Effective OpenGL

11 June 2016, [Christophe Riccio](mailto:mail@g-truc.net)

[](http://www.g-truc.net)

# Table of contents

Table of contents 2

Introduction 4

1. Internal texture formats 5

2. Configurable texture swizzling 6

3. BGRA texture swizzling using texture formats 7

4. Texture alpha swizzling 8

5. Half type constants 9

6. Color read format queries 10

7. Timer query 11

Ideas 12

1.5. Texture 16 bit norm formats 12

1.6. Texture float and half filtering 12

1.7. ETC texture formats 12

1.11. Seamless cubemap texture 12

1.12. Non power of two texture 12

2. Buffer differences 12

2.1. Buffer target 12

2.2. Mapped buffer 12

3. Framebuffer differences 12

3.1. glDrawBuffer 12

3.2. glDrawBuffer(GL\_NONE) 12

3.3. glDrawBuffers(1, &GL\_BACK) 12

3.4. GL\_DEPTH\_STENCIL\_ATTACHMENT 12

3.5. Framebuffer read 12

3.6. Framebuffer sRGB enable 12

3.7. sRGB texture format support 12

3.8. Rendering to float textures 12

3.9. Invalidate framebuffer 12

4. Shader differences 12

4.1. Precision qualifiers 13

5. Misc differences 13

5.1. GL\_MAX\_VERTEX\_UNIFORM\_VECTORS vs GL\_MAX\_VERTEX\_UNIFORM\_COMPONENTS 13

5.2. Multiple transform feedback buffers 13

5.3. Version and extension queries 13

6. Window system differences 13

5. Blending 13

Conclusions 13

References 13

# Introduction

When targeting a large amount of platform, by coincidence OpenGL 3.2 to 4.5, OpenGL ES 2.0 to ES 3.2 and WebGL 1.0 and 2.0, it takes quite some investigations to implement a feature optimally. The difference between doing this investigation and not doing it is basically shipping a buggy engine. As an example, let's study texture swizzle and how it's exposed in OpenGL, OpenGL ES and WebGL.

# 1. Internal texture formats

OpenGL expresses the texture format through the *internal format* and the *external format* which is composed of the *format* and the *type* as glTexImage2D declaration illustrates:

glTexImage2D(GLenum target, GLint level,

GLint **internalformat**, GLsizei width, GLsizei height, GLint border,

GLenum **format**, GLenum **type**, const void\* pixels);

Listing 1.1: Internal and external formats using glTexImage2D

The internal format is the format of the actual storage on the device while the external format is the format of the client storage. This API design allows the OpenGL driver to convert the external data into any internal format storage.

However, while designing OpenGL ES, the Khronos Group decided to simplify the design by forbidding texture conversions([ES 2.0, section 3.7.1](https://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.withchanges.pdf)) and allowing the actual internal storage to be platform dependent to ensure a larger hardware ecosystem support. As a result, it is specified in OpenGL ES 2.0 that the internalformat argument must match the format argument.

glTexImage2D(GL\_TEXTURE\_2D, 0, **GL\_RGBA**, Width, Height, 0, **GL\_RGBA**, GL\_UNSIGNED\_BYTE, Pixels);

Listing 1.2: OpenGL ES loading of a RGBA8 image

This approach is also supported by OpenGL compatibility profile however it will generate an OpenGL error with OpenGL core profile which requires sized internal formats.

glTexImage2D(GL\_TEXTURE\_2D, 0, **GL\_RGBA8**, Width, Height, 0, **GL\_RGBA**, GL\_UNSIGNED\_BYTE, Pixels);

Listing 1.3: OpenGL core profile and OpenGL ES 3.0 loading of a RGBA8 image

Additionally, texture storage (GL 4.2 / [GL\_ARB\_texture\_storage](https://www.opengl.org/registry/specs/ARB/texture_storage.txt) and ES 3.0 / [GL\_EXT\_texture\_storage](https://www.khronos.org/registry/gles/extensions/EXT/EXT_texture_storage.txt)) requires using sized internal formats as well.

glTexStorage2D(GL\_TEXTURE\_2D, 1, **GL\_RGBA8**, Width, Height);

glTexSubImage2D(GL\_TEXTURE\_2D, 0, 0, 0, Width, Height, **GL\_RGBA**, GL\_UNSIGNED\_BYTE, Pixels);

Listing 1.4: Texture storage allocation and upload of a RGBA8 image

#### Sized internal format support:

* Texture storage API
* OpenGL core and compatibility profile
* OpenGL ES 3.0
* WebGL 2.0

#### Unsized internal format support:

* OpenGL compatibility profile
* OpenGL ES
* WebGL

# 2. Configurable texture swizzling

OpenGL provides a mechanism to swizzle the components of a texture before they are returned to the shader. For example, it allows loading a BGRA8 or ARGB8 client texture to OpenGL RGBA8 texture object without a reordering of the CPU data.

This functionally was introduced with [GL\_EXT\_texture\_swizzle](https://www.opengl.org/registry/specs/EXT/texture_swizzle.txt) later promoted to OpenGL 3.3 specification through [GL\_ARB\_texture\_swizzle](https://www.opengl.org/registry/specs/ARB/texture_swizzle.txt) extension and included in OpenGL ES 3.0.

With OpenGL 3.3 and OpenGL ES 3.0, loading a BGRA8 texture can be done using the following approach shown in listing 2.1.

GLint const Swizzle[] = {GL\_BLUE, GL\_GREEN, GL\_RED, GL\_ALPHA};

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_SWIZZLE\_R, Swizzle[0]);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_SWIZZLE\_G, Swizzle[1]);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_SWIZZLE\_B, Swizzle[2]);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_SWIZZLE\_A, Swizzle[3]);

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGBA8, Width, Height, 0, GL\_RGBA, GL\_UNSIGNED\_BYTE, Pixels);

Listing 2.1: OpenGL 3.3 and OpenGL ES 3.0 BGRA texture swizzling, a channel at a time

Alternatively, OpenGL 3.3, [GL\_ARB\_texture\_swizzle](https://www.opengl.org/registry/specs/ARB/texture_swizzle.txt) and [GL\_EXT\_texture\_swizzle](https://www.opengl.org/registry/specs/EXT/texture_swizzle.txt) provides a slightly different approach allowing to setup all components at once as shown in listing 2.2.

GLint const Swizzle[] = {GL\_BLUE, GL\_GREEN, GL\_RED, GL\_ALPHA};

glTexParameteriv(GL\_TEXTURE\_2D, GL\_TEXTURE\_SWIZZLE\_RGBA, Swizzle);

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGBA8, Width, Height, 0, GL\_RGBA, GL\_UNSIGNED\_BYTE, Pixels);

Listing 2.2: OpenGL 3.3 BGRA texture swizzling, all channels at once:

Unfortunately, [neither WebGL 1.0 or WebGL 2.0 support texture swizzle](https://www.khronos.org/registry/webgl/specs/latest/2.0/#5.18) due to the performance impact that implementing such feature on top of Direct3D would have.

#### Support:

* Any OpenGL 3.3 or OpenGL ES 3.0 driver
* [MacOSX 10.8](https://developer.apple.com/opengl/capabilities/GLInfo_1085_Core.html) through [GL\_ARB\_texture\_swizzle](https://www.opengl.org/registry/specs/ARB/texture_swizzle.txt) using the OpenGL 3.2 core driver
* [Intel SandyBridge](http://opengl.gpuinfo.org/gl_listreports.php?listreportsbyextension=GL_EXT_texture_swizzle) through [GL\_EXT\_texture\_swizzle](https://www.opengl.org/registry/specs/EXT/texture_swizzle.txt)

# 3. BGRA texture swizzling using texture formats

OpenGL supports GL\_BGRA external format to load BGRA8 source textures without requiring the application to swizzle the client data. This is done using the following code:

glTexImage2D(GL\_TEXTURE\_2D, 0, **GL\_RGBA8**, Width, Height, 0, **GL\_BGRA**, GL\_UNSIGNED\_BYTE, Pixels);

Listing 3.1: OpenGL core and compatibility profiles BGRA swizzling with texture image

glTexStorage2D(GL\_TEXTURE\_2D, 1, **GL\_RGBA8**, Width, Height);

glTexSubImage2D(GL\_TEXTURE\_2D, 0, 0, 0, Width, Height, **GL\_BGRA**, GL\_UNSIGNED\_BYTE, Pixels);

Listing 3.2: OpenGL core and compatibility profiles BGRA swizzling with texture storage

This functionality isn't available with OpenGL ES. While, it's not useful for OpenGL ES 3.0 that has [texture swizzling support](#_2._Configurable_texture_1), OpenGL ES 2.0 relies on some extensions to expose this feature however it exposed differently than OpenGL because by design, OpenGL ES doesn’t [support format conversions](#_1._Internal_texture_1) including component swizzling.

Using the [GL\_EXT\_texture\_format\_BGRA8888](https://www.khronos.org/registry/gles/extensions/EXT/EXT_texture_format_BGRA8888.txt) or [GL\_APPLE\_texture\_format\_BGRA8888](https://www.khronos.org/registry/gles/extensions/APPLE/APPLE_texture_format_BGRA8888.txt) extensions, loading BGRA textures is done with the code in listing 3.3.

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_BGRA\_EXT, Width, Height, 0, GL\_BGRA\_EXT, GL\_UNSIGNED\_BYTE, Pixels);

Listing 3.3: OpenGL ES BGRA swizzling with texture image

Additional when relying on [GL\_EXT\_texture\_storage](https://www.khronos.org/registry/gles/extensions/EXT/EXT_texture_storage.txt) (ES2), BGRA texture loading requires sized internal format as shown by listing 3.4.

glTexStorage2D(GL\_TEXTURE\_2D, 1, **GL\_BGRA8\_EXT**, Width, Height);

glTexSubImage2D(GL\_TEXTURE\_2D, 0, 0, 0, Width, Height, GL\_BGRA, GL\_UNSIGNED\_BYTE, Pixels);

Listing 3.4: OpenGL ES BGRA swizzling with texture storage

#### Support:

* [Any driver supporting OpenGL 1.2 or GL\_EXT\_bgra including OpenGL core profile](http://delphigl.de/glcapsviewer/gl_listreports.php?listreportsbyextension=GL_EXT_bgra)
* [Adreno 200, Mali 400, PowerVR series 5, Tegra 3, Videocore IV and GC1000 through GL\_EXT\_texture\_format\_BGRA8888](http://delphigl.de/glcapsviewer/gles_listreports.php?extension=GL_EXT_texture_format_BGRA8888)
* [iOS and GC1000 through GL\_APPLE\_texture\_format\_BGRA8888](https://developer.apple.com/library/ios/documentation/DeviceInformation/Reference/iOSDeviceCompatibility/OpenGLESPlatforms/OpenGLESPlatforms.html)
* [PowerVR series 5 through GL\_IMG\_texture\_format\_BGRA8888](http://delphigl.de/glcapsviewer/gles_listreports.php?extension=GL_IMG_texture_format_BGRA8888)

# 4. Texture alpha swizzling

In this section, we call a texture alpha, a single component texture which data is accessed in the shader with the alpha channel (.a, .w, .q).

With OpenGL compatibility profile, OpenGL ES and WebGL, this can be done by creating a texture with an alpha format as demonstrated in listings 4.1 and 4.2.

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_ALPHA, Width, Height, 0, GL\_ALPHA, GL\_UNSIGNED\_BYTE, Data);

Listing 4.1: Allocating and loading an OpenGL ES 2.0 texture alpha

glTexStorage2D(GL\_TEXTURE\_2D, 1, GL\_ALPHA8, Width, Height);

glTexSubImage2D(GL\_TEXTURE\_2D, 0, 0, 0, Width, Height, GL\_ALPHA, GL\_UNSIGNED\_BYTE, Data);

Listing 4.2: Allocating and loading an OpenGL ES 3.0 texture alpha

Texture alpha formats have been removed in OpenGL core profile. An alternative is to rely on [rg\_texture formats](https://www.opengl.org/registry/specs/ARB/texture_rg.txt) and texture swizzle as shown by listings 4.3 and 4.4.

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_SWIZZLE\_R, GL\_ZERO);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_SWIZZLE\_G, GL\_ZERO);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_SWIZZLE\_B, GL\_ZERO);

glTexParameteri(GL\_TEXTURE\_2D, **GL\_TEXTURE\_SWIZZLE\_A**, **GL\_RED**);

glTexImage2D(GL\_TEXTURE\_2D, 0, **GL\_R8**, Width, Height, 0, **GL\_RED**, GL\_UNSIGNED\_BYTE, Data);

Listing 4.3: OpenGL 3.3 and OpenGL ES 3.0 texture alpha

Texture red format was introduced on desktop with OpenGL 3.0 and [GL\_ARB\_texture\_rg](https://www.opengl.org/registry/specs/ARB/texture_rg.txt). On OpenGL ES, it was introduced with OpenGL ES 3.0 and [GL\_EXT\_texture\_rg](https://www.khronos.org/registry/gles/extensions/EXT/EXT_texture_rg.txt). It is also supported by WebGL 2.0.

Unfortunately, OpenGL 3.2 core profile doesn't support either texture alpha format or texture swizzling. A possible workaround is to expend the source data to RGBA8 which consumes 4 times the memory but is necessary to support texture alpha on [MacOSX 10.7](https://developer.apple.com/opengl/capabilities/GLInfo_1075_Core.html).

#### Support:

* [Texture red format is supported on any OpenGL 3.0 or OpenGL ES 3.0 driver](http://delphigl.de/glcapsviewer/listreports2.php?listreportsbyextension=GL_ARB_texture_rg)
* [Texture red format is supported on PowerVR series 5, Mali 600 series, Tegra and Bay Trail on Android through](http://delphigl.de/glcapsviewer/gles_listreports.php?extension=GL_EXT_texture_rg) GL\_[EXT\_texture\_rg](https://www.khronos.org/registry/gles/extensions/EXT/EXT_texture_rg.txt)
* [Texture red format is supported on iOS through](http://delphigl.de/glcapsviewer/gles_listreports.php?extension=GL_EXT_texture_rg)GL\_[EXT\_texture\_rg](https://www.khronos.org/registry/gles/extensions/EXT/EXT_texture_rg.txt)

# 5. Half type constants

Half-precision floating point data was first introduced by [GL\_NV\_half\_float](https://www.opengl.org/registry/specs/NV/half_float.txt) for vertex attribute data and exposed using the constant GL\_HALF\_FLOAT\_NV whose value is 0x140B.

This extension was promoted to [GL\_ARB\_half\_float\_vertex](https://www.opengl.org/registry/specs/ARB/half_float_vertex.txt) renaming the constant to GL\_HALF\_FLOAT\_ARB but keeping the same 0x140B value. This constant was eventually reused for [GL\_ARB\_half\_float\_pixel](https://www.opengl.org/registry/specs/ARB/half_float_pixel.txt), [GL\_ARB\_texture\_float](https://www.opengl.org/registry/specs/ARB/texture_float.txt) and promoted to OpenGL 3.0 core specification with the name GL\_HALF\_FLOAT and the same 0x140B value.

Unfortunately, [GL\_OES\_texture\_float](https://www.khronos.org/registry/gles/extensions/OES/OES_texture_float.txt) took a different approach and exposed the constant GL\_HALF\_FLOAT\_OES with the value 0x8D61. However, this extension never made it to OpenGL ES core specification as OpenGL ES 3.0 reused the OpenGL 3.0 value for GL\_HALF\_FLOAT. [GL\_OES\_texture\_float](https://www.khronos.org/registry/gles/extensions/OES/OES_texture_float.txt) remains particularly useful for OpenGL ES 2.0 devices and WebGL 1.0 which also has a WebGL flavor of [GL\_OES\_texture\_float](https://www.khronos.org/registry/gles/extensions/OES/OES_texture_float.txt) extension.

Finally, just like regular RGBA8 format, OpenGL ES 2.0 requires an unsized internal format for floating point formats. Listing 5.1 shows how to correctly setup the enums to create a half texture across APIs.

GLenum const Type = isES20 || isWebGL10 ? GL\_HALF\_FLOAT\_OES : GL\_HALF\_FLOAT;

GLenum const InternalFormat = isES20 || isWebGL10 ? GL\_RGBA : GL\_RGBA16F;

…

// Allocation of a half storage texture image

glTexImage2D(GL\_TEXTURE\_2D, 0, InternalFormat, Width, Height, 0, GL\_RGBA, Type, Pixels);

…

// Setup of a half storage vertex attribute

glVertexAttribPointer(POSITION, 4, Type, GL\_FALSE, Stride, Offset);

Listing 5.1: Multiple uses of half types with OpenGL, OpenGL ES and WebGL

#### Support:

* All OpenGL 3.0 and OpenGL ES 3.0 implementations
* OpenGL ES 2.0 and WebGL 1.0 through [GL\_OES\_texture\_float](https://www.khronos.org/registry/gles/extensions/OES/OES_texture_float.txt) extensions

# 6. Color read format queries

OpenGL allows reading back pixels on the CPU side using glReadPixels. However, OpenGL ES requires implementation dependent formats which have to be queried. For OpenGL ES compatibility, these queries were added to OpenGL 4.1 core specification with [GL\_ARB\_ES2\_compatibility](https://www.opengl.org/registry/specs/ARB/ES2_compatibility.txt). When the format is expected to represent half data, we encounter enum issue discussed in [section 5](#_5._Half_type) in a specific corner case.

Additionally, NVIDIA ES drivers (353.12) don’t actually support OpenGL ES 2.0 anymore. When we request an OpenGL ES 2.0 context, we get a context for the latest OpenGL ES version supported by the driver, despite that OpenGL ES 3.0 is not backward compatible with OpenGL ES 2.0. Hence, on NVIDIA OpenGL ES implementation, queries will always return GL\_HALF\_FLOAT.

To workaround this issue, listing 6.1 proposes to always check for both GL\_HALF\_FLOAT and GL\_HALF\_FLOAT\_OES even when only targeting OpenGL ES 2.0.

GLint ReadType = DesiredType;

GLint ReadFormat = DesiredFormat;

if(HasImplementationColorRead)

{

glGetIntegerv(GL\_IMPLEMENTATION\_COLOR\_READ\_TYPE, &ReadType);

glGetIntegerv(GL\_IMPLEMENTATION\_COLOR\_READ\_FORMAT, &ReadFormat);

}

std::size\_t ReadTypeSize = 0;

switch(ReadType)

{

case GL\_FLOAT:

ReadTypeSize = 4; break;

case GL\_HALF\_FLOAT:

case GL\_HALF\_FLOAT\_OES:

ReadTypeSize = 2; break;

case GL\_UNSIGNED\_BYTE:

ReadTypeSize = 1; break;

default: assert(0);

}

std::vector<unsigned char> Pixels;

Pixels.resize(components(ReadFormat) \* ReadTypeSize \* Width \* Height);

glReadPixels(0, 0, Width, Height, ReadFormat, ReadType, &Pixels[0]);

Listing 6.1: OpenGL ES 2.0 and OpenGL 4.1 color read format

Unfortunately, an OpenGL program that chooses to only target OpenGL ES 2.0 with no regard for newer version will not possibly run correctly on NVIDIA implementation.

#### Support:

* All OpenGL 4.1, OpenGL ES 2.0 and WebGL 1.0 implementations supports read format queries.
* All OpenGL implementations will perform a conversion to any desired format

#### Known driver bugs:

* NVIDIA ES drivers (353.12) ignore the fact that OpenGL ES 3.0 isn’t backward compatible with OpenGL ES 2.0 returning GL\_HALF\_FLOAT while the OpenGL program expects GL\_HALF\_FLOAT\_OES