Candidate features for future OpenGL 5 / Direct3D 12 hardware and beyond

3 May 2014, Christophe Riccio

G-Truc Creation

Table of contents

Introduction

The announcement of Mantle has triggered a lot of discussions about graphics API design. I think there are technical issues in the OpenGL API 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 actual major graphics APIs issues that are ecosystem related in my opinion:

How to reliably solve cross compilation between shader languages?

Aras Pranckevičius is regularly discussing this issue on his blog as this remains a major headache. I believe the approach followed by HLSLCrossCompiler is potentially the most reliable way to solve this issue today but to my knowledge HLSLCrossCompiler hasn't been battlefield tested yet.

How to debug and profile graphics code including shader languages?

NVIDIA has Nsight; Intel has GPA; AMD has CodeXL; each mobile vendor also has its things. Debugging and profiling graphics code is a mess and the tools are not even good. Despite the fancy GUIs, I am still relying on GLIntercept that is still doing the job best to this point for me: It's reliable. Sadly, no Linux or MacOS support. Thanks to Valve, there is new hope with VOGL which is taking the right directions: open source and targetting Linux, Windows and MacOSX support. If IHVs really want to solve that problem, I think allocating some engineering time on that project is the move forward.

How to scale shaders compilation?

Most engines work with the concept of shader variations: Each little shader feature comes with multiple variations to support performance scalability and fallbacks. Combining all these little shader features create a shader but the engine needs to generate shaders for each variation, performance level and fallbacks. This quickly produce thousands of shaders with a significant compilation time making real time editing of shaders only hypothetical in the near future. Furthermore, topics like Physically Based Rendering might remove a lot of illegimate artist hand peaked edits but energy concervation is such a complex issue that the number of variations will keep increasing.

How to get a good understanding of how GPUs work?

Some vendors like Intel and AMD have made an amazing contributing to the graphics community by publishing their GPU specifications. NVIDIA has some documentation with the PTX ISA but it's far from exhaustive. For example, I still don't understand how NVIDIA manage to support any indexing of resources where other vendors only support dynamically uniform indexing. More importantly, I don't understand the concequences so it's a dead feature to me. Mobile vendors are kind of making an effort these days but it's not nearly as good as it is needed especially considering how specific they are compare with desktop GPUs. Finally, GPU specifications allow clearing out mythologies, pretty well spread these days because of too many missleading API design discussions.

How to ensure that ecosystem platforms will get updated drivers?

How good is a new feature or a driver bug fix if the user doesn't have the driver supporting these improvements? Vendors are putting in place new driver notifications systems or the drivers get package

with the OS updates. In practice, only OSX and iOS seems to really manage to keep their ecosystems up to date.

How to get consistent performance level between ecosystem platforms?

This is particularly an issue with OpenGL where the ecosystem is really complex: On Windows, Linux or Android, IHVs provide each the full software stacks. On MacOSX, Apple wrote most of the OpenGL drivers. Furthermore, each vendor has a different level of quality and coverage of OpenGL leaving the ecosytem particularly fragmented. Conscenquently, graphics engines are filled with arrays of flags for supported features, workarounds and implementation bugs.

Why should we care about new graphics APIs? For an essential side of real time rendering evolution: Exposing the new hardware features. The purpose of a graphics API is nothing but exposing the hardware to the graphics programmers. If we use the OpenGL API with the GPU architectures in mind, there is no reason to suffer of CPU overhead.

With Direct3D, Microsoft was able to drive the standardization of hardware features. Meanwhile, we experienced structural changes in the ecosystem: Windows became just another platform; new consoles are based on modern architecture; Valve announced the Steam Box on Linux; the mobile market became relevant for every actors; WebGL transformed the web browsers into real-time rendering platforms; etc. Furthermore, The Khronos Group became a force capable to resolve ecosystem issues through hardware standardization. ASTC texture format gives a good example, with an expected support on all future mobile and desktop GPUs.

The future of real-time graphics will pass by hardware standardization through a new hardware level beyond the OpenGL 4 hardware level. What would be 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 looking at hardware features available through OpenGL extensions and possible ideas that may or may not be standardize. We will have a particular focus on the two mains topics that will drive future GPU designs in my opinion: Programmable vertex pulling and programmable blending.

- What are the vendor specific features currently supported?
- What are the current limitations?
- What are the possible future hardware directions?

1. Draw submission

Draw submission has been a subject of API evolution since the very first version of OpenGL. With OpenGL 1.0, it was done through the *immediate mode* using **glBegin**/**glEnd**. 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 (GL1.1);
- Vertex Buffer Object (GL1.5);
- Vertex Array Object (GL3.0);
- Base Vertex (GL3.2);
- Instancing (GL3.2);
- Instanced Arrays (GL3.3);
- Base Instance (GL4.2); and
- Vertex Attrib Binding (GL4.3)

OpenGL 4.3 took a new direction aiming at providing a magnitude of draw performances thanks to **ARB_multi_draw_indirect** leading the way to Programmable Vertex Pulling. With this approach, the GPU takes over the CPU responsible to dispatch the draws.

1.1. GL_ARB_multi_draw_indirect

By batching the data of multiple vertex array objects (VAOs) into a single VAO and calling **glDrawArraysInstancedBaseInstance** or **glDrawElementsInstancedBaseVertexBaseInstance** many times in a tight loop, we can archive extremely good performances with a very low CPU overhead. We can push this concept further, by regrouping textures into texture arrays, by storing 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 charge of submitting the draws and 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**, 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** can be coupled with **ARB_compute_shader** 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** or **glMultiDrawArraysIndirect**.

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. 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 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

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** behaves for draw instancing. A massive difference between these new vertex shader inputs and **gl_InstanceID** 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; using **gl_BaseVertexARB** and **gl_VertexID** to fetch ourselves the vertex data.

Reading the Southern Islands programming guide, we see that implementing such functionality means in AMD architecture that the Constant Engine (a GPU block part of the command processor) has to write the value of **gl_DrawID** into SH registers. It seems that such operation should not be a real issue as this is already how **BaseVertex** and **BaseInstance** are passed to the vertex shader stage for the fetch shader. At the very least exposing **gl_BaseVertexARB or gl_BaseIntanceARB** is trivial on Southern Islands architecture. AMD has just release drivers supporting this extension so I haven't been able to test it yet.

Surviving without gl_DrawID presents an alternative approach to **gl_DrawID** to perform per-draw indexing of resources. Unfortunately, the presented technic based on **BaseInstance** and the divisor is faster than **gl_DrawID** on NVIDIA hardware for the moment.

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

1.3. GL ARB indirect parameters

An issue of **ARB_multi_draw_indirect** is that it requires that we submit the number of draws from the CPU side, as **drawcount** is a **glMultiDrawElementsIndirect** parameter. What if we use a compute shader to build the indirect multi-draw buffer? We need to query on the CPU side the number of draws stored in that buffer to feed the **drawcount** parameter: Very inefficient because we may stall the CPU waiting on the query result. 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 in a multi draw

With **ARB_multi_draw_indirect** 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 a *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 Southern Islands architecture, it appears that each execution unit has a dedicated shader code pointer. 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`**, it seems unlikely to be able to implement such behavior to provide **`ARB_multi_draw_indirect`** like performance for shader code path switching.

Southern Islands introduced a new class or register called SH that can contain frequently update registers including user data or shader code pointer (program base). Once again, Southern Islands architecture doesn't seem far off for such future idea.

Expected hardware support: Southern Islands, Future hardware

1.5. Shader indexed lose states

OpenGL has a lot of lose states. Many of them could disappeared thanks to fully programmable blending: these include the fixed function blend states, dithering and the logical operations. Other seems to remain relevant for a while, including the following states:

- Scissor test
- Depth test
- Stencil operations
- Face culling
- Polygon mode
- Polygon offset

- DrawBuffers indirection

With OpenGL 4.1 and **ARB_viewport_array**, the OpenGL ARB introduced shader indexed lose-states for the viewport in OpenGL. Writing to **gl_ViewportIndex** in the geometry shader, we can choose within the shader code which one of the **GL_MAX_VIEWPORTS** viewports should be used to rasterize a primitive.

Enabling such indexing of lose-states by either the command processor or the shader invocations will allow pushing forward the multi draw indirect approach for more complex scenarios. Tile based GPU architectures might benefit the most of such approach: it allows reducing the number of draw and dispatch calls but also it could be used to reduce the number of rendering passes necessary for a frame, particularly reducing the bandwidth consumption and the CPU overhead of tile based GPUs.

Expected hardware support: Future mobile hardware first followed by future desktop hardware

1.6. GL_NV_bindless_multi_draw_indirect

This is the last piece of NVIDIA bindless API allowing draw submission without binding vertex arrays or indirect draw buffer for lower CPU overhead. With OpenGL 5 hardware, we could consider vertex arrays deprecated but a bindless indirection draw buffer remains a step forward.

Current hardware support: NVIDIA Fermi
Expected hardware support: OpenGL 5 hardware

1.7. GL_AMD_interleaved_elements

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 some software actually generating meshes using multiple element arrays and the current solution is to duplicate attributes on the CPU. This functionality avoids the CPU cost for the attribute duplications and it saves the extra attributes bandwidth.

Current hardware support: AMD Southern Islands

2. Resources

2.1. GL_ARB_bindless_texture

GL_ARB_bindless_texture was promoted from **GL_NV_bindless_texture**. 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.

Unfortunately, this extension can't be implemented on Haswell as it is specified. Haswell supports bindless ressources but this extension requires having both the sampler states and the texture states to be bindless. Haswell only supports bindless texture states but sampler states remain registers which realistically makes more sense. Do we really need a different sampler per texture? Not really.

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

2.2. GL_NV_shader_buffer_load and GL_NV_shader_buffer_store

NVIDIA buffer load and store is pretty much a set of bindless buffer extensions. Going toward such design really emphasizes 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 way we manage any form of memory.

A major different with **ARB_shader_storage_buffer_object** extension in OpenGL 4.3 is that the access to the data is performed through a pointer in the shader code.

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

2.3. GL_ARB_sparse_texture

ARB_sparse_texture is a subset of **AMD_sparse_texture** enabling virtual texturing with seamless texture filtering. Thanks to this extension we can create 16K by 16K 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

ARB_sparse_texture is essentially a fixed design of **AMD_sparse_texture**. 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** 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** 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`** 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

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

`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.

Unfortunately, we can't map textures created with **GL_LAYOUT_DEFAULT_INTEL** memory layout but this is only a driver limitation as Intel exposes its memory layout in its Developer's Guide. 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 capability, IHVs 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

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

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 its 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

2.13. Unified memory

With Haswell architecture, Intel has announced a Direct3D extension called **`InstantAccess`** aiming at giving access to the same memory to the CPU and the GPU. With Southern Islands, AMD introduced a first step to this idea with **`AMD pinned memory`** that allows creating an OpenGL buffer object out of users CPU side memory.

Among the many possible use cases, the hardware support of **`InstantAccess`** could ensure the best possible efficiency for persistent mapped buffers introduced with OpenGL 4.4.

Current hardware support: Intel Haswell, AMD Southern Islands (partial support)
Expected hardware support: Future hardware

3. Shader operations

3.1. GL_ARB_shader_group_vote

Branching is a very interesting topic with GPUs. A view about GPUs is that they are designed arounf two concepts: How we access memory and how multiple shader invocations diverge. From that view, ALUs are just the cherry on the cake. Branching is a very important when it comes to performance. For example, in AMD Southern Islands architecture, 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`**, a small subset of **`NV_gpu_shader5`** which provides the GLSL functions **`anyInvocationARB`**, **`allInvocationsARB`**, **`allInvocationsEqualARB`** to compare values across shader invocations and take decisions based on those results.

```
if(allInvocationsARB(condition))
	result = do_fast_path();
else
	result = do_general_path();
```

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

3.2. GL_NV_shader_thread_group

This extension goes into the *super resolution* range of ideas where we no longer want to think at a fixed pixel resolutions but instead at higher or lower resolution than the native resolution.

GPUs don't actually execute anything on a per-pixel or a per-vertex rate but in many different kind of group. A first group, the warp/wavefront is actually an array of shader invocations. Another famous group is the quadpixel, a set of 4 fragments. The texture LOD calculation is computed per quadpixel because the analytic computation of derivatives required for the texture LOD computation is very complex. However, 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 swizzling the intermediate results across all fragment shader invocations. 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`**.

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

3.3. GL_NV_shader_thread_shuffle

This extension extends **`NV_shader_thread_group`** to any of the shader invocations of a wrap/wavefront. It seems very likely that we could use **`NV_shader_thread_group`** on any GPU because all GPUs use quadpixels however, the shader invocation group 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 framebuffer antialiazing and maybe things like soft shadows.

Current hardware support: NVIDIA Kepler
Expected hardware support: Future hardware

3.4. GL NV shader atomic float

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

Interaction with:	New GLSL atomic operation functions
`NV_shader_buffer_store`	`float imageAtomicAdd(IMAGE_PARAMS, float data)` `float imageAtomicExchange(IMAGE_PARAMS, float data)`
`ARB_shader_image_load_store`	`float atomicAdd(float *address, float data);` `float atomicExchange(float *address, float data);`
`ARB_shader_storage_buffer_object`	`float atomicAdd(inout float mem, float data);` `float atomicExchange(inout float mem, float data);`

Only atomic counter operations are not affected by this extension.

Current hardware support: NVIDIA Fermi
Expected hardware support: Future hardware

3.5. GL AMD shader atomic counter ops

`ARB_shader_atomic_counters` and OpenGL 4.2 introduced the concept of atomic counter operations: increment, decrement and query. Atomic counters are designed to expose the fastest atomic operations.

AMD GPUs support these atomic operations in GDS memory which is faster than image and buffer atomic operations. However, AMD GPUs support more GDS atomic operations: 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`** exposes all these operations.

Current hardware support: AMD Southern Islands

3.6. GL ARB compute variable group size

The purpose of this extension is simply to specify the sizes of a workgroup at dispatch time instead of compile time. This is an OpenCL 1.2 features but it can not be efficiently implemented by AMD current GPUs as the implementation would have to recompile the shaders for each different set of size. However, this extension is natively supported by NVIDIA architechtures.

```
// 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. Multi compute dispatch

Just like draw indirect benefits from multi draw indirect, it seems that compute dispatch indirect could benefit from multi compute dispatch. With such feature we could imagine compute shaders designed to build the list of compute dispatch to be executed.

Such feature would require a **gl_DispatchID** providing a way to index resources per dispatch. However, **gl_DispatchID** is equivalent to **gl_DrawID** which is pretty inefficient on NVIDIA hardware.

Expected hardware support: Future hardware

3.8. GL_NV_gpu_shader5

This extension was released with Fermi GPUs. It extends **ARB_gpu_shader5** with a variety of Fermi specific features at the time. It contains the features later promoted into **ARB_shader_group_vote**, the features picked up by **AMD_gpu_shader_int64** 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***);
- 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

3.9. GL_AMD_gpu_shader_int64

AMD_gpu_shader_int64 is a superset of the 64-bit support exposed by **NV_gpu_shader5** 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**, u**int64BitsToDouble**);
- 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 seems to be an obvious candidate to become an ARB extension and an OpenGL 5 hardware feature.

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

3.10. GL_AMD_gcn_shader

AMD has released an extension with miscellaneous features supported by Southern Islands ISA called AMD_gcn_shader:

- New cubemap addressing GLSL functions: **cubeFaceIndexAMD** to identify which cubemap face is addressed for a specific cubemap texture coordinate argument; **cubeFaceCoordAMD** to compute the 2D texture coordinates used to address that face.
- A new shader invocation group GLSL function: **ballotAMD** which returns a 64-bit bitfield indicating for each shader invocation of a wavefront whether the evaluation of an expression is true.

- A new timing GLSL function: **timeAMD** returns a 64-bit value representing the current clock as seen by the shader processor. Each shader invocation of a shader invocation group **timeAMD** may produce a different value.

timeAMD will be extremely useful to optimize shader code. While, being ALU bound is very unlikely on modern GPUs, this function can be used to study the behavior of atomic operations, texture fetching, branching, etc.

ballotAMD is also very useful in a similar way than **anyInvocationARB**, **allInvocationsARB** and **allInvocationsEqualARB** but with more control. With **ballotAMD** we can express ideas like if an expression is true for 75% of the shader invocations, choose code path A otherwise use code path B.

Current hardware support: AMD Southern Islands
Expected hardware support: Future hardware

3.11. GL_NV_vertex_attrib_integer_64bit

NV_vertex_attrib_integer_64bit requires **NV_gpu_shader5** 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

3.12. GL_AMD_shader_trinary_minmax

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

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. This is called temporal antialiasing.

Current hardware support: AMD Evergreen
Expected hardware support: All OpenGL 5 hardware

4.2. GL_EXT_framebuffer_multisample_blit_scaled

This extension is a collaboration between Apple and NVIDIA. It seems design to handle the high DPI screens by allowing in a single call of `glBlitFramebuffer` to resolve a multisampled framebuffer and scale the resulting framebuffer in a single operation.

Current hardware support: NVIDIA G80
Expected hardware support: All OpenGL 5 hardware

4.3. GL_NV_multisample_coverage and GL_NV_framebuffer_multisample_coverage

Multisample coverage is a method based on using more coverage samples than color samplers. The implementation can modulate the multisampling resolution according to the number of coverage samples covering the color sample.

For example, if each color sample is associated with four coverage samples but only 50% of its coverage samples are covered then we can apply an equivalent weight while resolving the multisampling for this color sample.

Current hardware support: NVIDIA G80

4.4. GL_AMD_depth_clamp_separate

`NV_depth_clamp` introduced the concept of depth clamping so that primitives rendered outside the view near and far planes can have their fragment depth values clamped in the depth range instead of clipping the primitives. `AMD_depth_clamp_separate` goes a step further, by independently enabling depth clamping on either the near or far planes.

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: By modifying the fragment shader stage, through a new per pixel shader stage or with a per tile shader stage. The Khronos Group released **`EXT_pixel_local_storage`** OpenGL ES extension for PowerVR Series 6 and ARM Mali T700 and Intel has released **`INTEL_fragment_shader_ordering`** that is also implemented by AMD drivers. It's a huge step toward the fragment shader stage approach but all three approaches are actualy pretty realistic for future hardware and standardization.

5.1. GL_NV_texture_barrier

Texture barrier is an NVIDIA extension but it has been largely implemented by others vendors (even Apple in MacOSX 10.9!). 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.

Effectively, this extension relaxes the interdiction to bind a texture that is also used as a framebuffer attachment. It also provides a mechanism to avoid read-after-write hazard.

Current hardware support: AMD R600, NVIDIA G80
Expected hardware support: Intel Sandy Bridge

5.2. GL_EXT_shader_framebuffer_fetch (OpenGL ES)

This extension provides a basic form of programmable blending providing an effective approach to replace the fixed function blending operations.

It allows declaring the fragment better outputs with an **`inout`** qualifier so that we can read the previous values stored in the framebuffer. We can logically think this behavior as giving access to the destination values of the blend equation to the fragment shader invocation.

Looking at the tiled base GPU architectures, this extension is a first step allowing reading the on-chip memory.

Current hardware support: Imagination Technologies PowerVR 5XT Series
Expected hardware support: Future hardware

5.3. GL_ARM_shader_framebuffer_fetch (OpenGL ES)

`ARM_shader_framebuffer_fetch` is a superset of **`EXT_shader_framebuffer_fetch`** providing a switchable mode to indicate that reading the framebuffer value should be performed once per sample or once per pixel. With **`EXT_shader_framebuffer_fetch`**, this is an undefined behavior.

Current hardware support: ARM Mali T700
Expected hardware support: Future hardware

5.4. GL_ARM_shader_framebuffer_fetch_depth_stencil (OpenGL ES)

ARM_shader_framebuffer_fetch and **EXT_shader_framebuffer_fetch** allow reading the framebuffer attachment color values previously stored. However, those extensions don't interact with the depth and stencil framebuffer attachments. **ARM_shader_framebuffer_fetch_depth_stencil** removes this limitation by exposing **gl_LastFragDepthARM** and **gl_LastFragStencilARM** input variables.

A use case of this extension is soft particle rendering in a single pass.

Current hardware support: ARM Mali T700
Expected hardware support: Future hardware

5.5. GL_EXT_pixel_local_storage (OpenGL ES)

EXT_pixel_local_storage provides the most advance programmable blending extension to date.

It allows writing data into a pixel local storage block instead of the framebuffer for on-chip memory storage. The application gets control over the reads, writes and invalidations of the on-chip memory for potential huge bandwidth and power savings.

For example, pixel local storage can be used to store the deferred shading / lighting G-Buffer and then subsequently resolve it with another fragment shader invocation using a different shader code. This is pretty similar to **EXT_shader_framebuffer_fetch** but it's explicit so that we could store and read the depth buffer. Furthermore, the pixel local storage can contain arrays so that we can store multiple fragments giving opportunities for Order Independent Transparency with no bandwidth cost.

Obviously, on-chip memory is pretty limited but GPU architectures could rely on the cache hierarchy to extend the pixel local storage size. This behavior is exposed in this extension by two constants: **GL_MAX_SHADER_PIXEL_LOCAL_STORAGE_FAST_SIZE_EXT** and **GL_MAX_SHADER_PIXEL_LOCAL_STORAGE_SIZE_EXT**.

```
#version 300 es
#extension GL_EXT_shader_pixel_local_storage : enable

__pixel_localEXT FragDataLocal {
        layout(r11f_g11f_b10f) mediump vec3 normal;
        layout(rgb10_a2) highp vec4 color;
        layout(rgba8ui) mediump uvec4 flags;
} gbuf;

/* .... */

void main()
{
        /* .... */
        gbuf.normal = v;
        gbuf.color = texture(sampler, coord);
        gbuf.flags = material_id;
}
```

Listing 5.5.1: Write data to pixel local storage block, example from **EXT_pixel_local_storage**

```
#version 300 es
#extension GL_EXT_shader_pixel_local_storage : enable

__pixel_localEXT FragDataLocal {
        layout(r11f_g11f_b10f) mediump vec3 normal;
```

```
        layout(rgb10_a2) highp vec4 color;
        layout(rgba8ui) mediump uvec4 flags;
} gbuf;

out highp vec4 fragColor;

void main()
{
        fragColor = do_lighting(gbuf.normal, gbuf.color, gbuf.flags, light_pos);
}
```

Listing 5.5.2: Resolve pixel local storage block, example from **EXT_pixel_local_storage**

As great this extension is, using the fragment shader stage for both filling the pixel local storage block and resolving a pixel local storage seem odd and limited. Other approach would be to enable compute shader resolution to store the final result in shader storage buffers or texture images. In such case we would have to figure out a way to express the window xy coordinates to locate the corresponding pixel local storage which is probably not easy.

Going further, we could imagine extending this feature to generalize local storage to any shader stage. It would be a form of compute shader shared memory that would be persistent after the completion of the shader invocations.

Another and more even more powerful approach would be adding a dedicated and independent tile shader stage and give it read only access to the pixel local storage blocks for all the pixel of a tile. This approach effectively provides access to the entire on-chip memory.

Unfortunately, this extension is written against OpenGL ES 3.0 so there is no interaction with OpenGL ES 3.1 or OpenGL 4.3 framebuffer with no attachments, shader storage buffer or texture images. Furthermore, it can't be used with multisampling. Effectively, the pixel local storage is tightly baked with the framebuffer and clears are even performed through the framebuffer clear functions as both pixel local storage and the fragment shader ouputs are aliased.

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

5.6. Tile shading

EXT_pixel_local_storage allows expressing the most sophisticated programmable blending to date but it seems that it should be possible to go one step further. What about adding a dedicated tile shader stage after the fragment shader stage? The fragment shader stage would only need to write data to the pixel local storage per pixel but the tile shader stage could read every data for each pixel of a tile to output a single pixel per tile shader invocation.

```
#extension GL_EXT_shader_pixel_local_storage2 : enable

__pixel_localEXT FragDataLocal {
        layout(r11f_g11f_b10f) mediump vec3 normal;
        layout(rgb10_a2) highp vec4 color;
        layout(rgba8ui) mediump uvec4 flags;
} gbuf[][];

const uvec2 gl_TileSize; // Sizes of the tile
```

```
in uvec2 gl_Tilecoord; // Pixel coordinates within the current tile
in uvec2 gl_TileID; // Identifier for the current tile reflecting the position in the window
```

Listing 5.5.3: Resolve pixel local storage block, example from **EXT_pixel_local_storage**

The use case of this could be to perform single pass, no bandwidth screen space antialiasing, some form of motion estimation, some blurring, per-tile evaluations, etc. Obviously, such shader stage would be bound to a tile which can quickly results in blocky artifacts so the graphics programmer would need to remain wise.

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

5.7. GL_INTEL_fragment_shader_ordering

GPUs guaranty that framebuffer writes are processed in primitive rasterization order however there is no guarantee for texture image or shader storage buffer reads and writes.

INTEL_fragment_shader_ordering introduces GLSL **beginFragmentShaderOrderingINTEL** function which stop the fragment shader invocation until all the previous fragment shader invocations for the same window **(x, y)** coordinates returns. All the memory transactions are guaranteed to be completed so that the fragment shader invocation could read the texture images and shader storage buffers previously written within a same draw.

Hence, this extension provides an elaborated form of programmable blending applying on both texture and buffer data. However, on the contrary to **EXT_pixel_local_storage**, this extension implies a high performance cost.

A possible approach for how AMD is implementing this extension is using the GDS memory (64KB of cache with built-in atomic operations shared across execution units). Tahiti (Radeon HD7900 ASIC) has 32 execution units, each working on 16 * 16 pixels tile for a total of 8192 pixels at a time (32 * 16 * 16). Hence, the GDS has 8 bytes per pixel which implies that trashing the cache will be very quick without screen space coherence. With atomic exchange, a wavefront would have to spin on **beginFragmentShaderOrderingINTEL** waiting for GDS changes preventing the launch of further shader invocation arrays to hide the wait. Southern Islands being capable to have up to 10 shader invocation array live per execution unit we are limiting ourselves to 1/10 of the GPU peak performance. Without careful profiling, using this extension will be particularly slow on AMD hardware.

I haven't found time yet to study Haswell hardware specification in enough detail but hopefully there are hardware improvements that may reduce this potential performance penalty.

Current hardware support: Intel Haswell, AMD Southern Islands
Expected hardware support: Future hardware

5.8. GL_KHR_blend_equation_advanced

This extension is the embodiment for why we need programmable blending. Clearly, most of the new blend equations of this extension are not computed by the ROPs but by shader invocations. It seems that an application could use image load and store and a compute shader to perform the same behavior: No more of that, thanks.

Current hardware support: NVIDIA Fermi

5.9. GL_AMD_blend_minmax_factor

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

This extension exposed the fragment shader built-in output variable **gl_FragStencilRefAMD**, allowing writing per fragment shader invocation the stencil reference value used for 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

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

AMD_stencil_operation_extended 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_INTEL_conservative_rasterization

If we explore glCapsViewer database we will see an extension called **`INTEL_conservative_rasterization`** extension exposed in an Intel HD 4600 GPUs. The specification hasn't been released but conservative rasterization has been largely described in the Chapter 42 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 non-empty.
- 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.

Another perspective for such feature would be to implementable a programmable form of *binning* on desktop and mobile GPUs. In tile based GPUs, *binning* is typically the fixed function step where primitives are sorted per tile.

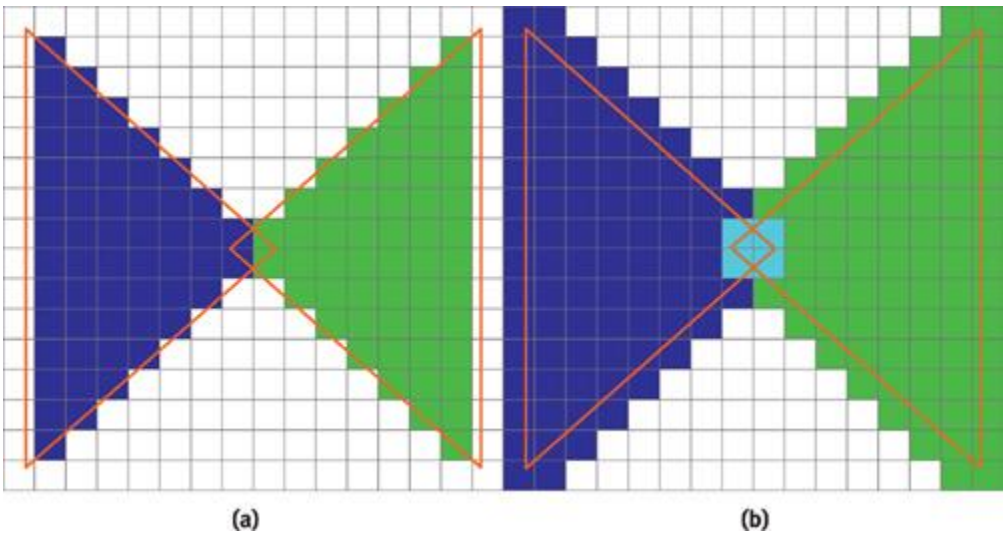

Figure 5.6.1: Comparing standard (a) and overestimated conservative (b) rasterization (GPU Gems 2)

Current hardware support: Intel Haswell
Expected hardware support: Future hardware

7.2. GL_AMD_vertex_shader_layer

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** per generated primitive. Using a geometry shader is not free on contrary of setting **gl_LayerID**. Following this reasoning, AMD published **AMD_vertex_shader_layer** which allows setting **gl_LayerID** in the vertex shader. Considering that mobile GPUs don't have a geometry shader, it would be particularly useful to use layered rendering.

Because most mobile hardware doesn't have a geometry shader stage, such feature would enable layered rendering on them.

Current hardware support: AMD Southern Islands
Expected hardware support: Future mobile hardware

7.3. GL_AMD_vertex_shader_viewport_index

This extension follows the same reasoning than **AMD_vertex_shader_layer**, enabling to choose the rendering view port **gl_ViewportIndex** from the vertex shader stage.

Current hardware support: AMD Southern Islands
Expected hardware support: Future mobile hardware

7.4. GL_AMD_transform_feedback3_lines_triangles

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** removes this restriction. Any primitive can be generated with any stream.

Current hardware support: AMD Southern Islands

7.5. GL_AMD_transform_feedback4

AMD_transform_feedback4 extends **AMD_transform_feedback3_lines_triangles** so that each stream can be rendered in a single draw even if the geometry shader output different type of primitives for each stream.

Hence, this extension allows rendering a single primitive both filled and in wireframe within a single draw.

Current hardware support: AMD Southern Islands

7.6. GL_AMD_occlusion_query_event

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

Current hardware support: Intel Haswell, AMD Sea Islands

7.7. WGL_AMD_gpu_association and WGL_NV_gpu_affinity

I have never really explored either **`AMD_gpu_association`** or **`NV_gpu_affinity`** by lack of interest of multi GPU solution. These extensions enable CrossFire and SLI on AMD and NVIDIA GPUs. Such feature could probably be standardized into an OpenGL ARB extension. Rendering a frame on one GPU and a second frame on the other GPU is a pretty relevant scenario for example.

Expected hardware support: OpenGL 3 hardware

8. Hardware rings and tasks parallelism

Graphics APIs such as OpenGL, Mantle or Direct3D12 exposes concepts such as display lists, command queues and command lists. Exploring hardware specifications, we see a gap between what these APIs exposed and the hardware architectures even on AMD hardware that is the friendliest to such concepts. What those feature expose, it is CPU side delayed compilation of states.

Considering command queues such as things that the GPU can execute in parallel is missing leading. On Southern Islands the GPU *Draw Engine* (DE) can execute a PM4 packet at a time. A PM4 packet can be seen as a macro instruction in the CPU world. Southern Islands introduce a new hardware block called the *Constant Engine* (CE) which aims at compensating the removal of the fixed hardware register for the shader resource descriptors. With Southen Islands, the shader resource descriptors are fetched from memory but cached by the Constant Engine. The Constant Engine has its own hardware ring so that both the Draw Engine and Constant Engine could execute PM4 packets in parallel. Neither the Draw Engine nor the Constant Engine can execute all the PM4 packets, each support dedicated subsets. DE and CE form what we call the *Command Processor* (CP) but really in Southern Islands it's two hardware blocks. Even if both can run in parallel, there are designed for instructions parallelism, not really tasks parallelism.

Southern Islands has two DMA engines that can do transfers on both directions: from device to client or from client to device memory. The DMA engines can run fully independently from the command processor.

Southern Islands also has 2 *Asynchronous Compute Engines* (ACE). These engines allow efficient multi-tasking with independent scheduling and workgroup dispatch. These engines can run in parallel with the Draw Engine without any form of contention. OpenCL 1.2 exposes them with something called device partitioning. Sea Islands raised the number of ACE to 8.

There is a lot of room for tasks parallelism in a GPU but the idea of submitting draws from multiple threads in parallel simply doesn't make any sense from the GPU architectures at this point. Everything will need to be serialized at some point and if applications don't do it, the driver will have to do it. This is true until GPU architectures add support for multiple command processors which is not unrealistic in the future.

For example, having multiple command processors would allow rendering shadows at the same time as filling G-Buffers or shading the previous frame. Having such drastically different tasks live on the GPU at the same time could make a better usage of the GPU as both tasks will probably have different hardware bottleneck.

Tasks parallelism is interesting as long as the architectures allow load balancing. On AMD Tahiti, there are 32 execution units and each of them can process independent tasks. However, Tahiti is a high-end GPU but the lower end parts have a lot less execution units. On mobile GPU, PowerVR GX6650 contains only 6 execution units despite that it is really high-end on mobile. NVIDIA Kepler uses fat execution units so that GK104 only has 8 execution units and Tegra K1 only has a single execution unit.

Current hardware support: AMD Southern Islands
Expected hardware support: Future hardware

Conclusions

I think we are going toward the convergence of the tile-based GPUs and the immediate mode GPUs. It's also particularly interesting to see architecture innovations coming from everywhere including both the desktop and mobile worlds. Few years ago, innovations would comes from either AMD or NVIDIA but this is long gone: ARM standalized ASTC texture format, Intel introduced fragment shader invocation ordering and I haven't even mentioned Imagination Technologies raytracing hardware for PowerVR 6 XT.

On current immediate mode hardware, it already makes a lot of sense to do tile based image computation like shading or data binning. On current tile based GPUs, it makes a lot of sense to enable programmable vertex pulling to ensure that the meshes will be process at fine granularity and with screen space coherence to avoid flushing on-chip memory to graphics memory.

I expect that future hardware will converge with both worlds moving toward each other. First, by enabling a full programmable vertex pulling allowing the GPUs to submit itself a lot of small draws with `MultiDrawIndirect` and executing significantly different shader code path per fine grain draws. Second, by introducing a tile shader stage used for a fully programmable blending allowing single pass deferred rendering, order-independent transparency or immediate antialiasing resolution for a massive bandwidth saving.

Could the OpenGL 5 hardware level be that fully programmable vertex pulling and programmable blending GPU architecture? Considering that is takes about three years to build a GPU, such architecture would have to already in the production pipeline of IHVs. That seems unlikely but it could be a nice OpenGL 6 hardware level.

Personnally, I would be happy with Southern Islands being used to define the OpenGL 5 hardware level and call the day. Southern Islands is the botton line for the new consoles hence a target for a large number of people for the years to come and it's amazingly documented. Hence, an OpenGL 5 hardware would have to support and expose:

- Hardware accelerated MultiDrawIndirect
- Per-draw shader code path in a multi draw (As a bonus, not sure that's possible on S.I.)
- Bindless / unlimited number of ressources
- Virtual memory for sparse ressources
- Two fully asynchronous DMA engines for upload and download
- Fragment shader invocation ordering

Thanks to Patrick Cozzi for the review of this article.

References

Jon Leech, The OpenGL® Graphics System: A Specification (Version 4.4 (Core Profile) - March 19, 2014)
http://www.opengl.org/registry/doc/glspec44.core.withchanges.pdf

John Kessenich, The OpenGL® Shading Language, Language Version: 4.40, Junuary 2014
http://www.opengl.org/registry/doc/GLSLangSpec.4.40.diff.pdf

Sascha Willems, OpenGL hardware database
http://delphigl.de/glcapsviewer/listreports2.php

Sascha Willems, OpenGL ES hardware database
http://delphigl.de/glcapsviewer/gles_launchpad.php

Christophe Riccio, OpenGL hardware matrix, Extensions exposed by OpenGL implementations, February 2014, http://www.g-truc.net/doc/OpenGL%20matrix%202014-02.pdf

Parallel Thread Execution ISA Version 4.0, February 2014
http://docs.nvidia.com/cuda/pdf/ptx_isa_4.0.pdf

Radeon Southern Islands Acceleration, April 2012
http://www.x.org/docs/AMD/old/si_programming_guide_v2.pdf

Southern Islands Series Instruction Set Architecture, August 2012
www.x.org/docs/AMD/old/AMD_Southern_Islands_Instruction_Set_Architecture.pdf

Radeon Southern Islands 3D/Compute Register Reference Guide, November 2011
http://www.x.org/docs/AMD/old/SI_3D_registers.pdf

Radeon Sea Islands 3D/Compute Register Reference Guide, September 2012
http://www.x.org/docs/AMD/old/CIK_3D_registers_v2.pdf

AMD Graphics Cores Next (GCN) Architecture, June 2012
http://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf#search=GCN%5FArchitecture%5Fwhitepaper

GPU Overview (Haswell), January 2014,
https://01.org/linuxgraphics/sites/default/files/documentation/intel-gfx-prm-osrc-hsw-gpu-overview_0.pdf

3D Media GPGPU Engine (Haswell), January 2014,
https://01.org/linuxgraphics/sites/default/files/documentation/intel-gfx-prm-osrc-hsw-3d-media-gpgpu-engine_0.pdf

Developer's Guide for Intel® Processor Graphics For 4th Generation Intel® Core™ Processors,
https://software.intel.com/sites/default/files/4th-gen-core-graphics-dev-guide.pdf