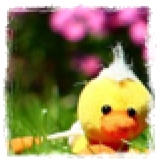
OpenGL 4.0 review

23 March 2010, Christophe Riccio



Introduction

OpenGL 4.0 is the new specification for GeForce GTX 400 and Radeon HD 5000 hardware series: aka Direct3D 11 hardware level. In this review we go through all the new features and explore their details.

1. Tessellation

OpenGL 4.0 extend the rendering pipeline with three new stages that take place between the vertex shader and geometry shader:

Control shader (Known as Hull shader in Direct3D 11)
Primitive generator
Evaluation shader (Known as Domain shader in Direct3D 11)

In a way, the tessellation stages replace the vertex shader stage in the graphics pipeline. Most of the vertex shader tasks will be dispathed in the control shader and the evalution shader. So far, the vertex shader stage is still required but the control shader and the evaluation shader are both optional.

Control shaders work on 'patches', a set of vertices. Their output per-vertex data and per-patch data used by the primitive generator and available as read only in the evaluation shader.

Input per-vertex data are stored in an array called `gl_in` which maximum size is `gl_MaxPatchVertices`. The elements of `gl_in` contain the variables `gl_Position`, `gl_PointSize`, `gl_ClipDistance` and `gl_ClipVertex`. The per-patch variables are `gl_PatchVerticesIn` (number of vertices in the patch), `gl_PrimitiveID` (number of primitives of the draw call) and `gl_InvocationID` (Invocation number).

The control shaders have a `gl_out` array of per-vertex data which members are `gl_Position`, `gl_PointSize` and `gl_ClipDistance`. They also output per-patch data with the variables `gl_TessLevelOuter` and `gl_TesslevelInner` to control the tessellation level.
A control shader is invoked several times, one by vertex of a patch and each invocation is identified by `gl_InvocationID`. These invocations can be synchronised by the built-in function barrier.

The primitive generator consumes a patch and produces a set of points, lines or triangles. Each vertex generated are associated with (u, v, w) or (u, v) position available in the evaluation shader thanks to the variable `gl_TessCoord` where u + v + w = 1.
The evaluation shaders provide a `gl_in` array like control shaders. The members of the elements of `gl_in` are `gl_Position`, `gl_PointSize` and `gl_ClipDistance` for each vertex of a patch. The evaluation shaders have the variables gl_PatchVerticesIn and `gl_PrimitiveID` but also some extra variables `gl_TessLevelOuter` and `gl_TesslevelInner` which contain the tessellation levels of the patch.

The evaluation shaders output `gl_Position`, `gl_PointSize` and `gl_ClipDistance`.
Tessellation has a lot more details to understand to work on a real implementation in a project! Those details are available in GL_ARB_tessellation_shader and obviously in OpenGL 4.0 specification. I think the API need some refinements but provides enough to start having fun with tesselation.

2. Subroutine

Subroutines are defined by GL_ARB_shader_subroutine as part of OpenGL 4.0 specification. This mechanism is some sort of C++ function pointer which allows to select, from the C++ program, a specific algorithm to be used in a GLSL program. This feature is a great enhancement for the 'uber-shader' type of software design where all the algorithms are included in a single shader to handle multiple/every cases. Subroutines allow to select specific shader code-pathes but also to keep the same program and program environment.

The following GLSL code sample defines 3 subroutine uniforms, which means 3 entries to change a shader behaviour. Several functions can be defined for a subroutine and a single subroutine function can be used for multiple subroutine uniforms. Subroutine function can't be overloaded. Subroutine uniforms are the sort of 'function pointer' but can only 'point' on subroutine functions.

Subroutine in GLSL 4.00:

```
subroutine vec4 greatFeature(in vec3 Var1, in vec3 Var2);
subroutine vec4 bestFeature(in vec3 Var1, in vec3 Var2);
subroutine mat4 otherFeature(in vec4 Var1, in float Var2, in int var3);

subroutine(greatFeature, bestFeature)
vec4 myFeature1(in vec3 Var1, in vec3 Var2)
{ ... } // Required function body

subroutine(greatFeature, bestFeature)
vec4 myFeature2(in vec3 Var1, in vec3 Var2)
{ ... } // Required function body

subroutine(bestFeature)
vec4 myBestFeature(in vec3 Var1, in vec3 Var2)
{ ... } // Required function body

subroutine(otherFeature)
subroutine mat4 myOtherFeature(in vec4 Var1, in float Var2, in int var3);
{ ... } // Required function body

// Could be set to myFeature1, myFeature2
subroutine uniform greatFeature GreatFeature;
// Could be set to myFeature1, myFeature2, myBestFeature
subroutine uniform bestFeature BestFeature;
// Could be set to myOtherFeature only...
// probably not a recommanded use of subroutines...
subroutine uniform otherFeature OtherFeature;

void main()
{
       // Subroutine uniform variables are called the same way functions are called.
       GreatFeature();
       …
       BestFeature();
       …
       OtherFeature();
}
```

The subroutine uniforms are assigned using the function `glUniformSubroutinesuiv` which parameters define the list of the subroutine functions used set to all subroutine uniforms. To get the subroutine function locations, OpenGL provides the function `glGetSubroutineIndex`.

3. Transform feedback

Transform feedback is the OpenGL name given to Direct3D output stream. It allows to capture processed vertices data before rasterisation and to be more accurate, just before clipping. A first extension proposed by nVidia (GL_NV_transform_feedback) has been promoted to GL_EXT_transform_feedback and finally included in OpenGL 3.0 specification.

From where we come (OpenGL 3.0):

```
char * Strings[] = {"Position", "Color"};
glTransformFeedbackVaryings(Program, 2, Strings, GL_SEPARATE_ATTRIBS);
glLinkProgram(Program);
...
glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 0, PositionBuffer);
glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 1, ColorBuffer);
// No rasterisation will bit performed, optional.
glEnable(GL_RASTERIZER_DISCARD);
...
glBeginQuery(GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN, Query);
glBeginTransformFeedback(GL_TRIANGLES);
glDrawElement(...);
glEndTransformFeedback();
glEndQuery(GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN);
...
// Query the number of primitives written in the transform buffer.
glGetQueryObjectuiv(Query, GL_QUERY_RESULT, &PrimitivesWritten);
```

This is basically how transform feedback works with OpenGL 3.0. `glTransformFeedbackVaryings` is a program state that must to be called before GLSL program linking. The last parameter can be either `GL_SEPARATE_ATTRIBS` or `GL_INTERLEAVED_ATTRIBS`. `GL_SEPARATE_ATTRIBS` is used to save each transform feedback varying in different buffers and `GL_INTERLEAVED_ATTRIBS` is used to save all transform feedback varying in the same buffer.

`GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN` is a query that can be used to get the number of primitives actually written in the transform feedback buffer(s). When data are captured from a vertex shader and the feedback primitive is the same as the drawn primitive, the number of primitives written in the buffer is likely to be the same as the number of primitives sent at draw call but transform feedback has such flexibility that transform feedback primitive can be different than draw call primitive.

Furthermore, transform feedback can capture geometry shader output. As geometry shader can generate or discard primitives, which output vertrices count become unpredictable. Transform feedback buffers can be used as vertex data of further draw calls where the vertrices polygon count might define the draw call primitive count. If you repeat a series of geometry shader and transform feedback, we might have a tessellator... but a really slow useless one!

This work has been followed by some more work by nVidia in the form of GL_NV_transform_feedback2 to finally give us 4 new extensions in OpenGL 4.0 which push the transform feedback boundaries.

- GL_ARB_transform_feedback2
- GL_ARB_transform_feedback3

GL_ARB_draw_indirect (Partially)
GL_ARB_gpu_shader5 (Partially)

GL_ARB_transform_feedback2 defines 3 features. First, it creates a transform feedback object (sometime called XBO) that encapsulates the transform feedback states... Well, that is to say the transform feedback buffers which with `GL_INTERLEAVED_ATTRIBS` is just 1 buffer... what's the point?!

This object allows to pause (`glPauseTransformFeedback`) and resume (`glResumeTransformFeedback`) transform feedback capture. XBO manages a behaviour 'state'. This way, multiple transform feedback objects can record the vertex attributes, one after the other but never at the same time. In an OpenGL command flow, we can imagine that some draw calls belong to one transform feedback and others belong to a second transform feedback.

Finally, this extension provides the function `glDrawTransformFeedback` to use transform feedback buffers as vertex shader source without having to query the primitives written count. When querying this count with `glGetQueryObjectuiv`, the function is going to stall the graphics pipeline waiting for the OpenGL commands to be completed. `glDrawTransformFeedback` replaces `glDrawArrays` in this case and doesn't need the vertices count, it's going to use automatically the count of written primitives in the transform feedback object to draw. GL_ARB_transform_feedback2 is part of OpenGL 4.0 but is also supported by GeForce GT200 as an extension.

GL_ARB_transform_feedback3 defines 2 features. First, with OpenGL 3.0 when we capture varying we are limited by 2 dispached methods: `GL_SEPARATE_ATTRIBS` to write a varying per buffer and `GL_INTERLEAVED_ATTRIBS` to write all the varyings in a single buffer.

GL_ARB_transform_feedback3 proposes a much more realistic scenario: It allows to write interleaved varyings in several buffers. Let's take an example. A transform feedback object could contains 3 buffers. The first buffer could capture 1 varying. The second buffer could capture 3 varying. and the third one could capture 2 varyings. This behaviour is defined with a simple very approach: In the name list given to `glTransformFeedbackVaryings`, we insert the name `gl_NextBuffer` as a separator between buffer.

Also, this extension has some interactions with GL_ARB_gpu_shader5 which defines multiple vertex streams in geometry shaders. Multiple vertex streams is a new concept for OpenGL 4.0. In a way, before OpenGL 4.0 we had a single vertex streams which was use by the rasterizer. The first vertex stream is still used by the raterizer but the others can be output to transform feedback objects. Such possibility requires an extra set of functions to query the written primitives per stream and to be able to draw directly using a specific vertex stream. This is done with `glDrawTransformFeedBackStream`, `glBeginQueryIndexed`, `glEndQueryIndexed` and `glGetQueryIndexediv`.

Where we are now (OpenGL 4.0):

```
// Create a transform feedback object
GLuint Feedback = 0;
glGenTransformFeedbacks(1, &TransformFeedback);
glBindTransformFeedbacks(GL_TRANSFORM_FEEDBACK, TransformFeedback);
glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 0, PositionBuffer);
glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 1, ColorBuffer);
```

```
glBindTransformFeedbacks(GL_TRANSFORM_FEEDBACK, 0);
...
// Setup transform feedback before linking
char * Strings[] = {"Position", "gl_NextBuffer", "Diffuse", "Specular"};
glTransformFeedbackVaryings(Program, 3, Strings, GL_INTERLEAVED_ATTRIBS);
glLinkProgram(Program);
...
// Draw and capture the streams
glBindTransformFeedbacks(GL_TRANSFORM_FEEDBACK, TransformFeedback);
glEnable(GL_RASTERIZER_DISCARD);
glBeginTransformFeedback(GL_TRIANGLES);
glDrawElement(...);
glEndTransformFeedback();
...
// Bind a stream buffer
glBindBuffer(GL_ARRAY_BUFFER, ColorBuffer);
glVertexAttribPointer(DiffuseLocation, 4, GL_FLOAT, GL_FALSE,
       sizeof(glm::vec4) * 2, 0);
glVertexAttribPointer(SpecularLocation, 4, GL_FLOAT, GL_FALSE,
       sizeof(glm::vec4) * 2, BUFFER_OFFSET(sizeof(glm::vec4)));
glBindBuffer(GL_ARRAY_BUFFER, 0);
...
// Draw the stream buffer without written primitives query
glDrawTransformFeedbackStream(GL_POINTS, TransformFeedback, 1);
```

Extra of the geometry shader:

```
layout(out = 0) out vec4 Position;
layout(out = 1) out vec4 Diffuse;
layout(out = 1) out vec4 Specular;
```

GL_ARB_draw_indirect provides new draw call functions (`glDrawArraysIndirect` and `glDrawElementsIndirect`) and a new buffer binding point called `GL_DRAW_INDIRECT_BUFFER`. They behave the same way than `glDrawArraysInstancedBasedVertex` and `glDrawElementsInstancedBasedVertex` except that the parameters are read from a buffer binded at `GL_DRAW_INDIRECT_BUFFER` point. This buffer could be generated by transform feedback or another APIs (OpenCL / Cuda) which avoid an undesired read back of GPU memory which would stall the rendering pipeline.

Format of the indirect buffer for arrays draw call:

```
typedef struct
{
       GLuint count;
       GLuint primCount;
       GLuint first;
       GLuint reservedMustBeZero;
} DrawArraysIndirectCommand;
```

Format of the indirect buffer or elements draw call:

```
typedef struct
{
       GLuint count;
       GLuint primCount;
       GLuint firstIndex;
       GLint baseVertex;
       GLuint reservedMustBeZero;
} DrawElementsIndirectCommand;
```

4. Geometry shader (r)evolution

OpenGL 4.0 Geometry shader provides streams and also a great improvement of this programmable stage: Geometry instancing. Where others OpenGL instancing techniques execute the entire graphics pipeline for each instance, this functionnality allows to run multiple times the geometry shader, each run being identified by `gl_InvocationID`. The number of time the geometry shader is invoked is indicated inside the geometry shader using the new input layout qualifier.

Geometry shader input layout qualifier:

```
	layout(triangles, invocations = 7) in;
```

The first parameter in the input layout is the input primitive type which can be `points`, `lines`, `lines_adjacency`, `triangles` and `triangles_adjacency`.

Geometry shader also provides new required output layout qualifiers.

Geometry shader input layout qualifier:

```
	layout(triangle_strip, max_vertices = 76, stream = 0) out;
```

This layout defines the geometry shader output primitive, points, line_strip or triangle_strip and the maximum number of vertices the shader will ever emit in a single invocation. The maximum value is `gl_MaxGeometryOutputVertices`. stream declares the default stream and can be different to 0 only when the output primitive is points. stream number can be declare in the global scoop, for a block or a non-block output variable. Vertrices and primitives are emited to specific streams using the GLSL functions `EmitStreamVertex` and `EndStreamPrimitive`.

This is just the changes in geometry shader, a few of all changes that GLSL 4.0 includes.

5. GLSL 4.0

GLSL 4.0 evolves a main change in the programming policy. Where GLSL until GLSL 3.3 was all about explicit conversions, GLSL 4.0 provides implicit conversions. It implies a lot of rules to define these conversions ... I'm really not sure about it, about how good this is for GLSL. At least a consequence is that GLM will use implicit conversion in a future version, a request widely requested which is probably why GLSL made this switch.

GLSL 4.0 introduces a new qualifier call precise that can be use to any variable to ensure that the result of an operation is performed the way the code request it. Compiler performs a lot of optimisations that can affect the result with a little lack of precision for a high performance benefit. precise effectively avoid those optimisations.

GLSL 4.00 extends the set of built-in functions with new integer functions (mainly bitfield manipulation functions), floating-point pack and unpack functions and extends the list of common functions.

6. Adaptative multisampling and per-sample processing

Thanks to GL_ARB_sample_shading and some GLSL 4.00 functionalities defined in GL_ARB_gpu_shader5, OpenGL 4.0 gives us much more control on multisampling.

With GL_ARB_sample_shading, the programmer can force the minimum number of samples that will be compute independently. To be efficient, most implementation share some values between samples like texture coordinates so that a texture fetch can be reused for every samples of a fragment. For example, in case of alpha test based on alpha texture, this behaviour can introduce aliasing. A problem quite obvious in Crysis.

The function `glMinSampleShading` is used to set this minimum number of samples. In GLSL, it gives us several built-in variables: in int `gl_SampleID` the number of the sample, a value between 0 and `gl_NumSamples` - 1uniform int `gl_NumSamples` is the total number of samples in the framebuffer; `gl_SamplePosition` the position of the sample in the pixel (between 0.0 and 1.0 where 0.5 is the pixel center); `gl_SampleMask` is used to changed the coverage of a sample, to exclude some samples from further fragment processing but it will never enable uncovered samples.

GL_ARB_gpu_shader5 provides further per-sample controls regarding how in/out data are interpolated using qualifiers. When centroid is used to qualify a variable, a single value can be assigned to that variable for all the samples in the pixel. However, when sample qualify a variable, a separate value must be assigned to that variable for each covered sampled.

New built-in interpolation functions `interpolateAtCentroid`, `interpolateAtSample` and `interpolateAtOffset` are available to compute interpolated value of a fragment shader input variable. `interpolateAtCentroid` will return the value of a variable a centroid location, `interpolateAtSample` at sample location and `interpolateAtOffset` at an offset location from the pixel center where (0, 0) is the center of the pixel. If an input variable is declared with the qualifier `noperspective`, the interpolation is computed without perspective correction.

7. Direct3D 10.1 features

GL_ARB_draw_buffers_blend extended per-colorbuffer blending with per-colorbuffer blend equation with the new functions `glBlendEquationi`, `glBlendEquationSeparatei`, `glBlendFunci` and `glBlendFuncSeparatei`.

```
// Reset the initial blend states of a specific colorbuffer.
void ResetColorbufferBlend(GLuint Colorbuffer)
{
	glDisablei(GL_BLEND, Colorbuffer);
	glColorMaski(Colorbuffer, GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
	glBlendFunci(Colorbuffer, GL_ONE, GL_ZERO);
	glBlendEquationi(Colorbuffer, GL_FUNC_ADD);
}
```

GL_ARB_texture_cube_map_array extended texture array to cube maps.

8. Programmable filtering

What is really about with this sampler object? It's about "die Die DIE fixed function sampler!". OpenGL 4.0 hardware brings an efficient programmable filtering.
This is possible thanks to 3 extensions GL_ARB_texture_gather, GL_ARB_texture_query_lod and GL_ARB_gpu_shader5.

GL_ARB_texture_gather is actually a Direct3D 10.1 hardware feature and provides the function textureGather. This function determines the "*2x2 footprint that are used for linear filtering in a texture lookup and it returns a vector consisting of the first component from each of the 4 texels in the footprint*". Nice... but very limited.

GL_ARB_gpu_shader5 extends this functionality so that we get a fully programmable texture filtering. The sampler object is no longer useful. It allows to gather any component of a 2x2 footprint. It allows to use an arbitrary offset to select the footprint and even extend this functionality to a per-component offset. Finally, it allows to perform a per-sample depth comparison.

Finally, GL_ARB_texture_query_lod defines the function `textureQueryLod` to query, in GLSL, the LOD that would be computed if a texture look up was performed.
With such feature, we can perform a per-fragment adaptive texture filtering. "Anisotropic filtering 16x" is no longer a meaningful concept.

9. Double-precision floating-point support aka FP64

GL_ARB_gpu_shader_fp64 is part OpenGL 4.0 and exposes support of FP64 uniforms and FP64 computations in GLSL. I'm really surprised and actually fairly sceptical about this choice. All Direct3D 11 cards doesn't have hardware support for doubles especially on AMD side. It quite makes sens as few users would need it. On the Radeon HD 5XXX series, the only cards that have double support are the Radeon HD 5830, Radeon HD 5850, Radeon HD 5870 and Radeon HD 5890 based on RV870 chip.

AMD said it will support OpenGL 4.0 on the entire Radeon HD 5XXX line which implies that double float will be emulated and, as a consequence, really slow... Well, what is the meaning of "Really slow"? AMD claims at FireStream 9170 release (a pro RV670 / Radeon HD 3870 card) that they emulate double float with 2 single floats and that this result in 1/2 to 1/4 of the single precision performance. According to a code to emulate FP64 on GPU with Cuda I found on nVidia forums, I would say that emulated FP64 would be about 10 to 20 times slower than FP32.

GL_ARB_gpu_shader_fp64 is a great extension. I just think it should have stay as an extension but I'm not sure it would be any problem in practice anyway as the use cases for FP64 are fairly limited, especially with OpenGL. Considering the GPUs that support FP64, we notice that this is the kind of feature that doesn't follow GPU generations but high-end graphics in general. The OpenGL 3.X cards that support hardware FP64 (and probably GL_ARB_gpu_shader_fp64) are Radeon HD 4770, 4830, 4850, 4870, 4890, 4850 X2, 4870 X2; GeForce GTX 260, 275, 280, 285, 295.

A double addition emulation with 2 floats by Norbert Juffa (nVidia):
```
vec2 dblsgl_add(vec2 x, vec2 y)
{
	vec2 z;
	float t1, t2, e;
	t1 = x.y + y.y;
	e = t1 - x.y;
	t2 = ((y.y - e) + (x.y - (t1 - e))) + x.x + y.x;
	z.y = e = t1 + t2;
	z.x = t2 - (e - t1);
	return z;
}
```

A double multiplication emulation with 2 floats by Norbert Juffa (nVidia):
```
vec2 dblsgl_mul(vec2 x, vec2 y)
{
        vec2 z;
        float up, vp, u1, u2, v1, v2, mh, ml;
        up = x.y * 4097.0;
        u1 = (x.y - up) + up;
        u2 = x.y - u1;
        vp = y.y * 4097.0;
        v1 = (y.y - vp) + vp;
        v2 = y.y - v1;
        //mh = __fmul_rn(x.y,y.y);
        mh = x.y*y.y;
        ml = (((u1 * v1 - mh) + u1 * v2) + u2 * v1) + u2 * v2;
        //ml = (fmul_rn(x.y,y.x) + __fmul_rn(x.x,y.y)) + ml;
        ml = (x.y*y.x + x.x*y.y) + ml;
        mh = mh;
        z.y = up = mh + ml;
        z.x = (mh - up) + ml;
        return z;
}
```

10. More texture formats! But not so many actually

Almost all OpenGL release implies more texture formats. This is again the case with OpenGL 4.0 as 2 new extensions on that side has been released. GL_ARB_texture_buffer_object_rgb32 also part of OpenGL 4.0 specification and GL_ARB_texture_compression_bptc excluded from OpenGL 4.0 specification, probably just because of an S3 patent just like GL_EXT_texture_compression_s3tc
GL_ARB_texture_buffer_object_rgb32 trivially adds 3 channels 32 bits texture buffers: GL_RGB32I, GL_RGB32UI and GL_RGB32F.

GL_ARB_texture_compression_bptc provides Direct3D 11 compressed formats known as BC6H and BC7 and called respectively `GL_BPTC_FLOAT` and `GL_BPTC` with OpenGL. They aim high dynamic range, low dynamic range texture compression and high quality compression of sharp edges. The compression ratio for `GL_BPTC_FLOAT` and `GL_BPTC` are 6:1 and 3:1.

Conclusion

Considering that OpenGL 3.2 has been released about 6 months ago, that OpenGL 3.3 has been release at the same time than OpenGL 4.0, I would like to say that OpenGL 4.0 is the most impressive work ever done by the ARB even if we still miss some important features. It provides the same hardware level of features than Direct3D 11 but some concepts are missing (Command lists are missing but this is more a software king of think).

OpenGL 4.0 is:

A set of big new features: Tessellation, programmable filtering, subroutines, programmable multisampling.
A set of features refinements: Tranform feedback, geometry shaders, configurable offset screen rendering blending.
A set of small useful features: More compressed texture formats, cube map arrays.
A clear graphics cards range.
A good graphics API.

On the drivers side, all the work from the past 2 years by ATI is remarkable. The drivers are know solid on Windows. nVidia has still great drivers and has already published beta drivers for OpenGL 3.3. OpenGL 4.0 beta drivers should be available at GeForce GTX 480 launch on the 26th of March.

If it continues this way, that developer tools follow, I really see OpenGL making it comes back as main API in games even on Windows. There is a long way before this happened but good signs keep popping from everywhere just like the recent news announcing Steam on Mac with actual game running OpenGL.

OpenGL 3.3 and OpenGL 4.0 are good graphics APIs and I think we are just 1 or 2 steps away from getting great APIs. I will cover, in the last series article, expectations (or wishes?) for OpenGL 3.4 and OpenGL 4.1 that might fill the gasp separating OpenGL from awesomeness!

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