OpenGL 4.1 review

18 August 2010, Christophe Riccio

1. The proper OpenGL 4 hardware features

1.1. Double float vertex attributes

OpenGL 4.1 includes the new extension called GL_ARB_vertex_attrib_64bit which trivially adds support for double precision floating point scalars and vectors for the vertex attributes. OpenGL already provides such support but usually the double values are converted to single precision floating point values. This extension ensures that this precision will be kept. An important detail is that double attribute can consume more than one location which will have some consequences in the software design. This behaviour isn't new as GL_ARB_gpu_shader_fp64 introduced it for double uniforms.

1.2. operations precision

OpenGL 4.1 also integrates GL_ARB_shader_precision which is the first extension that defines no value, API functions nor GLSL functions. It simply clarifies the accuracy of various GLSL functions.

List of the precision details where ULP is 'units in the last place':
- a+b, a-b, a*b: correctly rounded
- <, =<, ==, >, >=: correct result
- a/b, 1.0/b: <= 2.5 ULP
- a*b+c: correctly rounded single operation or sequence of two correctly rounded operations
- fma(): same as a*b+c
- pow: <= 16 ULP
- exp, exp2: <= 3 ULP
- log, log2: <= 3 ULP
- sqrt: <= 3 ULP
- inversesqrt: <= 2 ULP
- conversions: correctly rounded

All in all the proper OpenGL 4 hardware features are quite minors unlike what I was expecting but it provides much more interesting features which would actually work on OpenGL 2.1 and OpenGL 3 hardware.

2. A new development area through OpenGL debugging

2.1. Debugging with OpenGL until now

OpenGL 4.1 comes along with a new ARB extension which is just so great that I want to speak about it first: With GL_ARB_debug_output, the ARB has finally shows some mercy for us OpenGL developers by providing a more advanced mechanism based on callbacks than `glGetError` for debugging. This extension brings a new era for OpenGL!

2.2. Debugging improvements provided by GL_ARB_debug_output

GL_ARB_debug_output is an extension only and will probably stay as an extension as the ARB thought it is just good for developers, leaving the possibility of user of OpenGL applications to not have such feature. This may imply 2 types of OpenGL drivers in the future, one for developers and one for users. This is an interesting idea even if I actually see in the debug output capabilities, a way to generate a crash dump.

2.3. Current support

This extension is supported by nVidia drivers and through GL_AMD_debug_output on AMD drivers. For this extension "supported" doesn't mean much because it could just return the glGetError but we can also imagine more accurate messages so that it will be the reponsability of AMD and nVidia to make the best of this extension on a long term development process.

Catch all errors without distinction:

```
glDebugMessageControlARB(GL_DONT_CARE, GL_DONT_CARE, GL_DONT_CARE, 0, NULL, GL_TRUE);
glDebugMessageCallbackARB(&glf::debugOutput, NULL);
```

3. Separate program object

As part of OpenGL 4.1 core, GL_ARB_separate_program_objects is another welcome feature expected from a long time and quite successful in it's approach.

3.1. Increase the pipeline flexibility

It allows to independently use shader stages without changing others shader stages. I see two mains reasons for it: Direct3D, Cg and even the old OpenGL ARB program does it but more importantly it brings some software design flexibilities allowing to see the graphics pipeline at a lower granularity. For example, my best enemy the VAO, is a container object that links buffer data, vertex layout data and GLSL program input data. Without a dedicated software design, this means that when I change the material of an object (a new fragment shader), I need different VAO... It's fortunately possible to keep the same VAO and only change the program by defining a convention on how to communicate between the C++ program and the GLSL program. It works well even if some drawbacks remains.

With the separate programs, the fragment shader and vertex shader stages can be independant so that we are free to change the fragment program without touching the VAO. Finally! It's incredible all the consequences of an awfully designed API (VAOs)!

3.2. OpenGL 4.1 design

So just like Direct3D, OpenGL support separate programs but actually and we should get use to it, OpenGL outperform the Direct3D design. We have had a single program for ages for some reasons: the resource by name convention that requires a linking step across stages

but also because it allows some effective compiler optimizations. From those, the one I rank number 1 discards all the unused varying variables… This consideration has an impact on the design decision of the extension.

GL_ARB_separate_program_objects is a superset of GL_EXT_separate_program_objects extension including just the right improvements to transform a badly design extension to a great extension. With the ARB version, a new object called “pipeline program object” is used to attach multiple programs. It's possible to communicate between stages using user-defined variables instead of the deprecated varying variables… GLSL programs can contained multiple shader stages so that multiple stages can be linked and optimized all together. Chances are that vertex, control and evaluation shaders will be design to be use all together and consequently we can apply some extra optimizations by linking them. Finally, GL_ARB_separate_program_objects defines direct state access functions `glProgramUniform*` for all the `glUniform*` functions!

3.3. A quick code view

Create and setup a pipeline program object:

```
glGenProgramPipelines(1, &PipelineName);
glBindProgramPipeline(PipelineName);
glUseProgramStages(PipelineName, GL_VERTEX_SHADER_BIT, ProgramName[program::VERTEX]);
glUseProgramStages(PipelineName, GL_FRAGMENT_SHADER_BIT,
ProgramName[program::FRAGMENT]);
```

Vertex shader with explicit varying locations:

```
#version 410 core
// Declare all the semantics
#define ATTR_POSITION      0
#define ATTR_TEXCOORD      4
#define VERT_TEXCOORD      4
#define FRAG_COLOR    0

uniform mat4 MVP;

layout(location = ATTR_POSITION) in vec2 Position;
layout(location = ATTR_TEXCOORD) in vec2 Texcoord;
layout(location = VERT_TEXCOORD) out vec2 VertTexcoord;

void main()
{
       gl_Position = MVP * vec4(Position, 0.0, 1.0);
       VertTexcoord = Texcoord;
}
```

Fragment shader with explicit varying locations:

```
#version 410 core
// Declare all the semantics
#define ATTR_POSITION      0
#define ATTR_TEXCOORD      4
```

```
#define VERT_TEXCOORD      4
#define FRAG_COLOR         0

uniform sampler2D Diffuse;
layout(location = ATTR_TEXCOORD) in vec2 Texcoord;
layout(location = FRAG_COLOR, index = 0) out vec4 FragColor;

void main()
{
       FragColor = texture(Diffuse, Texcoord);
}
```

4. GLSL program binary

Another feature we have been waiting for a long time: Program binaries. I must say, this is one topic I haven't followed closely nor thought about it but for what I have understood, the capabilities of loading and saving GLSL binaries provided by GL_ARB_get_program_binary and OpenGL 4.1 is just a subset of the wishes out there.

I'm not sure about how useful this is. This extension comes from the OpenGL ES extensions GL_OES_get_program_binary with a subtle change: A hint to retrieve the program binary after being used or at the end of the program... and here is my problem. A GLSL program is sometime rebuild considering the context states so that there are effectively several different binaries per program. What actually involves a linking of the program? This is fairly undocumented so far but I have high expectations from GL_ARB_debug_output for that regard.

The goal is not to be able to release a software without the GLSL source. A GLSL binary is platform dependant and loading a GLSL binary might fail which involves GLSL sources rebuild. We might see some standard binary formats in the future but so far there is nothing in the OpenGL world. However, binary formats has been present on the OpenGL ES world for a while and many proprietary extensions have been released: GL_AMD_program_binary_Z400, GL_IMG_program_binary, GL_ARM_mali_shader_binary.

Conscequently, the program binary is just a cache system for GLSL binaries... It is really going to make the loading significantly faster? I have some doubt about it.

5. Multiple viewport

One difference from Direct3D and OpenGL is that OpenGL allows multiple rendertarget having different sizes. This is actually a contain relaxed from GL_EXT_framebuffer_object when is has been promoted to GL_ARB_framebuffer_object and OpenGL 3.0. Interestingly, this capability seems to me pretty useless without GL_ARB_viewport_array. This allows to render G-Buffers at various resolutions (and hence saving some memory bandwidth) in a single pass before the final G-Buffer compositing for example. I also imagine some interesting use for cascade shadows using layering rendering. It might look like a small feature at first look but for the rendering technique side, it's actually a key feature from which I expect a lot of performance

benefits in lot of cases.

To use the extra viewports, GLSL includes a new variable called `gl_ViewportIndex` that can be written in the geometry shader.

Setup 2 viewports for 2 render targets:

```
glViewportIndexedfv(0, &vec4(0.0f, 0.0f, Size)[0]);
glScissorIndexedv(0, &ivec4(0, 0, ivec2(Size)[0]);
glDepthRangeIndexed(0, 0.0f, 0.5f);
glViewportIndexedfv(1, &vec4(0.0f, 0.0f, Size * 0.5f)[0]);
glScissorIndexedv(1, &ivec4(0, 0, ivec2(Size * 0.5f)[0]);
glDepthRangeIndexed(1, 0.5f, 1.0f);
```

6. Compatibility with OpenGL ES 2.0

It could be hard to believe but OpenGL ES 2.0 has features that OpenGL 4 didn't had until OpenGL 4.1. GL_ARB_ES2_compatibility is the last extension integrated to OpenGL 4.1. This extension is fairly a boring one that only aims completeness and to ensure that OpenGL 4.1 is a superset of OpenGL ES 2.0. Chances are, I'm never going to use any of the new functions: `glReleaseShaderCompiler` allows to unload the GLSL compiler to save some memory; `glShaderBinary` to load shader binaries... but we have `glProgramBinary` now; `glGetShaderPrecisionFormat` to get GLSL precision of each variable types (`GL_LOW_FLOAT`, `GL_MEDIUM_FLOAT`, `GL_HIGH_FLOAT`, `GL_LOW_INT`, `GL_MEDIUM_INT`, `GL_HIGH_INT`). What if the variable is a `uvec*`?; `glDepthRangef` and `glClearDepthf` which takes float parameters instead of doubles. Finally, `glVertexAttribPointer` can take `GL_FIXED` for its type parameter, for fixed point data.

Features for the sake of... something, why not but I would mark most of them as deprecated.

7. A bunch of things for WebGL

I quite believe that a lot of things released with OpenGL 4.1 have been done for WebGL. It's incredible the entousiam around this technology, probably higher than OpenCL. WebGL involves a large number of developers and users in a medium term: basically everyone all in all!

7.1. Because WebGL is OpenGL ES 2.0 in Javascript

WebGL is based on OpenGL ES 2.0 so that we really need OpenGL ES 2.0 features on desktop. OpenGL 4.1 contains all the OpenGL ES 2.0 features hence WebGL implementations can rely on OpenGL 4.1 drivers to implement all the features. To reinforce OpenGL ES 2.0 on desktop, an OpenGL ES 2.0 profile have been created through WGL_EXT_create_context_es2_profile and GLX_EXT_create_context_es2_profile. I guess, this way it will be easier for low-end OpenGL implementations to support OpenGL ES 2.0 and it isn't require to load a large amount of function pointers that anyway shouldn't be used in a WebGL environment.

7.2. WebGL safety requirements

Following the WebGL requirements, I think the ARB also build the following extensions: WGL_ARB_create_context_robustness, GLX_ARB_create_context_robustness and GL_ARB_robustness which introduces functions which prevent buffer overflows in a similar way that `strncpy` is for `strcpy`... These extensions will be a great use for Chrome developers, Firefox developers, Opera developers, Safari developers (Internet Explorer developers?) and we really want them to do a good use of them, but for the rest of us, I'm not sure.

8. Improved cooperation with OpenCL

OpenCL has made steps toward OpenGL since its release through extensions but for once it's OpenGL which make a step toward OpenCL with the release of the extension GL_ARB_cl_event. This extension only allows to create an OpenGL sync object from an OpenCL sync object for more efficient images and buffers sharing between the 2 APIs.

9. Writing into the stencil buffer from a fragment shader

Finally, an other extension that might generate some clever ideas based on the stencil buffer. GL_ARB_shader_stencil_export allows to write the reference value of the stencil test from a fragment shader which actually means writing into the stencil buffer when `glStencilOp` is set to `GL_REPLACE`.

Conclusions

OpenGL 4.1 has reach Earth in a quite different way I expected it. I expected a OpenGL 4 hardware oriented release but it's definetly not the case and actually it comes closer to my wish-list than my expectations.

The BIG missing feature is the overload buffer and image load and store embody by GL_NV_shader_buffer_load,GL_NV_shader_buffer_store and GL_EXT_shader_image_load_store extensions. An OpenGL 4.2 release with only these features would already be amazing!

Unfortunately, OpenGL 4.1 doesn't include GL_EXT_direct_state_access as a lot of people would expect. However, AMD as release drivers supporting this extension so that I presume that the ARB has an agreement on the goods of this approach. Moreover, OpenGL 4.1 integrates through GL_ARB_separate_shader_objects a subset of the direct state access extension. Thus, I think we have here an idea on how the ARB want to bring this approach in core. I don't think that the deprecated feature will ever have direct state access in the OpenGL Compatibility profile. From my point of view, this would be for the best. For the core features, the direct state access will probably reach core through new extensions.

Finally an angry word about the lack of conscistency of this new 4.1 specification. It's almost

unbelievable! Sometimes it's so obvious that I stay stunned. I might be able to find in every extension something which doesn't follow the OpenGL convensions. This is a growing concern I have seen the OpenGL 3.0 release. OpenGL is a specification where every single word and API token mean something and it actually defines in the specification... at least, it uses to be. For example, OpenGL 3.0 introduces "geometry shader" while this stage was already cleary defines in the specification by "primitive". What is a "geometry" in OpenGL? Don't ask me I have no idea, it's undefined. OpenGL 4.1 is now using the token location and index for about everything, confronted to an API declarations or a specific paragraph it happens very often that the uses of some tokens is misleading. OpenGL 4.1 now introduces some new tokens like sub-pixel which comes from nowhere of arrays for a set of while arrays have been clearly defines since OpenGL 1.1. I quite believe that the overall "blocks" concept is mostly unspecified leading really different implementations on AMD and nVidia drivers. etc!!!

Download: OpenGL 4.1 core specification
Download: GLSL 4.1 specification
Link: OpenGL 4.1 man pages
Link: My OpenGL 4.0 review
Link: My OpenGL 4.1 wish-list