Candidate features for future OpenGL 5 and Direct3D 12 hardware

15 April 2014, [Christophe Riccio](mailto:mail@g-truc.net)



[G-Truc Creation](http://www.g-truc.net)

# Table of contents

Table of contents 2

Introduction 4

1. Draw submission 5

1.1. GL\_ARB\_multi\_draw\_indirect 5

1.2. GL\_ARB\_shader\_draw\_parameters 6

1.3. GL\_ARB\_indirect\_parameters 7

1.4. A shader code path per draw 7

1.5. GL\_NV\_bindless\_multi\_draw\_indirect 7

1.6. GL\_AMD\_interleaved\_elements 7

1.7. WGL\_AMD\_gpu\_association and WGL\_NV\_gpu\_affinity 8

2. Resources 9

2.1. GL\_ARB\_bindless\_texture 9

2.2. GL\_NV\_shader\_buffer\_load and GL\_NV\_shader\_buffer\_store 9

2.3. GL\_ARB\_sparse\_texture 9

2.4. GL\_AMD\_sparse\_texture 10

2.5. GL\_AMD\_sparse\_texture\_pool 11

2.6. Seamless texture stitching 11

2.7. 3D memory layout for sparse 3D textures 11

2.8. Sparse buffer 11

2.9. GL\_KHR\_texture\_compression\_astc 12

2.10. GL\_INTEL\_map\_texture 12

2.11. GL\_ARB\_seamless\_cubemap\_per\_texture 13

2.12. DMA engines 13

3. Shader operations 14

3.1. GL\_ARB\_shader\_group\_vote 14

3.2. GL\_NV\_shader\_thread\_group 14

3.3. GL\_NV\_shader\_thread\_shuffle 14

3.4. GL\_NV\_shader\_atomic\_float 15

3.5. GL\_AMD\_shader\_atomic\_counter\_ops 15

3.6. GL\_ARB\_compute\_variable\_group\_size 15

3.7. GL\_NV\_gpu\_shader5 16

3.8. GL\_AMD\_gpu\_shader\_int64 16

3.9. GL\_NV\_vertex\_attrib\_integer\_64bit 17

3.10. GL\_AMD\_ shader\_trinary\_minmax 17

4. Framebuffer 18

4.1. GL\_AMD\_sample\_positions 18

4.2. GL\_EXT\_framebuffer\_multisample\_blit\_scaled 18

4.3. GL\_NV\_multisample\_coverage and GL\_NV\_framebuffer\_multisample\_coverage 18

4.4. GL\_AMD\_depth\_clamp\_separate 18

5. Blending 19

5.1. GL\_NV\_texture\_barrier 19

5.2. GL\_EXT\_pixel\_local\_storage (OpenGL ES) 19

5.3. GL\_EXT\_shader\_framebuffer\_fetch (OpenGL ES) 19

5.4. GL\_ARM\_shader\_framebuffer\_fetch\_depth\_stencil (OpenGL ES) 19

5.5. GL\_INTEL\_fragment\_shader\_ordering 19

5.6. GL\_INTEL\_conservative\_rasterization 19

5.7. GL\_KHR\_blend\_equation\_advanced 20

5.8. GL\_AMD\_blend\_minmax\_factor 20

6. Stencil 21

6.1. GL\_AMD\_shader\_stencil\_export 21

6.2. GL\_AMD\_stencil\_operation\_extended 21

6.3. GL\_AMD\_shader\_stencil\_value\_export 21

7. Rendering pipeline 22

7.1. GL\_AMD\_vertex\_shader\_layer 22

7.2. GL\_AMD\_vertex\_shader\_viewport\_index 22

7.3. GL\_AMD\_transform\_feedback3\_lines\_triangles 22

7.4. GL\_AMD\_transform\_feedback4 22

7.5. GL\_AMD\_occlusion\_query\_event 22

8. Profiling 23

Conclusions 24

Is there room for an OpenGL 4.5 specification? Always! 24

Deprecation for OpenGL 5 hardware? 24

Features for OpenGL 5 hardware level? 25

Last words 25

# Introduction

The announcement of Mantle few months ago has triggered a lot of discussions about graphics API design. I think there are technical issues in OpenGL but those are precise issues that need to be address individually and following the GPU hardware designs

Sadly, these API design discussions have hidden the essential side that drives real time rendering evolution: the new hardware features. At the end of the day, the purpose of a graphics API is to expose the hardware to the graphics programmers

What I like with Direct3D is that Microsoft was important enough to drive IHVs to standardize hardware features. The Khronos Group is certainly getting better at it; the [ASTC texture format](http://www.opengl.org/registry/specs/KHR/texture_compression_astc_hdr.txt) is a good example as this format should be supported on all mobile and desktop GPUs in the future.

What's an OpenGL 5 hardware feature? Following the conversions for OpenGL 3 and OpenGL 4, it's any hardware feature that can't be implemented on all OpenGL 4 hardware but would be implementable on newer hardware by all IHVs.

In this article, we are going to have a look at hardware features available through OpenGL extensions and ideas that may or may not be interesting to standardize but it looks like programmable vertex pulling and programmable blending are active topics.

# 1. Draw submission

Draw submission has been a subject of API evolution since the very first version of OpenGL. With OpenGL 1.0 was done through the *immediate mode* using [glBegin](http://www.opengl.org/sdk/docs/man2/xhtml/glBegin.xml)/[glEnd](http://www.opengl.org/sdk/docs/man2/xhtml/glEnd.xml). Quickly, it appeared that an approach based on building and sending vertex one by one was way too slow to be efficient enough because the GPUs were faster to consume the primitives than the CPU was able to submit them. A lot of new features got introduced along the life of OpenGL to compensate this increasing GPU / CPU performance ratio:

* [Vertex Array](http://www.opengl.org/registry/specs/EXT/vertex_array.txt) (GL1.1);
* [Vertex Buffer Object](http://www.opengl.org/registry/specs/ARB/vertex_buffer_object.txt) (GL1.5);
* [Vertex Array Object](http://www.opengl.org/registry/specs/ARB/vertex_array_object.txt) (GL3.0);
* [Base Vertex](http://www.opengl.org/registry/specs/ARB/draw_elements_base_vertex.txt) (GL3.2);
* [Instancing](http://www.opengl.org/registry/specs/ARB/draw_instanced.txt) (GL3.2);
* [Instanced Arrays](http://www.opengl.org/registry/specs/ARB/instanced_arrays.txt) (GL3.3);
* [Base Instance](http://www.opengl.org/registry/specs/ARB/base_instance.txt) (GL4.2); and
* [Vertex Attrib Binding](http://www.opengl.org/registry/specs/ARB/vertex_attrib_binding.txt) (GL4.3)

OpenGL 4.3 took a new direction aiming at providing a magnitude of draw performances thanks to [ARB\_multi\_draw\_indirect](http://www.opengl.org/registry/specs/ARB/multi_draw_indirect.txt) leading the way to Programmable Vertex Pulling. With this approach, the GPU becomes responsible to dispatch the draws.

## 1.1. [GL\_ARB\_multi\_draw\_indirect](http://www.opengl.org/registry/specs/ARB/multi_draw_indirect.txt)

By batching the data of multiple VAOs into a single VAO and calling [glDrawArraysInstancedBaseInstance](http://www.opengl.org/sdk/docs/man4/xhtml/glDrawArraysInstancedBaseInstance.xml) or [glDrawElementsInstancedBaseVertexBaseInstance](http://www.opengl.org/sdk/docs/man4/xhtml/glDrawElementsInstancedBaseVertexBaseInstance.xml) many times in a tight loop, we can archive extremely good performances with a very low CPU overhead. We can push this concept further, my batching textures into texture arrays and uniforms into buffers sorted by update frequencies and indexing any resource used by a shader invocation in the tight loop.

However, the CPU remains in change of submitting the draws and worse, it needs to figure out which meshes needs to be drawn for a specific frame. That operation alone can require an entire CPU core.

Thanks to [ARB\_multi\_draw\_indirect](http://www.opengl.org/registry/specs/ARB/multi_draw_indirect.txt), the application can build a thinner OpenGL back-end by collecting all the draws that need to be dispatched in a C++ array from many threads having no interaction with the OpenGL API. Such array can be transfer to GPU memory to be read by the GPU command processor that will schedule the draws for execution at a speed that the CPU is not likely to follow.

Furthermore, if multi core CPU performance is not enough, [ARB\_multi\_draw\_indirect](http://www.opengl.org/registry/specs/ARB/multi_draw_indirect.txt) can be coupled with [ARB\_compute\_shader](http://www.opengl.org/registry/specs/ARB/compute_shader.txt) to move the draw array generation processing from CPU to the GPU. Effectively, the compute shader writes into a draw indirect buffer, storing the parameters for multiple draw calls. This buffer is consumed by [glMultiDrawElementsIndirect](http://www.opengl.org/sdk/docs/man4/xhtml/glMultiDrawElementsIndirect.xml) or [glMultiDrawArraysIndirect](http://www.opengl.org/sdk/docs/man4/xhtml/glMultiDrawArraysIndirect.xml).

The glMultiDraw\*Indirect functions are nothing more than an evolution 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\_BUFFER, BufferName);  for(std::size\_t DrawIndex = 0; DrawIndex < DrawCount; ++DrawIndex)  glDrawElementsIndirect(GL\_TRIANGLES, GL\_UNSIGNED\_SHORT, BUFFER\_OFFSET(Offset)); |

Listing 1.1.1: CPU draw dispatching in a tight loop.

|  |
| --- |
| glBindBuffer(GL\_DRAW\_INDIRECT\_BUFFER, BufferName);  glMultiDrawElementsIndirect(GL\_TRIANGLES, GL\_UNSIGNED\_SHORT, nullptr, GLsizei(DrawCount), 0); |

Listing 1.1.2: GPU draw dispatching by the command processor.

Multi draw indirect is part of OpenGL 4.3 core specification but it's arguably an OpenGL 5 hardware feature allowing the GPU to submit itself draws to be executed. This feature can be implemented through software emulation quite easily using the CPU to push each individual draw but this is really slow. Currently, all the Intel GPUs and AMD Evergreen support multi draw in software. Hardware implementations like AMD Southern Islands or NVIDIA Fermi give another magnitude of performance. For example, Kepler can submit up to 800000 draws per frame at 60Hz and Southern Islands can submit up to 300000 draws per frame on with a synthetic test rendering 2 triangles per draw on 4 pixels. That huge amount of draws provides such an amazing control over the rendering that not only the CPU overhead become insignificant but we can increase the GPU processing efficiency by submitting many more thin draws reducing overdraw and unnecessary processing of clipped primitives.

More details are given in [GPU Pro 4](http://www.amazon.com/GPU-Pro-Advanced-Rendering-Techniques/dp/1466567430/ref=pd_sim_b_4?ie=UTF8&refRID=09PXANNDHPFHZ120P8M5) chapter “*Introducing the Programmable Vertex Pulling Rendering Pipeline”* by Sean Lilley and I.

Current support: NVIDIA Fermi; AMD Evergreen (emulated), Southern Islands; Intel Haswell (emulated)

Expected support: All OpenGL 5 hardware

## 1.2. [GL\_ARB\_shader\_draw\_parameters](http://www.opengl.org/registry/specs/ARB/multi_draw_indirect.txt)

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](http://www.opengl.org/sdk/docs/man/html/gl_InstanceID.xhtml) for draw instancing. A massive difference between these new vertex shader inputs and [gl\_InstanceID](http://www.opengl.org/sdk/docs/man/html/gl_InstanceID.xhtml) 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](http://www.opengl.org/registry/specs/ARB/shader_storage_buffer_object.txt); using gl\_BaseVertexARB and gl\_VertexID to fetch ourselves the vertex data.

Unfortunately, this extension is only supported by NVIDIA OpenGL implementation at the moment so indexing is not that easy for other implementation. [*Surviving without gl\_DrawID*](http://www.g-truc.net/post-0518.html#menu) presents an approach to work around this limitation.

Current hardware support: NVIDIA Fermi

Expected hardware support: All OpenGL 5 hardware, AMD Southern Islands

## 1.3. [GL\_ARB\_indirect\_parameters](http://www.opengl.org/registry/specs/ARB/multi_draw_indirect.txt)

An issue of [ARB\_multi\_draw\_indirect](http://www.opengl.org/registry/specs/ARB/multi_draw_indirect.txt) is that it requires that we submit the number of draws from the CPU side, as <drawcount> is [glMultiDrawElementsIndirect](http://www.opengl.org/sdk/docs/man/xhtml/glMultiDrawElementsIndirect.xml) 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.

Current hardware support: NVIDIA Fermi

Expected hardware support: All OpenGL 5 hardware, AMD Southern Islands

## 1.4. A shader code path per draw

With [ARB\_multi\_draw\_indirect](http://www.opengl.org/registry/specs/ARB/multi_draw_indirect.txt) we can submit a huge number of draws however in many use cases, each draw would need to execute a dedicated shader code path. An application could choose to batch multiple code paths into an uber-shader as dynamically uniform indexing or unconditional branching is really fast on current hardware.

However, GPU execution units (CU on Southern Islands / SM on NVDIA) need to allocate an amount of registers according to the shader complexity. The GLSL compiler is in charge of figuring out how many registers is required to guarantee the execution of any code path in the uber-shader. Hence, using a trivial code path from an uber-shader will result in over-allocating GPU registers and underutilizing the GPU.

Looking at GPU architectures, it appears that each GPU execution unit has a shader code pointer that can be different for each CU. Hence, it seems possible to execute a different shader code path for each draw by providing a different pointer. Furthermore, we could allocate the correct number of registers if we had an API to bake the code paths per draw and if the DrawArraysIndirectCommand and DrawElementsIndirectCommand could add a parameter to identify the shader code path per draw.

|  |
| --- |
| typedef struct{  uint count;  uint primCount;  uint firstIndex;  int baseVertex;  uint baseInstance;  uint programID; // Added to identify a shader code path  } DrawElementsIndirectCommand;  typedef struct{  uint count;  uint primCount;  uint first;  uint baseInstance;  uint programID; // Added to identify a shader code path  } DrawArraysIndirectCommand; |

Listing 1.4: DrawElementsIndirectCommand and DrawArraysIndirectCommand with an extra parameter.

Sadly, considering the poor performance of NVIDIA gl\_DrawID implementation which is the only one to implement it at the moment, it seems unlikely that any current hardware would be able to implement such behavior to provide [ARB\_multi\_draw\_indirect](http://www.opengl.org/registry/specs/ARB/multi_draw_indirect.txt) like performance for shader code path switching.

Expected hardware support: Future hardware

## 1.5. [GL\_NV\_bindless\_multi\_draw\_indirect](http://www.opengl.org/registry/specs/NV/bindless_multi_draw_indirect.txt)

This is the last piece of NVIDIA bindless API allowing draw submission without binding vertex arrays or indirect draw buffer for lower CPU overhead.

Current hardware support: NVIDIA Fermi

Not needed with OpenGL 5 hardware

## 1.6. [GL\_AMD\_interleaved\_elements](http://www.opengl.org/registry/specs/AMD/interleaved_elements.txt)

This extension is really ugly but the functionality is really interesting. Instead of having a single element array, thanks to this extension we can have up to 4 element arrays and we can index each vertex attribute with the element array of our choice. There is quite a few software actually generating meshes using multiple element array so this functionality sounds extremely useful. Taking advantage of this feature can save bandwidth avoiding duplicating vertex attributes.

Current hardware support: AMD Southern Islands

## 1.7. [WGL\_AMD\_gpu\_association](http://www.opengl.org/registry/specs/AMD/wgl_gpu_association.txt) and [WGL\_NV\_gpu\_affinity](http://www.opengl.org/registry/specs/NV/gpu_affinity.txt)

I have never really explored either [AMD\_gpu\_association](http://www.opengl.org/registry/specs/AMD/wgl_gpu_association.txt) or [NV\_gpu\_affinity](http://www.opengl.org/registry/specs/NV/gpu_affinity.txt) by lack of interest. These extensions enable CrossFire and SLI on AMD and NVIDIA GPUs. Such feature could probably be standardized into an OpenGL ARB extension.

Expected hardware support: OpenGL 3 hardware

# 2. Resources

## 2.1. [GL\_ARB\_bindless\_texture](http://www.opengl.org/registry/specs/ARB/bindless_texture.txt)

[ARB\_bindless\_texture](http://www.opengl.org/registry/specs/ARB/bindless_texture.txt) was promoted from [NV\_bindless\_texture](http://www.opengl.org/registry/specs/NV/bindless_texture.txt). It allows a shader invocation to access an “infinite” number of textures, any texture resident in GPU memory. Texture handles are stored in a uniform buffer and accessed through indexing.

|  |
| --- |
| #version 420 core  #extension GL\_ARB\_bindless\_texture : require  #define handle uvec2  #define FRAG\_COLOR 0  #define MATERIAL 0  layout(binding = MATERIAL) uniform material  {  handle Diffuse; // This is the handle for the bindless texture  } Material;  in block  {  vec2 Texcoord;  } In;  layout(location = FRAG\_COLOR, index = 0) out vec4 Color;  void main()  {  Color = texture(sampler2D(Material.Diffuse), In.Texcoord.st);  } |

Listing 2.1.1: Example of a fragment shader sampling and bindless texture

One great side effect of that API is that textures can be part of data representing the materials for example.

Current hardware support: NVIDIA Kepler

Expected hardware support: All OpenGL 5 hardware, AMD Southern Islands

## 2.2. [GL\_NV\_shader\_buffer\_load](http://www.opengl.org/registry/specs/NV/shader_buffer_load.txt) and [GL\_NV\_shader\_buffer\_store](http://www.opengl.org/registry/specs/NV/shader_buffer_store.txt)

NVIDIA buffer load and store is pretty much a set of bindless buffer extensions. Going toward such design really emphasis that there is no element array buffer, array buffer, shader storage buffer, transform feedback buffer: It’s all just memory and we should manage the same wait we manage any form of memory.

Current hardware support: NVIDIA Fermi

Expected hardware support: All OpenGL 5 hardware, AMD Southern Islands

## 2.3. [GL\_ARB\_sparse\_texture](http://www.opengl.org/registry/specs/ARB/sparse_texture.txt)

[ARB\_sparse\_texture](http://www.opengl.org/registry/specs/ARB/sparse_texture.txt) is a subset of [AMD\_sparse\_texture](http://www.opengl.org/registry/specs/AMD/sparse_texture.txt) enabling virtual texturing with seamless texture filtering. Thanks to this extension we can create 16K by 16K texels 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 an independent task performed with glTexPageCommitmentARB on a subsection of that texture.

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.

Current hardware support: AMD Southern Islands, NVIDIA Fermi

Expected hardware support: All OpenGL 5 hardware

## 2.4. [GL\_AMD\_sparse\_texture](http://www.opengl.org/registry/specs/AMD/sparse_texture.txt)

[ARB\_sparse\_texture](http://www.opengl.org/registry/specs/ARB/sparse_texture.txt) is essentially a fixed design of [AMD\_sparse\_texture](http://www.opengl.org/registry/specs/AMD/sparse_texture.txt). However, AMD extension provides shader functions to query the status of a sparse texture fetch using dedicated sampling functions.

|  |
| --- |
| #version 420 core  #extension GL\_AMD\_sparse\_texture : require  #define handle uvec2  #define FRAG\_COLOR 0  #define MATERIAL 0  layout(binding = DIFFUSE) uniform sampler2D Diffuse;  in block  {  vec2 Texcoord;  } In;  layout(location = FRAG\_COLOR, index = 0) out vec4 Color;  void main()  {  if(GL\_AMD\_sparse\_texture)  {  vec4 Fetch = vec4(0);  int Code = sparseTexture(Diffuse, In.Texcoord.st, Fetch);  if(sparseTexelResident(Code))  Color = Fetch;  else  Color = vec4(0.0, 0.5, 1.0, 1.0);  }  else  {  texture(Diffuse, In.Texcoord.st);  }  } |

Listing 2.4.1: Example of texel residence query with [AMD\_sparse\_texture](http://www.opengl.org/registry/specs/AMD/sparse_texture.txt) extension

The following functions are used to interpret the status.

* bool sparseTexelResident(int code) : Returns true if the texture read that produced code retrieved valid data, and produced code retrieved valid data, and false otherwise ;
* bool sparseTexelMinLodWarning(int code) : Returns true if the texture read that produced code required a texel fetch from any LOD lower than the user specified LOD warning threshold ;
* int sparseTexelLodWarningFetch(int code) : Returns the LOD calculated by the texture read that generated <code> and resulted in a condition that would cause sparseTexelMinLodWarning to return true. If the LOD warning was not encountered, this function returns zero.

Current hardware support: AMD Southern Islands

Expected hardware support: Future hardware

## 2.5. GL\_AMD\_sparse\_texture\_pool

A limitation of [ARB\_sparse\_texture](http://www.opengl.org/registry/specs/ARB/sparse_texture.txt) is that each single texture page is backed by its own memory. We could imagine a design where multiple texture pages could share the same memory, providing new ways for texture compression.

This is what AMD\_sparse\_texture\_pool is aiming at sadly the specification haven’t been released yet.

Current hardware support: AMD Volcanic Islands

Expected hardware support: Future hardware

## 2.6. Seamless texture stitching

Unfortunately texture and sparse texture share the same limitations for the maximum texture sizes. 16K\*16K is a very large texture but it’s a very small sparse texture. What we really want is something like 1M\*1M pixels sparse texture however having such large texture would require a lot more precision for the texture coordinates in texture units.

An alternative to making texture bigger is called seamless texture stitching which is pretty much what the hardware does for seamless cubemap filtering. Applied to sparse textures, the hardware would be able to seamlessly filter across texture layers of a sparse texture 2D array.

Expected hardware support: Future hardware

## 2.7. 3D memory layout for sparse 3D textures

On AMD Southern Islands, when we query the texture page sizes of a sparse 3D texture, we realize that internally a 3D texture is stored as layers of 2D textures. This make filtering 3D textures less efficient but it also implies that sparse 3D texture pages are not little dices but 2D plans.

On NVIDIA Fermi, sparse 3D texture pages are stored as dices which make them better candidates for volumetric rendering technics.

Current hardware support: NVIDIA Fermi

Expected hardware support: All OpenGL 5 hardware

## 2.8. Sparse buffer

The OpenGL ARB released [ARB\_sparse\_texture](http://www.opengl.org/registry/specs/ARB/sparse_texture.txt) at Siggraph 2013 but sadly it didn’t came with an equivalent sparse buffer extension. Direct3D 11.2 has such feature and it would be nice to have it with OpenGL too.

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

## 2.9. [GL\_KHR\_texture\_compression\_astc](http://www.opengl.org/registry/specs/KHR/texture_compression_astc_hdr.txt)

The Khronos Group has standardized a new texture format called ASTC that provides very low bit rate and HDR support. Because, it's a KHR extension, it means that both the OpenGL ES group and the OpenGL ARB group voted to support that feature which gives me good hope that we will "soon" have support for this format on all desktop and mobile platforms.

|  |  |  |
| --- | --- | --- |
| Block size | Bits per pixels | Compression ratio |
| 4x4 | 8.00 | 4:1 |
| 5x4 | 6.40 | 5:1 |
| 5x5 | 5.12 | 6.25:1 |
| 6x5 | 4.27 | 7.5:1 |
| 6x6 | 3.56 | 9:1 |
| 8x5 | 3.20 | 10:1 |
| 8x6 | 2.67 | 12:1 |
| 8x8 | 2.00 | 16:1 |
| 10x5 | 2.56 | 12.5:1 |
| 10x6 | 2.13 | 15:1 |
| 10x8 | 1.60 | 20:1 |
| 10x10 | 1.28 | 25:1 |
| 12x10 | 1.07 | 30:1 |
| 12x12 | 0.89 | 36:1 |

Current hardware support: Imagination Technologies PowerVR6XT, ARM Mali T700, NVIDIA Maxwell

Expected hardware support: Future hardware

## 2.10. [GL\_INTEL\_map\_texture](http://www.opengl.org/registry/specs/INTEL/map_texture.txt)

GL\_INTEL\_map\_texture allows choosing the memory layout of a texture between a customs swizzle order (GL\_LAYOUT\_DEFAULT\_INTEL) and a linear order (GL\_LAYOUT\_LINEAR\_INTEL) that can be cached (GL\_LAYOUT\_LINEAR\_CPU\_CACHED\_INTEL). Textures created with a linear memory layout can be mapped just like a buffer with:

|  |
| --- |
| void\* glMapTexture2DINTEL(GLuint texture,  GLint level, GLbitfield access, GLint \*stride, GLenum \*layout); |

Caching the texture on the client side can result in better performance when reading texture on CPU but might negatively impact the GPU side access to the texture. Thus the option is intended only for cases when volume of the read access from CPU justifies such effect.

Sadly, we can’t map textures created with GL\_LAYOUT\_DEFAULT\_INTEL memory layout. For that Intel would have to expose its memory layout. It’s probably not a big issue as such layout is exposed on consoles for example. However on PC, the memory layout of each format; each architecture; each vendor; can be different which is quickly not tractable for any software. For each, capabilities IHV would have to agree on a standard memory layout.

Current hardware support: Intel Sandy Bridge

Expected hardware support: All OpenGL 5 hardware, AMD Evergreen, NVIDIA Fermi

## 2.11. [GL\_ARB\_seamless\_cubemap\_per\_texture](http://www.opengl.org/registry/specs/ARB/seamless_cubemap_per_texture.txt)

OpenGL 3.2 and [ARB\_seamless\_cube\_map](http://www.opengl.org/registry/specs/ARB/seamless_cube_map.txt) 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](http://www.opengl.org/sdk/docs/man/html/glEnable.xhtml)(GL\_TEXTURE\_CUBE\_MAP\_SEAMLESS). If we want to don’t want to use it on another texture, we need to call [glDisable](http://www.opengl.org/sdk/docs/man/html/glEnable.xhtml)(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](http://www.opengl.org/registry/specs/ARB/seamless_cubemap_per_texture.txt) 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.

Current hardware support: AMD RV700, NVIDIA Kepler

Expected hardware support: All OpenGL 5 hardware

## 2.12. DMA engines

NVIDIA Fermi and AMD Northern Islands have dedicated DMA engines that can live their lives on their own. Hence a dedicated thread could be in charge of streaming resources because at some point the application figure out that they might become useful. During these transfers, the graphics engine can continue his life independently without any required synchronization. Obviously, the transfers would have to be completed before using the resources but with enough anticipation we could need a synchronization object only for the purpose of guarantying correctness on all possible hardware but without actually hitting that fence.

Currently NVIDIA supports this behavior but only by creating a separated context on a dedicated thread. This is workable but cumbersome and it costs thread safety penalty for the entire OpenGL implementation.

An explicit use of the DMA engine for fully asynchronous transfers and performing transfer outside of the rendering code would be really nice to have.

Current hardware support: AMD Northern Islands, NVIDIA Fermi

Expected hardware support: All OpenGL 5 hardware

# 3. Shader operations

## 3.1. [GL\_ARB\_shader\_group\_vote](http://www.opengl.org/registry/specs/ARB/shader_group_vote.txt)

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 cake. 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](http://developer.amd.com/wordpress/media/2012/12/AMD_Southern_Islands_Instruction_Set_Architecture.pdf), I found five different methods to handle branching.

The OpenGL ARB has tackled this issue by considering how we could help the compiler to produce more efficient branching code. The proposed solution is exposed by [ARB\_shader\_group\_vote](http://www.opengl.org/registry/specs/ARB/shader_group_vote.txt), a small subset of [NV\_gpu\_shader5](http://www.opengl.org/registry/specs/NV/gpu_shader5.txt) which provides the GLSL functions anyInvocationARB, allInvocationsARB, allInvocationsEqualARB to compare values across shader invocations.

Current hardware support: NVIDIA Fermi

Expected hardware support: All OpenGL 5 hardware, AMD Southern Islands

## 3.2. [GL\_NV\_shader\_thread\_group](http://www.opengl.org/registry/specs/NV/shader_thread_group.txt)

This extension goes into the "super resolution" range of idea where we no long want to think at a fixed pixel resolutions but instead we want to think at higher or lower resolution than the native resolution. GPU doesn't actually execute anything on per pixel or per vertex base but in many different kind of grouping. The warp/wavefront is the grouping for shader invocation and another famous one is the quadpixel, a set of 4 fragments. The texture LOD calculation is computed per quadpixel because it is very complex to compute analytically the derivatives required for the texture LOD computation but it is really easy to compute within a quadpixel: It's only the different between the values across quadpixels.

This extension gives access to quadpixels allowing to swizzle the intermediate results accross each fragments. Let's say that fragment shader requires 4 texture sampling. In some areas, we could consider that it is not that useful to sample per fragment and we can deal will sampling per quadpixels. This feature should interact pretty well with [ARB\_shader\_group\_vote](http://www.opengl.org/registry/specs/ARB/shader_group_vote.txt).

Current hardware support: NVIDIA Fermi

Expected hardware support: All OpenGL 5 hardware, AMD Evergreen

## 3.3. [GL\_NV\_shader\_thread\_shuffle](http://www.opengl.org/registry/specs/NV/shader_thread_shuffle.txt)

This extension extends [NV\_shader\_thread\_group](http://www.opengl.org/registry/specs/NV/shader_thread_group.txt) to any of the shader invocations of a wrap/wavefront. It seems very likely that we could use [NV\_shader\_thread\_group](http://www.opengl.org/registry/specs/NV/shader_thread_group.txt) on any GPU because all GPUs use quadpixels however, the warp/wavefront size is different for each GPU vendors: 32 for NVIDIA; 64 for AMD; and variable for Intel, between 4 to 16 shader invocations. This feature sounds particularly useful for post processed antialiazing and maybe things like soft shadows.

Current hardware support: NVIDIA Kepler

Expected hardware support: Future hardware

## 3.4. [GL\_NV\_shader\_atomic\_float](http://www.opengl.org/registry/specs/NV/shader_atomic_float.txt)

This extension is simply extending add and exchange to float atomic operations.

Current hardware support: NVIDIA Fermi

Expected hardware support: Future hardware

## 3.5. [GL\_AMD\_shader\_atomic\_counter\_ops](http://www.opengl.org/registry/specs/AMD/shader_atomic_counter_ops.txt)

[ARB\_shader\_atomic\_counters](http://www.opengl.org/registry/specs/ARB/shader_atomic_counters.txt) and OpenGL 4.2 introduced the concept of atomic counter operations be those where limited to increment, decrement and query. AMD GPUs support these atomic operations in GDS memory which is faster than image and buffer atomic operations. However, AMD GPUs support more atomic operations from GDS: Increment and decrement with wrap ; addition and subtraction ; minimum and maximum ; bitwise operators (AND, OR, XOR, etc.) ; masked OR operator ; exchange, and compare and exchange operators. [AMD\_shader\_atomic\_counter\_ops](http://www.opengl.org/registry/specs/AMD/shader_atomic_counter_ops.txt) exposes all these operations.

Current hardware support: AMD Southern Islands

## 3.6. [GL\_ARB\_compute\_variable\_group\_size](http://www.opengl.org/registry/specs/ARB/compute_variable_group_size.txt)

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.

|  |
| --- |
| // GLSL side  #define LOCAL\_SIZE\_X \* #define LOCAL\_SIZE\_Y \*  #define LOCAL\_SIZE\_Z \*  layout(  local\_size\_x = LOCAL\_SIZE\_X,  local\_size\_y = LOCAL\_SIZE\_Y,  local\_size\_z = LOCAL\_SIZE\_Z) in;  // C++ side  void glDispatchCompute(GLuint num\_groups\_x, GLuint num\_groups\_y, GLuint num\_groups\_z); |

Listing 3.6.1: Compute shader invocation with built-in local sizes.

With ARB\_compute\_variable\_group\_size the sizes of a workgroup can change between compute dispatches.

|  |
| --- |
| // C++ side  void glDispatchComputeGroupSizeARB(  GLuint num\_groups\_x, GLuint num\_groups\_y, GLuint num\_groups\_z,  GLuint group\_size\_x, GLuint group\_size\_y, GLuint group\_size\_z); |

Listing 3.6.2: Compute shader invocation with per-draw group sizes.

Current hardware support: NVIDIA Fermi

Expected hardware support: All OpenGL 5 hardware

## 3.7. [GL\_NV\_gpu\_shader5](http://www.opengl.org/registry/specs/NV/gpu_shader5.txt)

This extension was released with Fermi GPUs and it extends [ARB\_gpu\_shader5](http://www.opengl.org/registry/specs/ARB/gpu_shader5.txt) with a variety of Fermi specific features at the time. It contains the features later promoted into [ARB\_shader\_group\_vote](http://www.opengl.org/registry/specs/ARB/shader_group_vote.txt), the features picked up by [AMD\_gpu\_shader\_int64](http://www.opengl.org/registry/specs/AMD/gpu_shader_int64.txt) for AMD Southern Islands and the following:

* support for a full set of 8-, 16-, 32-, and 64-bit scalar and vector data types, including uniform API, uniform buffer object, and shader input and output support (int8\_t, int16\_t, int64\_t, uint8\_t, uint16\_t, uint64\_t, float16\_t, i8vec2, i8vec3, i8vec4, i16vec2, i16vec3, i16vec4, i64vec2, i64vec3, i64vec4, u8vec2, u8vec3, u8vec4, u16vec2, u16vec3, u16vec4, u64vec2, u64vec3, u64vec4, f16vec2, f16vec3, f16vec4) ;
* the ability to aggregate samplers into arrays, index these arrays with arbitrary expressions, and not require that non-constant indices be uniform across all shader invocations ;
* new built-in functions to pack and unpack 32-bit unsigned integer types into a two-component 16-bit floating-point vector (uint packFloat2x16(f16vec2 v), f16vec2 unpackFloat2x16(uint v)) ;
* vector relational functions supporting comparisons of vectors of 8-, 16-, and 64-bit integer types or 16-bit floating-point types ;
* extending texel offset support to allow loading texel offsets from regular integer operands computed at run-time, except for lookups with gradients ([textureGrad\*](http://www.opengl.org/sdk/docs/man/html/textureGrad.xhtml)) ;
* relaxing the requirement of a tessellation shader stage when processing patches. This allows the “patches” layout qualifier can be used for geometry shader input, as input to transform feedback and as input to the fixed-function rasterization stages where each point is drawn as independent points ; and
* the capability to read per-patch variable written by a tessellation control shader by the geometry shader.

Current hardware support: NVIDIA Fermi

## 3.8. [GL\_AMD\_gpu\_shader\_int64](http://www.opengl.org/registry/specs/AMD/gpu_shader_int64.txt)

[AMD\_gpu\_shader\_int64](http://www.opengl.org/registry/specs/AMD/gpu_shader_int64.txt) is a superset of the 64-bit support exposed by [NV\_gpu\_shader5](http://www.opengl.org/registry/specs/NV/gpu_shader5.txt) and supported by AMD Southern Islands. This extension introduces the following features:

* support for 64-bit scalar (int64\_t, uint64\_t) and vector integer data types (i64vec\*, u64vec\*), including uniform API, uniform buffer object, transform feedback, and shader input and output support;
* new built-in functions to pack and unpack 64-bit integer types into a two-component 32-bit integer vector (int64BitsToDouble, uint64BitsToDouble);
* new built-in functions to convert double-precision floating-point values to or from their 64-bit integer bit encodings (doubleBitsToInt64, doubleBitsToUint64);
* vector relational functions supporting comparisons of vectors of 64-bit integer types; and
* common functions abs, sign, min, max, clamp, and mix supporting arguments of 64-bit integer types.

[AMD\_gpu\_shader\_int64](http://www.opengl.org/registry/specs/AMD/gpu_shader_int64.txt) seems to be an obvious candidate to become an ARB extension and OpenGL 5 hardware feature.

Current hardware support: AMD Southern Islands

Expected hardware support: All OpenGL 5 hardware, NVIDIA Fermi

## 3.9. [GL\_NV\_vertex\_attrib\_integer\_64bit](http://www.opengl.org/registry/specs/NV/vertex_attrib_integer_64bit.txt)

[NV\_vertex\_attrib\_integer\_64bit](http://www.opengl.org/registry/specs/AMD/gpu_shader_int64.txt) requires [NV\_gpu\_shader5](http://www.opengl.org/registry/specs/NV/gpu_shader5.txt) to provide 64-bit integer and unsigned integer support for vertex attributes.

Current hardware support: NVIDIA Fermi

Expected hardware support: Not needed in OpenGL 5 hardware

## 3.10. [GL\_AMD\_ shader\_trinary\_minmax](http://www.opengl.org/registry/specs/AMD/shader_trinary_minmax.txt)

This extension adds functions to find the minimum, maximum and median of three float or integer scalar or vector inputs.

|  |  |
| --- | --- |
| Syntax | Description |
| genType min3(genType x, genType y, genType z) genIType min3(genIType x, genIType y, genIType z) genUType min3(genUType x, genUType y, genUType z) | Returns the per-component minimum value of x, y, and z |
| genType max3(genType x, genType y, genType z) genIType max3(genIType x, genIType y, genIType z) genUType max3(genUType x, genUType y, genUType z) | Returns the per-component maximum value of x, y, and z |
| genType mid3(genType x, genType y, genType z) genIType mid3(genIType x, genIType y, genIType z) genUType mid3(genUType x, genUType y, genUType z) | Returns the per-component median value of x, y, and z |

Current hardware support: AMD Southern Islands

Expected hardware support: Future Hardware

# 4. Framebuffer

## 4.1. [GL\_AMD\_sample\_positions](http://www.opengl.org/registry/specs/AMD/sample_positions.txt)

Setting the sample positions is to me a very useful feature for post processing antialiasing but also for very high multisample rendering using multiple passes. Another approach is based on considering that the eye is a continuous integrator of a signal. Using different sample position per frame each frame will be slightly different and the eye will perceive less aliasing.

Current hardware support: AMD Evergreen

Expected hardware support: All OpenGL 5 hardware

## 4.2. [GL\_EXT\_framebuffer\_multisample\_blit\_scaled](http://www.opengl.org/registry/specs/EXT/framebuffer_multisample_blit_scaled.txt)

This extension is collaboration between Apple and NVIDIA. It seems design to handle the high DPI screens by allowing in a single call of [glBlitFramebuffer](http://www.opengl.org/sdk/docs/man/html/glBlitFramebuffer.xhtml) to resolve a multisampled framebuffer and scale the resulting framebuffer.

Current hardware support: NVIDIA G80

Expected hardware support: All OpenGL 5 hardware

## 4.3. [GL\_NV\_multisample\_coverage](http://www.opengl.org/registry/specs/NV/multisample_coverage.txt) and [GL\_NV\_framebuffer\_multisample\_coverage](http://www.opengl.org/registry/specs/NV/framebuffer_multisample_coverage.txt)

NVIDIA uses something call multisample coverage which allows to have coverage samples than color samplers. Hence, the implementation can adapted the multisampling according to the number of coverage samples covering the color sample.

In the second party of this article, we will discuss the possibilities for blending, stencil, profiling, rendering pipeline and misc features. We will discuss the announcements at GDC if there is enough public information given away that I could discuss about relevant things. We will conclude by my personal wish list for OpenGL 5 and Direct3D 12 hardware class.

Current hardware support: NVIDIA G80

## 4.4. [GL\_AMD\_depth\_clamp\_separate](http://www.opengl.org/registry/specs/AMD/depth_clamp_separate.txt)

[NV\_depth\_clamp](http://www.opengl.org/registry/specs/NV/depth_clamp.txt) introduced the concept of depth clamping so that primitive rendered outside the view near and far planes can be no longer clipped but the depth value is clamped in the depth range. [AMD\_depth\_clamp\_separate](http://www.opengl.org/registry/specs/AMD/depth_clamp_separate.txt) goes a step further, by independently enabling depth clamping on either the near or far plans.

Current hardware support: AMD RV670

# 5. Blending

Programmable blending has been on the wish list of many graphics programmers for a long time. There are three possible approaches forward: Through a per tile shader stage, through a new per pixel shader stage or by modifying the fragment shader stage. The Khronos Group released [EXT\_pixel\_local\_storage](https://www.khronos.org/registry/gles/extensions/EXT/EXT_shader_pixel_local_storage.txt) OpenGL ES extension for PowerVR Series 6 and ARM Mali T700. It’s a huge step toward the fragment shader stage approach. However it seems that the three approaches are pretty realistic for future hardware.

## 5.1. [GL\_NV\_texture\_barrier](http://www.opengl.org/registry/specs/NV/texture_barrier.txt)

Texture barrier is an NVIDIA extension but that extension has been implemented by all vendors (even Apple!). This extension was the very first step toward programmable blending allowing reading once and writing once at the same pixel location within a fragment shader invocation.

Current hardware support: AMD R600, NVIDIA G80

Expected hardware support: Intel Sandy Bridge

## 5.2. [GL\_EXT\_pixel\_local\_storage](https://www.khronos.org/registry/gles/extensions/EXT/EXT_shader_pixel_local_storage.txt) (OpenGL ES)

Current hardware support: Imagination Technologies PowerVR Rogue, ARM Mali T700

Expected hardware support: Future hardware

## 5.3. [GL\_EXT\_shader\_framebuffer\_fetch](https://www.khronos.org/registry/gles/extensions/EXT/EXT_shader_framebuffer_fetch.txt) (OpenGL ES)

## 5.4. [GL\_ARM\_shader\_framebuffer\_fetch\_depth\_stencil](https://www.khronos.org/registry/gles/extensions/ARM/ARM_shader_framebuffer_fetch_depth_stencil.txt) (OpenGL ES)

## 5.5. [GL\_INTEL\_fragment\_shader\_ordering](http://www.opengl.org/registry/specs/INTEL/fragment_shader_ordering.txt)

Current hardware support: Intel Haswell, AMD Southern Islands

Expected hardware support: Future hardware

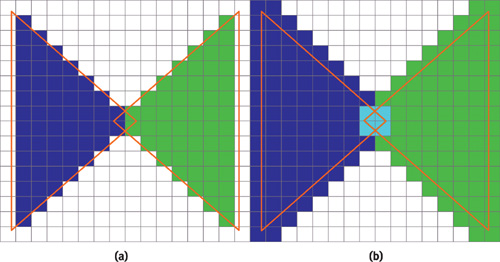
## 5.6. [GL\_INTEL\_conservative\_rasterization](http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter42.html)

If we explore [glCapsViewer](http://delphigl.de/glcapsviewer/listreports.php?listreportsbyextension=GL_INTEL_conservative_rasterization) database we will see an extension called [INTEL\_conservative\_rasterization extension exposed in an Intel HD 4600 GPUs](http://delphigl.de/glcapsviewer/listreports.php?listreportsbyextension=GL_INTEL_conservative_rasterization). The specification hasn’t been released but conservative rasterization has been largely described in the [Chapter 42](http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter42.html) of GPU Gem 2.

There are two variants of conservative rasterization:

* Overestimated conservative rasterization: A polygon includes all pixels for which the intersection between the pixel cell and the polygon is nonempty.
* Underestimated conservative rasterization: A polygon includes only the pixels whose pixel cell lies completely inside the polygon.

Use cases for conservative rasterization are GPU based collision detections and occlusion culling. With currently OpenGL 4 hardware to ensure somewhat correct result with need to keep framebuffer resolution high to reduce (but not avoid!) missing intersections. With conservative rasterization, we can get all the intersections and even save some fill-rate and bandwidth if this is useful.

**  
Figure 5.6.1: Comparing standard (a) and overestimated conservative (b) rasterization (GPU Gems 2)

Current hardware support: Intel Haswell

Expected hardware support: Future hardware

## 5.7. [GL\_KHR\_blend\_equation\_advanced](http://www.opengl.org/registry/specs/NV/blend_equation_advanced.txt)

This extension is the embodiment for why we need programmable blending. Clearly, most of the new blend equations of this extension are no computed by the ROPs but by the shader invocations. It seems that an application could use image load and store and a compute shader to perform the same behavior.

Current hardware support: NVIDIA Fermi

## 5.8. [GL\_AMD\_blend\_minmax\_factor](http://www.opengl.org/registry/specs/AMD/blend_minmax_factor.txt)

This extension provides two new blend equations that produce the minimum or maximum of the products of the source color and source factor, and the destination color and destination factor.

|  |  |  |
| --- | --- | --- |
| Mode | RGB Components | Alpha Component |
| GL\_FACTOR\_MIN\_AMD | R = min(Rs \* Sr, Rd \* Dr) G = min(Gs \* Sg, Gd \* Dg) B = min(Bs \* Sb, Bd \* Db) | A = min(As \* Sa, Ad \* Da) |
| GL\_FACTOR\_MAX\_AMD | R = max(Rs \* Sr, Rd \* Dr) G = max(Gs \* Sg, Gd \* Dg) B = max(Bs \* Sb, Bd \* Db) | A = max(As \* Sa, Ad \* Da) |

Current hardware support: AMD Northern Islands

# 6. Stencil

## 6.1. GL\_[AMD\_shader\_stencil\_export](http://www.opengl.org/registry/specs/AMD/shader_stencil_export.txt)

This extension exposed the fragment shader built-in output variable gl\_FragStencilRefAMD, allowing writing per invocation the stencil reference value used by the stencil test. For example, the extension allows writing directly to the stencil buffer when the stencil operation is set to GL\_REPLACE.

Current hardware support: AMD RV670

## 6.2. GL\_[AMD\_stencil\_operation\_extended](http://www.opengl.org/registry/specs/AMD/stencil_operation_extended.txt)

The stencil operation takes the three arguments sfail, dpfail and dppass describing the operations for updating the stencil buffer. With OpenGL 4.4 the available operation a pretty trivial: GL\_KEEP, GL\_ZERO, GL\_REPLACE, GL\_INCR, GL\_DECR, GL\_INVERT, GL\_INCR\_WRAP and GL\_DECR\_WRAP. This AMD extension adds new possible operations for the stencil buffer:

* GL\_SET\_AMD ; (setting to the maximum representable value)
* GL\_AND ;
* GL\_XOR ;
* GL\_OR ;
* GL\_NOR ;
* GL\_EQUIV ;
* GL\_NAND ; and
* GL\_REPLACE\_VALUE\_AMD (replacing with the operation source value instead of the reference value)

This extension also separate the value used for the stencil tests from the value used for the stencil operation. The operation value can be set with glStencilOpValueAMD for either face.

Current hardware support: AMD Southern Islands

## 6.3. GL\_[AMD\_shader\_stencil\_value\_export](http://www.opengl.org/registry/specs/AMD/shader_stencil_export.txt)

[AMD\_stencil\_operation\_extended](http://www.opengl.org/registry/specs/AMD/stencil_operation_extended.txt) decouples the stencil reference value (gl\_FragStencilRefAMD) from the stencil operation value.

This extension introduces a new fragment shader built-in output variable called gl\_FragStencilValueAMD allowing writing the operation value per shader invocation.

Current hardware support: AMD Southern Islands

# 7. Rendering pipeline

## 7.1. [GL\_AMD\_vertex\_shader\_layer](http://www.opengl.org/registry/specs/AMD/vertex_shader_layer.txt)

OpenGL 3 hardware introduced layered rendering allowing rendering each primitive to a different framebuffer attachment. However, to leverage this functionality, we need to use a geometry shader to specify [gl\_LayerID](http://www.opengl.org/sdk/docs/man/html/gl_Layer.xhtml) per generated primitive. Using a geometry shader is not free on contrary of setting [gl\_LayerID](http://www.opengl.org/sdk/docs/man/html/gl_Layer.xhtml). Following this reasoning, AMD published [AMD\_vertex\_shader\_layer](http://www.opengl.org/registry/specs/AMD/vertex_shader_layer.txt) which allows to set [gl\_LayerID](http://www.opengl.org/sdk/docs/man/html/gl_Layer.xhtml) in the vertex shader. Considering that mobile GPUs don't have a geometry shader, it would be particularly useful to use layered rendering.

## 7.2. [GL\_AMD\_vertex\_shader\_viewport\_index](http://www.opengl.org/registry/specs/AMD/vertex_shader_viewport_index.txt)

This extension follows the same reasoning than [AMD\_vertex\_shader\_layer](http://www.opengl.org/registry/specs/AMD/vertex_shader_layer.txt), enabling to choose the rendering view port [gl\_ViewportIndex](http://www.opengl.org/sdk/docs/man/html/gl_ViewportIndex.xhtml) from the vertex shader stage.

## 7.3. [GL\_AMD\_transform\_feedback3\_lines\_triangles](http://www.opengl.org/registry/specs/AMD/transform_feedback3_lines_triangles.txt)

With OpenGL 4.4, the application can use multiple transform feedback streams but only the first stream can output primitives that are not points. [AMD\_transform\_feedback3\_lines\_triangles](http://www.opengl.org/registry/specs/AMD/transform_feedback3_lines_triangles.txt) removes this restriction. Any primitive can be generated with any stream.

## 7.4. [GL\_AMD\_transform\_feedback4](http://www.opengl.org/registry/specs/AMD/transform_feedback4.txt)

[AMD\_transform\_feedback4](http://www.opengl.org/registry/specs/AMD/transform_feedback4.txt) extends [AMD\_transform\_feedback3\_lines\_triangles](http://www.opengl.org/registry/specs/AMD/transform_feedback3_lines_triangles.txt) so that each stream can be rendered in a single draw.

## 7.5. [GL\_AMD\_occlusion\_query\_event](http://www.opengl.org/registry/specs/AMD/occlusion_query_event.txt)

OpenGL provides occlusion queries to count the number of fragments that pass the tests. This extension provides finer queries to determine the number of fragments that pass specific tests.

|  |  |
| --- | --- |
| GL\_QUERY\_DEPTH\_PASS\_EVENT\_BIT\_AMD | Indicates that the fragment passed all tests |
| GL\_QUERY\_DEPTH\_FAIL\_EVENT\_BIT\_AMD | Indicates that the fragment passed the depth bounds and stencil tests, but failed the depth test |
| GL\_QUERY\_STENCIL\_FAIL\_EVENT\_BIT\_AMD | Indicates that the fragment passed the depth bounds test but failed the stencil test |
| GL\_QUERY\_DEPTH\_BOUNDS\_FAIL\_EVENT\_BIT\_AMD | Indicates that the fragment failed the depth bounds test |
| GL\_QUERY\_ALL\_EVENT\_BITS\_AMD | Indicates that any event generated by the fragment should be counted |

## 7.6. Hardware rings

# 8. Profiling

# 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](http://www.opengl.org/registry/specs/EXT/direct_state_access.txt). 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 comfortable 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](http://www.khronos.org/registry/cl/specs/spir_spec-1.2-provisional.pdf), 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](http://www.opengl.org/registry/specs/ARB/sparse_texture.txt). 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](http://www.opengl.org/registry/specs/ARB/indirect_parameters.txt) 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](https://www.google.co.uk/search?q=order+independent+transparency&spell=1&sa=X&ei=vZf1UeLjGMXEtAbim4CwCA&ved=0CCoQvwUoAA) (OIT). [NV\_blend\_equation\_advanced](http://www.opengl.org/registry/specs/NV/blend_equation_advanced.txt) or [AMD\_blend\_minmax\_factor](http://www.opengl.org/registry/specs/AMD/blend_minmax_factor.txt) 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](http://www.opengl.org/registry/doc/glspec44.core.withchanges.pdf)
* [GLSL 4.4 specification](http://www.opengl.org/registry/doc/GLSLangSpec.4.40.diff.pdf)
* [OpenGL 4.4 review](http://www.g-truc.net/doc/OpenGL%204.4%20review.pdf)
* [OpenGL 4.3 review](http://www.g-truc.net/doc/OpenGL%204.3%20review.pdf)
* [OpenGL 4.2 review](http://www.g-truc.net/doc/OpenGL%204.2%20review.pdf)
* [OpenGL 4.1 review](http://www.g-truc.net/doc/OpenGL%204.1%20review.pdf)
* [OpenGL 4.0 review](http://www.g-truc.net/doc/OpenGL%204.0%20review.pdf)
* [OpenGL 3.3 review](http://www.g-truc.net/doc/OpenGL%203.3%20review.pdf)
* [OpenGL 4.4 Pipeline Map](http://www.g-truc.net/doc/OpenGL%204.4%20Pipeline%20Map.pdf)
* [OpenGL ES 3.0 Pipeline Map](http://www.g-truc.net/doc/OpenGL%20ES%203.0%20Pipeline%20Map.pdf)

A big thanks to [Vladimir Hrincar](https://twitter.com/LokimanSK) for the detailed review of the review and [Daniel Rákos](https://twitter.com/aqnuep) for sending his comments on the new extensions.