OpenGL 3.3 review

16 March 2010, Christophe Riccio

Introduction

1. RGB10A2 format for vertex attributes and textures

I keep saying: "It's all about bandwidth!" A GPU is so much computational power, it need to be feed.

Usually, for normals and tangents attributes we use floating point data. It is a lot of precision but a big cost in memory bandwidth. GL_ARB_vertex_type_2_10_10_10_rev provide RGB10A2 format for vertex attribute. The bandwidth is reduced by 2 or 3 times and it keep a really good precision, actually higher than most normal maps.

OpenGL supports RGB10A2 textures since OpenGL 1.1 but GL_ARB_texture_rgb10_a2ui allows an unnormalized access to the texture data just like GL_EXT_texture_integer allowed it for common interger textures in OpenGL 3.0.

2. Sampler object

Finally! The OpenGL community has debated a lot about this feature before the release. OpenGL texture objects is a conceptual non sens which blend data and operations on a single object. It results in a lot of limitations that various extensions try to remove. GL_ARB_sampler_objects allows sampling a single image with multiple different filters and sampling multiple images with the same filter. This could be a huge benefice both in texture memory (no data copy) and texture filtering processing thanks to an adaptative filtering per-fragment. Why using an amazing filtering method when the fragment is in a blurred part of the image?

This extension still rely on the "texture unit" semantic. Many developers (include me) wished to see this extension but in a form that avoid avoid texture unit. To be fair, I would say that GL_ARB_sampler_objects provides a simple solution for a complex problem. I can't be against sure approach. Moreover, this extension is the first one to use 'direct state access' instead of 'bind to edit' API.

Samplers apply on all texture targets.

Some dreamed code:

```
GLuint Sampler = 0;
glGenSamplers(1, &Sampler);
glSamplerParameteri(Sampler, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glSamplerParameteri(Sampler, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glSamplerParameteri(Sampler, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
glSamplerParameteri(Sampler, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
glSamplerParameteri(Sampler, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glSamplerParameteri(Sampler, GL_TEXTURE_MAX_ANISOTROPY_EXT, 16);
```

And I woke up:

```
// Bind 1 texture to be sample by 2 samplers.
glBindSampler(0, Sampler);
glBindSampler(1, SamplerB);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, Texture);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, Texture);
```

3. GLSL update

GLSL 3.30 is few changes away from GLSL 1.50. One good thing is the version change. It's easier to match GLSL and OpenGL versions from now on.

GL_ARB_shader_bit_encoding provides functions for getting and setting the bit encoding for floating-point values. 4 fonctions: `floatBitsToInt`, `floatBitsToUint`, `intBitsToFloat`, `uintBitsToFloat`.

More important feature and actually I think it's quite a great improvement: GL_ARB_explicit_attrib_location. This extension allows setting the location of the input variables inside the vertex shader and the output variables inside the fragment shader. This is actually a great and beautiful way to implement Cg 'semantics' in GLSL.

Vertex shader input declaration:

```
#version 330
#define POSITION 0
#define COLOR0 1
#define NORMAL 4

layout(location = POSITION) in vec4 Position;
layout(location = COLOR0) in vec4 Colors[3]; // Reserve 2 and 3 locations.
layout(location = NORMAL) in vec4 Normal;
```

Fragment shader output declaration:

```
#version 330
#define COLOR0 0
#define NORMAL 3

// Reserve location 0, 1 and 2
layout(location = COLOR0) out vec4 Colors[3];
// First parameter of the blend equation, index = 0 by default
layout(location = NORMAL) out vec3 Normal;
// Second parameter of the blend equation
layout(location = NORMAL, index = 1) out vec4 Factor;
```

Less or even no more `glGetFragDataLocation`, `glBindFragDataLocation`, `glGetAttribLocation`, `glBindAttribLocation` calls needed! These functions are quite an overload and unpleasant for software design.

This location idea is such a main feature for me. However, that's not enough to be really a great one. I would like to see some kind of GL_ARB_explicit_uniform_location to set the uniform variable locations and extend the concept for 'varying' variables which will fix all the complaint I have against seperate shader objects (to make each stage independent) before promoting this extension to core (in OpenGL 3.4...).

Wait a minute? From where come this 'index' of blend equation parameter? I would say from GL_ARB_blend_func_extended extension, a community request and also it seems to be a Direct3D 10 feature (Anyone has some documentation on that?). To use the second index, new values can be used in `glBlendEquation` and `glBlendEquationSeparate`: `GL_SRC1_COLOR`, `GL_SRC1_ALPHA`, `GL_ONE_MINUS_SRC1_COLOR`, `GL_ONE_MINUS_SRC1_ALPHA`.

4. Occlusion query refinement

GL_ARB_occlusion_query2 is the one extension that's made me laught. It's a nice extension that adds a new occlusion query: `GL_ANY_SAMPLES_PASSED`. The difference with `GL_SAMPLES_PASSED`? `GL_SAMPLES_PASSED` returns a count of samples and `GL_ANY_SAMPLES_PASSED` is just a boolean value that returns not null is any sample pass. Just like the super old GL_HP_occlusion_test extension behave! An example in this extension shows some conditional rendering using `GL_ANY_SAMPLES_PASSED` without using the OpenGL conditional rendering funtions...

Ok, this extension is a why not... but I don't think this extension improves anything. It's nice and will of course work as well as `GL_SAMPLES_PASSED` with conditional rendering and maybe slightly faster.

5. Promoted extensions to OpenGL core specification

GL_ARB_texture_swizzle
GL_ARB_intanced_arrays
GL_ARB_timer_query

Texture swizzle provides some new texture sampler states to swizzle the texture components which provides a great freedom to interpret texture format.

Instanced arrays provides an alternative to the current OpenGL instancing techniques. We can already use uniform buffer and texture buffer to store the per instance data, GL_ARB_intanced_arrays proposed to use array buffers for per instance data. Using the function `glVertexAttribDivisor` for each per-instance array buffer, we specify that the draw call must use the first attribute for N vertices where N should be the count of instances vertices. This extension allows to draw multiples different objects as well, as far as they have the same number of vertices. Using attributes for instance data is likely to avoid the latency of a texture buffer fetch but might fill up the attribute data flow if the size of per instance data is

quite large. Probably the fastest instancing method for small per instance data rendering and huge number of instance.

Instanced arrays is an extension that came along with OpenGL 3.0 release without being promoted to core until now. This extension have been widely supported by ATI but lacks on nVidia drivers, being part of the core specification will hopefully brings this feature on nVidia hardware.

Finally, GL_ARB_timer_query use the query object to request the time in nanosecond spend by OpenGL calls without stalling the graphics pipeline. A great extension for optimisation and maybe for dynamically adjusting the quality level to keep a good enough frame rate.

6. GLSL #include directive extension

GL_ARB_shading_language_include is an extension that provides an `#include` directive to GLSL but it hasn't been promoted to OpenGL 3.3 specifications. GLSL 3.30 specification shows that initially, this extension was planned to be part of OpenGL 3.3 core.

The goal of this #include directive is to reuse the same shader text in multiple shader across multiple contexts. The way GL_ARB_shading_language_include allows this, it introduces 'named strings' to create some kind of paths in the GLSL compiler space that contains the shader texts. These name strings are created and deleted by `glNamedStringARB` and `glDeleteNamedStringARB`. From these name strings, the shader text can be compiled with the function `glCompileShaderIncludeARB` of `glCompileShader`.

This extension is an interesting step on the topic of GLSL build management. Developpers are asking for improvement on that side, #include but also shader binaries (blobs). I'm not completely convinced by this extension. It might require just few improvements. Direct loading from OS filesystem have been considered but involved portability issues.

Until the next series post

And this is absolutely all for OpenGL 3.3? Yes, it's not so much! The others extensions release at GDC 2010 are either Direct3D 10.1 and Direct3D 11 hardware extensions. OpenGL 3.3 is a small update of OpenGL 3.2 which provides some good refinements for API. However, I already expect an OpenGL 3.4 release because some subsets of OpenGL 4.0 features seem to be perfectly compatible with OpenGL 3.x hardware. Moreover, GLSL 3.3 and GLSL 4.0 suffer of incompatibilities independent from the hardware.

Link: OpenGL 3.3 core specification
Link: OpenGL 3.3 compatibility specification
Link: GLSL 3.3 specification
Link: OpenGL registry with everything
Link: OpenGL 4.0 quick reference card