OpenGL 4.3 review

6 August 2012, Christophe Riccio

G-Truc Creation

Introduction

I don't believe most in the OpenGL community were expecting to see OpenGL 4.3 being such an important update. I think the most notable aspect of this new version of the specification is the increase flexibility of the rendering pipeline thanks to four main extensions:

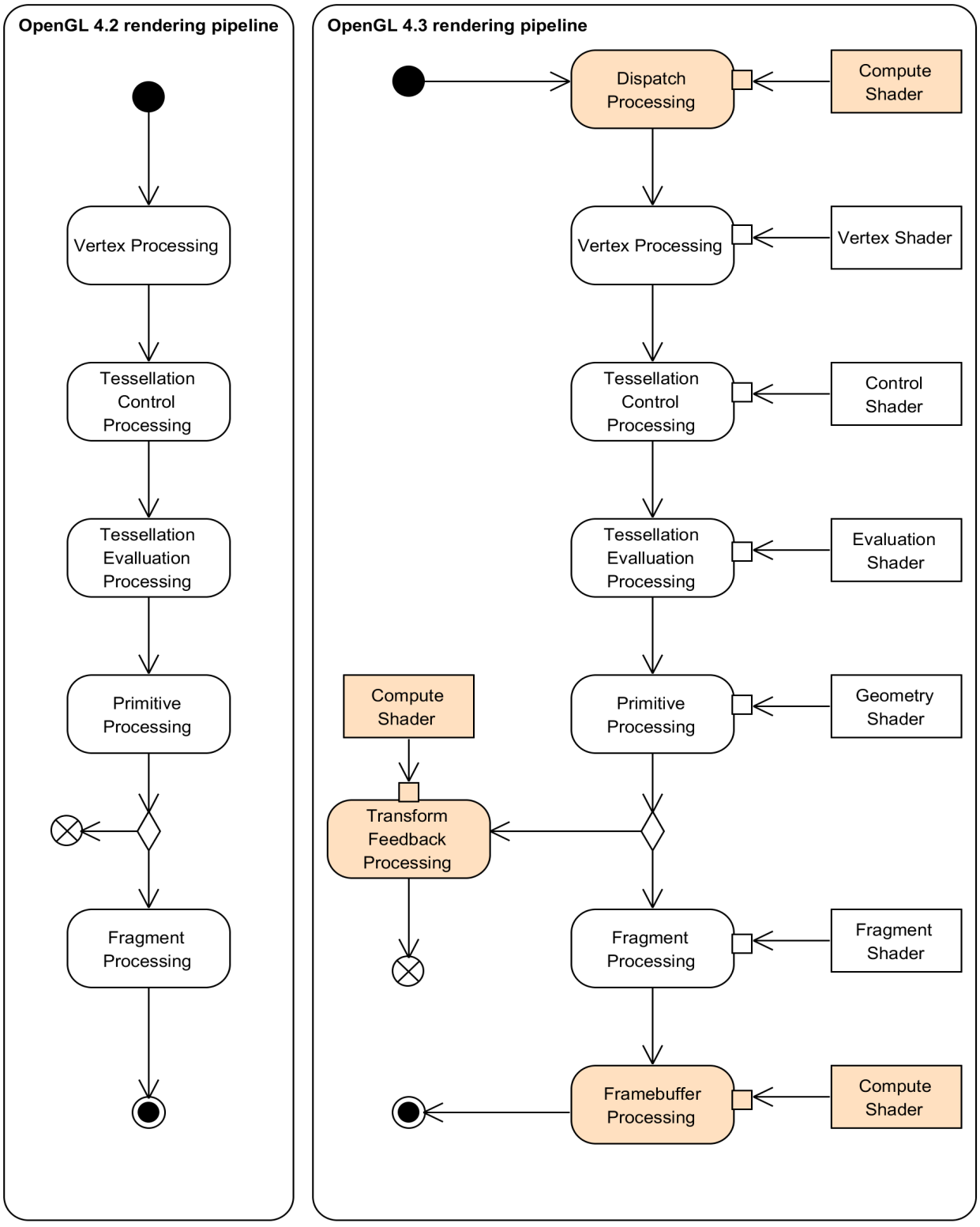
- ARB_compute_shader adding a compute shader stage
- ARB_shader_storage_buffer_object adding a read and write large array of structures
- ARB_multi_draw_indirect to dispatch a large number of draws in a single call
- ARB_framebuffer_no_attachment to execute the graphics pipeline without framebuffer

OpenGL 4.3 is also adding the debug output functionality to core through a new extension called KHR_debug designed only for OpenGL but also for OpenGL ES implementation and eventually WebGL. This specification will allows much easier debugging and profiling for applications for the whole ecosystem allowing programmers to spend less time doing these tasks and focusing in actually writing innovative software. KHR_debug is mainly a promoted version of ARB_debug_output but it also takes advantage of the OpenGL ES group ideas, including debug markers and debug labels. Also, on the level of debugging ARB_program_interface_query will be very useful to tackle the issue of *silent errors,* case where OpenGL is not reporting any error but the rendering is obviously wrong.

OpenGL is a very fragmented ecosystem probably for two reasons in my opinion: The ARB is releasing specifications every years and actually writing an OpenGL implementation is a very challenging and expensive work. This results in having only two OpenGL 4.2 available implementations from AMD and NVIDIA, an OpenGL 4.0 implementation by Intel, an OpenGL 3.2 implementation by Apple and an OpenGL 3.0 implementation for MESA. Working into this environment implies compromises from having multiple code paths to support all the platforms or lowering down to the common denominator. The OpenGL ARB has started tackling this issue by introducing ARB_internalformat_query2 making easier cross platform development on the sides of textures and framebuffers.

Work on memory is main topic for this version considering the addition of ARB_copy_image allowing to memcpy image tiles from GPU memory to GPU memory or ARB_texture_view that allows referencing a texture storage subset from multiple textures. Furthermore, with ARB_clear_buffer_object we can reset a buffer to a specific value. ARB_texture_buffer_range allows binding a range of a texture buffer. ARB_invalidate_subdata opens opportunities to avoid unnecessary data movement and ARB_stencil_texture enables sampling of the stencil framebuffer attachment.

Finally a lot of extension appears as bug fixes from previous extensions: ARB_shader_image_size allows querying the size of a texture images; ARB_texture_storage_multisample allows creating immutable multisample textures; ARB_texture_query_level exposes the GL_TEXTURE_MAX_LEVEL state in the shader stages; ARB_fragment_layer_viewport exposes gl_LayerID and gl_ViewportIndex in the fragment shader stage; ARB_arrays_of_arrays allows to declare arrays of arrays without declaring unnecessary complex data structures; ARB_explicit_uniform_location allows to declare location or index qualifier for uniform variables and subroutines; and last but not least ARB_vertex_attrib_binding finally decoupled the vertex format from the array buffer.



Evolution of the OpenGL Pipeline between OpenGL 4.2 and OpenGL 4.3

For this review, we will study in more details the improvements offered by OpenGL 4.3.

1. A more flexible rendering pipeline

1.1. Compute shader stage: GL_ARB_compute_shader

OpenGL has a new shader stage thanks to ARB_compute_shader that provides a lightweight compute capability to OpenGL. This stage can't be attached to a program pipeline that already has any other stage attached to it. Hence, it can be executed before a vertex program, after a transform feedback or a fragment program but never in between shader stages.

The OpenGL compute shader stage has several advantages over OpenCL. It natively support GLSL types (`vec*`, `mat*`) and the shader code can be reused across the compute shader stage and the others stages. All the infrastructures available with others stages like the sampler, the uniform block or the shader storage buffer is available in this stage. Only missing: the input variables (for vertex attributes) and the output variables (for framebuffer attachments). Also, just like OpenCL kernels, it supports a shared amount of memory, giving access to GPUs Local Data Store (LDS).

```
// work group dimensions
in    uvec3 gl_NumWorkGroups;
const uvec3 gl_WorkGroupSize;

// work group and invocation IDs
in    uvec3 gl_WorkGroupID;
in    uvec3 gl_LocalInvocationID;

// derived variables
in    uvec3 gl_GlobalInvocationID;
in    uint  gl_LocalInvocationIndex;
```

Built-in variables of a compute shader stage controlling the stage execution.

By itself, ARB_compute_shader is not really a big deal to me but when put into perspective with the OpenGL pipeline it starts to shine: OpenGL 4.3 enables Programmable Vertex Pulling and some forms of Programmable Blending!

I expect to see rising a issue with the compute shader. Today's GPUs are capable of performing a level of task parallelism on the GPU. However, because OpenGL has strict requirements for the execution order of the rendering it seems challenging to expect any GPU to be able to process multiple graphics command queues in parallel like we can safely execute multiple OpenCL kernels in parallel. Hence, it seems that it could be useful to be able to create OpenGL contexts only capable of compute processing.

1.2. Shader Storage buffer: GL_ARB_shader_storage_buffer_object

OpenGL 4.2 introduced two important extensions. First `ARB_shader_atomic_counters` exposing atomic counters in GLSL that are backed by buffers and second `ARB_shader_image_load_store` for reading and writing data from image (a texture level or a buffer) and even performing atomic operations on these resources. It was great but quite not enough and many not very practical so that OpenG 4.3 is rising with a new extension called `ARB_shader_storage_buffer_object`. Basically, it offers a new type of blocks called *shader storage block* that are very similar to uniform block except that:

- They are not limited to 64KB but 256MB.

- We can read from them but also write into them.
- They can contain atomic variables.
- The last element of the block can be an unsized array.

```
struct control
{
	int VertexFormatID;
};

struct vertex
{
	vec4 Position;
	vec3 Normal;
	vec3 Tangent;
	vec2 Texcoord;
};

buffer block
{
	control Control;
	vertex Vertices[];
};
```

A possible declaration for a shader storage block

Shader storage block doesn't implicitly deprecated uniform blocks because an implementation may perform optimization on uniform blocks that it would be impossible to do on shader storage blocks.

It seems that atomic counter operations can't be performed on shader storage buffer so that we still need to use dedicated buffers for them? Would it be possible to have atomic counter in shader storage block and even perform atomic counter operation on texture image?

1.3. ARB_multi_draw_indirect for Programmable Vertex Pulling

The idea hiding Programmable Vertex Pulling is composed with two main desires: We want to control the draw dispatched by the GPU on the GPUs and we want to build ourselves the inputs of a vertex shader. All in all Programmable Vertex Pulling is about taking control of the beginning of the pipeline by controlling the vertex invocations on the GPUs.

With OpenGL 4.2 the application is responsible to dispatch the *draws* to the GPU. With OpenGL 1.0 is was done using `glBegin/glEnd`, the immediate mode but quickly it appears that such approach building and sending vertex one by one was way too slow to be interesting because the GPUs were faster to consume the primitives than the CPU was able to submit them.

OpenGL has evolved many times to ensure that this submission could be done quick enough, Vertex Array (GL1.1), Vertex Buffer Object (GL1.5), Vertex Array Object(GL3.0), Base Vertex (GL3.2), Instancing (GL3.2), Base Instance (GL4.2) and probably more features that I am missing here.

By batching multiple VAOs into a single VAO by copying multiple sets of buffers into a single set of buffers and then calling `glDrawElementsInstancedBaseVertexBaseInstance` or `glDrawArraysInstancedBaseInstance` multiple times on that single VAO to draw the needed meshes we can already archive extremely good performances and the CPU overhead taken by this approach will be very low. However, the CPU remains in change of figuring out which meshes needs to be drawn for a specific frame which for many applications can occupy an entire CPU core…

Thanks to `ARB_multi_draw_indirect` and `ARB_compute_shader` it is possible to move this processing on the GPU, the compute shader becoming responsible to create a draw indirect buffer, which will store the parameters for multiple draw calls. This buffer will then be consumed by `glMultiDrawElementsIndirect` or `glMultiDrawArraysIndirect`.

The `glMultiDraw*Indirect` functions are nothing more than evolutions of the `glDraw*Indirect` introduced with OpenGL 4.0. Instead of processing a single draw per draw call, the new functions can submit many draws per calls.

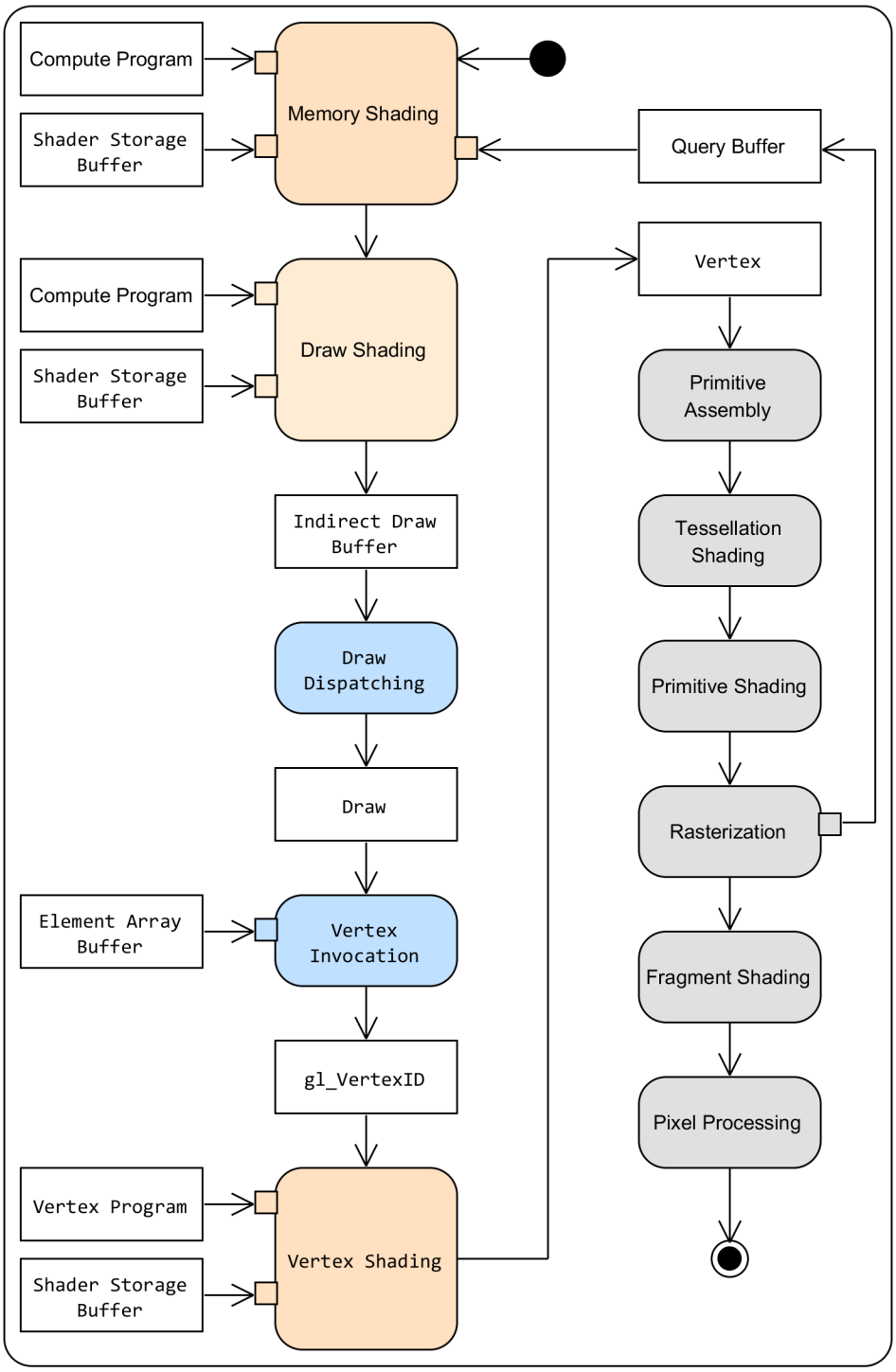
```
glBindBuffer(GL_DRAW_INDIRECT, BufferName);
for(std::size_t DrawIndex = 0; DrawIndex < DrawCount; ++DrawIndex)
{
	glDrawElementsIndirect(
		GL_TRIANGLES, GL_UNSIGNED_INT,
		BUFFER_OFFSET(Offset));
}
```

CPU draws dispatching

```
glBindBuffer(GL_DRAW_INDIRECT, BufferName);
glMultiDrawElementsIndirect(
	GL_TRIANGLES, GL_UNSIGNED_INT, NULL, GLsizei(drawCount), 0);
```

Command processor draws dispatching

A first usage of the shader storage buffer is when it is brought together with rendering without vertex attribute and multi draw indirect (ARB_multi_draw_indirect promoted from AMD_multi_draw_indirect). We can imagine that a compute shader can select in advance which parts of a mesh actually need to be rendered. This detection generated the multi draw indirect buffer, which will be used to pull specific draws from shader storage buffers that store a larger mesh. This method will be demonstrated in my GPU Pro article to be released.



A proposed Programmable Vertex Pulling rendering pipeline enabled by OpenGL 4.3, ARB_compute_shader, ARB_shader_storage_buffer_object and ARB_multi_draw_indirect

To improve Programmable Vertex Pulling, it could be interesting to revive the hardware subroutine functionality so that for each draw, the vertex could use a dedicated vertex format that various subroutine functions could implement. Could it be done? How should it be implemented? These are questions that would need some answers. An approach would be to have a subroutine buffer that would

be filled by a compute shader stage and for each draw, the command processor would automatically setup the right set of subroutines for all stages. Effectively, this is moving the subroutine set as a draw call parameter, which from an application point of view is close to what is currently happening due to the unpractical design of the subroutines API. Another possibility would be to provide the capability to index the subroutines. From the many conversations I had about AMD_multi_draw_indirect, it appears that a missing element is definitely a gl_DrawID, a capability to identify each draw, just like gl_InstanceID identify each instance. This gl_DrawID could be used to access to a uniform block for example that would store for each ID which subroutine needs to be selected.

It has been said before but it would be great to be able to store the number of draw stored into the indirect draw buffer because even if an implementation can discard a draw if the primitive count is zero, this is not really fast and it might force an application to execute a lot of empty draws if it doesn't want to go back to the CPU to submit the right number of draws that a compute shader stage figure said the rendering needed.

GLSL has relied on normalization to exposed vertex attributes of various types as vec3 in the vertex shader. This functionality is not available with shader storage buffers and I personally don't want of this dark magic. Instead, it would be great to expose the actually types even if the type is something specific like RGB10A2U. The shader would then be responsible to decide how it wants to interpret this variable maybe using some built-in functions.

1.4. Decoupled vertex format and vertex binding: GL_ARB_vertex_attrib_binding

If you have been reading G-Truc Creation, you probably now that I hate so much the Vertex Array Object and I believe it is the biggest mistake ever made in OpenGL. `ARB_vertex_attrib_binding` is kind of the vertex array object done right: The vertex format and the vertex array buffer are now separated. It should be easy to update an existing application to take advantage of `ARB_vertex_attrib_binding`.

```
struct vertex
{
	static GLuint const POSITION = 0;
	static GLuint const NORMAL = 1;
	static GLuint const TEXCOORD = 2;

	static GLuint const POSITION_OFFSET = 0;
	static GLuint const NORMAL_OFFSET = sizeof(glm::vec3);
	static GLuint const TEXCOORD_OFFSET = sizeof(glm::vec3) * 3;

	glm::vec3 Position;
	glm::vec3 Normal;
	glm::vec2 Texcoord;
};

glGenVertexArrays(1, &VertexArrayName);
glBindVertexArray(VertexArrayName);
	glVertexAttribFormat(
		vertex::POSITION, 3, GL_FLOAT, GL_FALSE, vertex::POSITION_OFFSET);
	glVertexAttribBinding(vertex::POSITION, 0);
	glEnableVertexAttribArray(vertex::POSITION);
	glVertexAttribFormat(
```

```
			vertex::NORMAL, 3, GL_FLOAT, GL_FALSE, vertex::NORMAL_OFFSET);
	glVertexAttribBinding(vertex::NORMAL, 0);
	glEnableVertexAttribArray(vertex::NORMAL);
	glVertexAttribFormat(
			vertex::TEXCOORD, 2, GL_FLOAT, GL_FALSE, vertex::TEXCOORD_OFFSET);
	glVertexAttribBinding(vertex::TEXCOORD, 0);
	glEnableVertexAttribArray(vertex::TEXCOORD);
glBindVertexArray(0);

for(std::size_t VertexFormatIndex(0); VertexFormatIndex < VertexFormatCount;
++VertexFormatIndex)
{
	glBindVertexArray(VertexArrayName[VertexFormatIndex]);
	for(std::size_t MeshIndex(0); MeshIndex < MeshCount; ++MeshIndex)
	{
		glBindVertexBuffer(0,
			BufferName[MeshIndex][buffer::VERTEX], 0, sizeof(vertex));
		glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,
			BufferName[MeshIndex][buffer::ELEMENT]);

		glDrawElementsInstancedBaseVertexBaseInstance(
			GL_TRIANGLES, ElementCount, GL_UNSIGNED_SHORT, 0,
			Instance[MeshIndex], BaseVertex[MeshIndex], 0);
	}
}
```

Efficient draw submissions on the CPU side by sorting draws by vertex formats

It's pretty unfortunate to see this extension being released now when not only the battle is finished but a new war as began with programmable vertex pulling where ultimately we won't even need vertex array objects, array buffers and maybe element array buffers.

1.4. Rendering without framebuffer attachment: GL_ARB_framebuffer_no_attachment

Thanks to OpenGL 4.2 and ARB_shader_image_load_store we can write a shader that will effectively never need to write to the framebuffer attachment. However, to setup the rasterizer, an application needs to create a framebuffer and an attachment. It is particularly embarrassing when we want to setup the rasterize for layered cube map rendering or HDR multisample rendering. The memory will be reserved by the drivers but never used! ARB_framebuffer_no_attachment resolves this issue. ARB_shader_storage_buffer_object can be used in place of ARB_shader_image_load_store in this context.

```
glGenFramebuffers(1, &Name);
glBindFramebuffer(GL_FRAMEBUFFER, Name);
glFramebufferParameteri(GL_FRAMEBUFFER, GL_FRAMEBUFFER_DEFAULT_WIDTH, Width);
glFramebufferParameteri(GL_FRAMEBUFFER, GL_FRAMEBUFFER_DEFAULT_HEIGHT, Height);
glFramebufferParameteri(GL_FRAMEBUFFER, GL_FRAMEBUFFER_DEFAULT_LAYERS, Layers);
glFramebufferParameteri(GL_FRAMEBUFFER, GL_FRAMEBUFFER_DEFAULT_SAMPLES, Samples);
glFramebufferParameteri(GL_FRAMEBUFFER
	GL_FRAMEBUFFER_DEFAULT_FIXED_SAMPLE_LOCATIONS, Fixed);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Initialization of a framebuffer object without attachment

Combined with a compute shader stage we can arguably say that this extension enables a form of programmable blending. For a more effective form, we could imagine that allows using the shared blocks in a fragment shader stage could give us an extra edge.

2. Debug functionalities

2.1. Debug output reach core: GL_KHR_debug

I keep saying that ARB_debug_output is one of the greatest evolutions that OpenGL has known for the past years. There is really no reason on platforms supporting it to keep using glGetError. Using an API is only viable when it's practical to use it. ARB_debug_output brings this level of viability to OpenGL. We can instantly figure out where an error happens and have a descriptive comment for the nature of the error. OpenGL Insights includes a chapter called ARB_debug_output: A Helping Hand for Desperate Developers by Antonio Ramires Fernandes and Bruno Oliveira describing in details this extension and what we can expect from implementations.

This extension didn't reach OpenGL core as it is. Instead is got promoted to KHR_debug, which is a superset of ARB_debug_output not only designed for OpenGL but also for OpenGL ES! KHR_debug is a superset because it includes ARB_debug_output but add something features like debug groups, debug markers and debug label, features inspired by the OpenGL ES extensions EXT_debug_marker and EXT_debug_label.

The debug label allows attaching a descriptive string for any OpenGL object. Then this label can be reused to generate the debug output messages generated from this object. With debug marker, the OpenGL programmer can annotate the debug output stream to notify specific events. Debug group allows encapsulating a section of the code so that a specific debug output volume setup can be used for this group. Both entering and leaving a debug group can generate messages so that debug group can just be used as debug group markers.

```
glEnable(GL_DEBUG_OUTPUT_SYNCHRONOUS);
glDebugMessageCallback(&glf::debugOutput, NULL);
glPushDebugGroup(GL_DEBUG_SOURCE_APPLICATION, 76, -1, "My debug group");
glDebugMessageControl(GL_DONT_CARE, GL_DONT_CARE, GL_DONT_CARE, 0, NULL, GL_TRUE);
```

Basic debug output setup asking the implementation to generate all the feedback it can.

Last but not least, KHR_debug is part of the core specification so that the debug APIs can't only be available when a debug context is created. Hence, the APIs are available anytime however an implementation might decide to just do nothing with a release context. The advantage of this approach is that if a user is running into issues, it might be interesting for him of the vendor to run the application using a debug context either while using an application specific setting or just forcing the use of the debug context with an entry into the drivers control panel.

With ARB_debug_output many programmers ran into issues to enable the debug context because they were using third party libraries to create the OpenGL context but those libraries didn't expose this part of the code. A classic example here is Qt. For this purpose the OpenGL ARB added an enable OpenGL state to enable the debug context without modify the WGL, GLX or EGL code.

```
glEnable(GL_DEBUG_OUTPUT);
```

Enabling the debug context on a non-debug context

The OpenGL specification doesn't require that his enable actually enable the debug context or that disabling it will actually disable the debug context. Hence it will be very interesting to see how vendors actually implement this feature.

2.2. Debug query: GL_ARB_program_interface_query

OpenGL provides a long list of state queries and it is typically recommended to avoid using them for performance. Yes, the query API is not efficient but for debug builds they can provide valuable information to avoid looking for hours why a silent error is happening. (OpenGL doesn't generate an error but the rendering is "obviously" wrong). As an example, the query API could be used to validate whether the vertex input interface match the bound vertex array object. Unfortunately, the query API is not complete: We can't query the fragment shader output or we can't query the varying input and output variable either. `ARB_program_interface_query` resolves these issues and actually provides a unified new query API so that with only seven functions we can do all the possible queries, including transform feedback, uniform blocks, sampler, etc.

I also think it is important to notice that the language defining the name strings to use for the queries has been clarify which should lead to less bugs on that side.

3. Convergence

3.1. Cross-version programming with GL_ARB_internalformat_query2

The OpenGL ecosystem is fragmented. Not only we have OpenGL and OpenGL ES but we have also many different versions of OpenGL and all the implementors have reach the same level of implementation. In theory, these days an OpenGL software need to support OpenGL 4.2 for OpenGL 4 hardware, OpenGL 3.3 for OpenGL 3 hardware and maybe OpenGL 2.1 for OpenGL 2.1 hardware. In practice, this is only possible on AMD and NVIDIA implementations because Intel only support OpenGL 4.0 and Apple only supports OpenGL 3.2. `GL_ARB_internalformat2` is a great opportunity, which may partially resolve this issue for texture and framebuffer capabilities.

3.2. Compatibility with OpenGL ES 3.0: GL_ARB_ES3_compatibility

Convergence with the OpenGL ES 3.0 API is very important to provide a way to handle the fragmentation of the OpenGL ecosystem. However, I remain shuffle about this extension that mainly adds the EAC, and ETC2 texture formats support. I don't believe that any OpenGL 4 hardware has such support which implies that vendors will be required to develop a transcoder from these formats to another format which is not necessary a strength forward task and I don't believe that anyone could recommend relying on this transcoder. On this time where making OpenGL drivers is very expensive, I think this is a cost the OpenGL community would have ben willing to spare.

4. Memory

4.1. Image copy: ARB_copy_image

This maybe a basic functionality but it is a functionality that was missing: The copy of a sub-image to another sub-image directly from GPU memory to GPU memory. Yes, it was already somehow possible with using two framebuffer objects and glBlitFramebuffer but this is not exactly convenient. ARB_copy_image (promoted from NV_copy_image) resolves this issue but also provides some extra goodness like the possibility of copying a GL_RGBA16UI texture to a GL_COMPRESSED_RED_RGTC1 texture. This is going to make on-GPU texture compression a lot easier, avoiding to setup an unpack buffer object to read the GL_RGBA16UI texture and a pack buffer object to write the data to the GL_COMPRESSED_RED_RGTC1 texture and possibly gain some performance on the way, doing a single memory transfer instead of two. ARB_copy_image strictly works like a memcpy for images, being either a level of a texture or the data of a renderbuffer object.

4.2. Texture views: ARB_texture_view

Texture views are a new type of texture object, which are created from a subset an existing texture object. There are somehow similar to image copy except that it is designed to avoid any memory copy by preserving the same memory layout across the original texture object and all the views. For this to be possible, texture views can only be created from immutable textures.

Texture views allow reinterpreting the internal format and provide a way to handle sRGB decode (EXT_texture_srgb_decode). For example, the original texture may be created with a sRGB internal format but a texture view could be created with a vintage RGB internal format which effectively disable the sRGB decode.

4.3. Stentil textures: ARB_stencil_texture

OpenGL 3.0 introduced with the new framebuffer object API the depth stentil formats:
- GL_DEPTH32F_STENCIL8
- GL_DEPTH24_STENCIL8

However, when binding textures using one of these formats, it was only possible to sample the depth component, (only if GL_TEXTURE_COMPARE_MODE is GL_NONE). Thanks to ARB_stencil_texture, it is now possible to sample the stencil value by using the new <pname> GL_DEPTH_STENCIL_TEXTURE_MODE to GL_STENCIL_INDEX.

Thanks to texture views we can even sample both the depth values and the stencil values of a single depth stencil texture.

```
glGenTextures(3, TextureName);

glBindTexture(GL_TEXTURE_2D, TextureName[SOURCE]);
glTexStorage2D(GL_TEXTURE_2D, 1, GL_DEPTH24_STENCIL8, Width, Height);
glBindTexture(GL_TEXTURE_2D, TextureName[DEPTH]);
glTextureView(
        TextureName[DEPTH], GL_TEXTURE_2D, TextureName[SOURCE],
        GL_DEPTH24_STENCIL8, 0, 1, 0, 1);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE, GL_NONE);
glBindTexture(GL_TEXTURE_2D, TextureName[STENCIL]);
glTextureView(
        TextureName[STENCIL], GL_TEXTURE_2D, TextureName[SOURCE],
        GL_DEPTH24_STENCIL8, 0, 1, 0, 1);
```

```
glTexParameteri(
        GL_TEXTURE_2D, GL_DEPTH_STENCIL_TEXTURE_MODE, GL_STENCIL_INDEX);
```

Create a depth stencil texture and views to access either the depth channel or the stencil channel.

4.4. ARB_clear_buffer_object

This extension provides in my opinion one of the oddest features designed for OpenGL (This way better that the original ARB_vertex_array_object). It provides a useful functionality design to initialize a buffer but the buffer API is designed with the texture API conventions, using both a format and an internalformat.

Reading this extension we understand that this is API is actually designed to initialize a texture buffer, not generic buffer even if it can work on any buffer… Considering that OpenGL 4.3 is released with the storage buffer from ARB_shader_storage_buffer_object, one could consider the texture buffer simply deprecated. However, ARB_clear_buffer_object can't be used to initialized a storage buffer…

I picture one really useful use case of this extension with atomic buffer. In some scenarios, only the shaders are writing into the atomic buffer but for example for every frame, draw call, etc., the atomic counter needs to be reinitialized because it stores a count, an offset which is only useful for these specific ranges. However really for these cases atomics don't even need to be backed by a buffer and it seems to me that it only force the implementation to take a slower path than what the hardware could do.

4.5. ARB_invalidate_subdata

ARB_invalidate_subdata is an extension designed to avoid unnecessary memory transfers. By invalidating buffers or images either entirely or by ranges implementations no longer need to maintain these data allowing them to avoid memory transfer from a memory space to another. In practice it sounds like whenever we don't need the content of a resource we should use new functions. We can expect that the behaviours of this set of new functions will depend from implementation to implementation. We can also imagine that to be effective a minimum range of the data will need to be invalidated per call.

4.6. ARB_texture_buffer_range

One of the good concepts of OpenGL is the possibility to bind ranges of a buffer so that for a specific draw call, only a subset of the buffer will be used. Interestingly, the only exception what for texture buffer but this is fixed with OpenGL 4.3.

4.7. ARB_texture_storage_multisample

Interestingly the ARB_texture_storage extension supports all the king of OpenGL textures but multisample texture. The purpose of immutable textures (created with glTexStorage*) was to remove the constant required validation required my mutable textures (Created with glTexImage*) checking whether the texture is still complete. A multisample texture doesn't have mipmaps so the problem doesn't happen. However, texture views require that the GL_TEXTURE_IMMUTABLE_FORMAT is set to GL_TRUE to be able to create a view. For this purpose, we now have immutable multisample textures, which is great news for consistence.

Bonus of this extension, it adds the missing Direct State Access function for creating multisample textures.

5. Shading Language Functionalities

5.1. Arrays of arrays: GL_ARB_arrays_of_arrays

Arrays of arrays with GLSL was a pretty messy area because it was kind of supported using blocks, structures, and embedded arrays but that was more a work around than anything. This is now fixed: we can freely declare array of array of arrays which is going to be pretty useful especially for the compute shader stage.

5.2. Explicit location for uniforms: GL_ARB_explicit_uniform_location

Querying the location for uniforms and subroutines are quite annoying because the application needs to track the variables storing these locations and pass them from class to class. Not so nice but OpenGL 4.3 improves this experience by allowing to declare the location layout qualifier to uniforms and the index layout qualifier to subroutine.

In practice we can already consider that uniform variables from the default uniform block are deprecated. Using the location qualifier on those is possible but non practical because if we change the type of a variable to another, all the variable located at a higher location will need to have their locations updated. There is no practicality in such idea so that uniform buffer and block remains the only viable solution here for me.

Indexed on subroutine doesn't have such limitation making this new qualifier more useful. Unfortunately, the subroutine API is not practically option with its "I am neither really a context or a program state".

With ARB_explicit_uniform_location, all the GLSL resources have an explicit location/index/binding qualify to identify where to find the data. All but one: Varying blocks! This is very unfortunate because we can legitimately recommend to always relying on varying blocks instead of independent varying variable except that it is impossible to do partial interface matching with separated programs.

5.3. GL_ARB_texture_query_levels

With `textureLod(Sampler, Texcoord, 0)`, we don't access to the image/mipmap at the level 0 of a texture but at the base level of this texture. This is a way to expose the `GL_TEXTURE_BASE_LEVEL` state in GLSL shaders. Unfortunately, until GLSL 4.30 the `GL_TEXTURE_MAX_LEVEL` state was not exposed but this is now fixed thanks to the new function `textureQueryLevels()` that basically returns `GL_TEXTURE_MAX_LEVEL - GL_TEXTURE_BASE_LEVEL + 1`. It is highly recommended to only use it only with immutable textures to avoid running over complicated scenarios involving complete and mipmap complete textures inherent to old fashion mutable textures.

5.4. Texture image size: GL_ARB_shader_image_size

OpenGL 3.0 introduced the new GLSL function `textureSize` that allows a shader to query the dimensions of a texture level. OpenGL 4.2 introduced texture images and image unit that allows binding

a single texture level to a shader. Unfortunately, it wasn't possible to query the size of an image. ARB_shader_image_size fixes this issue.

```
int imageSize(gimage1D image)
ivec2 imageSize(gimage2D image)
ivec3 imageSize(gimage3D image)
ivec2 imageSize(gimageCube image)
ivec3 imageSize(gimageCubeArray image)
ivec2 imageSize(gimageRect image)
ivec2 imageSize(gimage1DArray image)
ivec3 imageSize(gimage2DArray image)
int imageSize(gimageBuffer image)
ivec2 imageSize(gimage2DMS image)
ivec3 imageSize(gimage2DMSArray image)
```

List of the new imageSize GLSL functions.

5.5. GL_ARB_fragment_layer_viewport

This extension is pretty trivial. OpenGL 3.2 introduced the built-in variable `gl_LayerID` that allows dispatching primitives to a layered framebuffer. Then OpenGL 4.2 and ARB_viewport_array introduced viewport arrays and the built-in variable `gl_ViewportIndex` that allows selecting the viewport used by the rasterizer. Both variables are geometry shader output variables. With OpenGL 4.3 and ARB_fragment_layer_viewport, these built-in variables are now available as inputs of the fragment shader stage so that we have access at the result of the geometry shader stage choice in the fragment shader.

Conclusions

OpenGL 4.3 is a lot of goodness that gives another dimension to OpenGL like each minor revision managed to provide. Somehow OpenGL 4.3 gives a feeling of tackling post OpenGL 4 hardware type of problems to resolve which I personally envisioned as a massive step forward in matter of scene complexity and amount of data feeding the GPU. However, we are far from reaching the end of the road.

Direct State Access is still missing

Since the release of EXT_direct_state_access, a lot of desire has been formulate by the community for this extension to be improve and promoted. Since, no news about it, which is once again a big disappointment but true believers in the future of OpenGL won't give up. CPU overhead because of resources switching is important on both driver side and application side. If an API strategy provides a solution to avoid a portion of resources switching by providing guarantees that the resources have been switched, why wouldn't we care about this feature? I can't explain this mystery by anything else but arbitrary decisions but eventually I remain confident that we will have a direct state access API sooner or later.

New features

This could tempted me to write: "OpenGL 4 hardware done, Now... what about future hardware? :D". However, I think that there are still a lot of topics to explore for a potential new revision of the OpenGL

4 specification. In a precedent post, I mentioned how much I like AMD_query_buffer_object. This extension significantly improves the interoperability between the fixed-function part of the pipeline and the programmable parts building new use case like using it as a predicate in the Programmable Vertex Pulling rendering pipeline without ever falling back to the CPU.

Moving beyond OpenGL 4 hardware level, both AMD and NVIDIA has started to bring some inputs: AMD_sparse_texture, AMD_sparse_buffer and NV_bindless_texture. If we want to significantly increase the scene complexity, we can't possibility expect that this amount of data required for such will hold in graphics memory but we need solution to address this data and this is the purpose of these complementary extensions. Furthermore, I expect that removing the binding limitation will allow to simplify GLSL so that it becomes more like C with vector and matrix types.

GPUs also seem to be pretty limited for the maximum size of a buffer. OpenGL 4.3 required 256 MB for the minimum maximum of a shader storage buffer. This is clearly not enough. If we put in perspective Programmable Vertex Pulling with a single call to render an entire pass, we can't be satisfy with 256 MB for a shader storage buffer.

The entire memory model is actually questionable. I am still dreaming about a model that would take advantage of the synchronization object a lot more. The application would explicitly be responsible of the synchronization. Also, instead of having the drivers moving my data around from memory space to memory space, I expect that requiring the application to explicitly move the data would produce more reliable performance behaviour.

Finally, there are many game developers complaining about the GLSL compile time. I haven't personally explored this question but I would enjoy seeing some actual facts beyond these words and identifying how slow it is because especially in the frame of the development cycle, building GLSL shader might be an issue without practical solution at the moment.

Deprecation

Deprecation is one of these things that is only a matter of time before it happened. Considering the maturity of the OpenGL 4 hardware support, it might be time to consider deprecation once again: Mutable texture and buffer, GLSL implicitly sized arrays, default uniform and varying blocks, online texture compression, unified draw dispatching, monolithic programs, old shader interface query API, etc. In my most radical point of view, I would even remove the Vertex Array Object and the whole attribute API but also the sampler state from the texture and sampler objects and move them to the shader.

More generally speaking, I think that OpenGL needs deprecation to leverage the fragmentation issue that the OpenGL community encounters. If deprecation mean less work to build a good OpenGL implementation then it would be easier for all the vendors to catch up with the last OpenGL revisions. Furthermore, I hope this would allow more vendors to enter the OpenGL area building more competition and innovations in the world of graphics. Finally, this would simplify the API providing a smoother experience to the OpenGL programmers.

- Download: OpenGL 4.3 core specification
- Download: GLSL 4.3 specification
- Link: OpenGL 4.2 review

- Link: OpenGL 4.1 review
- Link: OpenGL 4.0 review
- Link: OpenGL 3.3 review