Effective OpenGL

12 June 2016, Christophe Riccio

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