGetProgramBinaryOES(shaderProgram, length, &lengthWritten, &binaryFormat,
       programBinaryData);
}
else
{
 // Get the binary data, how much data there is, and what format it's in from whatever
 // cache it's stored in (e.g. a file)
 // ...
 // Instead of creating, compiling, attaching and linking shaders, upload a binary
 // program data from a program that previously underwent all of these stages.
 glProgramBinaryOES(shaderProgram, binaryFormat, programBinaryData,
    programBinaryDataLength);
}
```

GL_IMG_read_format

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x, 2.0, 3.x

Description

Reading pixels from a framebuffer in OpenGL ES is performed via the function glReadPixels(). This function is able to read data out in only two formats; RGBA8888, and another, implementation defined format, which can be queried with glGetIntegerv(). In OpenGL ES 1.x, only one read format is available in core, so GL_OES_read_format is required for this extension. Normally these additional formats are limited to being RGBA, RGB or Alpha-Only formats. This extension adds a new readable format (GL_BGRA_IMG) and a new readable type (UNSIGNED_SHORT_4_4_4_4_REV_IMG), allowing the function to be extended to use a different native format than typically allowed.

Registry Link

http://www.khronos.org/registry/gles/extensions/IMG/IMG_read_format.txt

Example

```
// Read pixels can now be performed with GL_BGRA data as well as any other default formats.
glReadPixels(0, 0, 1024, 1024, GL_BGRA_IMG, GL_UNSIGNED_SHORT_4_4_4_4_REV_IMG, pixelDataOutput);
```

GL_IMG_shader_binary

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 2.0, 3.x

Description

The OpenGL ES 2.0 and 3.0 specifications include functions "glGetShaderBinary/ glShaderBinary" - allowing an application developer to retrieve, store and load binary shaders after compilation. By using this extension to cache compiled shaders, it is possible to avoid the run time cost of compiling shaders on subsequent runs. The specifications do not, however, define any binary formats for shaders, which is instead left up to each implementer. This extension enables Imagination's shader binary format to be exposed through these mechanisms.

Typically, it is more efficient to use GL_OES_get_program_binary/ GL_IMG_program_binary where available, as these also avoid the cost of linking and binding attributes - saving additional time on start up.

Registry Link

http://www.khronos.org/registry/gles/extensions/IMG/IMG_shader_binary.txt

Example

```
// Create a fragment shader
GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
// Instead of uploading shader source and compiling, upload pre-compiled binary data
glShaderBinary(1, &fragmentShader, GL_SGX_BINARY_IMG, binaryShaderData,
 lengthOfBinaryShaderData);
```

GL_IMG_shader_group_vote

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 2.0, 3.0

Description

This extension provides new built-in functions to compute the composite of a set of boolean conditions across a group of shader invocations. These composite results may be used to execute shaders more efficiently on a single-instruction multiple-data (SIMD) processor. The set of shader invocations across which boolean conditions are evaluated is implementation-dependent, and this extension provides

no guarantee over how individual shader invocations are assigned to such sets. In particular, the set of shader invocations has no necessary relationship with the compute shader local work group -- a pair of shader invocations in a single compute shader work group may end up in different sets used by these built-ins.

Compute shaders operate on an explicitly specified group of threads (a local work group), but many implementations of OpenGL ES 3.0 will even group non-compute shader invocations and execute them in a SIMD fashion. When executing code like

```
if (condition)
{
  result = do_fast_path();
}
else
{
  result = do_general_path();
}
```

where `condition` diverges between invocations, a SIMD implementation might first call `do_fast_path()` for the invocations where `condition` is true and leave the other invocations dormant. Once `do_fast_path()` returns, it might call `do_general_path()` for invocations where `condition` is false and leave the other invocations dormant. In this case, the shader executes **both** the fast and the general path and might be better off just using the general path for all invocations.

This extension provides the ability to avoid divergent execution by evaluating a condition across an entire SIMD invocation group using code like:

```
if (allInvocationsIMG(condition))
{
    result = do_fast_path();
}
else
{
    result = do_general_path();
}
```

The built-in function `allInvocationsIMG()` will return the same value for all invocations in the group, so the group will either execute `do_fast_path()` or `do_general_path()`, but never both. For example, shader code might want to evaluate a complex function iteratively by starting with an approximation of the result and then refining the approximation.

Some input values may require a small number of iterations to generate an accurate result (`do_fast_path`) while others require a larger number (`do_general_path`). In another example, shader code might want to evaluate a complex function (`do_general_path`) that can be greatly simplified when assuming a specific value for one of its inputs (`do_fast_path`).

Note

If the GL_ARB_shader_group_vote is present, the ARB variant of this extension will be recommended to use.

Example

```
#extension GL_IMG_shader_group_vote : require
if (allInvocationsIMG(condition))
{
```

```
    result = do_fast_path();
}
else
{
    result = do_general_path();
}
```

GL_IMG_texture_compression_pvrtc

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x, 2.0, 3.x

Description

Compressed textures are a core part of modern rendering solutions, enabling much more texture content to be pushed and with greater performance due to the lower bandwidth and memory requirements. PVRTC is PowerVR's proprietary texture compression format which achieves high quality, full RGBA compression at data rates of 2 or 4 bits per pixel. Textures are restricted to being square, with power of two dimensions - though typically rectangular textures are often supported; there is no way to query for this.

This extension specifically enables supported devices to use these textures directly, often dramatically improving performance over uncompressed textures. This extension does not enable compression of these textures as it is too computationally expensive; this can be done on Desktop platforms with PVRTexTool (*http://www.imgtec.com/powervr/insider/powervr-pvrtextool.asp*)

Registry Link

http://www.khronos.org/registry/gles/extensions/IMG/IMG_texture_compression_pvrtc.txt

Example

```
// Upload a 4bpp PVRTC1 texture, RGBA and RGB are uploaded in the same way.
GLuint bitsPerPixel = 4;
glCompressedTexImage2D(GL_TEXTURE_2D, 0, GL_COMPRESSED_RGBA_PVRTC_4BPPV1_IMG, 1024, 1024, 0,
 (width*height*bitsPerPixel)/8, pixelData);
// Upload a 2bpp PVRTC1 texture, RGBA and RGB are uploaded in the same way.
GLuint bitsPerPixel = 2;
glCompressedTexImage2D(GL_TEXTURE_2D, 0, GL_COMPRESSED_RGB_PVRTC_2BPPV1_IMG, 1024, 1024, 0,
                       (width*height*bitsPerPixel)/8, pixelData);
```

GL_IMG_texture_compression_pvrtc2

Supported Hardware

Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x, 2.0, 3.x

Description

Compressed textures are a core part of modern rendering solutions, enabling much more texture content to be pushed and with greater performance due to the lower bandwidth and memory requirements. PVRTC2 is PowerVR's proprietary texture compression format which achieves high quality, full RGBA compression at data rates of 2 or 4 bits per pixel. PVRTC2 is related to the original PVRTC format but adds a number of additional features which allow it to improve further on the original's quality levels. In particular, textures with arbitrary dimensions are now supported, and there is a hard edge mode which can improve areas of high discontinuity and non-tiled textures.

This extension specifically enables supported devices to use these textures directly, often dramatically improving performance over uncompressed textures. This extension does not enable compression of these textures as it is too computationally expensive; this can be done on Desktop platforms with PVRTexTool (*http://www.imgtec.com/powervr/insider/powervr-pvrtextool.asp*)

Registry Link

http://www.khronos.org/registry/gles/extensions/IMG/IMG_texture_compression_pvrtc2.txt

Example

```
// Upload a 4bpp PVRTC2 texture
GLuint bitsPerPixel = 4;
glCompressedTexImage2D(GL_TEXTURE_2D, 0, GL_COMPRESSED_RGBA_PVRTC_4BPPV2_IMG, 1024, 1024, 0,
 (width*height*bitsPerPixel)/8, pixelData);
// Upload a 2bpp PVRTC2 texture
GLuint bitsPerPixel = 2;
glCompressedTexImage2D(GL_TEXTURE_2D, 0, GL_COMPRESSED_RGBA_PVRTC_2BPPV2_IMG, 1024, 1024, 0,
                           (width*height*bitsPerPixel)/8, pixelData);
```

GL_IMG_texture_filter_cubic

Supported Hardware

Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x, 2.0, 3.0

Description

OpenGL ES provides two sampling methods available; nearest neighbour or linear filtering, with optional MIP Map sampling modes added to move between differently sized textures when downsampling.

This extension adds an additional, high quality cubic filtering mode, using a Catmull-Rom bicubic filter. Performing this kind of filtering can be done in a shader by using 16 samples, but this can be inefficient. The cubic filter mode exposes an optimized high quality texture sampling using fixed functionality.

This extension affects the way textures are sampled, by modifying the way texels within the same MIP-Map level are sampled and resolved. It does not affect MIP-Map filtering, which is still limited to linear or nearest.

Registry Link

https://www.khronos.org/registry/gles/extensions/IMG/IMG_texture_filter_cubic.txt

Example

```
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, CUBIC_MIPMAP_LINEAR_IMG);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, CUBIC_IMG);
```

GL_IMG_texture_format_BGRA8888

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x, 2.0, 3.x

Description

Typically, textures in OpenGL ES run in RGBA order; this extension adds BGRA8888 as a new format so that textures in this byte order can be used instead. This can be effectively simulated in OpenGL ES 2.0 and 3.0 by using texture swizzling when it is not supported. GL_EXT_texture_format_BGRA8888 mirrors this extension and should be preferentially used. Apple hardware exposes a variation on this extension: GL_APPLE_texture_format_BGRA888, which is identical except that when loading the texture, the internal format should be RGBA instead of being equivalent to the format.

Example

```
// Create a BGRA8888 texture
glTexImage2D(GL_TEXTURE_2D, 0, GL_BGRA_IMG, 1024, 1024, 0, GL_BGRA_IMG, GL_UNSIGNED_BYTE,
 pixelData);
```

GL_IMG_texture_npot

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 2.0

Description

This extension adds MIPMap capable minification filters to OpenGL ES 2 for non-power of two (NPOT) textures. OpenGL ES 2 limits NPOT textures as it does not support MIPMapping for them. This extension allows users to call 'glGenerateMipMap' to create a full MIP chain, and then adds minification filters that allow them to be accessed. This extension does not add functionality allowing applications to supply a full NPOT MIP Map chain prior to runtime, however.

Note

This functionality is core to OpenGL ES 3.0.

Example

```
// Upload a texture with dimensions of 15 by 47. Typically, this isn't supported without
// extension support.
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 15, 47, 0, GL_RGBA, GL_UNSIGNED_BYTE, pixelData);
// Mip-Maps can't be specified by this extension, but can be auto-generated for non-compressed
// textures
glGenerateMipmap(GL_TEXTURE_2D);
// Any filter mode specified by OpenGL is now valid with this extension, whereas it would not
// have been previously.
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
// Only the following parameters are specifiable, anything else is invalid (including the
// default texture parameters!).
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

GL_KHR_blend_equation_advanced

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 2.0, 3.x

Description

On its own OpenGL ES provides only a limited number of blend equations for users, with a number of different sources and destinations. This extension adds several additional blending equations to the API as follows:

- MULTIPLY_KHR
- SCREEN_KHR
- OVERLAY_KHR
- DARKEN_KHR
- LIGHTEN_KHR

- COLORDODGE_KHR
- COLORBURN_KHR
- HARDLIGHT_KHR
- SOFTLIGHT_KHR
- DIFFERENCE_KHR
- EXCLUSION_KHR
- HSL_HUE_KHR
- HSL_SATURATION_KHR
- HSL_COLOR_KHR
- HSL_LUMINOSITY_KHR

The exact meaning of these is detailed in the specification itself, but those familiar with various rendering or imaging suites should recognise most of them.

To use these functions, both API functions and shader code are required to enable them. There is a layout qualifier for output variables determining the likely blend modes, and new enums for the glBlendEquation* functions. Any mismatches in the two when rendering will cause a draw-time error.

Registry Link

http://www.khronos.org/registry/gles/extensions/KHR/blend_equation_advanced.txt

Example

```
// Set the colour dodge blend equation
glBlendEquationi(GL_COLORDODGE_KHR);
```

Example

```
// Appropriate shader code for color dodge blending
layout(location = 0, blend_support_colordodge) highp vec4 outputColour;
```

GL_KHR_blend_equation_advanced_coherent

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 2.0, 3.x

Description

This extension is actually part of GL_KHR_blend_equation_advanced, and is an additional guarantee that blending is done coherently and in API primitive order. An enable is provided to opt out of this behaviour, as if this guarantee was not supported.

Registry Link

http://www.khronos.org/registry/gles/extensions/KHR/blend_equation_advanced.txt

Example

```
// Turn off coherent blending.
glDisable(GL_BLEND_ADVANCED_COHERENT_KHR);
```

GL_KHR_debug

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 2.0, 3.x

Description

KHR_debug combines a number of debug features into a single package that is made consistent across OpenGL and OpenGL ES. There are three main features provided by this extension:

- Message Logging: While running in Debug mode, if the driver detects any errors, performance problems or any sort of problem, it can raise a human readable message to a log. An application is also free to add its own messages to this log. A callback API is also provided, so that the user can be notified when a message is generated. This is similar to the GL_ARB_debug_output extension.
- Group Markers: Push and Pop functions are provided to encapsulate a block of functionality with a label. This is primarily to aid tracing and other debug tools when examining a recorded call stream. This functionality is similar to that provided by EXT_debug_marker.
- Object Labels: Finally, this extension also adds the ability to tag objects with a name. Again, this is primarily for tracing tools when examining a recorded call stream, to allow users to quickly identify specific objects, rather than looking them up by their assigned identifier.

In conjunction with debug tools, this is a very useful extension as it allows users to get a better idea of how their application's GL calls are laid out. Typical usage includes things like putting group markers around the start and end of application functions or labelling key textures or vertex buffers to match the assets they contain.

Registry Link

http://www.khronos.org/registry/gles/extensions/KHR/debug.txt

Example

```
// Debug callback function
void (APIENTRY) debugMessageCallback(GLenum source,
                                     GLenum type,
```

```
                                    GLuint id,
                                    GLenum severity,
                                    GLsizei length,
                                    const GLchar* message,
                                    const void* userParam)
{
 /*
  Handle the message based on severity and such. Here we're using printf to
  output a message.
  This example only handles the type, severity and message. Useful information
  can be gleaned
  from the other parameters as well but would unnecessarily clutter this
  example.
 */
 // Output the message type
 switch (type)
 {
 case GL_DEBUG_TYPE_ERROR:
  {
   printf("%s", "Error ");
   break;
  }
 case GL_DEBUG_TYPE_DEPRECATED_BEHAVIOUR:
  {
   printf("%s", "Deprecated Behaviour ");
   break;
  }
 case GL_DEBUG_TYPE_UNDEFINED_BEHAVIOUR:
  {
   printf("%s", "Undefined Behaviour ");
   break;
  }
 case GL_DEBUG_TYPE_PORTABILITY:
  {
   printf("%s", "Portability Issue ");
   break;
  }
 case GL_DEBUG_TYPE_PERFORMANCE:
  {
   printf("%s", "Performance Issue ");
   break;
  }
 case GL_DEBUG_TYPE_OTHER:
  {
   printf("%s", "Misc Message ");
   break;
  }
 case GL_DEBUG_TYPE_MARKER:
  {
   printf("%s", "Marker Hit ");
   break;
  }
 case default:
  {
   printf("%s", "Unknown Message Type ");
   break;
  }
 }
 // Output the message severity
 switch (severity)
 {
 case GL_DEBUG_SEVERITY_NOTIFICATION:
  {
   printf("%s", "(Notification): ");
   break;
  }
 case GL_DEBUG_SEVERITY_LOW:
  {
   printf("%s", "(Low Severity): ");
   break;
  }
 case GL_DEBUG_SEVERITY_MEDIUM:
  {
   printf("%s", "(Medium Severity): ");
   break;
  }
 case GL_DEBUG_SEVERITY_HIGH:
  {
```

```
    printf("%s", "(High Severity): ");
    break;
   }
  case default:
   {
    printf("%s", "(Unknown Severity): ");
    break;
   }
  }
  // Append the message and a newline.
  if (length==0)
  {
   // NULL terminated strings are easy enough
   printf("%s\n", message);
  }
  else
  {
   // Strings with a specified length are not NULL terminated
   printf("%.*s", length, message);
  }
}
void initialise()
{
 /*
  Use a debug marker to encapsulate the initialise function, to mark this group
  in a tracing tool like PVRTrace.
 */
 glPushDebugMarker(GL_DEBUG_SOURCE_APPLICATION, 0, 0, "Function 'initialise'");
 ...
 // Create and label a texture object
 GLuint importantTexture;
 glGenTextures(1, &importantTexture);
 glObjectLabel(GL_TEXTURE, importantTexture, 0, "importantTexture");
 ...
 // Pop the debug marker
 glPopDebugMarker();
}
```

GL_KHR_robustness

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x, OpenGL ES 2.0, OpenGL ES 3.x

Description

Several recent trends in how OpenGL ES integrates into modern computer systems have created new requirements for robustness and security for GL rendering contexts.

Additionally, GPU architectures now support hardware fault detection; for example, video memory supporting ECC (error correcting codes) and error detection. GL contexts should be capable of recovering from hardware faults such as uncorrectable memory errors. Along with recovery from such hardware faults, the recovery mechanism can also allow recovery from video memory access exceptions and system software failures. System software failures can be due to device changes or driver failures.

GL queries that return (write) some number of bytes to a buffer indicated by a pointer parameter introduce risk of buffer overflows that might be exploitable by

malware. To address this, queries with return value sizes that are not expressed directly by the parameters to the query itself are given additional API functions with an additional parameter that specifies the number of bytes in the buffer and never writing bytes beyond that limit. This is particularly useful for multi-threaded usage of GL contexts in a "share group" where one context can change objects in ways that can cause buffer overflows for another context's GL queries.

The original ARB_vertex_buffer_object extension includes an issue that explicitly states program termination is allowed when out-of-bounds vertex buffer object fetches occur. Modern graphics hardware is capable of well-defined behaviour in the case of out-of-bounds vertex buffer object fetches. Older hardware may require extra checks to enforce well-defined (and termination free) behavior, but this expense is warranted when processing potentially untrusted content.

The intent of this extension is to address some specific robustness goals:

- For all existing GL queries, provide additional "safe" APIs that limit data written to user pointers to a buffer size in bytes that is an explicit additional parameter of the query.
- Provide a mechanism for a GL application to learn about graphics resets that affect the context. When a graphics reset occurs, the GL context becomes unusable and the application must create a new context to continue operation. Detecting a graphics reset happens through an inexpensive query.
- Define behaviour of OpenGL calls made after a graphics reset.
- Provide an enable to guarantee that out-of-bounds buffer object accesses by the GPU will have deterministic behaviour and preclude application instability or termination due to an incorrect buffer access. Such accesses include vertex buffer fetches of attributes and indices, and indexed reads of uniforms or parameters from buffers.

Registry Link

https://www.khronos.org/registry/gles/extensions/KHR/robustness.txt

Example

```
if(NO_ERROR == GetGraphicsResetStatus())
{
 ReadnPixels(x, y, width, height, format, type, bufSize, data);
 GetnUniformfv(program, location, bufSize, params);
}
```

GL_KHR_texture_compression_astc_ldr

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x, OpenGL ES 2.0, OpenGL ES 3.x

Description

Adaptive Scalable Texture Compression (ASTC) is the latest texture compression scheme that is intended to eventually become ubiquitous across mobile devices, replacing existing texture compression technologies. It is very flexible, offering a large range of bit rates from 8bpp (bits per pixel) down to 0.88bpp. Each block is always 128 bits, but the number of texels represented by each block is highly variable - ranging from 4x4 to 12x12 - which allows different images to be compressed at varying degrees of quality, whilst using the same compression scheme.

Example

```
// Upload an ASTC 4x4 texture
glCompressedTexImage2D(GL_TEXTURE_2D, 0, GL_COMPRESSED_RGBA_ASTC_4x4_KHR, width, height, 0,
 astcDataSize, astcData);
// Upload an ASTC 12x12 texture
glCompressedTexImage2D(GL_TEXTURE_2D, 0, GL_COMPRESSED_RGBA_ASTC_12x12_KHR, width, height, 0,
 astcDataSize2, astcData2);
```

GL_OES_blend_equation_separate

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x

Description

This function extends GL_OES_blend_subtract, which adds functionality allowing a blend equation to operate on all the channels in an output colour at once. However, often a different operation is wanted for the alpha and colour channels. This extension adds a new function, glBlendEquationSeparateOES, which operates in the same way as glBlendEquationOES, except that different equations can now be specified for RGB and Alpha.

Note

This extension is part of the OpenGL ES 1.x Extension Pack Specification and is core functionality in OpenGL ES 2.0 and 3.0.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_blend_equation_separate.txt

Example

```
// Set the blend equation to use additive blending for the RGB channels, but subtractive for
// the Alpha channel.
glBlendEquationSeparate(GL_FUNC_ADD_OES, GL_FUNC_SUBTRACT_OES);
```

GL_OES_blend_func_separate

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x

Description

OpenGL ES 1.x has functionality allowing a blend function to operate on all the channels in an output colour at once. However, often a different operation is wanted for the alpha and colour channels. This extension adds a new function, glBlendFuncSeparateOES, which operates in the same way as glBlendFunc, except that different equations can now be specified for RGB and Alpha.

Note

This extension is part of the OpenGL ES 1.x Extension Pack Specification and is core functionality in OpenGL ES 2.0 and 3.0.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_blend_func_separate.txt

Example

```
// Set the blend equation to use a value of transparent white for the source colour, but the
// destination colour remains as it otherwise would.
glBlendFuncSeparate(GL_ONE, GL_DST_COLOR, GL_ZERO, GL_DST_ALPHA);
```

GL_OES_blend_subtract

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x

Description

OpenGL ES 1.x has functionality allowing various basic blending functions to occur via glBlendFunc, by allowing the user to pick the source and destination values used in blend calculations. This feature enables further specification of how blending works by adding a new function - glBlendEquationOES, which allows a developer to specify, further what actually happens to those values once they have been selected. Usual GL operation adds the values together, using "GL_FUNC_ADD_OES" as a default. Two new modes are included by this function; "subtraction" and "reverse subtraction".

The equations used look as follows:

- GL_FUNC_ADD_OES:
  - Result = Source + Destination
- GL_FUNC_SUBTRACT_OES:
  - Result = Source - Destination
- GL_FUNC_REVERSE_SUBTRACT_OES:
  - Result = Destination - Source

This extension is further extended by GL_OES_blend_equation_separate.

Note

This extension is part of the OpenGL ES 1.x Extension Pack Specification and is core functionality in OpenGL ES 2.0 and 3.0.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_blend_subtract.txt

Example

```
// Set the blend equation to use additive blending
glBlendEquation(GL_FUNC_ADD_OES);
// Set the blend equation to use subtractive blending: source - destination
glBlendEquation(GL_FUNC_SUBTRACT_OES);
// Set the blend equation to use reverse subtractive blending: destination - source
glBlendEquation(GL_FUNC_REVERSE_SUBTRACT_OES);
```

GL_OES_byte_coordinates

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x

Description

This extension allows users to pass byte values as texture coordinates and vertex positions in OpenGL ES 1.x, which are otherwise not allowed by the API. Byte sized data has the advantage of consuming less bandwidth than larger types and can be useful for certain scenes.

Note

This functionality is superseded by programmable shader support in subsequent versions of OpenGL ES.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_byte_coordinates.txt

Example

```
// Input byte coordinates
GLbyte coordinates[] =
{
 0, 255, 0,
 255, 255, 0,
 255, 0, 0,
};
// Give the coordinates to OpenGL ES
glVertexPointer(3, GL_BYTE, 0, coordinates);
```

GL_OES_compressed_ETC1_RGB8_texture

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x, 2.0, 3.x

Description

This extension adds support for loading textures that have been compressed with the ETC1 texture compression scheme, directly into the API via glCompressedTexImage2D. This format provides RGB texture encoding at a lower bitrate than would otherwise be available, whilst still providing reasonable quality.

As this specification was written against OpenGL ES1, when used in ES2, some features applicable to ES2 are not available due to the non-trivial nature of colour read back from an ETC texture. In particular; Non-power of two ETC1 is not directly supported in some first generation ES2 hardware, so it's generally best to ensure that all ETC1 textures are encoded with square, power-of-two dimensions.

Note

Whilst this functionality can be exposed in OpenGL ES 3.0, it is effectively enabled anyway by the inclusion in core 3.0 of ETC2, which is backwards compatible. By passing the same data with the new enum "GL_COMPRESSED_RGB8_ETC2" the behaviour can be replicated.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_compressed_ETC1_RGB8_texture.txt

Example

```
// Upload an ETC1 texture.
GLuint bitsPerPixel = 4;
glCompressedTexImage2D(GL_TEXTURE_2D, 0, GL_COMPRESSED_RGB8_ETC2, 1024, 1024, 0,
```

```
                        (width*height*bitsPerPixel)/8, pixelData);
```

GL_OES_compressed_paletted_texture

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x

Description

This extension allows the use of paletted textures in OpenGL ES as a compressed texture. Paletted formats are textures which store a small number of colours as a palette, and then the texel array is a simple array of indices into this palette. The benefit of this is that the palette can be cached, and the indices have a lower bandwidth cost than a full colour. This extension allows the colour formats of RGB565, RGBA4444, RGBA551, RGB888 or RGBA8888. Each texel is then allowed to either have a 4- or 8-bit index value, and an appropriate number of palette colours to match.

Note

In subsequent versions of OpenGL ES, this functionality can be easily replicated using programmable shaders by having a small palette texture, and a vertex attribute or another texture specifying the index to use. This allows the functionality to be much more flexible than provided by this extension, so it was dropped.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_compressed_paletted_texture.txt

Example

```
// Upload a palletted texture. The size of the data will be dependent on how it was
// compressed and is thus dependent on whatever tool was used.
glCompressedTexImage2D(GL_TEXTURE_2D, 0, GL_PALETTE4_RGB8_OES, 1024, 1024, 0, pixelDataSize,
                       pixelData);
```

GL_OES_depth_texture

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 2.0

Description

When using a framebuffer, z-buffers need to be specifically supported by attaching a depth render target. Originally this was done with a depth renderbuffer; but these are limited as they cannot be read back later. This extension adds depth textures, which allow the user to read back depth output from a render, enabling techniques such as shadow mapping to occur without resorting to storing the depth in a colour texture.

Note

This functionality is core to OpenGL ES 3.0.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_depth_texture.txt

Example

```
// Create a depth texture - generally texture data isn't uploaded for this - but it can be.
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, 1024, 1024, 0, GL_DEPTH_COMPONENT,
                GL_UNSIGNED_INT, NULL);
```

GL_OES_depth24

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x, 2.0

Description

This extension adds the sized internal format GL_DEPTH_COMPONENT24 to the list of internal formats accepted by the function "glRenderBufferStorage". This format is a single channel data format with 24-bits per channel, used to represent the depth values stored in a z-buffer.

Note

This functionality is core to OpenGL ES 3.0.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_depth24.txt

Example

```
// Create a 24-bit depth renderbuffer
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT24_OES, 1024, 1024);
```

GL_OES_draw_buffers_indexed

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 2.0

Description

This extension builds upon the EXT_draw_buffers extension. In EXT_draw_buffers (part of OpenGL ES 3.0), separate values could be written to each colour buffer, but the blend enable, blend functions, blend equations and colour write masks are global and apply to all colour outputs.

This extension provides the ability to independently:

- enable or disable blending,
- set the blend equations,
- set the blend functions, and
- set the colour write masks

per colour output.

This extension introduces indexed versions of the enable, blend equation, blend function, and colour mask commands, as well as associated indexed queries in order to control and query these states independently on a per-colour output basis.

Note

This functionality is core to OpenGL ES 3.0, so the extension is no longer needed.

Registry Link

https://www.khronos.org/registry/gles/extensions/OES/OES_draw_buffers_indexed.txt

Example

```
EnableiOES(BLEND, 0);
DisableiOES(BLEND, 0);
IsEnablediOES(BLEND, 0);
BlendEquationiOES(0, FUNC_ADD);
BlendEquationSeparateiOES(0, FUNC_ADD, FUNC_SUBTRACT);
BlendFunciOES(0, ZERO, ONE);
BlendFuncSeparateiOES(0, SRC_COLOR, DST_COLOR, CONSTANT_ALPHA, SRC_ALPHA_SATURATE);
ColorMaskiOES(0, EGL_TRUE, EGL_TRUE, EGL_TRUE, EGL_TRUE);
```

GL_EXT_draw_elements_base_vertex

Supported Hardware
Series6, Series6XE, Series6XT

Valid APIs

Requires OpenGL ES 2.0

Description

This extension provides a method to specify a "base vertex offset" value which is effectively added to every vertex index that is transferred through DrawElements.

This mechanism can be used to decouple a set of indices from the actual vertex array that it is referencing. This is useful if an application stores multiple indexed models in a single vertex array. The same index array can be used to draw the model no matter where it ends up in a larger vertex array simply by changing the base vertex value. Without this functionality, it would be necessary to rebind all the vertex attributes every time geometry is switched and this can have larger performance penalty.

Registry Link
https://www.khronos.org/registry/gles/extensions/OES/OES_draw_elements_base_vertex.txt

Example

```
glDrawElementsBaseVertexOES(TRIANGLES, 6, UNSIGNED_BYTE, &indices, 100)
//[[ If OpenGL ES 3.0 is supported: ]]
glDrawRangeElementsBaseVertexOES(TRIANGLES, 0, 200,
                                 6, UNSIGNED_BYTE,
                                     &indices, 100);
glDrawElementsInstancedBaseVertexOES(TRIANGLES, 6,
                                          UNSIGNED_BYTE, &indices,
                                          10, 100);
//[[ If EXT_multi_draw_arrays is supported: ]]
glMultiDrawElementsBaseVertexOES(TRIANGLES,
                                       &count,
                                       UNSIGNED_BYTE,
                                       &indices,
                                       10,
                                       &basevertex);
```

GL_OES_draw_texture

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x

Description

This extension adds a simple method of blitting a texture to the screen, for things such as backgrounds and 2D GUI elements. The user can provide a crop rectangle to specify the part of the texture that they wish to draw, and then call DrawTex**OES() with coordinates for the part of the screen they wish to draw to.

Note

This functionality as specified cannot be made to work with the programmable pipeline of OpenGL ES 2.0. In OpenGL ES 3.0, this functionality is effectively replaced by the function "glBlitFramebuffer".

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_draw_texture.txt

Example

```
// Bind a texture
glBindTexture(GL_TEXTURE_2D, texture);
// Set the texture crop rectangle to set the texture coordinates to draw from
GLint textureCropRectangle[4] = {0, 0, 256, 256};
glTexParameteriv(GL_TEXTURE_2D, GL_TEXTURE_CROP_RECT_OES, textureCropRectangle);
// Draw the texture to the whole screen, which will be resized appropriately from the texture
// rectangle
glDrawTexfOES(-1.0, -1.0, 2.0, 2.0);
```

GL_OES_EGL_image

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x, 2.0, 3.x

Description

The concept of an EGLImage is introduced by EGL_KHR_image_base or EGL_KHR_image, and this extension enables an OpenGL ES implementation to bind EGLImages as if they were textures or renderbuffers, using the functions glEGLImageTargetTexture2DOES and glEGLImageTargetRenderbufferStorageOES, respectively. The format/type of the images must already be supported in OpenGL ES (or analogous to them) so that they work with any existing OpenGL ES queries. Image formats not supported in OpenGL ES natively are supported via the additional extension: GL_OES_EGL_image_external. Textures specified in this way can be sampled as textures or used as framebuffer attachments as if they were native objects.

It should be noted that if a user modifies the storage of the image by calling something like glGenerateMipMaps or glTexImage2D, then the driver will allocate new memory and copy relevant data across (if applicable). This will dissociate the GL

texture object from the original EGLImage, and the two will be completely separate and independent.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_EGL_image.txt

Example

```
// Create an EGL image via EGL_KHR_image_base. In this case from an EGL pixmap
// (EGL_KHR_image_pixmap), but the source is irrelevant to this example
EGLImageKHR eglImage = eglCreateImageKHR(eglDisplay, eglContext, EGL_NATIVE_PIXMAP_KHR,
 (EGLNativeBuffer)eglPixmap, NULL);
// Generate a texture object to map the EGLImage to, and bind it.
GLint texture;
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture);
// Map the EGLImage into an OpenGL ES texture using this extension.
glEGLImageTargetTexture2DOES(GL_TEXTURE_2D, (GLeglImageOES)eglImage);
// This extension now allows an application to attach this to a framebuffer
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, texture, 0);
```

GL_OES_EGL_image_external

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x, 2.0, 3.x

Description

This extension is similar to GL_OES_EGL_image, and allows formats not natively supported in the OpenGL ES API to be used as texture objects. Images bound in this way have restrictions to enable this, such as not being targetable by gl*Tex*Image*D functions, they can never be MIP mapped, and can only be sampled with the wrap mode GL_CLAMP_TO_EDGE. Images bound in this way use the texture target GL_TEXTURE_EXTERNAL_OES rather than GL_TEXTURE_2D etc., via the function glEGLImageTargetTexture2DOES. Sampling these textures will perform an implicit conversion from whatever format they are in to a linear RGB colour.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_EGL_image_external.txt

Example

```
// Create an EGL image via EGL_KHR_image_base. In this case from an EGL pixmap
 (EGL_KHR_image_pixmap), but the source is irrelevant to this example
EGLImageKHR eglImage = eglCreateImageKHR(eglDisplay, eglContext, EGL_NATIVE_PIXMAP_KHR,
 (EGLNativeBuffer)eglPixmap, NULL);
// Generate a texture object to map the EGLImage to, and bind it.
GLint texture;
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_EXTERNAL_OES, texture);
// Map the EGLImage into an OpenGL ES texture using this extension.
glEGLImageTargetTexture2DOES(GL_TEXTURE_EXTERNAL_OES, (GLeglImageOES)eglImage);
```

GL_OES_EGL_image_external_essl3

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 3.0 and ESSL 3.0.

Description

OES_EGL_image_external provides a mechanism for creating EGLImage texture targets from EGLImages, but only specified language interactions for the OpenGL ES Shading Language version 1.0. This extension adds support for versions 3.x of the OpenGL ES Shading Language.

Registry Link

https://www.khronos.org/registry/gles/extensions/OES/OES_EGL_image_external_essl3.txt

Example

```
#extension GL_OES_EGL_image_external_essl3 : enable
samplerExternalOES sampler;
vec4 color = texture(sampler, UV );
```

GL_OES_EGL_sync

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x, 2.0, 3.x

Description

This extension enables the use of EGLSync objects in an OpenGL ES command stream. EGLSync objects are defined in separate extensions, such as EGL_KHR_fence_sync.

Note

This functionality is superseded by GLSync objects in OpenGL ES 3.0, but is still useful for cross-API communication.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/EGL_KHR_fence_sync.txt

Example

```
// Disclaimer: None of this is good thread-safe code, it's pseudo-code for illustration. The
 exact CPU-side synchronisation mechanism will depend on your code.
// A fence sync handle accessible from both threads.
EGLSyncKHR g_eglFenceSync = 0;
// Pseudo-Function representing operations on a thread, in particular this thread is acting as
// a data producer.
void Thread1()
{
 // Render something
 glDraw(...);
 // Create a fence sync object in the command stream after the draw call, so a second
 // thread knows when the render operation is completed.
 g_eglFenceSync = eglCreateSyncKHR(eglDisplay, EGL_SYNC_FENCE_KHR, NULL);
 /*  Call eglClientWaitSync to flush the context, so that it's guaranteed to finish at
     some point. It only flushes the current context for the thread, hence calling it
     here - Flushing from the thread which actually needs to wait on this sync object
     would not flush the previous draw operations. By setting the timeout to 0 we've
     effectively called "glFlush" but a little more precisely, and it must work on all
     drivers.
 */
 EGLint waitResult = eglClientWaitSyncKHR(eglDisplay, eglFenceSync,
       EGL_SYNCH_FLUSH_COMMANDS_BIT_KHR, 0);
}
// Pseudo-Function representing operations on a second thread. This thread acts as a data
 consumer.
void Thread2()
{
 while(g_eglFenceSync == 0)
 {
  // Do some other work, the first thread isn't ready.
 }
 // Wait for the synchronisation object
 EGLint waitResult = eglClientWaitSyncKHR(eglDisplay, eglFenceSync,
       EGL_SYNCH_FLUSH_COMMANDS_BIT_KHR, EGL_FOREVER_KHR);
 // Destroy the fence sync object once we're done with it.
 EGLBoolean success = eglDestroySyncKHR(eglDisplay, eglFenceSync);
 // Now that the first thread's rendering has completed, do something with the result.
 // For example, composite the result of the draw into a windowing system.
 glDraw(...);
}
```

GL_OES_element_index_uint

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT (ES2/3 Only)

Valid APIs

OpenGL ES 1.x, 2.0

Description

This extension adds the ability to use an array of unsigned integers (GL_UNSIGNED_INT) as indices in glDrawElements - typically only unsigned short or unsigned byte types are supported. This is more costly than using byte or short in terms of bandwidth, but does allow a much greater range of data to be specified - allowing larger models to be sent through one draw call where they would otherwise have to be batched.

Note

This functionality is core to OpenGL ES 3.0

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_element_index_uint.txt

Example

```
// Draw with 32-bit, unsigned integer indices
glDrawElements(GL_TRIANGLES, 32, GL_UNSIGNED_INT, indices);
```

GL_OES_extended_matrix_palette

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x

Description

This extension extends GL_OES_matrix_palette, which specifies that a minimum of only 9 matrices are paletted, and that only 3 matrices can be blended per vertex. Based on developer feedback, this extension has been created to up the minimum palette size to 32, and now 4 matrices can be blended per vertex.

Note

This extension is part of the OpenGL ES 1.x Extension Pack Specification and is core functionality in OpenGL ES 2.0 and 3.0.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_extended_matrix_palette.txt

Example

```
// Bind the 32nd matrix as the current matrix, not normally allowed but defined by this
// extension.
glCurrentPaletteMatrixOES(31);
```

GL_OES_fixed_point

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x

Description

This extension provides fixed point functions for hardware that supports the OpenGL ES 1.x common profile but has inefficient floating-point support. The functions provided are equivalent to the Common-Lite profile and are denoted by ending in an "x" instead of an "f" for floating-point functions. Fixed point values are unsigned integers that are scaled to a S15.16 representation. The S15.16 representation means that there is one sign bit, the top 15 bits represent integer values, and the bottom 16 bits represent values below zero. In essence, it represents a signed short with 16 bits of sub-zero values.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_fixed_point.txt

Example

```
// Use the fixed-point version of a function to scale by 1.0, 0.5 and 2.0 in the x, y and z
// axes respectively. The value "65536" is effectively treated as 1.0.
GLfixed x = 65536;
GLfixed y = 65536 / 2;
GLfixed z = 65536 * 2;
glScalex(x, y, z);
```

GL_OES_fragment_precision_high

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 2.0

Description

This extension allows the use of high precision floating point values in fragment shaders. This affects GLSL ES 1.00 only, as it is a core feature in GLSL ES 3.00. High precision floating point allows a higher range of values to be used than would otherwise be provided by medium precision floating point, allowing more complex and accurate work to be done in the shader.

Note

This functionality is core to OpenGL ES 3.0.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_fragment_precision_high.txt

Example

```
// Declare a highp variable in a fragment shader, not allowed in Core OpenGL ES
varying highp vec4 someVariable;
```

GL_OES_framebuffer_object

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x

Description

This extension defines a simple interface for drawing to rendering destinations other than the buffers provided to the GL by the windowing-system. This allows things to be drawn independently of VSync and enables multi-pass algorithms that previously would have been limited to using CopyTexImage2D on the main framebuffer. Framebuffers allow you to attach colour, depth and stencil attachments to be used as drawing buffers, and then bind the framebuffer to be drawn into. Subsequent draw operations will then take place in this framebuffer, until the main framebuffer is rebound. Textures attached in this way can then be read back in subsequent drawing operations. This extension also introduces "Renderbuffers", which work similarly to a texture but are managed by the GPU entirely, and cannot be read back later. They are typically used for things like a depth buffer where the end result does not matter and will never need flushing.

Note

This extension is part of the OpenGL ES 1.x Extension Pack Specification and is core functionality in OpenGL ES 2.0 and 3.0.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_framebuffer_object.txt

Example

```
void Initialise()
{
 // Create a framebuffer object
 GLint framebufferObject;
 glGenFramebuffersOES(1, &framebufferObject);
 // Bind the framebuffer object to the current state
 glBindFramebufferOES(GL_FRAMEBUFFER_OES, framebufferObject)
 // Create a depth renderbuffer
 GLint renderBuffer;
 glGenRenderbuffersOES(1, &renderBuffer);
 // Bind and initialise it.
 glBindRenderbufferOES(GL_RENDERBUFFER_OES, renderBuffer);
 glRenderbufferStorageOES(GL_RENDERBUFFER_OES, GL_DEPTH_COMPONENT16_OES, 1024, 1024);
 // Attach the renderbuffer to the framebuffer
 glFramebufferRenderbufferOES(GL_FRAMEBUFFER_OES, GL_DEPTH_ATTACHMENT,
      GL_RENDERBUFFER_OES,  renderBuffer);
 // Create a colour texture
 GLint texture;
 glGenTextures(1, &texture);
 // Bind and initialise the texture
 glBindTexture(GL_TEXTURE_2D, texture);
 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 1024, 1024, 0, GL_RGBA,
       GL_UNSIGNED_BYTE, NULL);
 // Attach the texture to the framebuffer
```

```
 glFramebufferTexture2DOES(GL_FRAMEBUFFER_OES, GL_COLOUR_ATTACHMENT0_OES,
        GL_TEXTURE_2D, texture, 0);
 // Check framebuffer is complete
 if (glCheckFramebufferStatusOES(GL_FRAMEBUFFER_OES) != GL_FRAMEBUFFER_COMPLETE_OES)
 {
  // An error has occurred
 }
}
void Render()
{
 // Bind the framebuffer for rendering
 glBindFramebufferOES(GL_FRAMEBUFFER_OES, framebufferObject)
 // Draw something
 glDraw(...);
 // Bind the default (EGL provided) framebuffer to actually draw to the screen
 glBindFramebufferOES(GL_FRAMEBUFFER_OES, 0)
 // Draw something else, generally using the result of the framebuffer object's render.
 glDraw(...);
 // SwapBuffers
 eglSwapBuffers();
}
```

GL_OES_geometry_point_size

Supported Hardware

Series6XT

Valid APIs

OpenGL ES 3.1

Description

This extension adds programmable point size to the geometry shader and allows resizing of generated point sprites that come through as input - useful for things like particle effects.

Registry Link

https://www.khronos.org/registry/gles/extensions/OES/OES_geometry_shader.txt

Example

```
#extension GL_OES_geometry_shader : require
#extension GL_OES_geometry_point_size : require
#extension GL_OES_shader_io_blocks : require
// Take in point primitives.
layout(points) in;
// Output up to 1 line primitive - needs at least two vertices (one line).
layout(points, max_vertices = 1) out;
void main()
{
 // Re-emit the point primitive - at twice the size
 gl_Position = gl_in[0].gl_Position + vec4(0.0,-0.1, 0.0, 0.0);
 gl_PointSize = gl_in[0].gl_PointSize * 2.0f;
 EmitVertex();
 // End the (point) primitive
 EndPrimitive();
}
```

GL_OES_geometry_shader

Supported Hardware

Series6XT

Valid APIs

OpenGL ES 3.1

Description

Geometry Shaders are a new programmable pipeline step that sits in the current OpenGL ES pipeline directly after primitive assembly, and before clipping, culling etc. This stage allows access to the vertices in the primitive constructed by the earlier phases, in order to interpret them and re-emit new geometry. For example, the vertex shader and primitive assembly could output single point primitives, and the geometry shader could convert this into a set of triangles for further processing. Geometry shaders can be used for a form of tessellation, but this is considered superseded by OES_tessellation_shader. Geometry shaders can also discard primitives.

Note

Adjacency Primitives

As well as the standard GL primitives, this extension adds four new primitive types: LINES_ADJACENCY, LINE_STRIP_ADJACENCY, TRIANGLES_ADJACENCY and TRIANGLE_STRIP_ADJACENCY. These new types function the same as their non-adjacency counterparts everywhere except the geometry shader. In the geometry shader, these modes allow access to neighbouring vertices for each vertex. These neighbouring vertices can be useful as control points to bound any transformations done on input geometry.

Layered Rendering

For use cases such as stereoscopic (or other multi-view) rendering, it is often desirable to do some vertex shading only once, rather than repeating it over and over for each view. Geometry shader functionality described so far can allow this by using multiple invocations to transform geometry from different angles, but each view may need to be output to a separate render target. To handle this use case, this extension adds functionality called layered rendering, which allows framebuffers to be created with multiple layers. The geometry shader can then choose which layer its outputs will be sent to via the special gl_Layer output value.

Inputs

Geometry shader inputs come in the form of Shader IO blocks - including the built-in inputs. Built-in inputs match the built-in vertex shader outputs, but are passed through a built-in io block: gl_in. This is to distinguish them from output values.

Note: This block does not include point size unless OES_geometry_point_size is supported and enabled.

Each io block is actually an array of blocks, and each is either implicitly (or explicitly) sized to the number of vertices available for each primitive type (e.g. 1 for a point, 2 for a line, 3 for a triangle, etc.).

As well as the built-in block for vertex shader outputs, two additional values are provided:

- gl_PrimitiveIDIn, which is a counter that describes how many primitives have been processed by the shader during this render.
- gl_InvocationID, which indicates how many times this shader has been invoked using Multiple Invocations.

Outputs

Geometry shader outputs are the same as inputs, except for the built-in values, which are not part of an io block - instead they are free variables in the same way as they are in a vertex shader. gl_PrimitiveID is also provided as a modifiable output value that is passed on towards the fragment shader.

Registry Link

https://www.khronos.org/registry/gles/extensions/OES/OES_geometry_shader.txt

Example

```
// Create a Geometry Shader
GLuint geometryShader = glCreateShader(GL_GEOMETRY_SHADER_OES);

#extension GL_OES_geometry_shader : require
#extension GL_OES_shader_io_blocks : require

// Take in point primitives.
layout(points) in;
// Output up to 1 line primitive - needs at least two vertices (one line).
layout(lines, max_vertices = 2) out;
void main()
{
 // Emit the first vertex for the output primitive - below the original point
 gl_Position = gl_in[0].gl_Position + vec4(0.0,-0.1, 0.0, 0.0);
 EmitVertex();
 // Emit the second vertex for the output primitive - above the original point
 gl_Position = gl_in[0].gl_Position + vec4( 0.0, 0.1, 0.0, 0.0);
 EmitVertex();
 // End the (line) primitive
 EndPrimitive();
}
```

GL_OES_get_program_binary

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 2.0

Description

One of the larger costs of using shaders is that when initialising your application, they need to be compiled by an implementation's default compiler. Mechanisms exist to get the compiled shader binary back from the API after compilation, but there is still a linking stage after this which needs to be performed to create a program. This extension allows users to retrieve the fully compiled and linked program binary from the API, and then load it in on subsequent runs. Using this extension can save additional time on application start up compared to shader binaries as it also includes the attribute location binding and linking stages. It's worth noting that this extension does not itself define any binary formats, this is up to each vendor. For PowerVR hardware, these formats are defined by GL_IMG_program_binary.

Note

This functionality is core to OpenGL ES 3.0.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_get_program_binary.txt

Example

```
// Create a shader program object
GLuint shaderProgram = glCreateProgram();
// If there isn't already a binary, create a program
if (!binaryExists)
{
 // Create shaders, compile them, attach them and link them as per normal, then...
 // Retrieve the program binary size
 GLint length=0;
 glGetProgramiv(shaderProgram, GL_PROGRAM_BINARY_LENGTH, &length);
 // Pointer to the binary shader program in memory, needs to be allocated with the
 // right size.
 GLvoid* programBinaryData = (GLvoid*)malloc(length);
 // The format that the binary is retrieved in.
 GLenum binaryFormat=0;
 // Error checking variable - this should be greater than 0 after
 // glGetProgramBinaryOES, otherwise there was an error.
 GLsizei lengthWritten=0;
 glGetProgramBinaryOES(shaderProgram, length, &lengthWritten,
      &binaryFormat, programBinaryData);
}
else
{
 // Get the binary data, how much data there is, and what format it's in from whatever
 // cache it is stored in (e.g. a file)
 // ...
 // Instead of creating, compiling, attaching and linking shaders, upload a binary
 // program data from a program that previously underwent all of these stages.
 glProgramBinaryOES(shaderProgram, binaryFormat, programBinaryData,
 programBinaryDataLength);
}
```

GL_OES_gpu_shader5

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 3.10

Description

This extension provides a set of new features to the OpenGL ES Shading Language and related APIs to support capabilities of new GPUs, extending the capabilities of version 3.10 of the OpenGL ES Shading Language. Shaders using the new functionality provided by this extension should enable this functionality via the construct:

#extension GL_OES_gpu_shader5 : require (or enable)

This extension provides a variety of new features for all shader types, including:

- support for indexing into arrays of opaque types (samplers, and atomic counters) using dynamically uniform integer expressions;
- support for indexing into arrays of images and shader storage blocks using only constant integral expressions;
- extending the uniform block capability to allow shaders to index into an array of uniform blocks;
- a "precise" qualifier allowing computations to be carried out exactly as specified in the shader source to avoid optimization-induced invariance issues (which might cause cracking in tessellation);
- new built-in functions supporting:
- fused floating-point multiply-add operations;
- extending the textureGather() built-in functions provided by OpenGL ES Shading Language 3.10:
- allowing shaders to use arbitrary offsets computed at run-time to select a 2x2 footprint to gather from; and
- allowing shaders to use separate independent offsets for each of the four texels returned, instead of requiring a fixed 2x2 footprint.

Note

This specification is written against the OpenGL ES 3.1 (March 17, 2014) and OpenGL ES 3.10 Shading Language (March 17, 2014) Specifications.

This extension interacts with OES_geometry_shader.

Registry Link

https://www.khronos.org/registry/gles/extensions/OES/OES_gpu_shader5.txt

Example

```
#extension GL_OES_gpu_shader5 : require
in vec4 a, b, c, d;
precise out vec4 v;
float func(float e, float f, float g, float h)
{
```

```
 return (e*f) + (g*h);            // no special precision
}
float func2(float e, float f, float g, float h)
{
 precise result = (e*f) + (g*h);  // ensures a precise return value
 return result;
}
float func3(float i, float j, precise out float k)
{
 k = i * i + j;                   // precise, due to <k> declaration
}
void main(void)
{
 vec4 r = vec3(a * b);           // precise, used to compute v.xyz
 vec4 s = vec3(c * d);           // precise, used to compute v.xyz
 v.xyz = r + s;                     // precise
 v.w = (a.w * b.w) + (c.w * d.w);   // precise
 v.x = func(a.x, b.x, c.x, d.x);    // values computed in func()
                                    // are NOT precise
 v.x = func2(a.x, b.x, c.x, d.x);   // precise!
 func3(a.x * b.x, c.x * d.x, v.x);  // precise!
}
```

GL_OES_mapbuffer

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x, 2.0

Description

This extension enables the user to upload data directly into memory that the driver provides for them, by mapping the storage of a buffer object into client address space. In many cases this means that the user can avoid an additional memory allocation on top of one performed by the driver.

Note

This functionality is core to OpenGL ES 3.0 in the form of glMapBufferRange

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_mapbuffer.txt

Example

```
// Bind a buffer
glBindBuffer(GL_ARRAY_BUFFER, aVertexBuffer);
// Map the buffer's data to a CPU addressable pointer, in a way that allows us to write to it,
// but not read the data back.
void* bufferData = glMapBufferOES(GL_ARRAY_BUFFER, GL_WRITE_ONLY_OES);
// Get the size of the buffer
GLint bufferSize = 0;
glGetBufferParameteriv(GL_ARRAY_BUFFER, GL_BUFFER_SIZE, &bufferSize);
// Write some data into the pointer
memcpy(bufferData, newDataForTheBuffer, bufferSize);
// Unmap the pointer
glUnmapBufferOES(GL_ARRAY_BUFFER);
// The pointer is now invalid, so NULL it
bufferData = NULL;
```

GL_OES_matrix_get

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x

Description

This extension is specifically targeted at platforms where only fixed-point functionality is supported on the CPU, but data may be represented internally to a GPU as floating-point. In these cases, an application developer is only able to query floating-point matrices via glGetFixedv, which will result in a loss of information. To work around this, this extension allows floating point matrices to be queried back from the hardware by packing each floating-point value into an integer instead, using the IEEE 754 floating point representation. This value can then be queried using glGetIntegerv, using a number of new "float as int" tokens for affected matrix types.

Note

This functionality is core to OpenGL ES 2.0 and 3.0.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_matrix_get.txt

Example

```
// Get the values of the projection matrix that was previously loaded into OpenGL ES, but with
// the floating point bits stored as integers (for systems that don't support floats)
GLint projectionMatrix[16];
glGetIntegerv(GL_MODELVIEW_MATRIX_FLOAT_AS_INT_BITS_OES, projectionMatrix);
```

GL_OES_matrix_palette

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x

Description

Rather than using a fixed matrix for an entire model, allowing a mixture of different matrices with appropriate weightings to be used in a single piece of geometry can allow greater flexibility in a renderer. In particular this enables better control of

skeletal animation. This extension adds a palette of matrices to be specified, with indices passed in an array to specify which matrix to use for each vertex, and an array of weights to control how much it affects the vertex.

Note

This extension is effectively part of the OpenGL ES 1.x Extension Pack Specification (required for GL_OES_extended_matrix_palette) and is core functionality in OpenGL ES 2.0 and 3.0.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_matrix_palette.txt

Example

```
// Set the matrix palette stack as the current matrix mode
glMatrixMode(GL_MATRIX_PALETTE_OES);
// Bind the 9th matrix as the current matrix - the maximum guaranteed by this extension
glCurrentPaletteMatrixOES(8);
// Load a matrix into the palette
glLoadMatrix(someTranslationMatrix);
// Use translation functions as with any other matrix
glTranslatef(0.0f, 10.0f, 0.0f);
```

GL_OES_packed_depth_stencil

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT (ES2/3 Only)

Valid APIs

OpenGL ES 1.x, 2.0

Description

Typical use of a depth buffer can generally be satisfied with a 24-bit buffer, but this is generally unaligned with respect to power-of-two boundaries. Having 8-bits of stencil data is typically more than enough for most applications, so it is a natural fit to interleave the two buffer types. By doing so, the overall buffer can be aligned to power-of-two boundaries for speed, without wasting any memory.

Note

This functionality is core to OpenGL ES 3.0.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_packed_depth_stencil.txt

Example

```
// Create a packed depth and stencil texture format, normally you would not upload data here
```

```
// for this
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_STENCIL_OES, 1024, 1024, 0, GL_DEPTH_STENCIL_OES,
GL_UNSIGNED_INT_24_8_OES, NULL);
```

GL_OES_point_size_array

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x

Description

Points are usually rendered at a fixed size by OpenGL ES, according to the value set by glPointSize. This size is applied uniformly to all rendered points, which limits their capabilities. This extension adds the ability to use an array of values instead, so that each point can have different sizes.

Note

This functionality is core to OpenGL ES 2.0 and 3.0.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_point_size_array.txt

Example

```
// Setup an array of points to be rendered - three in this case
GLfloat pointPositions[] =
{
  0.4f, 0.7f, 0.2f,
  0.1f,-0.5f, 0.2f,
 -0.0f, 0.7f, 0.2f
};
// Set the vertex pointer as normal
glVertexPointer(3, GL_FLOAT, 0, pointPositions);
// Enable the vertex array
glEnableClientState(GL_VERTEX_ARRAY);
// Create an array of sizes for these points
GLfloat pointSizes[] =
{
 1.0f,
 2.0f,
 0.7f
};
// Set the point sizes via this extension
glPointSizePointerOES(3, GL_FLOAT, 0, pointPositions);
// Enable the point size array
glEnableClientState(GL_POINT_SIZE_ARRAY_OES);
// Draw the points
glDraw(GL_POINTS, ...);
```

GL_OES_point_sprite

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x

Description

This extension extends the standard functionality of OpenGL points to allow more flexible rendering than otherwise provided. For a core point object, OpenGL ES specifies that texture coordinates are identical across the entire body of the point, and it is then anti-aliased to fade out. To work around this, developers often use a quad with an alpha blended/tested texture to allow the full range of a texture to be expressed. When in use, this extension disables the anti-aliasing on the point, and instead interpolates the texture coordinates provided over the entire body of the point, allowing a point to be effectively textured to represent a sprite.

Note

This functionality is core to OpenGL ES 2.0 and 3.0.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_point_sprite.txt

Example

```
// Enable point sprite drawing
glEnable(GL_POINT_SPRITE_OES);
// Set the texture environment to use point sprite coordinate replace to true, so that texture
// coordinates are interpolated automatically across the rendered point, rather than using a
// single coordinate.
glTexEnvi(GL_POINT_SPRITE_OES, GL_COORD_REPLACE_OES, GL_TRUE);
// Draw some point sprites at the locations specified in
glDraw(GL_POINTS, ...);
```

GL_OES_query_matrix

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x

Description

This extension allows a user to query a matrix in a type independent of the underlying implemented value, by returning the mantissa and exponent of each matrix value separately. This solves a similar issue to OES_matrix_get, but in a slightly more flexible way. The mantissa value is returned as an S15.16 fixed point value, and the exponents are returned as integers. It works similarly to the C function "modf". This extension relies on either the Common-Lite profile or the GL_OES_fixed_point extension to provide the definition of a fixed-point value.

Note

This functionality is core to OpenGL ES 2.0 and 3.0.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_query_matrix.txt

Example

```
// Get the project matrix back as separate exponents and mantissas
// Set the projection matrix stack as the current matrix mode
glMatrixMode(GL_PROJECTION);
// Get the exponents and mantissa values of the projection matrix that was previously loaded
// into OpenGL ES
GLfixed projectionMatrixMantissas[16];
GLint projectionMatrixExponents[16];
glQueryMatrixxOES(projectionMatrixMantissas, projectionMatrixExponents);
```

GL_OES_read_format

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x

Description

Reading pixels from a framebuffer in OpenGL ES is performed via the function glReadPixels(). This function is able to read data out in only one format for OpenGL ES 1.x; RGBA8888. This extension adds another, implementation defined format, which can be queried with glGetIntegerv(). These additional formats are limited to being RGBA, RGB or Alpha-Only formats.

Note

This functionality is core to OpenGL ES 2.0 and 3.0.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_read_format.txt

Example

```
// Find out the preferred format and type to read back pixels on a platform
GLint preferredFormat;
GLint preferredType;
glGetIntegerv(GL_IMPLEMENTATION_COLOR_READ_FORMAT_OES, &preferredFormat);
glGetIntegerv(GL_IMPLEMENTATION_COLOR_READ_FORMAT_OES, &preferredType);
// Figure out how big the texture is going to be.
GLuint pixelReadDataSize = 0;
switch(preferredFormat)
{
case GL_RGBA:
 {
  switch(preferredType)
```

```
   {
   case GL_UNSIGNED_BYTE:
    pixelReadDataSize = readWidth * readHeight * 4;
    break;
   case GL_UNSIGNED_SHORT_4_4_4_4:
   case GL_UNSIGNED_SHORT_5_5_5_1:
    pixelReadDataSize = readWidth * readHeight * 2;
    break;
   }
   break;
  }
 case GL_RGB:
  {
   switch(preferredType)
   {
   case GL_UNSIGNED_BYTE:
    pixelReadDataSize = readWidth * readHeight * 3;
    break;
   case GL_UNSIGNED_SHORT_5_6_5:
    pixelReadDataSize = readWidth * readHeight * 2;
    break;
   }
   break;
  }
 case GL_LUMINANCE_ALPHA:
  {
   pixelReadDataSize = readWidth * readHeight * 2;
   break;
  }
 case GL_LUMINANCE:
  {
   pixelReadDataSize = readWidth * readHeight;
   break;
  }
 case GL_ALPHA:
  {
   pixelReadDataSize = readWidth * readHeight;
   break;
  }
 }
 // Allocate enough space to read the data
 GLvoid* pixelReadData = malloc(pixelReadDataSize);
 // Specify the pixel unpack operation to be tightly packed (no padding) when reading back
 // data, otherwise this needs to be handled when working out the size
 glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
 // Read pixels using these formats
 if (pixelReadData)
 {
  glReadPixels(readOriginX, readOriginY, readWidth, readHeight, preferredFormat, preferredType,
  pixelReadData);
 }
```

GL_OES_required_internalformat

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x, 2.0

Description

By default, OpenGL ES implementations are free to store texture data as they see fit when using unsized types such as "GL_RGBA". This extension adds a number of "sized" internal formats to the specification which mandate a minimum storage precision for these data types. More precision may be used internally, but it can

never store data at a lower precision than requested which would otherwise cause a loss of information.

Note

This functionality is core to OpenGL ES 3.0.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_required_internalformat.txt

Example

```
// Create a texture with RGBA8 so that it uses no less than 8 bits per pixel
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8_OES, 1024, 1024, 0, GL_RGBA, GL_UNSIGNED_BYTE,
 pixelData);
```

GL_OES_rgb8_rgba8

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x, 2.0

Description

This extension adds the sized internal formats RGB8 and RGBA8 to the list of internal formats accepted by the function "glRenderBufferStorage". These formats represent 3 and 4 channel data formats with 8-bits per channel, used to represent colour data.

Note

This functionality is core to OpenGL ES 3.0.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_rgb8_rgba8.txt

Example

```
// Create a renderbuffer with RGBA8 so that it uses no less than 8 bits per pixel
glRenderbufferStorage(GL_RENDERBUFFER, GL_RGBA8_OES, 1024, 1024);
```

GL_OES_sample_shading

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 3.x

Description

This extension is one of three extensions which enhances OpenGL ES's programmability with respect to how multisampling is performed:

- GL_OES_sample_variables
- GL_OES_sample_shading
- GL_OES_shader_multisample_interpolation

GL_OES_sample_shading adds the ability to force an implementation to process a minimum number of unique samples per-pixel, as a percentage of the total number of samples.

Example

```
// Enable minimum sample shading
glEnable(GL_SAMPLE_SHADING_OES);
// Set the minimum number of samples to half the total number of samples per-pixel in the
 framebuffer
// E.g. if the framebuffer has 4xMSAA enabled, then a value of 0.5 will mean at least 2 (0.5 x 4
 = 2) samples must be processed.
glMinSampleShading(0.5);
```

GL_OES_sample_variables

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 3.x

Description

This extension is one of three extensions which enhances OpenGL ES's programmability with respect to how multisampling is performed:

- GL_OES_sample_variables
- GL_OES_sample_shading
- GL_OES_shader_multisample_interpolation

GL_OES_sample_variables is required by the other two extensions, and adds a number of variables accessible to a fragment shader that allow control over each individual sample that contributes to the current fragment:

```
    in lowp int gl_SampleID*  // The ID of the current sample being shaded.
    in mediump vec2 gl_SamplePosition* // The sub-pixel position of the sample, in the range
 [0,1], relative to the bottom left of the pixel.
    in highp int gl_SampleMaskIn[(gl_MaxSamples+31)/32] // The sample mask generated for the
 current fragment by the fixed function pipeline.
```

```
    out highp int gl_SampleMask[(gl_MaxSamples+31)/32]** // Output sample mask, allowing the
 shader to generate a different mask, programmably.
```

* Any use of these variables will force the fragment shader to run at sample rate, increasing the amount of fragment work.

** gl_SampleMask will be filled in with the value of gl_SampleMaskIn if it is never written to. If any execution path might write to it, all paths must write to it, or the value will be undefined.

Example

```
#extension GL_OES_sample_variables : require
void main()
{
 ...
 // Safely invert the sample mask
 for (uint i = 0; i < ((gl_MaxSamples+31)/32); ++i)
 {
  gl_SampleMask[i] == !gl_SampleMaskIn[i];
 }
 ...
}
```

GL_OES_shader_image_atomic

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 3.x

Description

This extension adds GLSL built-ins that provide guaranteed atomic read/write access to images when called. Image atomic operations perform a guaranteed read/modify/write operation to a given pixel in an image without overlapping with any other shader performing another atomic operation at the same time (causing a race condition). These operations can only work on integer or unsigned integer textures. They are all labelled as imageAtomic*(), and perform the same types of atomic operations added by ES 3.1 and in many Operating Systems on the CPU:

- imageAtomicAdd
- imageAtomicMin
- imageAtomicMax
- imageAtomicAnd
- imageAtomicOr
- imageAtomicXor
- imageAtomicExchange
- imageAtomicCompSwap

These are identical to the already available atomic operations for buffer objects, but function on images instead. These functions only operate on integer or unsigned integer images.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_shader_image_atomic.txt

Example

```
layout(binding = 0) coherent uimage2D exampleImage;
void main()
{
 ...
 // Increment the value at the current fragment coordinate in the example image.
 imageAtomicAdd(exampleImage, gl_FragCoord.xy, 1);
 ...
}
```

GL_OES_shader_io_blocks

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 3.0, 3.1

Description

This extension extends the functionality of interface blocks to support input and output interfaces in the OpenGL ES Shading Language.

Input and output interface blocks are used for forming the interfaces between vertex, tessellation control, tessellation evaluation, geometry and fragment shaders. This accommodates passing arrays between stages, which otherwise would require multi-dimensional array support for tessellation control outputs and for tessellation control, tessellation evaluation, and geometry shader inputs.

This extension provides support for application defined interface blocks which are used for passing application-specific information between shader stages.

This extension moves the built-in "per-vertex" in/out variables to a new built-in gl_PerVertex block. This is necessary for tessellation and geometry shaders which require separate instance for each vertex, but it can also be useful for vertex shaders.

Finally, this extension allows the redeclaration of the gl_PerVertex block in order to reduce the set of variables that must be passed between shaders.

Registry Link

https://www.khronos.org/registry/gles/extensions/OES/OES_shader_io_blocks.txt

Example

```
// Vertex Shader
#extension GL_EXT_shader_io_blocks : require
layout(location = 0) out SurfaceData
{
 highp vec3 normal;
 highp vec3 tangent;
} surfaceData;
// Fragment Shader
#extension GL_EXT_shader_io_blocks : require
layout(location = 0) in SurfaceData
{
 highp vec3 normal;
 highp vec3 tangent;
} surfaceData;
```

GL_OES_shader_multisample_interpolation

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 3.x

Description

This extension is one of three extensions which enhances OpenGL ES's programmability with respect to how multisampling is performed:

- GL_OES_sample_variables
- GL_OES_sample_shading
- GL_OES_shader_multisample_interpolation

GL_OES_shader_multisample_interpolation adds functions to GLSL that allow fragment inputs to be interpolated at locations other than just the centroid, such as specific sample locations, or at a given offset (within an implementation-defined range).

Example

```

#extension GL_OES_sample_variables : require
#extension GL_OES_shader_multisample_interpolation : require
// A sample-rate input. Operations using this must be calculated separately for each sample.
// It is automatically interpolated at each sample location, rather than the centroid.
sample in highp vec2 sampleTextureCoords;
// A standard input
in highp vec4 normal;
void main()
{
 ...
 // Get the normals for each sample location - as if it were declared with 'sample'
 highp vec4 sampleNormal = interpolateAtSample(normal, gl_SampleID);
 ...
}
```

GL_OES_single_precision

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x

Description

This function is basically the equivalent of the GL_OES_fixed_point, but for floating point functions. This extension doesn't actually express anything, as these functions exist in the common profile anyway. Theoretically it could have made some sort of sense in a Common-Lite Profile (fixed point only), but this never happened in practice.

Note

This functionality is core to OpenGL ES 2.0 and 3.0.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_single_precision.txt

Example

```
// Use a floating-point function
GLfloat rgbaBlendValues[4] = {0.0f, 0.0f, 0.0f, 0.0f};
glGetFloatv(GL_BLEND_COLOR, rgbaBlendValues);
```

GL_OES_standard_derivatives

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 2.0

Description

Standard derivative functions are optionally available in the GL shading language to give an approximate delta to the values in neighbouring fragments in the x or y directions. The function will evaluate the local difference for any value passed to it. It uses dFdx() to return the difference in the x direction, and dFdy() to return the difference in the y direction.

For example, if you were to call 'dFdx(gl_FragCoord.x)', you'd get a result of 1.0, as the neighbouring fragment will have an x coordinate exactly one position over.

'dFdx(gl_FragCoord.y)' on the other hand would return 0.0, as the y coordinate is static as you move left or right.

Typical use of this functionality is to estimate the filter width used for anti-aliasing procedural textures. To facilitate this use case, a third function is provided: fwidth(), which returns the sum of the absolute difference in x and y (e.g. abs(dFdx(val)) + abs(dFdy(val))).

Note

This functionality is core to OpenGL ES 3.0.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_standard_derivatives.txt

Example

```
// Get the texture colour for a given fragment
lowp vec4 colourHere = texture2D(sTexture, TexCoord.xy);
// Get the value that was used in a neighbouring fragment in the x direction
lowp vec4 colourNextDoor = dFdx(colourHere) + colourHere;
// Get the value that was used in a neighbouring fragment in the y direction
lowp vec4 colourNextDoorVertically = dFdy(colourHere) + colourHere;
```

GL_OES_stencil_wrap

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x

Description

Core OpenGL ES 1.0 stencil buffers require that when a value is incremented to the maximum value or decreased to the minimum, further operations will clamp that value. However, a number of algorithms use this buffer as a counter, calculating the difference between the total number of increments and the total number of decrements. This extension adds two new stencil operations: INCR_WRAP and DECR_WRAP. These allow the value to wrap round when the value changes and give more flexibility to the stencil buffer.

Note

This extension is part of the OpenGL ES 1.x Extension Pack Specification and is core functionality in OpenGL ES 2.0 and 3.0.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_stencil_wrap.txt

Example

```
// Change the stencil operations to decrease and wrap when failing the depth or stencil test,
// and increase and wrap when succeeding
glStencilOp(GL_DECR_WRAP, GL_DECR_WRAP, GL_INCR_WRAP);
```

GL_OES_stencil8

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x

Description

Despite OpenGL ES 1.x supporting Depth and Stencil testing in core, the specification doesn't define any stencil buffer representations, and does not mandate any stencil buffer support. This extension exposes an 8-bit stencil buffer for stencil operations.

Note

This functionality is core to OpenGL ES 2.0 and 3.0.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_stencil8.txt

Example

```
// Create a 1024x1024, 8-bit stencil renderbuffer
glRenderbufferStorage(GL_RENDERBUFFER, GL_STENCIL_INDEX8_OES, 1024, 1024);
```

GL_OES_surfaceless_context

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x, 2.0, 3.x

Description

Applications that want to render to framebuffer objects only, without touching any sort of main framebuffer, currently still need to create an EGLSurface for a context. EGL_KHR_surfaceless_context adds the ability to create a context without a surface, but vanilla OpenGL ES cannot function correctly with such a context. This extension

provides mechanisms whereby this behaviour is defined so that there is no need to create a default surface.

A single new feature is added in the form of an enum, GL_FRAMEBUFFER_UNDEFINED_OES, which is returned when what would otherwise be the default framebuffer is bound. Any commands relying on the framebuffer will behave as if an incomplete framebuffer, of size 0 by 0, is currently bound.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_surfaceless_context.txt

Example

```
// Query the current framebuffer status
GLenum status = glCheckFramebufferStatus(GL_FRAMEBUFFER);
if (status == GL_FRAMEBUFFER_UNDEFINED_OES)
{
 // There's no framebuffer bound!
}
```

GL_OES_tessellation_point_size

Supported Hardware

Series6XT

Valid APIs

OpenGL ES 3.1

Description

This extension adds programmable point size to the tessellation shaders and allows resizing of generated point sprites that come through as input - useful for things like particle effects.

Registry Link

https://www.khronos.org/registry/gles/extensions/OES/OES_tessellation_shader.txt

Example

```
// Tessellation control shader
#extension GL_OES_tessellation_shader : require
#extension GL_OES_tessellation_point_size : require
#extension GL_OES_shader_io_blocks : require
layout(vertices = 1) out;
void main(void)
{
 gl_TessLevelOuter[0] = 2.0;
 gl_TessLevelOuter[1] = 4.0;
 gl_TessLevelOuter[2] = 6.0;
 gl_TessLevelOuter[3] = 8.0;
 gl_TessLevelInner[0] = 8.0;
 gl_TessLevelInner[1] = 8.0;
 gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
 gl_out[gl_InvocationID].gl_PointSize = gl_in[gl_InvocationID].gl_PointSize * 1.1;
}
```

GL_OES_tessellation_shader

Supported Hardware

Series6XT

Valid APIs

OpenGL ES 3.1

Description

This extension adds new tessellation stages to the OpenGL ES pipeline, and two new shader types - tessellation control and tessellation evaluation. When tessellation is active, a number of pipeline stages are affected. Tessellation operates after primitive assembly and before geometry shading (if supported).

Patch Primitives

Tessellation operates on a new primitive type - patches, which are a collection of vertices that are combined to form a patch. The actual number of vertices is application defined, allowing it to be fairly general-purpose.

Tessellation Control

The first step of tessellation is the control shader. This shader exists to allow further transformation of vertices and set the level of detail required by the fixed function tessellation stage. It may also add or subtract vertices from a patch before passing it on - though it cannot change the number of patches.

Fixed Function Tessellation

The fixed function stage turns patches into a number of primitives, which are then available to the tessellation evaluation shader. The algorithm is fairly flexible and has a number of determining factors depending on the control shader's outputs. Full details of this are available in the specification.

Note: This stage is also affected by the type of primitive specified in the tessellation evaluation shader.

Tessellation Evaluation

After the fixed function tessellation stage, this shader consumes those generated primitives, doing final evaluation of the vertices before passing them on to Geometry Shading. The values output by the fixed function tessellation are usually abstract and not suitable for rasterization directly, so are modified by this shader stage into a more final output, typically involving interpolation.

Registry Link

https://www.khronos.org/registry/gles/extensions/OES/OES_tessellation_shader.txt

Example

```
// Create a Tessellation Control Shader
GLuint geometryShader = glCreateShader(GL_TESS_CONTROL_SHADER_OES);
// Create a Tessellation Control Shader
GLuint geometryShader = glCreateShader(GL_TESS_EVALUATION_SHADER_OES);
```

Example

```
// Tessellation control shader
#extension GL_OES_tessellation_shader : require
#extension GL_OES_shader_io_blocks : require
layout(vertices = 3) out;
void main(void)
{
 gl_TessLevelOuter[0] = 2.0;
 gl_TessLevelOuter[1] = 4.0;
 gl_TessLevelOuter[2] = 6.0;
 gl_TessLevelOuter[3] = 8.0;
 gl_TessLevelInner[0] = 8.0;
 gl_TessLevelInner[1] = 8.0;
 gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
}
// Tessellation evaluation shader
#extension GL_OES_tessellation_shader : require
#extension GL_OES_shader_io_blocks : require
layout(triangles, equal_spacing, ccw) in;
vec3 interpolate(vec3 v0, vec3 v1, vec3 v2)
{
    return (gl_TessCoord.x * v0) + (gl_TessCoord.y * v1) + (gl_TessCoord.z * v2);
}
void main(void)
{
 gl_Position = interpolate(gl_in[0].gl_Position, gl_in[1].gl_Position, gl_in[2].gl_Position);
}
```

GL_OES_texture_border_clamp

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 2.0, OpenGL ES 3.x

Description

When sampling from a texture, if the requested UV coordinates reference a place outside of the texture, a wrapping behaviour is needed to specify what happens. OpenGL ES has three strategies depending on version:

- OpenGL ES 2.0 allows either clamping the UVs to the edge of the texture or wrapping around to the start of the valid range.
- OpenGL ES 3.0 added mirrored repeat, which acts as if the texture was flipped each time it wraps the coordinates.

This extension adds an additional behaviour - returning an application-defined border colour. This allows a shader to draw a texture to the screen with a dedicated border colour if so desired. It also gives the shader a cheap way to detect that a UV coordinate has gone out of range, without complex shader logic.

Registry Link

https://www.khronos.org/registry/gles/extensions/OES/OES_texture_border_clamp.txt

Example

```
// Set the texture's border colour to opaque red
//                        Red   Green Blue  Alpha
GLfloat borderColour[] = {1.0f, 0.0f, 0.0f, 1.0f};
glBindTexture(GL_TEXTURE_2D, texture)
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR_OES, borderColour);
// Set the texture to use the border clamp wrapping mode.
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER_OES);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER_OES);
```

GL_OES_texture_buffer

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 3.1

Description

This extension adds a way to effectively treat a buffer as a 1-dimensional texture, and sample from that buffer in a shader. This is done by attaching a buffer object as the data store for a texture, in place of the TexStorage* or TexImage* functions.

This allows applications to have access to a linear texture in memory which can be mapped into CPU space and the data can be manipulated more readily than typically opaque textures.

Registry Link

https://www.khronos.org/registry/gles/extensions/OES/OES_texture_buffer.txt

Example

```
// Bind a texture object
GLuint textureObject;
glBindTexture(GL_TEXTURE_2D, textureObject);
// Set a buffer as the bound texture object's data store, with RGBA8 data
GLuint bufferObject;
glTexBufferEXT(GL_TEXTURE_2D, GL_RGBA8, bufferobject);
[Example]
#extension GL_EXT_texture_buffer : require
uniform samplerBuffer myTextureBuffer;
in float textureBufferCoords;
void main()
{
 ...
 // Sample the texture buffer.
 vec4 value = texture(myTextureBuffer, textureBufferCoords);
 ...
}
```

GL_OES_texture_cube_map

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x

Description

The standard way to render a cube of textures, such as a skybox, is to render from six individual 2D textures, one for each face. This means that the render has to be split, however, so that each face can be rendered with its own texture. This extension adds a new texture layout scheme for efficient rendering of things like skyboxes, by storing the 6 faces of a cubemap in one texture object. As well as this, texture lookups are performed by using a vector that points at the cube, rather than using explicit x,y coordinates.

This extension also adds texture coordinate generation functions to automatically create texture coordinates based on the eye direction of the user, via one of two modes:

1. **Reflection map mode**: Generates the coordinates matching the eye-space reflection vector. This is used for standard skybox or other cubemapping.
2. **Normal map mode**: Generates the coordinates matching the vertex's transformed eye-space normal. This is often useful for more advanced techniques and diffuse lighting models.

Note

This extension is part of the OpenGL ES 1.x Extension Pack Specification and is core functionality in OpenGL ES 2.0 and 3.0.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_texture_cube_map.txt

Example

```
// Create a cubemap texture - have to specify all faces individually.
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X_OES, 0, GL_RGBA 1024, 0, GL_RGBA, GL_UNSIGNED_BYTE,
 pixelData[0]);
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Y_OES, 0, GL_RGBA 1024, 0, GL_RGBA, GL_UNSIGNED_BYTE,
 pixelData[1]);
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Z_OES, 0, GL_RGBA 1024, 0, GL_RGBA, GL_UNSIGNED_BYTE,
 pixelData[2]);
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_X_OES, 0, GL_RGBA 1024, 0, GL_RGBA, GL_UNSIGNED_BYTE,
 pixelData[3]);
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_Y_OES, 0, GL_RGBA 1024, 0, GL_RGBA, GL_UNSIGNED_BYTE,
 pixelData[4]);
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_Z_OES, 0, GL_RGBA 1024, 0, GL_RGBA, GL_UNSIGNED_BYTE,
 pixelData[5]);
```

GL_OES_texture_cube_map_array

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 3.1

Description

OpenGL ES 3.1 supports two-dimensional array textures. An array texture is an ordered set of images with the same size and format. Each image in an array texture has a unique level. This extension expands texture array support to include cube map textures.

A cube map array texture is a two-dimensional array texture that may contain many cube map layers. Each cube map layer is a unique cube map image set. Images in a cube map array have the same size and format limitations as two-dimensional array textures. A cube map array texture is specified using TexImage3D or TexStorage3D in a similar manner to two-dimensional arrays. Cube map array textures can be bound to a render targets of a frame buffer object just as two-dimensional arrays are, using FramebufferTextureLayer.

When accessed by a shader, a cube map array texture acts as a single unit. The "s", "t", "r" texture coordinates are treated as a regular cube map texture fetch. The "q" texture is treated as an unnormalized floating-point value identifying the layer of the cube map array texture. Cube map array texture lookups do not filter between layers.

Note

OpenGL ES 3.1 and OpenGL ES Shading Language 3.10 are required.

This specification is written against the OpenGL ES 3.1 (March 17, 2014) and OpenGL ES 3.10 Shading Language (March 17, 2014) Specifications.

OES_geometry_shader or EXT_geometry_shader is required.

OES_texture_border_clamp or EXT_texture_border_clamp affect the definition of this extension.

This extension interacts with OES_shader_image_atomic.

Registry Link

https://www.khronos.org/registry/gles/extensions/OES/OES_texture_cube_map_array.txt

Example

```
glGenTextures(1, &texid);
glBindTexture(TEXTURE_CUBE_MAP_ARRAY_OES, texid);
...
```

```
// In the shader the texture value is determined by (s, t, r, q) coordinates
// where "s", "t", "r" is defined to be the same as for TEXTURE_CUBE_MAP
// and "q" is defined as the index of a specific cube map in the cube map array
vec4 sampler_coord;
vec4 color = texture(samplerCubeArray, sampler_coord)
```

GL_OES_texture_env_crossbar

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x

Description

This extension extends the combining functionality in OpenGL ES 1 to allow application developers to choose textures from specific active texture units as the source, rather than just the current texture.

Note

This extension is part of the OpenGL ES 1.x Extension Pack Specification and is core functionality in OpenGL ES 2.0 and 3.0.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_texture_env_crossbar.txt

Example

```
// Set the first source alpha texture environment variable
// to sample from the third texture unit
glTexEnvi(GL_TEXTURE_2D, GL_SOURCE0_ALPHA, GL_TEXTURE2);
```

GL_OES_texture_float

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT (ES2/3 Only)

Valid APIs

OpenGL ES 1.x, 2.0

Description

This extension adds the ability to use 32-bit floating point type (GL_FLOAT) representations of texture data as a readable texture format when shading an object. This extension does not add linear sampling from this texture type however due to the high cost of such an operation, which is instead enabled by

GL_OES_texture_float_linear. This extension also does not allow these textures to be used as a colour attachment for a framebuffer.

The float data format allows greater range than an integer representation would and is typically used for High Dynamic Range (HDR) colour data.

Note

This functionality is core to OpenGL ES 3.0.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_texture_float.txt

Example

```
// Create a floating-point texture
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 1024, 1024, 0, GL_RGBA, GL_FLOAT, pixelData);
// Only nearest filtering is supported for these textures for this extension. Linear filtering
// is enabled by separate extensions.
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST_MIPMAP_NEAREST);
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
```

GL_OES_texture_half_float

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT (ES2/3 Only)

Valid APIs

OpenGL ES 1.x, 2.0

Description

This extension adds the ability to use 16-bit floating point type (GL_HALF_FLOAT) representations of texture data as a readable texture format when shading an object. This extension does not add linear sampling from this texture type however due to the high cost of such an operation, which is instead enabled by GL_OES_texture_half_float_linear. This extension also does not allow these textures to be used as a colour attachment for a framebuffer. The extension GL_EXT_color_buffer_half_float allows half float textures to be attached to a framebuffer.

The half-float data format consists of 1 sign bit, 5 exponent bits and 10 mantissa bits, and allows greater range than an integer representation would, whilst reducing the memory footprint to that of a short data type.

Note

This functionality is core to OpenGL ES 3.0.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_texture_float.txt

Example

```
// Create a floating-point texture
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 1024, 1024, 0, GL_RGBA, GL_HALF_FLOAT_OES, pixelData);
// Only nearest filtering is supported for these textures for this extension. Linear filtering
// is enabled by separate extensions.
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST_MIPMAP_NEAREST);
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
```

GL_OES_texture_mirrored_repeat

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x

Description

This extension adds another wrap mode to OpenGL ES, similar to GL_REPEAT, but the image is flipped each time it is repeated. For horizontal repeats, the texture is flipped horizontally, and for vertical repeats it is flipped vertically. The benefit of this is that it means tiled textures can be created without having to worry about making the edges match up.

Note

This extension is part of the OpenGL ES 1.x Extension Pack Specification and is core functionality in OpenGL ES 2.0 and 3.0.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_texture_mirrored_repeat.txt

Example

```
// Set the wrap mode for a texture to mirrored repeat
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_MIRRORED_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_MIRRORED_REPEAT);
```

GL_OES_texture_npot

Supported Hardware

Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x, 2.0

Description

This extension removes almost all of the limitations surrounding non-power of two textures in OpenGL ES 1.x and 2.0. MIP Map specification is now allowed, and minification filters that include a MIP Map filter will now work as expected for Power of Two textures. All available wrap modes will now work as well, rather than just GL_CLAMP. The only restriction that this extension does not lift is that in OpenGL ES 1.x, glGenerateMIPMaps will not work unless GL_OES_framebuffer_object is also supported.

Note

This functionality is core to OpenGL ES 3.0.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_texture_npot.txt

Example

```
// Upload a texture with dimensions of 15 by 47. Typically, this isn't supported without
// extension support.
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 15, 47, 0, GL_RGBA, GL_UNSIGNED_BYTE, pixelData);
// This extension lifts any restrictions on non-power of two textures, and all filter/wrap
// modes can be used.
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

GL_OES_texture_stencil8

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 3.1

Description

This extension provides support for a new type of immutable texture - two dimensional multisample array textures.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_texture_stencil8.txt

Example

```
// Create a 2D Multisample array texture
GLuint ms2dArrayTexture;
glGenTextures(1, &ms2dArrayTexture);
glBindTexture(GL_TEXTURE_2D_MULTISAMPLE_ARRAY_OES, ms2dArrayTexture);
// Allocate a 512x512 texture with 4 samples, 128 array layers, and RGBA data
glTexStorage3DMultisampleOES(GL_TEXTURE_2D_MULTISAMPLE_ARRAY_OES, 4, GL_RGBA8, 512, 512, 128,
```

```
                                    GL_FALSE);
```

GL_OES_texture_storage_multisample_2d_array

Supported Hardware

Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 3.1

Description

This extension provides support for a new type of immutable texture - two dimensional multisample array textures.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_texture_storage_multisample_2d_array.txt

Example

```
// Create a 2D Multisample array texture
GLuint ms2dArrayTexture;
glGenTextures(1, &ms2dArrayTexture);
glBindTexture(GL_TEXTURE_2D_MULTISAMPLE_ARRAY_OES, ms2dArrayTexture);
// Allocate a 512x512 texture with 4 samples, 128 array layers, and RGBA data
glTexStorage3DMultisampleOES(GL_TEXTURE_2D_MULTISAMPLE_ARRAY_OES, 4, GL_RGBA8, 512, 512, 128,
 GL_FALSE);
```

GL_OES_vertex_array_object

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT

Valid APIs

OpenGL ES 1.x, 2.0

Description

This extension provides a level of encapsulation for bound vertex state such as that set by, for example 'glVertexPointer'. With this extension, any modifications to the vertex array state will bind specifically to the bound VAO, rather than being bound directly to the context. Changing between bound VAOs will modify the vertex state to whatever was last set within the newly bound VAO. This allows users to quickly switch between various vertex states, allowing them to, for example, switch between different model objects with far fewer API calls than are traditionally needed. To maintain compatibility with the Core ES APIs, this extension employs a "default

VAO". What this means is that rather than the object name '0' being a special "nothing bound" name as in most OpenGL objects, it is instead a fully usable VAO.

More specifically, vertex array objects all the state set by the following methods:

```
ES1: gl*Pointer(), glEnableClientState(), glBindBuffer()*
ES2: glVertexAttribPointer(), glEnableVertexAttribArray(), glBindBuffer()*
*VAOs only store the bound GL_ELEMENT_ARRAY_BUFFER. Any binding to GL_ARRAY_BUFFER or other
buffer types are separate and distinct from VAOs.
```

Note

This functionality is core to OpenGL ES 3.0.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_vertex_array_object.txt

Example

```
// Generate a vertex array object
GLuint vertexArray;
glGenVertexArraysOES(1, &vertexArray);
// Bind the vertex array object
glBindVertexArrayOES(vertexArray);
// Set some vertex attribute pointers
glVertexAttribPointer(0, 4, GL_FLOAT, GL_TRUE, 0, dataPointer);
// Rebind the default VAO
glBindVertexArrayOES(0);
```

GL_OES_vertex_half_float

Supported Hardware

Series5, Series5XT, Series6, Series6XE, Series6XT (ES2/3 Only)

Valid APIs

OpenGL ES 1.x, 2.0

Description

This extension adds the ability to use 16-bit floating point (GL_HALF_FLOAT) representations of vertex data when uploading it to GL. The half-float data format consists of 1 sign bit, 5 exponent bits and 10 mantissa bits, and allows greater range than an integer representation would whilst reducing the memory footprint to that of a short data type.

Note

This functionality is core to OpenGL ES 3.0.

Registry Link

http://www.khronos.org/registry/gles/extensions/OES/OES_vertex_half_float.txt

Example

```
// Set a vertex attribute to sample vertices as half float values.
glVertexAttribPointer(0, 4, GL_HALF_FLOAT_OES, GL_TRUE, 0, dataPointer);
```

GL_OVR_multiview

Valid APIs

OpenGL ES 3.0+

Description

The method of stereo rendering supported in OpenGL is currently achieved by rendering to the two eye buffers sequentially. This typically incurs double the application and driver overhead, despite the fact that the command streams and render states are almost identical.

This extension seeks to address the inefficiency of sequential Multiview rendering by adding a means to render to multiple elements of a 2D texture array simultaneously. In multiview rendering, draw calls are instanced into each corresponding element of the texture array. The vertex program uses a new ViewID variable to compute per-view values, typically the vertex position and view-dependent variables like reflection.

Registry Link

https://www.khronos.org/registry/OpenGL/extensions/OVR/OVR_multiview.txt

Example

```
glFramebufferTextureMultiviewOVR( target, attachment, texture, level, baseViewIndex, numViews );
```

GL_OVR_multiview_multisampled_render_to_texture

Valid APIs

OpenGL ES 3.0+

Description

This extension brings multisampling to multiview rendering.

Registry Link

https://www.khronos.org/registry/OpenGL/extensions/OVR/OVR_multiview_multisampled_render_to_texture.txt

Example

```
GLsizei width  = ...;
GLsizei height = ...;
GLint samples = ...;
```

```
GLsizei views = 2;
glGenTextures(views, tex);
/* Create a colour texture */
glBindTexture(GL_TEXTURE_2D_ARRAY, tex[0]);
glTexStorage3D(GL_TEXTURE_2D_ARRAY, 1, GL_RGBA8, width, height, views );
/* Create a depth texture */
glBindTexture(GL_TEXTURE_2D_ARRAY, tex[1]);
glTexStorage3D(GL_TEXTURE_2D_ARRAY, 1, GL_DEPTH_COMPONENT24, width, height, views )
/* attach the render targets */
glFramebufferTextureMultisampleMultiviewOVR(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, tex[0],
 0, samples, 0, views);
glFramebufferTextureMultisampleMultiviewOVR(GL_DRAW_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, tex[1], 0,
 samples, 0, views);
/* .. draw to multisampled multiview .. */
```

GL_OVR_multiview2

Valid APIs

OpenGL ES 3.0+

Description

This extension relaxes the restriction in OVR_multiview that only gl_Position can depend on ViewID in the vertex shader. With this change, view-dependent outputs like reflection vectors and similar are allowed.

Registry Link

https://www.khronos.org/registry/OpenGL/extensions/OVR/OVR_multiview2.txt

4. Contact Details

For further support, visit our forum:

http://forum.imgtec.com

Or file a ticket in our support system:

https://pvrsupport.imgtec.com

For general enquiries, please visit our website:

http://imgtec.com/corporate/contactus.asp