Effective OpenGL

3 August 2016, Christophe Riccio

0. Cross platform support

Initially released on January 1992, OpenGL has a long history which led to many versions; market specific variations such as OpenGL ES in July 2003 and WebGL in 2011; a backward compatibility break with OpenGL core profile in August 2009; and many vendor specifics, multi vendors (EXT), standard (ARB, OES), and cross API extensions (KHR).

OpenGL is massively cross platform but it doesn't mean it comes automagically. Just like C and C++ languages, it allows cross platform support but we have to work hard for it. The amount of work depends on the range of the application targeted market. Across vendors? Eg: AMD, ARM, Intel, NVIDIA, PowerVR and Qualcomm GPUs. Across hardware generations? Eg: Tesla, Fermi, Kepler, Maxwell and Pascal architectures. Across platforms? Eg: macOS, Linux and Windows or Android and iOS. Across languages? Eg: C with OpenGL ES and Javascript with WebGL.

Before the early 90s, vendor specific graphics APIs were the norm driven by hardware vendors. Nowadays, vendor specific graphics APIs are essentially business decisions by platform vendors. For example, in my opinion, Metal is design to lock developers to the Apple ecosystem and DirectX 12 is a tool to force users to upgrade to Windows 10. Only in rare cases, such as Playstation libgnm, vendor specific graphics APIs are actually designed for the purpose of providing better performance.

Using vendor specific graphics APIs leads applications to cut themselves out a part of a possible market share. Metal or DirectX based software won't run on Android or Linux respectively. However, this might be just fine for the purpose of the software or the company success. For example, PC gaming basically doesn't exist outside of Windows, so why bothering using another API than DirectX? Similarly, the movie industry is massively dominated by Linux and NVIDIA GPUs so why not using OpenGL like a vendor specific graphics API? Certainly, vendor extensions are also designed for this purpose. For many software, there is just no other choice than supporting multiple graphics APIs.

Typically, minor platforms rely on OpenGL APIs because of platform culture (Linux, Raspberry Pi, etc) or because they don't have enough weight to impose their own APIs to developers (Android, Tizen, Blackberry, SamsungTV, etc). Not using standards can lead platform to failure because the developer entry cost to the platform is too high. An example might be Windows Phone. However, using standards don't guarantee success but at least developers can leverage previous work reducing platform support cost.

In many cases, the multiplatform design of OpenGL is just not enough because OpenGL support is controlled by the platform vendors. We can identify at least three scenarios: The platform owner doesn't invest enough on its platform; the platform owner want to lock developers to its platform; the platform is the bread and butter of the developers.

On Android, drivers are simply not updated on any devices but the ones from Google and NVIDIA. Despite, new versions of OpenGL ES or new extensions being released, these devices are never going to get the opportunity to expose these new features let alone getting drivers bug fixes. Own a Galaxy S7 for its Vulkan support? #lol. This scenario is a case of lack of investment in the platform, after all, these devices are already sold so why bother?

Apple made the macOS OpenGL 4.1 and iOS OpenGL ES 3.0 drivers which are both crippled and outdated. For example, this result in no compute shader available on macOS or iOS with OpenGL/ES. GPU vendors have OpenGL/ES drivers with compute support, however, they can't make their drivers available on macOS or iOS due to Apple control. As a result, we have to use Metal on macOS and iOS for compute shaders. Apple isn't working at enabling compute shader on its platforms for a maximum of developers; it is locking developers to its platforms using compute shaders as a leverage. These forces are nothing new: Originally, Windows Vista only supported OpenGL through Direct3D emulation…

Finally, OpenGL is simply not available on some platform such as Playstation 4. The point is that consoles are typically the bread and butter of millions budgets developers which will either rely on an exist engine or implement the graphics API as a marginal cost, because the hardware is not going to move for years, for the benefit of an API cut for one ASIC and one system.

This document is built from experiences with the OpenGL ecosystem to ship cross-platform software. It is designed to assist the community to use OpenGL functionalities where we need them within the complex graphics APIs ecosystem.

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) 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** and ES 3.0 / **GL_EXT_texture_storage**) 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** later promoted to OpenGL 3.3 specification through **GL_ARB_texture_swizzle** 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** and **GL_EXT_texture_swizzle** 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 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 through **GL_ARB_texture_swizzle** using the OpenGL 3.2 core driver
- Intel SandyBridge through **GL_EXT_texture_swizzle**

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, 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 including component swizzling.

Using the **GL_EXT_texture_format_BGRA8888** or **GL_APPLE_texture_format_BGRA8888** 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** (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
- Adreno 200, Mali 400, PowerVR series 5, Tegra 3, Videocore IV and GC1000 through **GL_EXT_texture_format_BGRA8888**
- iOS and GC1000 through **GL_APPLE_texture_format_BGRA8888**
- PowerVR series 5 through **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 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`**. On OpenGL ES, it was introduced with OpenGL ES 3.0 and **`GL_EXT_texture_rg`**. 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.

Support:

- Texture red format is supported on any OpenGL 3.0 or OpenGL ES 3.0 driver
- Texture red format is supported on PowerVR series 5, Mali 600 series, Tegra and Bay Trail on Android through **`GL_EXT_texture_rg`**
- Texture red format is supported on iOS through **`GL_EXT_texture_rg`**

5. Half type constants

Half-precision floating point data was first introduced by **GL_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 **GL_ARB_half_float_vertex** 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**, **GL_ARB_texture_float** and promoted to OpenGL 3.0 core specification with the name **GL_HALF_FLOAT** and the same **0x140B** value.

Unfortunately, **GL_OES_texture_float** 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** remains particularly useful for OpenGL ES 2.0 devices and WebGL 1.0 which also has a WebGL flavor of **GL_OES_texture_float** 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** 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`**. When the format is expected to represent half data, we encounter enum issue discussed in section 5 in a specific corner case.

Additionally, many OpenGL ES drivers 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 drivers. Hence, these OpenGL ES implementations, 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, a program that chooses to only target OpenGL ES 2.0 and extensions with no regard for newer versions will not possibly run correctly on OpenGL ES implementations that automatically promote the context version such as NVIDIA driver.

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

7. sRGB texture

sRGB texture is the capability to perform sRGB to linear conversions while sampling a texture. It is a very useful feature for linear workflows.

sRGB textures have been introduced to OpenGL with **`GL_EXT_texture_sRGB`** extensions later promoted to OpenGL 2.1 specification. With OpenGL ES, it has been introduced with **`GL_EXT_sRGB`** which was promoted to OpenGL ES 3.0 specification.

Effectively, this feature provides an internal format variation with sRGB to linear conversion for some formats: **`GL_RGB8 => GL_SRGB8`** ; **`GL_RGBA8 => GL_SRGB8_ALPHA8`**.

The alpha channel is expected to always store linear data, as a result, sRGB to linear conversions are not performed on that channel.

OpenGL ES supports one and two channels sRGB formats through **`GL_EXT_texture_sRGB_R8`** and **`GL_EXT_texture_sRGB_RG8`** but these extensions are not available with OpenGL. However, OpenGL compatibility profile supports **`GL_SLUMINANCE8`** for single channel sRGB texture format.

Why not storing directly linear data? Because the none linear property of sRGB allows increasing the resolution where it matters more of the eyes. Effectively, sRGB formats are trivial compression formats. Higher bit-rate formats are expected to have enough resolution that no sRGB variations is available.

Compressed formats have sRGB variants when there are expected to be used for non linear data. These variants are typically introduced at the same time than the compression formats are introduced. This is the case for BPTC, ASTC or ETC2 however for older compression formats, the situation is more complex.

PVRTC and PVRTC2 sRGB variants are defined in **`GL_EXT_pvrtc_sRGB`**. ETC1 doesn't have a sRGB variations but **`GL_ETC1_RGB8_OES`** is equivalent to **`GL_COMPRESSED_RGB8_ETC2`**, despite using different values, which sRGB variation is **`GL_COMPRESSED_SRGB8_ETC2`**.

For S3TC, the sRGB variations are defined in **`GL_EXT_texture_sRGB`** which is exclusively an OpenGL extensions. With OpenGL ES, only **`GL_NV_sRGB_formats`** exposed sRGB S3TC formats despite many hardware, such as Intel GPUs, being capable. ATC doesn't have any sRGB support.

Support:

- All OpenGL 2.1, OpenGL ES 3.0 and WebGL 2.0 implementations.
- sRGB R8 is supported by PowerVR 6 and Adreno 400 GPUs on Android.
- sRGB RG8 is supported by PowerVR 6 on Android.
- Adreno 200, GCXXX, Mali 4XX, PowerVR 5 and Videocore IV doesn't support sRGB textures.
- WebGL doesn't exposed sRGB S3TC, only Chrome exposes **`GL_EXT_sRGB`**.

Known bugs:

- Intel OpenGL ES drivers (4352) doesn't expose sRGB S3TC formats while it's supported.
- NVIDIA ES drivers (355.00) doesn't list sRGB S3TC formats with **`GL_COMPRESSED_TEXTURE_FORMATS`** query.
- AMD driver (16.7.1) doesn't perform sRGB conversion on texelFetch[Offset] functions

8. sRGB framebuffer object

sRGB framebuffer is the capability of converting from linear to sRGB on framebuffer writes and reading converting from sRGB to linear on framebuffer read. It requires sRGB textures used as framebuffer color attachments and only apply to the sRGB color attachments. It is a very useful feature for linear workflows.

sRGB framebuffers have been introduced to OpenGL with **GL_EXT_framebuffer_sRGB** extension later promoted to **GL_ARB_framebuffer_sRGB** extension and into OpenGL 2.1 specification. On OpenGL ES, the functionality was introduced with **GL_EXT_sRGB** which was promoted to OpenGL ES 3.0 specification.

OpenGL and OpenGL ES sRGB framebuffer have few differences. With OpenGL ES, framebuffer sRGB conversion is automatically performed for framebuffer attachment using sRGB formats. With OpenGL, framebuffer sRGB conversions must be explicitly enabled:

```
glEnable(GL_FRAMEBUFFER_SRGB)
```

OpenGL ES has the **GL_EXT_sRGB_write_control** extension to control the sRGB conversion however a difference remains: With OpenGL, framebuffer sRGB conversions are disabled by default while on OpenGL ES sRGB conversions are enabled by default.

WebGL 2.0 supports sRGB framebuffer object. However, WebGL 1.0 has very limited support through **GL_EXT_sRGB** which is only implemented by Chrome to date.

A possibility workaround is to use a linear format framebuffer object, such as **GL_RGBA16F**, and use a linear to sRGB shader to blit results to the default framebuffer. With this is a solution to allow a linear workflow, the texture data needs to be linearized offline. HDR formats are exposed in WebGL 1.0 by **GL_OES_texture_half_float** and **GL_OES_texture_float** extensions.

With WebGL, there is no equivalent for OpenGL ES **GL_EXT_sRGB_write_control**.

Support:

- All OpenGL 2.1+, OpenGL ES 3.0 and WebGL 2.0 implementations.
- **GL_EXT_sRGB** is supported by Adreno 200, Tegra, Mali 60, Bay Trail.
- **GL_EXT_sRGB** is supported by WebGL 1.0 Chrome implementations.
- **GL_EXT_sRGB_write_control** is supported by Adreno 300, Mali 600, Tegra and Bay Trail

Bugs:

- OSX 10.8 and older with AMD HD 6000 and older GPUs have a bug where sRGB conversions are performed even on linear framebuffer attachments if **GL_FRAMEBUFFER_SRGB** is enabled.

References:

- The sRGB Learning Curve
- The Importance of Terminology and sRGB Uncertainty

9. sRGB default framebuffer

While sRGB framebuffer object is pretty straightforward, sRGB default framebuffer is pretty complex. This is partially due to the interaction with the window system but also driver behaviors inconsistencies that is in some measure the responsibility of the specification process.

On Windows and Linux, sRGB default framebuffer is exposed by **`[WGL|GLX]_EXT_framebuffer_sRGB`** extensions for AMD and NVIDIA implementations but on Intel and Mesa implementations, it is exposed by the promoted **`[WGL|GLX]_ARB_framebuffer_sRGB`** extensions… which text never got written...

In theory, these extensions provide two functionalities: They allow performing sRGB conversions on the default framebuffer and provide a query to figure out whether the framebuffer is sRGB capable as shown in listing 9.1 and 9.2.

```
glGetIntegerv(GL_FRAMEBUFFER_SRGB_CAPABLE_EXT, &sRGBCapable);
```
Listing 9.1: Using [WGL|GLX]_EXT_framebuffer_sRGB, is the default framebuffer sRGB capable?

```
glGetFramebufferAttachmentParameteriv(
    GL_DRAW_FRAMEBUFFER, GL_BACK_LEFT,
    GL_FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING, &Encoding);
```
Listing 9.2: Using [WGL|GLX]_ARB_framebuffer_sRGB, is the default framebuffer sRGB capable?

AMD and NVIDIA drivers support the approach from listing 9.2 but regardless the approach, AMD drivers claims the default framebuffer is sRGB while NVIDIA drivers claims it’s linear. Intel implementation simply ignore the query. In practice, it’s better to simply not rely on the queries, it’s just not reliable.

All OpenGL implementations on desktop perform sRGB conversions when enabled with **`glEnable(GL_FRAMEBUFFER_SRGB)`** on the default framebuffer.

The main issue is that with Intel and NVIDIA OpenGL ES implementation on desktop, there is simply no possible way to trigger the automatic sRGB conversions on the default framebuffer. An expensive workaround is to do all the rendering into a linear framebuffer object and use an additional shader pass to manually performance the final linear to sRGB conversion. A possible format is **`GL_RGB10A2`** to maximum performance when the alpha channel is not useful and when we accept a slight loss of precision (sRGB has the equivalent of up to 12-bit precision for some values). Another option is **`GL_RGBA16F`** with a higher cost but which can come for nearly free with HDR rendering.

EGL has the **`EGL_KHR_gl_colorspace`** extension to explicitly specify the default framebuffer colorspace. This is exactly what we need for CGL, WGL and GLX. HTML5 canvas doesn’t support color space but there is a proposal.

Bugs:

- Intel OpenGL ES drivers (4331) **`GL_FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING`** query is ignored.
- NVIDIA drivers (368.22) returns **`GL_LINEAR`** with **`GL_FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING`** query on the default framebuffer but perform sRGB conversions anyway.
- With OpenGL ES drivers on WGL (NVIDIA & Intel), there is no possible way to perform sRGB conversions on the default framebuffer.

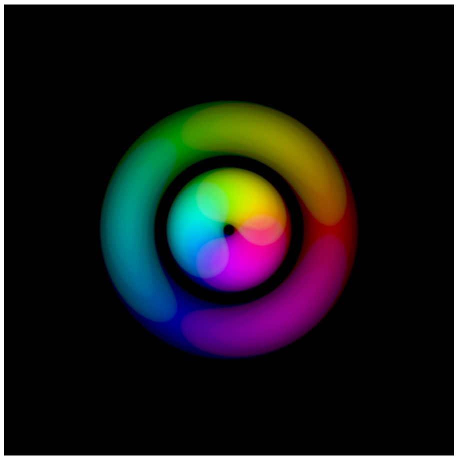
10. sRGB framebuffer blending precision

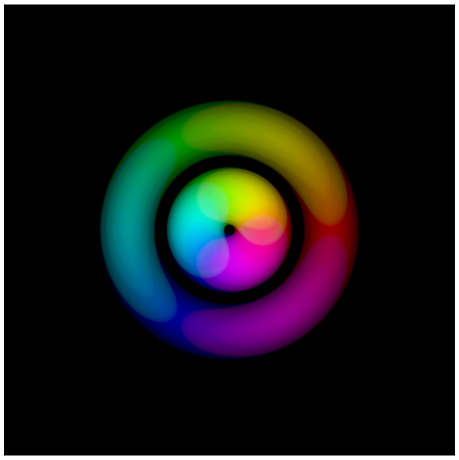
sRGB8 format allows a different repartition of the precisions on a RGB8 storage. Peak precision is about 12bits on small values but this is at the cost of only 6bits precision on big values. sRGB8 provides a better precision where it matters the most for the eyes sensibility and tackle perfectly some use cases just particle systems rendering. While rendering particle systems, we typically accumulate many small values which sRGB8 can represent with great precisions. RGB10A2 also has great RGB precision however a high precision alpha channel is required for soft particles.

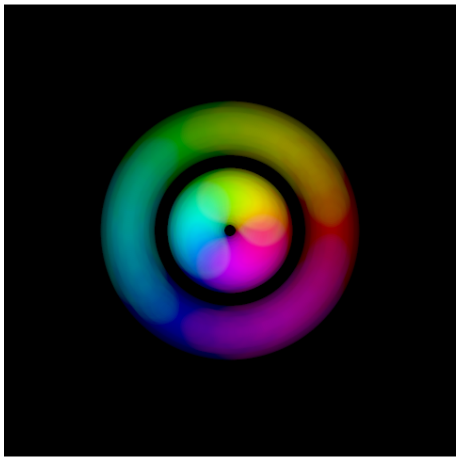
To guarantee that the framebuffer data precision is preserved during blending, OpenGL has the following language:

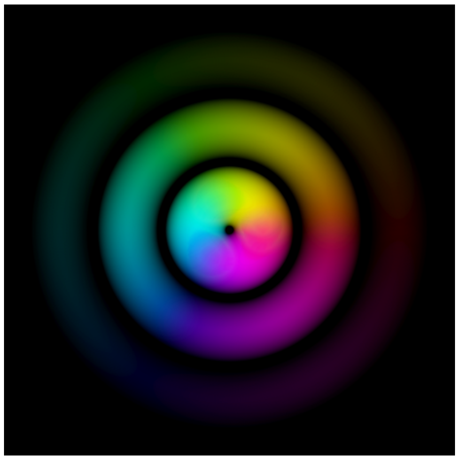
"Blending computations are treated as if carried out in floating-point, and will be performed with a precision and dynamic range no lower than that used to represent destination components."
OpenGL 4.5 - 17.3.6.1 Blend Equation / OpenGL ES 3.2 - 15.1.5.1 Blend Equation

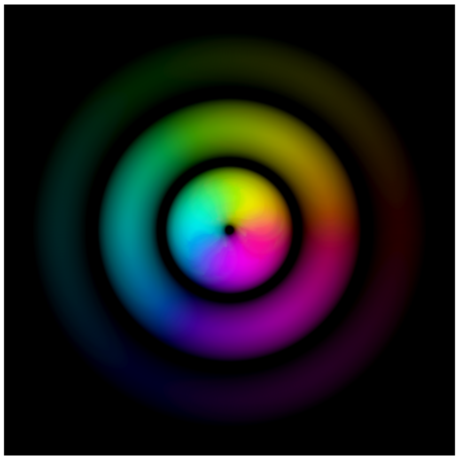
Unfortunately, figure 10.1 shows that NVIDIA support of sRGB blending is really poor.


RGB8 blending on AMD C.I.


RGB8 blending on Intel Haswell


RGB8 blending on NV Maxwell


sRGB8 blending on AMD C.I.


sRGB8 blending on Intel Haswell

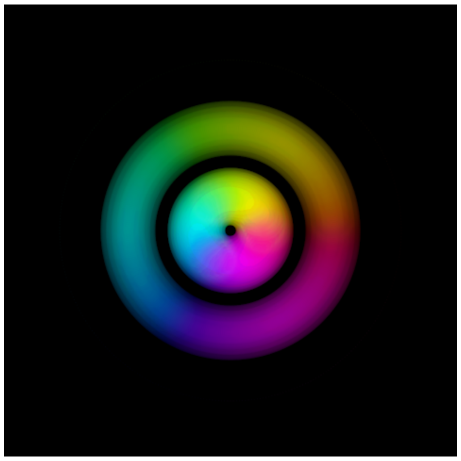

sRGB8 blending on NV Maxwell

Figure 10.1: Blending precision experiment: Rendering with lot of blended point sprites.
Outer circle uses very small alpha values, inner circle uses relative big alpha values.

Tile based GPUs typically perform blending using the shader core ALUs avoiding the blending precision concerns.

Bug:

- NVIDIA drivers (368.69) seem to crop sRGB framebuffer precision to 8 bit linear while performing blending

11. Compressed texture internal format support

OpenGL, OpenGL ES and WebGL provide the queries in listing 11.1 to list the supported compressed texture formats by the system.

```
GLint NumFormats = 0;
glGetIntegerv(GL_NUM_COMPRESSED_TEXTURE_FORMATS, &NumFormats);
std::vector<GLint> Formats(static_cast<std::size_t>(NumFormats));
glGetIntegerv(GL_COMPRESSED_TEXTURE_FORMATS, &Formats);
```

Listing 11.1: Querying the list of supported compressed format

This functionality is extremely old and was introduced with **GL_ARB_texture_compression** and OpenGL 1.3 later inherited by OpenGL ES 2.0 and WebGL 1.0. Unfortunately, drivers support is unreliable on AMD, Intel and NVIDIA implementations with many compression formats missing. However, traditionally mobile vendors (ARM, Imagination Technologies, Qualcomm) seems to implement this functionality correctly.

An argument is that this functionality, beside being very convenient, is not necessary because the list of supported compressed formats can be obtained by checking OpenGL versions and extensions strings. The list of required compression formats is listed appendix C of the OpenGL 4.5 and OpenGL ES 3.2 specifications. Unfortunately, due to patent troll, S3TC formats are supported only through extensions. To save time, listing 11.2 summarizes the versions and extensions to check for each compression format.

	OpenGL	OpenGL ES
S3TC	GL_EXT_texture_compression_s3tc	GL_EXT_texture_compression_s3tc
sRGB S3TC	GL_EXT_texture_compression_s3tc & GL_EXT_texture_sRGB	GL_NV_sRGB_formats
RGTC1, RGTC2	3.0, GL_ARB_texture_compression_rgtc	
BPTC	4.2, GL_ARB_texture_compression_bptc	
ETC1	4.3, GL_ARB_ES3_compatibility	GL_OES_compressed_ETC1_RGB8_texture
ETC2, EAC	4.3, GL_ARB_ES3_compatibility	3.0
ASTC 2D	GL_KHR_texture_compression_astc_ldr	3.2 GL_OES_texture_compression_astc GL_KHR_texture_compression_astc_ldr
Sliced ASTC 3D	GL_KHR_texture_compression_astc_sliced_3d	
ASTC 3D		GL_OES_texture_compression_astc
ATC		GL_AMD_compressed_ATC_texture
PVRTC1		GL_IMG_texture_compression_pvrtc
PVRTC2		GL_IMG_texture_compression_pvrtc2
sRGB PVRTC 1 & 2		GL_EXT_pvrtc_sRGB

Listing 11.2: OpenGL versions and extensions to check for each compressed texture format.

WebGL 2.0 supports ETC2 and EAC and provides many extensions: **WEBGL_compressed_texture_s3tc**, **WEBGL_compressed_texture_s3tc_srgb**, **WEBGL_compressed_texture_etc1**, **WEBGL_compressed_texture_es3**, **WEBGL_compressed_texture_astc**, **WEBGL_compressed_texture_atc** and **WEBGL_compressed_texture_pvrtc**

Support:

- Apple OpenGL drivers don't support BPTC.
- Only Broadwell support ETC2 & EAC formats and Skylake support ASTC on desktop in hardware.
- **GL_COMPRESSED_RGB8_ETC2** and **GL_ETC1_RGB8_OES** are different enums that represent the same data.

Bugs:

- NVIDIA GeForce and Tegra driver don't list RGBA DXT1, sRGB DXT and RGTC formats and list ASTC formats and palette formats that aren't exposed as supported extensions.
- AMD driver (13441) and Intel driver (4454) doesn't list sRGB DXT, LATC and RGTC formats.
- Intel driver (4474) doesn't support ETC2 & EAC (even through decompression) on Haswell.

12. Sized texture internal format support

Required texture formats are described section 8.5.1 of the OpenGL 4.5 and OpenGL ES 3.2 specifications. On the contrary to compression formats, there is no query to list them and it's required to check both versions and extensions. To save time, listing 12.1 summarizes the versions and extensions to check for each texture format.

	OpenGL	OpenGL ES	WebGL
GL_R8, GL_RG8	3.0, GL_ARB_texture_rg	3.0, GL_EXT_texture_rg	2.0
GL_RGB8, GL_RGBA8	1.1	2.0	1.0
GL_SR8	N/A	GL_EXT_texture_sRGB_R8	N/A
GL_SRG8	N/A	GL_EXT_texture_sRGB_RG8	N/A
GL_SRGB8, GL_SRGB8_ALPHA8	3.0, GL_EXT_texture_sRGB	3.0, GL_EXT_sRGB	2.0, GL_EXT_sRGB
GL_R16, GL_RG16, GL_RGB16, GL_RGBA16,	1.1	GL_EXT_texture_norm16	N/A
GL_R8_SNORM, GL_RG8_SNORM, GL_RGBA8_SNORM	3.0, GL_EXT_texture_snorm	3.0, GL_EXT_render_snorm	2.0
GL_RGB8_SNORM,	3.0, GL_EXT_texture_snorm	3.0	2.0
GL_R16_SNORM, GL_RG16_SNORM, GL_RGBA16_SNORM	3.0, GL_EXT_texture_snorm	GL_EXT_render_snorm, GL_EXT_texture_norm16	
GL_RGB16_SNORM	3.0, GL_EXT_texture_snorm	GL_EXT_texture_norm16	
GL_R8UI, GL_RG8UI, GL_R16UI, GL_RG16UI, GL_R32UI, GL_RG32UI, GL_R8I, GL_RG8I, GL_R16I, GL_RG16I, GL_R32I, GL_RG32I	3.0, GL_ARB_texture_rg	3.0	2.0
GL_RGB8UI, GL_RGBA8UI, GL_RGB16UI, GL_RGBA16UI, GL_RGB32UI, GL_RGBA32UI, GL_RGB8I, GL_RGBA8I, GL_RGB16I, GL_RGBA16I, GL_RGB32I, GL_RGBA32I	3.0, GL_EXT_texture_integer	3.0	2.0
GL_RGBA4, GL_R5G6B5, GL_RGB5A1	1.1	2.0	1.0
GL_RGB10A2	1.1	3.0	2.0
GL_RGB10_A2UI	3.3, GL_ARB_texture_rgb10_a2ui	3.0	2.0
GL_R16F, GL_RG16F, GL_RGB16F, GL_RGBA16F	3.0, GL_ARB_texture_float	3.0, GL_OES_texture_half_float	2.0
GL_R32F, GL_RG32F, GL_RGB32F, GL_RGBA32F	3.0, GL_ARB_texture_float	3.0, GL_OES_texture_float	2.0
GL_RGB9_E5	3.0, GL_EXT_texture_shared_exponent	3.0	2.0
GL_R11F_G11F_B10F	3.0, GL_EXT_packed_float	3.0	2.0
GL_DEPTH_COMPONENT16	1.0	2.0	1.0
GL_DEPTH_COMPONENT24, GL_DEPTH24_STENCIL8	1.0	3.0	2.0
GL_DEPTH_COMPONENT32F, GL_DEPTH32F_STENCIL8	3.0, GL_ARB_depth_buffer_float	3.0	2.0
GL_STENCIL8	4.3, GL_ARB_texture_stencil8	3.1	N/A

Listing 12.1: OpenGL versions and extensions to check for each texture format.

Many restrictions apply on texture formats: Multisampling support, mipmap generation, renderable, filtering mode, etc.

For multisampling support, a query was introduced in OpenGL ES 3.0 and then exposed in OpenGL 4.2 and **GL_ARB_internalformat_query**. However, typically all these restrictions are listed in the OpenGL specifications directly.

To expose these limitations through queries, **GL_ARB_internalformat_query2** was introduce with OpenGL 4.3.

A commonly used alternative to checking versions and extensions, consists in creating a texture and then calling **glGetError** at the beginning of the program to initial a table of available texture formats. If the format is not supported, then **glGetError** will return a **GL_INVALID_ENUM** error. However, OpenGL doesn't guarantee the implementation behavior after an error. Typically, implementations will just ignore the OpenGL command but an implementation could simply quit the program. This is the behavior chosen by SwiftShader.

13. Surviving without gl_DrawID

With GPU driven rendering, we typically need an ID to access per draw data just like instancing has with **gl_InstanceID**. With typically draw calls, we can use a default vertex attribute or a uniform. Unfortunately, neither is very efficient and this is why Vulkan introduced push constants. However, GPU driven rendering thrives with multi draw indirect but default attributes, uniforms or push constants can't be used to provide an ID per draw of a multi draw call. For this purpose, **GL_ARB_shader_draw_parameters** extension introduced the **gl_DrawID** where the first draw has the value 0, the second the value 1, etc. Unfortunately, this functionality is only supported on AMD and NVIDIA GPUs since Southern Islands and Fermi respectively. Furthermore, on implementation to date, **gl_DrawID** doesn't always provide the level of performance we could expect…

A first native, and actually invalid, alternative consists in emulating the per draw ID using a shader atomic counter. Using a first vertex provoking convention, when **gl_VertexID** and **gl_InstanceID** are both 0, the atomic counter is incremented by one. Unfortunately, this idea is wrong due to the nature of GPU architectures so that OpenGL doesn't guarantee the order of executions of shader invocations and atomics. As a result, we can't expect even expect that the first draw will be identified with the value 0. On AMD hardware, we almost obtain the desired behavior but not 100% all the time. On NVIDIA hardware atomic counters are executed asynchronously which nearly guarantee that we will never get the behavior we want.

Fortunately, there is an alternative method which turns out to be faster and more flexible. This method leverages the computation of element of a vertex attribute shown in listing 13.1.

```
floor(<gl_InstanceID> / <divisor>) + <baseinstance>
```
Listing 13.1: Computation of the element of a vertex array for a non-zero attribute divisor.

Using 1 has a value for divisor, we can use the **<baseinstance>** parameter as an offset in the DrawID array buffer to provide an arbitrary but deterministic **DrawID** value per draw call. The setup of a vertex array object with one attribute used as per draw identifier is shown in listing 13.2.

```
glGenVertexArrays(1, &VertexArrayName);
glBindVertexArray(VertexArrayName);
glBindBuffer(GL_ARRAY_BUFFER, BufferName[buffer::DRAW_ID]);
glVertexAttribIPointer(DRAW_ID, 1, GL_UNSIGNED_INT, sizeof(glm::uint), 0);
glVertexAttribDivisor(DRAW_ID, 1);
glEnableVertexAttribArray(DRAW_ID);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, BufferName[buffer::ELEMENT]);
```
Listing 13.2: Creating a vertex array object with a DrawID attribute

This functionality is a great fit for multi draw indirect but it also works fine with tight loops, providing an individual per draw identifier per call without setting a single state.

Support:

- Base instance is an OpenGL 4.0 and **GL_ARB_base_instance** feature.
- Base instance is available on GeForce 8, Radeon HD 2000 series and Ivry Bridge.
- Base instance is exposed to OpenGL ES through **GL_EXT_base_instance**.
- Base instance is only exposed on mobile on Tegra SoCs since K1.

14. Cross architecture control of framebuffer restore and resolve to save bandwidth

A good old trick on immediate rendering GPUs (IMR) is to avoid clearing the colorbuffers when binding a framebuffer to save bandwidth. Additionally, if some pixels haven't been written during the rendering of the framebuffer, the rendering of an environment cube map must take place last. The idea is to avoid writing pixels that will be overdraw anyway. However, this trick can cost performance on tile based GPUs (TBR) where the rendering is performed on on-chip memory which behaves as an intermediate buffer between the execution units and the graphics memory as shown in figure 14.1.

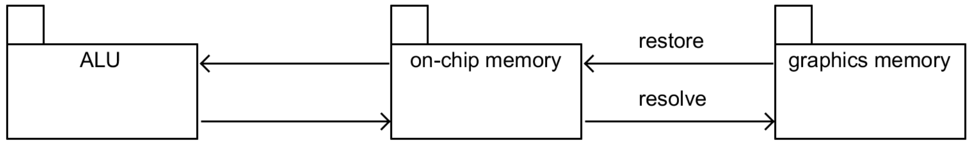


Figure 14.1: Data flow on tile based GPUs.

On immediate rendering GPUs, clearing the framebuffer, we write in graphics memory. On tile based GPUs, we write in on-chip memory. Replacing all the pixels by the clear color, we don't have to **restore** the graphics memory into tile based memory. To further optimize, we can simply invalidate the framebuffer to notify the driver that the data is not needed.

Additionally, we can control whether we want to save the data of a framebuffer into graphics memory. When we store the content of a tile, we don't only store the on-chip memory into the graphics memory, we also process the list of vertices associated to the tile and perform the multisampling resolution. We call these operations a **resolve**.

The OpenGL/ES API allows to control when the restore and resolve operations are performed as shown in listing 14.2.

```
void BindFramebuffer(GLuint FramebufferName, GLenum Target,
    GLsizei NumResolve, GLenum Resolve[], GLsizei NumRestore, GLenum Restore[], bool ClearDepthStencil)
{
    if(NumResolve > 0) // Control the attachments we want to flush from on-chip memory to graphics memory.
        glInvalidateFramebuffer(Target, NumResolve, Resolve);

    glBindFramebuffer(Target, FramebufferName);

    if(NumRestore > 0) // Control the attachments we want to fetch from graphics memory to on-chip memory.
        glInvalidateFramebuffer(Target, NumRestore, Restore);

    if(ClearDepthStencil && Target != GL_READ_FRAMEBUFFER)
        glClear(GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
}
```

Listing 14.2: Code sample showing how to control tile restore and resolve with the OpenGL/ES API

A design trick is to encapsulate `glBindFramebuffer` in a function with arguments that control restore and resolve to guarantees that each time we bind a framebuffer, we consider the bandwidth consumption. These considerations should also take place when calling `SwapBuffers` functions.

We can partial invalidate a framebuffer, however, this is generally suboptimal as parameter memory can't be freed. When `glInvalidateFramebuffer` and `glDiscardFramebufferEXT` are not supported, `glClear` is a good fallback to control restore but `glInvalidateFramebuffer` can be used all both IMR and TBR GPUs in a unique code path without performance issues.

Support:

- `GL_EXT_discard_framebuffer` extension is largely supported: Adreno 200, BayTrail, Mali 400, Mesa, PowerVR SGX, Videocore IV and Vivante GC 800 and all ES3.0 devices.
- `glInvalidateFramebuffer` is available on all ES 3.0 devices ; GL4.3 and `GL_ARB_invalidate_subdata` including GeForce 8, Radeon HD 5000, Intel Haswell devices.

References:

- Performance Tuning for Tile-Based Architectures, Bruce Merry, 2012
- How to correctly handle framebuffers, Peter Harris, 2014

15 Building platform specific code paths

It is often necessary to detect the renderer to workaround bugs or performance issues and more generally to define a platform specific code path. For example, with PowerVR GPUs, we don't have to sort opaque objects front to back. Listing 15.1 shows how to detect the most common renderers.

```
enum renderer {
    RENDERER_UNKNOWN, RENDERER_ADRENO, RENDERER_GEFORCE, RENDERER_INTEL, RENDERER_MALI, RENDERER_POWERVR,
    RENDERER_RADEON, RENDERER_VIDEOCORE, RENDERER_VIVANTE, RENDERER_WEBGL
};

renderer InitRenderer() {
    char const* Renderer = reinterpret_cast<char const*>(glGetString(GL_RENDERER));
    if(strstr(Renderer, "Tegra") || strstr(Renderer, "GeForce") || strstr(Renderer, "NV"))
       return RENDERER_GEFORCE; // Mobile, Desktop, Mesa
    else if(strstr(Renderer, "PowerVR") || strstr(Renderer, "Apple"))
       return RENDERER_POWERVR; // Android, iOS PowerVR 6+
    else if(strstr(Renderer, "Mali"))
       return RENDERER_MALI;
    else if(strstr(Renderer, "Adreno"))
       return RENDERER_ADRENO;
    else if(strstr(Renderer, "AMD") || strstr(Renderer, "ATI"))
       return RENDERER_RADEON; // Mesa, Desktop, old drivers
    else if(strstr(Renderer, "Intel"))
       return RENDERER_INTEL; // Windows, Mesa, mobile
    else if(strstr(Renderer, "Vivante"))
       return RENDERER_VIVANTE;
    else if(strstr(Renderer, "VideoCore"))
       return RENDERER_VIDEOCORE;
    else if(strstr(Renderer, "WebKit") || strstr(Renderer, "Mozilla") || strstr(Renderer, "ANGLE"))
       return RENDERER_WEBGL; // WebGL
    else return RENDERER_UNKNOWN;
}
```
Listing 15.1: Code example to detect the most common renderers

WebGL is a particular kind of renderer because it typically hides the actual device used because such information might yield personally-identifiable information to the web page. **WEBGL_debug_renderer_info** allows detecting the actual device.

The renderer is useful but often not enough to identify the cases to use a platform specific code path. Additionally, we can rely on the OS versions, the OpenGL versions, the availability of extensions, compiler macros, etc.

GL_VERSION query may help for further accuracy as hardware vendor took the habit to store the driver version in this string. However, this work only on resent desktop drivers (since 2014) and it requires a dedicated string parser per renderer. On mobile, vendor generally only indicate the OpenGL ES version. However, since Android 6, it seems most vendors expose a driver version including a source version control revision. This is at least the case of Mali, PowerVR and Qualcomm GPUs.

When building a driver bug workaround, it's essential to write in the code a detail comment including the OS, vendor, GPU and even the driver version. This workaround will remain for years, with many people working on the code. The comment is necessary to be able to remove some technical debt when the specific platform is no longer supported and to avoid breaking that platform in the meantime. Workarounds are typically hairy, hence, without a good warning, the temptation is huge to just remove it.

On desktop, developers interested in very precise identification of a specific driver may use OS specific drivers detection.

Support:

- **WEBGL_debug_renderer_info** is supported on Chrome, Chromium, Opera, IE and Edge

Change log

2016-08-03

- Added item 14: Cross architecture control of framebuffer restore and resolve to save bandwidth
- Added item 15: Building platform specific code paths

2016-07-22

- Added item 13: Surviving without gl_DrawID

2016-07-18

- Updated item 0: More details on platform ownership

2016-07-17

- Added item 11. Compressed texture internal format
- Added item 12. Sized texture internal format

2016-07-11

- Updated item 7: Report AMD bug: texelFetch[Offset] missing sRGB conversions
- Added item 10: sRGB framebuffer blending precision

2016-06-28

- Added item 0: Cross platform support
- Added item 7: sRGB textures
- Added item 8: sRGB framebuffer objects
- Added item 9: sRGB default framebuffer

2016-06-12

- Added item 1: Internal texture formats
- Added item 2: Configurable texture swizzling
- Added item 3: BGRA texture swizzling using texture formats
- Added item 4: Texture alpha swizzling
- Added item 5: Half type constants
- Added item 6: Color read format queries