Effective OpenGL

7 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

Blablabla. 12

1.8. Texture conversion 12

1.9. BGRA texture format 12

1.10. Texture swizzle 12

1.11. Seamless cubemap texture 13

1.12. Non power of two texture 13

2. Buffer differences 14

2.1. Buffer target 14

2.2. Mapped buffer 14

3. Framebuffer differences 15

3.1. glDrawBuffer 15

3.2. glDrawBuffer(GL\_NONE) 15

3.3. glDrawBuffers(1, &GL\_BACK) 15

3.4. GL\_DEPTH\_STENCIL\_ATTACHMENT 15

3.5. Framebuffer read 15

3.6. Framebuffer sRGB enable 15

3.7. Rendering to float textures 15

3.8. Invalidate framebuffer 15

4. Shader differences 16

4.1. Precision qualifiers 16

5. Misc differences 17

5.1. GL\_MAX\_VERTEX\_UNIFORM\_VECTORS vs GL\_MAX\_VERTEX\_UNIFORM\_COMPONENTS 17

5.2. Multiple transform feedback buffers 17

5.3. Version and extension queries 17

6. Window system differences 18

5. Blending 19

Conclusions 20

References 21

# Introduction

When targeting a large amount of platform, by coincidence OpenGL 3.2 to 4.5, OpenGL ES 2.0 to ES 3.1+ 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 texture formats through the *internal formats* and the *external formats* composed of the *formats* and the *types* 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 4.1: Internal and external formats expressed by glTexImage2D

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

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 for a larger hardware ecosystem support. As a result, it is specified in OpenGL ES 2.0 that the internalformat argument must match format argument.

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

Listing 1.1: OpenGL ES loading of a RGBA8 image

This approach is also supported by OpenGL compatibility profile how ever 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, Data);

Listing 1.2: OpenGL core profile and OpenGL ES 3.0 loading of a RGBA8U image

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

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

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

Listing 1.3: Texture storage loading of a RGBA8U image

Sized internal format support:

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

Unsized internal format support:

* OpenGL compatibility profile
* OpenGL ES
* WebGL

# 2. BGRA texture swizzling using texture formats

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

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

Listing 2.1: 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, Data);

Listing 2.2: OpenGL core and compatibility profiles BGRA swizzling with texture storage

However, such functionality isn't available with OpenGL ES. While, it's not useful for OpenGL ES 3.0 that has texture swizzling support, OpenGL ES 2.0 relies on some extensions to expose this feature but it does it differently than OpenGL.

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 following code:

OpenGL core and compatibility profiles BGRA swizzling with texture image:

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

When [GL\_EXT\_texture\_storage](https://www.khronos.org/registry/gles/extensions/EXT/EXT_texture_storage.txt) (ES2) is supported, BGRA texture loading is perform with the following code.

OpenGL core and compatibility profiles BGRA swizzling with texture image:

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

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

To understand the difference between OpenGL, OpenGL ES and OpenGL ES with texture storage the reasoning is the following: On the contrary to OpenGL, OpenGL ES doesn't support external format to internal format conversion of any kind, including texture component swizzling. Additionally, OpenGL ES 2.0 requires unsized format for the internal format while OpenGL ES 3.0 and texture storage requires sized internal format.

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)

# 3. Configurable texture swizzling

[EXT\_texture\_swizzle](https://www.opengl.org/registry/specs/EXT/texture_swizzle.txt), promoted to OpenGL 3.3 core specification through [ARB\_texture\_swizzle](https://www.opengl.org/registry/specs/ARB/texture_swizzle.txt) extension, introduced a functionality to handle texture with arbitrary ordering or the components without requiring CPU reordering. For example, it allows loading BGRA8 or ARGB8 source textures to OpenGL RGBA8 texture object.

With OpenGL 3.3 and OpenGL ES 3.0, loading a BGRA8 texture can be done using the following approach shown in listing 1.1.

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, Data);

Listing 1.1: OpenGL 3.3 and OpenGL ES 3.0 BGRA texture swizzling, a channel at a time

OpenGL 3.3, [ARB\_texture\_swizzle](https://www.opengl.org/registry/specs/ARB/texture_swizzle.txt) and [EXT\_texture\_swizzle](https://www.opengl.org/registry/specs/EXT/texture_swizzle.txt) provides a slightly different approach that is not available with OpenGL ES 3.0 showing in listing 1.2.

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, Data);

Listing 1.2: 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 performance issue of implementation such feature on top of Direct3D which doesn't have any equivalent functionality.

Support:

* Any OpenGL 3.3 or OpenGL ES 3.0 driver
* [MacOSX 10.8 through ARB\_texture\_swizzle using the OpenGL 3.2 core driver](https://developer.apple.com/opengl/capabilities/index.html)
* [Intel SandyBridge through EXT\_texture\_swizzle](http://delphigl.de/glcapsviewer/gl_generatereport.php?reportID=888)

# 4. Texture alpha swizzling

In this section, we call a texture alpha, a texture we using the alpha channel (.a, .w, .q) in a shader. With OpenGL ES and WebGL, this can be done by creating a texture with an alpha format as shown with the following code samples.

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

Listing 3.1: 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 3.2: Allocating and loading an OpenGL ES 3.0 texture alpha

Texture alpha format is also available in OpenGL compatibility profile and WebGL but was 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 with the following code samples.

On one hand, both OpenGL ES and OpenGL compatibility profile supports texture alpha and luminance alpha. However, OpenGL core profile doesn't. On other hand, OpenGL 3.0 and OpenGL ES 3.0 support texture red and texture red green.

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 3.3: OpenGL 3.3 and OpenGL ES 3.0 texture alpha

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);

glTexStorage2D(GL\_TEXTURE\_2D, 1, GL\_R8, Width, Height);

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

Listing 3.4: OpenGL 4.2 and OpenGL ES 3.0 texture alpha with texture storage

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).

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 which might be necessary to support texture alpha on MacOSX 10.7.

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 NV\_half\_float for vertex attribute data and exposed using the constant GL\_HALF\_FLOAT\_NV whose value is 0x140B.

This extension was promoted to ARB\_half\_float\_vertex renaming the constant to GL\_HALF\_FLOAT\_ARB but keeping the same 0x140B value. This constant was eventually reused for ARB\_half\_float\_pixel, ARB\_texture\_float and promoted to OpenGL 3.0 core specification with the name GL\_HALF\_FLOAT and the same 0x140B value.

Unfortunately, OES\_texture\_float took a different approach and exposed the constant GL\_HALF\_FLOAT\_OES with the value 0xXXXX. However, this extension never made it to OpenGL ES core specification as OpenGL ES 3.0 reused OpenGL 3.0 value for GL\_HALF\_FLOAT however OES\_texture\_float remains particularly useful for OpenGL ES 2.0 devices and WebGL 1.0 which also has a WebGL flavor of OES\_texture\_float extensions.

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, Data);

…

// Setup of a half storage vertex attribute

glVertexAttribPointer(POSITION, 4, Type, GL\_FALSE, 2 \* 4, BUFFER\_OFFSET(0));

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 OES\_texture\_float 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 can be queried. For OpenGL ES compatibility, these queries were added to OpenGL 4.1 core specification with ARB\_ES2\_compatibility.

When the format is expected to represent half data, the issues discussed in item 5 will happen.

Additionally, NVIDIA ES drivers (353.12) don’t actually support ES2. If we request an OpenGL ES 2.0 context we will get an OpenGL ES 3.1 context despite that ES 3.0 is not being backward compatible with ES 2.0. Hence, on NVIDIA OpenGL ES implementation, queries will always return GL\_HALF\_FLOAT.

As a result and taking the color read format case, 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 of OpenGL ES 3.0 will not possibly run correctly on NVIDIA implementation.

Support:

* All OpenGL 4.1, OpenGL ES 2.0 and WebGL 1.0 implementations
* 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

# 7. Timer query

# Ideas

## 1.5. [Texture](http://www.opengl.org/registry/specs/ARB/multi_draw_indirect.txt) 16 bit norm formats

## 1.6. [Texture](http://www.opengl.org/registry/specs/ARB/multi_draw_indirect.txt) float and half filtering

## 1.7. [ETC](http://www.opengl.org/registry/specs/ARB/multi_draw_indirect.txt) texture formats

## 1.11. Seamless cubemap [texture](http://www.opengl.org/registry/specs/ARB/multi_draw_indirect.txt)

## 1.12. Non power of two texture

# 2. Buffer differences

## 2.1. Buffer target

## 2.2. Mapped buffer

# 3. Framebuffer differences

## 3.1. glDrawBuffer

## 3.2. glDrawBuffer(GL\_NONE)

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

## 3.4. GL\_DEPTH\_STENCIL\_ATTACHMENT

## 3.5. Framebuffer read

## 3.6. Framebuffer sRGB enable

## 3.7. sRGB texture format support

## 3.8. Rendering to float textures

## 3.9. Invalidate framebuffer

# 4. Shader differences

## 4.1. Precision qualifiers

# 5. Misc differences

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

## 5.2. Multiple transform feedback buffers

## 5.3. Version and extension queries

# 6. Window system differences

# 5. Blending

# Conclusions

# References