OpenGL 4.4 review

29 July 2013, Christophe Riccio

G-Truc Creation

Table of contents

Introduction

OpenGL 4.4 specifications have been released at Siggraph 2013. It is composed of eight core extensions but none of them is really a big new feature. There are nice additions. The big announcements for OpenGL weren't the new core specifications but a set of ARB extensions designed for post-OpenGL 4 hardware and the availability of the conformance test for OpenGL 4.4.

Since OpenGL 3.0, the ARB has designed OpenGL specifications so that all the revisions of OpenGL for a major version would work on the same hardware. Each major OpenGL number defines what we can call a hardware level. Hence, OpenGL 3 hardware can all implement OpenGL 3.0, OpenGL 3.1, OpenGL 3.2 and OpenGL 3.3. Of course, it's the same logic for OpenGL 4 hardware. There is a reason for that: Reducing the ecosystem fragmentation. Sadly, implementing OpenGL takes a lot of time so effectively the ecosystem remains fragmented because all vendors are not catching up. Intel (Linux) with OpenGL 3.1, Apple with OpenGL 3.2, Intel (Windows) with OpenGL 4.0, AMD with OpenGL 4.3 and NVIDIA with OpenGL 4.4 at this date.

At Siggraph 2013, the Khronos Group has released ARB extensions that can't be supported by all OpenGL 4 hardware, which automatically makes them a glance at OpenGL 5 hardware level features. Does the future look exciting? Yes!

Furthermore, the Khronos Group has announced an OpenGL 4.4 conformance test that is said to be mandatory from OpenGL 4.4 onwards. This is better than the declaration of intention that we have every year at the BOF but we lack of details regarding the test coverage and hence the impact that the conformance test will have on OpenGL implementations. Considering the fragmentation of the implementation support, wouldn't it have been better to focus on an OpenGL 3.2 core profile conformance test? Or can we expect a great OpenGL 4.4 coverage?

The answer to that question is no in my opinion. There are the experiences of OpenGL ES 2.0 and OpenCL 1.x that are a lot less complex specifications. Basically all implementations ended up passing conformance quite quickly. If it's that easy, it is because the coverage is poor. If the test coverage was absolute then we could write code for one OpenGL implementation and guarantee that it works for all implementations. Our experiences tell us that this is not the case.

A full OpenGL 4.4 coverage conformance test would be amazing and even fantastic but the amount of work that this represents is so huge that it's an illusion to believe in this idea. Actually, if such test could actually exist, it would become so much easier and quicker to write an OpenGL implementation. So what's really the conformance test? A marketing tool? A minimum specification support? Somehow, I have higher interest for private initiatives like the work from DrawElements and Piglit to make a serious impact on drivers quality.

In this review we will cover each extension released, give some perspectives on those features before concluding on possible future OpenGL specifications and hardware.

1. A shift of the resource model

I have been focussing my attention on adding scene complexity for the passed three or four years and I still believe this will be archive thanks to multi draw indirect and what I call programmable vertex pulling. However, to consider this paradigm, we need to have an API for accessing all the resources we need in a single multi draw call. **ARB_sparse_texture** and **ARB_bindless_texture** are great steps into this direction.

1.1. ARB_sparse_texture

ARB_sparse_texture is a subset of **AMD_sparse_texture** enabling virtual texturing with seamless texture filtering. Thanks to this extension we can create 16K by 16K texel textures that memory isn't fully resident. In practice when we create a sparse texture, a large table of pointers to memory pages is allocated. The allocation of these memory pages is potentially an independent task performed with **glTexPageCommitmentARB** on a subsection of that texture.

The API has changed a bit from the original extension. It's worth noticing that all the shader queries are gone. Allocation and copy have been decoupled so that we are no longer required to pass data for a sparse texture that we are actually going to fill afterward.

Sparse textures can be used for sampling and rendering. Supported formats are not specified. Multisample textures are explicitly not supported. For others formats, including compressed and depth stencil formats, it's a matter of querying **GL_NUM_VIRTUAL_PAGE_SIZES_ARB**. Supporting depth formats is a serious advantage to do high-resolution shadow map generation.

This extension is a really nice new feature for OpenGL that Direct3D 11.2 has the good idea to inspire itself. We can regret that the ARB didn't take time to create an **ARB_sparse_buffer** extension at the same time.

This feature remains too limited as it only relies on the virtual addressing of GPUs but it is bound by the current texture filtering capabilities. A 16K x 16K x 2K (arrays) texture is nice but 1M x 1M x 1M sparse textures would be so much better. AMD supports texture 3D sparse textures however the tiles are only 2D planes on Southern Islands. This is inefficient and it doesn't make sparse texture on this platform a great candidate for sparse voxel storage. For OpenGL 5 hardware? Let's wish it!

ARB_sparse_texture
Expected hardware support: OpenGL 5 hardware, AMD Southern Islands, NVIDIA Fermi

1.2. ARB_bindless_texture

ARB_bindless_texture is a very similar extension to **NV_bindless_texture**.

If there is still something wrong with the OpenGL API it's the texture binding API. Bindless texture API doesn't fixed the current design but it provides an alternative that allows using about an infinite number of different textures simultaneously. I would be particularly interested to see some work on Ptex using this extension.

This is a great feature but then what about ARB bindless buffer?

ARB_bindless_texture
Expected hardware support: OpenGL 5 hardware, AMD Southern Islands, NVIDIA Kepler

1.3. ARB_seamless_cubemap_per_texture

OpenGL 3.2 and **ARB_seamless_cube_map** provide a state for sampling a cube map accessing multiple faces to avoid seams. This functionality is embodied by a global state that affects every cubemaps. If we want to use seamless cube map filtering for one cube map we need to call **glEnable(GL_TEXTURE_CUBE_MAP_SEAMLESS)**. If we want to don't want to use it on another texture, we need to call **glDisable(GL_TEXTURE_CUBE_MAP_SEAMLESS).** If we would to apply these two textures on a single mesh, then we need to do two rendering passes. **ARB_seamless_cubemap_per_texture** changes this behavior giving to each cube map texture and sampler a state to enable or not the seamless cubemap filtering so that we can in a single pass sample a single cube map both ways.

ARB_seamless_cubemap_per_texture
Expected hardware support: OpenGL 5 hardware, AMD Evergreen, NVIDIA Kepler

1.4. Immutable buffers (OpenGL 4.4, ARB_buffer_storage)

Searching performance, we might mislead ourselves very easily. Good performance for a real time software is not about having the highest average frame rate but having the highest minimum framerate by avoiding latency spikes. Sadly, most benchmarks don't really show the performance of hardware but how bad there are to do whatever they do and they will typically report average FPS ignoring the real user experience.

The OpenGL specifications provide buffer objects since OpenGL 1.5 and that includes the famous usage hints. Why is it called "hints"? Because an implementation can do whatever it wants with a buffer regardless of the hint set at the buffer creation. In average the drivers do a good job and increase the average frame rate however the behavior of the drivers is nearly unpredictable and it could decide to move a buffer at the worse time in our program leading to latency spikes.

The purpose of **ARB_buffer_storage** is to resolve this issue and make performance more reliable by replacing the usage hints by usage flags and creating immutable buffers.

Immutable buffers are buffer that we can't change the size or orphans. To create an immutable buffer we use **glBufferStorage** commands and because it is immutable we can't possible call **glBufferData** on that buffer.

If we want to modify a buffer we need to use the flag **GL_MAP_WRITE_BIT**. If we want to read it's content, we need to use the flag **GL_MAP_READ_BIT**. Only using these flags implies that we can't use **glBufferSubData** to modify an immutable buffer.

On January 2012 OpenGL drivers status, I was pointing out the use of rendering using a mapped buffer. While, not allowed by the specifications, it was working on AMD implementation to a certain point. With **ARB_buffer_storage**, the ARB has tackled this use case by introducing persistent client mapping through the flag **GL_MAP_PERSISTENT_BIT** and **GL_MAP_COHERENT_BIT**. **GL_MAP_PERSISTENT_BIT** simply ensures that the mapped pointer remains valid even if rendering commands are executed during the mapping.

GL_MAP_COHERENT_BIT opens opportunities to ensure that whenever the client of server side performs charges on a buffer, these changes will be visible on this other side.

So, no more hints on buffers? Unfortunately, these are not dead as two hints have been included: **GL_DYNAMIC_STORAGE_BIT** and **GL_CLIENT_STORAGE_BIT**. I guess if we want to be serious about OpenGL programming and performance level, we should never use **GL_DYNAMIC_STORAGE_BIT** and **GL_CLIENT_STORAGE_BIT**.

All in all, I think **ARB_buffer_storage** is an interesting extension and I am looking forward to get some feedbacks from people doing some serious buffer transfer / streaming (like Outerra team) to see if this extension hold its promises. It's unfortunate that **GL_DYNAMIC_STORAGE_BIT** and **GL_CLIENT_STORAGE_BIT** have been added and I am wondering whether it's because the buffer API is still not enough low level. I would have enjoyed specifying explicitly whether the memory allocated in the host, in accessible memory or device memory and even provided by the host like we can do with **AMD_pinned_memory**.

ARB_buffer_storage - OpenGL 4.4 core specification
Expected hardware support: OpenGL 3 hardware

2. A step further toward programmable vertex pulling

A common misconception in rendering is that draw calls are expensive. They are not. What's expensive is switching resources between draw calls as that introduces a massive CPU overhead while the draw call itself it just the GPU command processor launching a job. Hence, we need to avoid switching resources relying on batching and on shader based dynamically uniform resource indexing. I have largely cover this topic in my GPU Pro 4 chapter titled "Introducing the programmable pulling rendering pipeline". For that purposes, OpenGL introduces two new extensions: **ARB_shader_draw_parameters** and **ARB_indirect_parameters**.

2.1. ARB_shader_draw_parameters

This extension exposes three new built-in inputs to the vertex shader stage: **gl_BaseIntanceARB**, **gl_BaseVertexARB** that exposes the values passed in the draw commands but also **gl_DrawIDARB** that behaves for multi-draws just like **gl_InstanceID** for draw instancing. A massive difference between these new vertex shader inputs and **gl_InstancesID** is that there are dynamically uniform variables so that we can use them to address arrays of resources.

```
layout(binding = INDIRECTION) uniform indirection {
	int Transform[MAX_DRAW];
} Indirection;

layout(binding = TRANSFORM0) uniform transform {
	mat4 MVP[MAX_DRAW];
} Transform;

layout(location = POSITION) in vec3 Position;
layout(location = TEXCOORD) in vec3 Texcoord;

out gl_PerVertex {
	vec4 gl_Position;
};

out block {
	vec2 Texcoord;
} Out;

void main(){
	Out.Texcoord = Texcoord.st;
	gl_Position = Transform.MVP[Indirection.Transform[gl_DrawIDARB]] * vec4(Position, 1.0);
}
```

Listing 2.1.1: Use sample of **gl_DrawIDARB** to use a different matrix per draw in a multi draw call.

With the perspective of programmable vertex pulling, we could imagine removing the Position and Texcoord input variables and store them into a shader storage buffer and using **gl_BaseVertexARB** and **gl_VertexID** to fetch ourselves the vertex data.

ARB_shader_draw_parameters
Expected hardware support: OpenGL 5 hardware, AMD Southern Islands, NVIDIA Fermi

2.2. ARB_indirect_parameters

An issue of **ARB_multi_draw_indirect** is that it requires that we submit the number of draws from the CPU side, as **<drawCount>** is **glMultiDrawElementsIndirect** parameter. What if we use a compute shader to build the indirect multi-draw buffer? We need to query on the CPU side the number of draws stored in that buffer to feed the **<drawCount>** parameter: Very inefficient because we stall the CPU waiting on the query. Another workaround is to reserve a large buffer of **<drawcount>** elements and set to zero the primitive counts of each draw we want to skip. Unfortunately, according to my measurements, that solution is inefficient because GPUs are barely faster at skipping a draw than executing it.

The proposed solution is to add a parameter called **<maxdrawcount>** which value is sourced from an indirect parameter buffer. The maximum of executed draws becomes **min(drawcount, maxdrawcount)**. Why not only source **drawcount** from a buffer? Because some command processors needs to know from the CPU the **drawcount**.

ARB_indirect_parameters
Expected hardware support: OpenGL 5 hardware, AMD Southern Islands, NVIDIA Fermi

3. Decoupled per-SIMD and SIMD component execution

Branching is a very interesting topic with GPUs. I guess GPU design is essentially based on how we access memory and how multiple shader invocations diverge. After that, ALUs are just the cherry on the cacke. As a result, branching is a very important (if not the most important) topic when it comes to performance. For example, in AMD Southern Islands architecture, I found five different methods to handle branching.

Nevertheless, the OpenGL ARB has tackled this issue by considering how could we help the compiler to produce more efficient branching code. The proposed solution is exposed by **`GL_ARB_shader_group_vote`**, which provides the GLSL functions **`anyInvocationARB`**, **`allInvocationsARB`**, **`allInvocationsEqualARB`** to compare values across shader invocations.

ARB_shader_group_vote
Expected hardware support: OpenGL 4 hardware, AMD Evergreen, NVIDIA Fermi, Intel Ivy Bridge

4. Still reducing the CPU interventions

4.1. Multiple binding per calls

As mentioned earlier in this review, what's expensive is not submitting draw calls but switching resources. With **ARB_multi_bind** we can all the textures, all the buffers, all the texture images, all the samplers in one call, reducing the CPU overhead. I like the API because we no longer need the target parameter or selectors to bind a texture as shown in listing 4.1.1. Furthermore, derived texture states are also use to bind texture images.

```
// Binding textures with OpenGL 4.3
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, TextureNames[0]);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, TextureNames[1]);

// Binding textures with OpenGL 4.4
glBindTextures(0, 2, TextureNames);
```

Listing 4.1.1: Binding textures with OpenGL 4.3 and OpenGL 4.4

Maybe one limitation of this extension is that we must bind consecutives units.

ARB_multi_bind - OpenGL 4.4 core specification
Expected hardware support: Any OpenGL 3 hardware

4.2. Direct state access texture clearing

Clearing a texture with OpenGL 4.3 is the most cumbersome thing to do. We need to create a FBO with this texture attached and clear it as a framebuffer attachment. Ah! Not only it costs a lost of CPU overhead to create that FBO but this affects the rendering pipeline as we need to bind this FBO to clear it.

Fortunately OpenGL 4.4 introduces **glClearTexImage** to clear a texture image and glClearTexSubImage to clear a part of this texture image. This feature provides a nice interaction with sparse textures and image load and store.

ARB_clear_texture - OpenGL 4.4 core specification
Expected hardware support: Any OpenGL 3 hardware

5. Better interactions with the fixed pipeline functionalities

Looking at my OpenGL Pipeline Map, we see that GPUs do a lot of things using fixed function hardware. Having both programmable and fixed functions functionalities invites us at considering the interoperability between both. This is what does **ARB_query_buffer_object** by capturing directly query results into a buffer object that we can directly within shaders without CPU round trip.

```
// --- Application side ---
// Create a buffer object for the query result
glGenBuffers(1, &queryBuffer);
glBindBuffer(GL_QUERY_BUFFER_AMD, queryBuffer);
glBufferData(GL_QUERY_BUFFER_AMD, sizeof(GLuint), NULL, GL_DYNAMIC_COPY);

// Perform occlusion query
glBeginQuery(GL_SAMPLES_PASSED, queryId)
...
glEndQuery(GL_SAMPLES_PASSED);

// Get query results to buffer object
glBindBuffer(GL_QUERY_BUFFER_AMD, queryBuffer);
glGetQueryObjectuiv(queryId, GL_QUERY_RESULT, BUFFER_OFFSET(0));

// Bind query result buffer as uniform buffer
glBindBufferBase(GL_UNIFORM_BUFFER, 0, queryBuffer);
...

// --- Shader ---
...
uniform queryResult
{
	uint samplesPassed;
}

void main()
{
	...
	if (samplesPassed > threshold) {
		// Complex processing
		...
	} else {
		// Simplified processing
		...
	}
}
```

Listing 5.1: Usage sample of query buffers.

Furthermore, with query buffers, we can request the result of many queries at the same time by mapping the buffer instead of submitting one OpenGL command per query to get each result.

ARB_query_buffer_object - OpenGL 4.4 core specification
Expected hardware support: OpenGL 3 hardware, AMD Evergreen, NVIDIA Tesla

6. Stronger shader interface between shader stages

I have discussed many times about the issues with the GLSL shader interfaces and even ended up writing an article for OpenGL Insights about this topic. Since then, many things have changed and GLSL 4.40 shader interfaces are a lot more robust than what they used to be in GLSL 1.50. Unfortunately, these improvements come at the cost of complexity. In the following section we will cover the features provided by **`ARB_enhanced_layouts`** to OpenGL 4.4 core specifications.

- The shader interface is very complex topic with OpenGL that contains multiple subtopics:
  - Interface matching
  - Location definition
  - Fixed function and programmable inputs and outputs
  - Naming conventions

This topic is so complex that the ARB had to patch up the design on the way adding up more complicity. OpenGL 4.4 continues on this patch up tradition to cover most cases. A good pass of deprecation and a real vision for it would be welcome!

ARB_enhanced_layouts - OpenGL 4.4 core specification
Expected hardware support: OpenGL 3 hardware

6.1. Partial matching of shader interfaces using blocks for separate shader programs.

OpenGL 4.1 introduced separate shader stages to enable changing shader stage programs without affecting the bound program of another shader stage. Hence, separated programs aren't all linked together. The absence of linking raises the question of the communication between shader stages, how are handled shader outputs and inputs matching.

The rules for shader matching are fully described in **section 7.4.1 Shader Interface Matching** of the OpenGL 4.4 specification. Unfortunately, there is one case of matching that was not supported with OpenGL 4.3: Partial matching of block with separated programs.

There are two kind of matching:
- Full interface matching
- Partial interface matching

With full interface matching both side of the interface use the same definition. In this case, even if we don't link we can assume that the GLSL compiler is going to follow the same rules for each side so that the subsequent shader stage will find its inputs where it expects to find it.

With partial interface matching, some variables or blocks might be missing in the subsequent shader stage because the GLSL compiler notices that a particular variable isn't declared. As a result, the compiler can't generate a matching shader interface without linking. When linking, the linker has the two sides of the shader interface and it can use the variable names to do the matching. If an input variable is missing, it could even remove the output variable in the preceding shader stage and all the code used to give it a value. Linking partial matching interfaces can produce a more efficient code but this is at the cost of flexibility, it's a tradeoff.

```
out vec4 A;
out vec3 B;
out float C;
```
Listing 6.1.1: Example of a shader output interface matching by name.

```
in vec4 A;
in float C;
```
Listing 6.1.2: Example of a shader input interface matching by name.

Listing 6.1.2 shows an input shader interface that will match the output shader interface in listing 6.1.1 only if we link the two stages so that the compiler could do a matching my name.

What happen if we don't link? The compiler can't possibly guarantee that the partial matching shader interface will match. To resolve this issue and support partial matching with separated shader stages, the ARB has created the location qualifier for input and output variables. This qualifier says to the compiler: "Here is where to write and read this variable."

```
layout(location = 0) out vec4 A;
layout(location = 1) out vec3 B;
layout(location = 2) out float C;
```
Listing 6.1.3: Example of a shader output interface matching by location.

```
layout(location = 0) in vec4 M;
layout(location = 2) in float N;
```
Listing 6.1.4: Example of a shader input interface matching by location.

The name of the variables or the sequence of variables doesn't matter anymore because the location qualifier provides an abstraction of where to find the variables in memory. Just like for blocks, the layout location qualifier make variable naming conventions a lot easier because there is no longer a dependent chain of names from shader stage to shader stage. For more information, see The GLSL Shader Interfaces chapter in OpenGL Insights.

With OpenGL 4.3, we can also match interfaces using blocks.

```
out blockA // This is called the block name
{
	vec4 A;
	vec3 B;
} OutA; // This is called the instance name

out blockB // This is called the block name
{
	vec2 A;
	vec2 B;
} OutB; // This is called the instance name
```
Listing 6.1.5: Example of a shader output interface matching by block name.

```
in blockB
{
	vec2 A;
	vec2 B;
} InB;
```

Listing 6.1.6: Example of a shader input interface matching by block name.

What happen if we don't link? It is reasonable to expect that the compiler will pack the block memory but because blockA is missing in the input interface, the interface won't match.

Just like OpenGL 4.1 and **GL_ARB_separate_shader_objects** introduced the layout location qualifier for variables, OpenGL 4.4 and **GL_ARB_enhanced_layout** introduced the layout location qualifier for blocks as shown in listing 6.1.7 and 6.1.8.

```
layout(location = 0) out blockA
{
	vec4 A;
	vec3 B;
} OutA;

layout(location = 2) out blockB
{
	vec2 A;
	vec2 B;
} OutB;
```

Listing 6.1.7: Example of a shader output interface matching by block name.

```
layout(location = 2) in blockB
{
	vec2 A;
	vec2 B;
} InB;
```

Listing 6.1.8: Example of a shader input interface matching by block name.

6.2. Packed shader inputs and outputs

If you fully read section 6.1, you have already a good feeling about how complicated the shader interfaces can be. Sadly, there is a lot more to say!

In listing 6.1.7 and 6.1.8, I used 0 and 2 as the locations for the two blocks. Why not using 0 and 1? In that case OutB.A and OutA.B would use the same location for these two variables.

A location is not an index but an abstract representation of the memory layout. Basically, all scalar and vector types consume 1 location except dvec3 and dvec4 that may consume 2 locations. However, it doesn't mean that the OpenGL implementations necessarily consume 4 components when we use a float variable for example. Locations provide a way to locate where variables are stored but giving enough freedom to the compiler to pack the inputs and outputs the way it wants.

AMD GPUs are reported to have lower performance in tessellation and many people only point out the primitive rate of the GPUs, which is a lot lower than NVIDIA GPUs. However, I think it's a too quick view of the problem. A quick test with OpenGL shows that AMD implementation uses a less optimal packing of the output and input variables than NVIDIA. The specifications require that implementations could export up to 128 components per shader input and output interface. However, if we create an array of floats on AMD, the size of that array can't be bigger than 32. This is because each location will consume 4 components; hence consuming more memory when performing the tessellation.

GL_ARB_enhanced_layout gives a finer granularity control of how the components are consumed by the locations. Along with the location qualifier we have a new component layout qualifier to assign for each component of a single location multiple variables. The specifications give a good code sample (listing 6.2.1) to understand the feature.

```
// Consume W component of 32 vectors
layout(location = 0, component = 3) in float kueken[32];

// Consume X/Y/Z components of 32 vectors
layout(location = 0) in vec3 ovtsa[32];
```

Listing 6.2.1 Using the component layout qualifier

6.3. Future directions and deprecation for the input and output interfaces

The shader input and output interface might be one of the most complex aspects of OpenGL. On one side it gives a lot of opportunity for the implementation to optimize deeply the shaders and the memory passing between stages, on other side it is way too complicated for 99% of the OpenGL programmers: Matching rules, location assignment, component utilization, blocks, built-ins, linked or separated. WTF!

What about that for future specifications: Deprecating all that stuff and only output a single structure with no matching. If the shaders are linked then the implementation can do crazy optimizations. If we don't link we work by assumption and the entire structure is considered active.

```
out struct [STRUCT_NAME] // The structure name could be optional.
{
	position vec4 Position; // position would be a qualifier for built-in gl_Position
	flat vec4 Color;
	vertex Vertex;
} Out;
```

Listing 6.3.1: An idea for a simpler shader input and output interface

6.4. Offset and alignment in uniform and storage buffer.

Two new qualifiers have been added to uniform and storage blocks: offset and align. These qualifiers give a control per variable on they map the memory. Listing 6.4.1 gives a usage example of the offset qualifier.

```
uniform Block
{
	layout(offset = 0) vec4 kueken;
	layout(offset = 64) vec4 ovtsa;
};
```

Listing 6.4.1: Using the offset qualifier on a uniform buffer.

The value passed to the offset qualifier is expressed in bytes and it points to the memory used to store the variable in memory. If the compiler removes a variable between kueken and ovtsa because it is not an active variable, thanks to the offset qualifier on ovtsa we can guarantee that the variable will back the right memory.

The align qualifier provides guarantees that the variables will use the correct padding of the memory as shown by listing 6.4.2.

```
uniform Block
{
	vec4 kueken;
	layout(offset = 44, align = 8) vec4 ovtsa; // Effectively store at offset 48
};
```
Listing 6.4.2: Using the align qualifier on a uniform block.

6.5. Constant expressions for qualifiers.

With the new offset and align qualifiers but also with the location qualifier, there might be a rise of interest for applying a constant expression calculation on them which OpenGL 4.4 and **GL_ARB_enhanced_layout** provide.

Let's say that we want to do partial interface matching on the block in listing 6.5.1.

```
out block
{
	A a; // A is a structure
	B b; // B is a structure
	C c; // C is a structure
} Out;
```
Listing 6.5.1: A block with three structures used for partial matching

If we remove B in the corresponding input block, the shader interface won't match when using separated programs so we need to use the layout location qualifier as shown in listing 6.5.2.

```
out block
{
	layout(location = 0) A a;
	layout(location = LOCATION_OF_A) B b;
	layout(location = LOCATION_OF_A + LOCATION_OF_B) C c;
} Out;

// In the subsequent shader stage:
in block
{
	layout(location = 0) A a;
	layout(location = LOCATION_OF_A + LOCATION_OF_B) C c;
} In;
```
Listing 6.5.2: A block with three structures used for partial matching using the location qualifier.

Unfortunately, as reported in issue 1 of **GL_ARB_enhanced_layout**, the ARB sees as a non-use case the idea of a **locationof** equivalent for locations of **sizeof** so we need to preprocess the value of **LOCATION_OF_A** and **LOCATION_OF_B**. A classic **sizeof** doesn't seem to be a use case either even if now we have the offset and align qualifiers. This is a very annoying aspect of this extension. What's the use case for offset and align if we can't compute their value relative to others elements of the blocks?

6.6. Embedded transform feedback interface in shaders

With OpenGL 4.3 every single shader interface could reference its resource unit in the shader… every interface but one! The one used for transform feedback. This is done using new ugly namespaced qualifier as shown in listing 6.6.1.

```
layout(xfb_buffer = 0, xfb_offset = 0)  out vec3 var1;
layout(xfb_buffer = 0, xfb_offset = 24) out vec3 var2;
layout(xfb_buffer = 1, xfb_offset = 0)  out vec4 var3;
layout(xfb_buffer = 1, xfb_stride = 32) out;
```

Listing 6.6.1: Transform feedback setup in a geometry shader.

The process is actually simpler that the old API approach. We just need to bind our buffers to transform feedback unit and then set at which offset we will store the variable data. The stride qualifier is used to specify at which offset to output the next vertex.

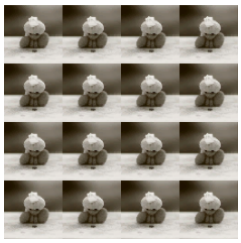
6.7. More embedded shader interfaces for resources?

If using the shader to specify where to store the data is a good idea, and it's rather a good one to me, why not providing also such design for the vertex shader inputs? It could be a good idea but this is missing that in hardware the vertex shader doesn't pull the vertex attributes it only receives a pulled vertex inputs. This implies that the transform feedback approach is necessarily limited and if this fixed function hardware feature had to support things like conversions, the shader would need to be switched to another one just to assume a tight packing of transformed feedback.

7. Texturing

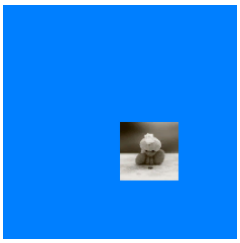
7.1. A new mirror clamp to edge texture coordinate mode

Some features seem to live forgotten by all until someone realized: "Hey, we all support it now, let's put it in core". This is what happens to **ARB_texture_mirror_clamp_to_edge** that has been promoted from **EXT_texture_mirror_clamp**, ifself promoted from **ATI_texture_mirror_one** and stripped down to one feature, a new wrap mode called **GL_MIRROR_CLAMP_TO_EDGE**. It looks like that **GL_MIRROR_CLAMP_TO_BORDER** has been removed because every vendor doesn't support it. A quick look at glCapsViewers database shows that AMD and NVIDIA support **EXT_texture_mirror_clamp** but not Intel. Why not an **ARB_mirror_clamp_to_border** extension only then? Or why not simply supporting it. Is that too much of a big deal to use an absolute instruction on the shader side to cover that mode?

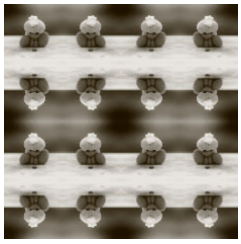


OpenGL 1.0
GL_REPEAT

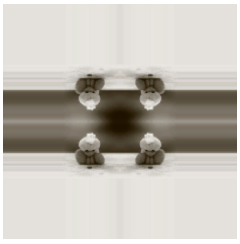
OpenGL 1.0
GL_CLAMP_TO_EDGE



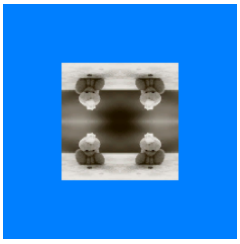
OpenGL 1.0
GL_CLAMP_TO_BORDER



OpenGL 1.4
ARB_texture_mirrored_repeat
GL_MIRROR_REPEAT



OpenGL 4.3
ARB_texture_mirror_clamp_to_edge
GL_MIRROR_CLAMP_TO_EDGE



EXT_texture_mirror_clamp
GL_MIRROR_CLAMP_TO_BORDER

Ultimately, the GPU texture units will remain fixed function at least for the 10 years to come. This is because a texture unit gathers many texels but only outputs a single filtered sample. Sending all these texels to the shader units would cost too much transistor just to convoy them and it consumes a lot more local registers. Using the **fetchTexel** instruction it is possible to program even an anisotropic sampling but this is magnitude slower according to my experience. However, GPU could already execute part of the texture coordinate wrapping on the shading units and I hope they do. There is the case of all the mirror-wrapping modes that we can implement on the shader side using listing 7.1.1 in combination with **GL_CLAMP_TO_EDGE** or **GL_CLAMP_TO_BORDER**.

```
// For mirror_repeat
float mirror_repeat(float Coord)
{
	return int(floor(Coord)) % 1 ? s - floor(Coord) : 1 - (s - floor(Coord));
}

// For mirror_to_egde and mirror_to_border
```

```
float mirror_once(float Coord)
{
	return abs(Coord);
}
```
Listing 7.1.1: Shader code to implement the mirror modes

GL_MIRROR_REPEAT: mirror_repeat with **GL_CLAMP_TO_EDGE**
GL_MIRROR_CLAMP_TO_EDGE: mirror_once with **GL_CLAMP_TO_EDGE**
GL_MIRROR_CLAMP_TO_BORDER: mirror_once with **GL_CLAMP_TO_BORDER**

My recommendation for the future: Just deprecate all the mirror modes, **GL_MIRROR_REPEAT**, **GL_MIRROR_CLAMP_TO_EDGE** and **GL_MIRROR_CLAMP_TO_BORDER** in the OpenGL API and in the hardware.

ARB_texture_mirror_clamp_to_edge - OpenGL 4.4 core specification
Expected hardware support: Any OpenGL 3 hardware

7.2. No more need for stencil renderbuffer

This extension is a good sense extension. The only case where we would want to use renderbuffer with OpenGL 4.3 would be to create a stencil only framebuffer attachment. With this new extension, we can create a stencil texture, which bury renderbuffers. They could have been useful for tile based GPUs but it seems that nobody cared to keep them special enough to legitimate their existence. Hence, there are deprecated in my mind and if I were to use them, it would be only to write cross API code between OpenGL ES 3.0 and OpenGL desktop.

ARB_texture_stencil8 - OpenGL 4.4 core specification
Expected hardware support: Any OpenGL 3 hardware

8. Misc

ARB_compute_variable_group_size

The purpose of this extension is simply to specify the sizes of a workgroup at dispatch time instead of compile time. This is an OpenCL 1.2 features that is not effectively implemented by AMD implementation. Also it sounds like an interesting feature, the implementation needs to be capable to set this size without recompiling the shaders.

ARB_compute_variable_group_size
Expected hardware support: NVIDIA Fermi, Intel Ivy Bridge

ARB_vertex_type_10f_11f_11f_rev

This extension exposes a new vertex format called GL_UNSIGNED_INT_10F_11F_11F_REV which is the same format used by texture already. It could be useful to store high precision compressed normal for example.

We can regret that no feature allows storing and reading packed formats from uniform or shader storage buffers.

ARB_vertex_type_10f_11f_11f_rev - OpenGL 4.4 core specification
Expected hardware support: Any OpenGL 3 hardware

Conclusions

Is there room for an OpenGL 4.5 specification? Always!

Continuing to rant about not having a better support of direct state access sounds like a desperate cause 5 years after the release of `EXT_direct_state_access`. Maybe, but this vision for the API remains as legitimate as even. That issue has to be tackle even if it will be particularly disrupting for the OpenGL API.

I guess a massive work that needs to be undertaken is separating the sampler and the texture in the shader. Direct3D 11 requires 128 texture units but only 32 samplers but with OpenGL we are stock at 32 texture units because the sampler and the texture is a monolithic unit.

I am also still missing the concept of `sizeof` and `locationof` in GLSL to be able to set my location depending on the definition of structures and variable that I don't necessarily know or that might constantly change during the development process. We, OpenGL programmers, are used to use sizeof in C++ programs, so we will feel confortable working with sizeof in the shaders.

Subroutines could be improved. They are such a weak API that we still need to setup on the CPU side. With the multi draw indirect capabilities we have a great approach to indexing. With sparse textures and bindless textures we have resources to use for a large number of draws. However, we need have bad API to choose the shader code we want to execute in the shader. Subroutine could leverage that if we get a way to avoid the CPU interventions.

Some developers would like something like Direct3D 11 deferred context. That approach doesn't work so we shouldn't expect it. However, we could consider the idea of exposing the command buffer.

Finally, OpenCL continues moving forward with SPIR, an IR for OpenCL, we could picture many developers happy to have an equivalent for shader code robustness, obfuscation and compile time performances.

Deprecation for OpenGL 5 hardware?

Considering the idea of an OpenGL 5 hardware level we could imagine a new pass of deprecations:

- Deprecate vertex array object and the array buffer: Use shader storage buffer instead.
- Deprecate texture buffer object. Use shader storage buffer instead.
- Deprecate transform feedback buffer. Use shader storage buffer instead.
- Deprecate atomic counter buffer. Use shader storage buffer instead.
- Deprecate MIRROR wrapping modes.
- Deprecate the concept of locations for input and output variables.
- Deprecate compressed formats that have better alternatives.
- Deprecate mutable textures and buffers.
- Deprecate all conversions not directly supported by the DMA controllers.
- Deprecate renderbuffers.
- And probably many more!

Features for OpenGL 5 hardware level?

On the domain of OpenGL 5 hardware level features, the OpenGL ARB has strike in the recent past leaving Direct3D just following what OpenGL has been doing. This is the case for example of the "tiled resources" which is nothing more than `AMD_sparse_texture`. On other aspects the Direct3D 11 API doesn't even reach parity with OpenGL 4.4.

The new ARB extensions for OpenGL 5 hardware are great but we can expect more from Kepler and Southern Islands architectures. Could we have ARB sparse buffers and ARB bindless buffers?

Then there is feature that probably no available hardware can do: Could we have a new pixel shader stage following the fragment shader stage to do programmable blending. This is more a tile based rendering view of the GPU world but I quite believe that convergence between desktop and mobile will happen and that it will turn out to be a programmable tile based rendering architecture. There are actually three approaches to imagine programmable blending:

- 1/ In the fragment shader where output would be array where we accumulate processed fragment.
- 2/ In a dedicated pixel shader stage where the input would be a list of fragments.
- 3/ In a dedicated tile shader stage where we could access all the fragments within a tile.

All three methods require on-chip memory to be efficient but when we consider the 64 KB of LDS in AMD Southern Islands CU, it sounds like that we are no very far from enough on-chip memory.

Hardware vendors also need to do some works on their command processors. Target for OpenGL 5 hardware: Being capable to process 1 million draws per seconds. Draws need to become smaller to allow finer granularity culling and to reduce partially used large batches from brut-force processing.

`ARB_indirect_parameters` opens interesting perspectives but the feature might be too limited. We could imagine an approach where per-draw user-defined parameters could be defined to index resources in the shaders. Technically, we can only already do this by using vertex attributes with the divisor set to 1 and base instance set to 0. However in practice each vertex attributes are fetched per vertex, which make it quite wasteful. It's hard to imagine whether this approach could be efficiently implemented on any current hardware but probably not for a target of 1 million draws per seconds.

We need a good support for sparse resources in OpenGL 5 hardware. This means textures and buffers with shader queries to check the residence of memory pages and capable to address a terabyte of data, 1 million layers textures, 1 million by 1 million texel textures and a 3D tile order for texture 3D to allow more efficient storage of sparse voxels.

Last words

We are reaching the end of the evolution of the OpenGL 4/Direct3D generation with a pretty mature OpenGL 4.4 API. While that generation will remains for years a base line for productions, innovations through a new base line of GPU features are coming as the new ARB extensions rise to our attentions.

The medium term future of real time rendering is already written: The renderer will submit less than 10 draw calls per frames but generating thousands of draws for each of them, each indexing only the resources it needs thanks to sparse and bindless resources. Each draw will also dynamically select the

code it needs to execute with-in über-shaders reducing the total number of shaders to a few. Bandwidth remains more critical as ever which will reduce the size of the batch and multiply the draws for a finer granularity GPU culling.

Ultimately, I think that mobile and desktop GPU architectures will converge to a form of programmable tile based GPUs where the triangle lists would be created by a compute shader and assigned to a multi draw indirect buffer. Going for tiles could also open more opportunities for programmable blending to do for example on-chip order independent transparency (OIT). **`NV_blend_equation_advanced`** or **`AMD_blend_minmax_factor`** show recent interests in more blending programmability. My bet? All that could happen with Maxwell and I will keep my fingers crossed for it. 2014 should be another pretty interesting year!

- OpenGL 4.4 core specification
- GLSL 4.4 specification
- OpenGL 4.4 review
- OpenGL 4.3 review
- OpenGL 4.2 review
- OpenGL 4.1 review
- OpenGL 4.0 review
- OpenGL 3.3 review
- OpenGL 4.4 Pipeline Map
- OpenGL ES 3.0 Pipeline Map

A big thanks to Vladimir Hrincar for the detailed review of the review and Daniel Rákos for sending his comments on the new extensions.