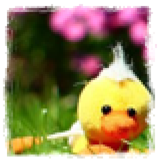
OpenGL 4.2 review

08 August 2011, Christophe Riccio



Introduction

Despite major new features like program separate, program binary caching and multiple viewports, OpenGL 4.1 didn't really expose major new OpenGL 4 hardware capabilities as all these functionalities are for OpenGL 3 or even OpenGL 2 hardware classes as well. Yes, we had 64 bits vertex attributes and some sort of clarification on the precision of some GLSL functions but that was all.

With OpenGL 4.2, it's a very different outcome especially regarding to two new extensions included in OpenGL 4.2 core specifications: `ARB_shader_atomic_counters` and `ARB_shader_image_load_store`. These two extensions open a new world of opportunities for rendering which I find much more interesting than the OpenGL 4 hardware tessellation.

Despite a lot of improvement and new features, once again this release is missing a full support of direct state access and we are still living without redistributable GLSL binary programs neither in core or extensions.

1. Atomic counter: GL_ARB_shader_atomic_counters

The atomic counter is a new opaque type which can be declare in any stage with up to 8 instances of them. The atomic counter must be backed by a buffer object which allows accessing on the application side to the values of an atomic counter.

An atomic counter is represented by a 32 bits unsigned int and only three operations can be performed on it: We can read the current value (`atomicCounter`), get the counter value then increment it (`atomicCounterIncrement`) and decrement the counter value then get the counter value (`atomicCounterDecrement`).

It is safe to increment or decrement in any shader stage and for any execution of this stage. However, incrementing and decrementing within an execution should be considered as if the operations on the atomic counter were unordered. The OpenGL specification doesn't define any mechanism (memory barrier) to ensure operations ordering.

```
// binding: unit where the atomic buffer is bound
// offset: buffer data offset, where the atomic counter value is stored.
layout(binding = 0, offset = 0) uniform atomic_uint a;
```

This extension is extremely simple to use. An atomic counter is an opaque type declared as a uniform variable just like the sampler object. A buffer object is bound to an atomic counter binding point but unlike sampler, we can't set the index of the bounding point with `glUniform1i`. It requires to be set within the GLSL program using the layout qualifier `<binding>`. This is also possible for samplers thanks to `ARB_shading_language_420pack`.

2. Image load and store: GL_ARB_shader_image_load_store

2.1. Load and store

`ARB_shader_image_load_store` is wonderful new extension but also the promoted extension to core of `EXT_shader_image_load_store` thanks to few changes between the two extensions. To introduce this extension, I think that there is nothing like the first sentence of the extension overview:

"*This extension provides GLSL built-in functions allowing shaders to load from, store to, and perform atomic read-modify-write operations to a single level of a texture object from any shader stage.*"

Incredible? This is exactly the feeling I had when I first discovered it.

OpenGL 4.2 clarifies and maybe actually specify what is an “image” in OpenGL. An image could be a single mipmap level or a single texture 2d array layer which composes a texture. Images are bound to “image units” using the command `glBindImageTexture` for a maximum minimum of 8 units in the fragment shader stage only according to the OpenGL 4.2 specifications. In practice, I believe that both AMD and NVIDIA hardware allow binding these units to the vertex shader stage as well and maybe any shader stage.

```
void glBindImageTexture(
	GLuint unit,
	GLuint texture,
	GLint level,
	GLboolean layered,
	GLint layer,
	GLenum access,
	GLenum format);
```

Most of the rest of this extension happens in GLSL and take the following shape for example:

```
layout(binding = 0, rgba8ui) uniform readonly image2D Image;
```

Two main GLSL functions are available to access any image of any format: `imageLoad` and `imageStore`. However, the following extended set of atomic functions is only available for images using the formats r32i and r32ui.

Atomic operations on integer types:
- `imageAtomicAdd`
- `imageAtomicMin`
- `imageAtomicMax`
- `imageAtomicAnd`
- `imageAtomicOr`
- `imageAtomicXor`
- `imageAtomicExchange`
- `imageAtomicCompSwap`

2.2. Memory barriers

These dynamic load and store involve some pretty complex memory management issues: When we are reading a data, are we sure the result of a previous operation is stored already? For this purpose, this extension provides a GLSL function and an OpenGL command which will make sure that the previous operations are executed: `glMemoryBarrier` and `memoryBarrier`.

`glMemoryBarrier` orders that the memory transactions of certain types are issued prior the commands after this barrier. This barrier apply only to selected types of data by the OpenGL programmer.

Flags that can be passed to glMemoryBarrier:

- GL_VERTEX_ATTRIB_ARRAY_BARRIER_BIT
- GL_ELEMENT_ARRAY_BARRIER_BIT
- GL_UNIFORM_BARRIER_BIT
- GL_TEXTURE_FETCH_BARRIER_BIT
- GL_SHADER_IMAGE_ACCESS_BARRIER_BIT
- GL_COMMAND_BARRIER_BIT
- GL_PIXEL_BUFFER_BARRIER_BIT
- GL_TEXTURE_UPDATE_BARRIER_BIT
- GL_BUFFER_UPDATE_BARRIER_BIT
- GL_FRAMEBUFFER_BARRIER_BIT
- GL_TRANSFORM_FEEDBACK_BARRIER_BIT
- GL_ATOMIC_COUNTER_BARRIER_BIT
- GL_ALL_BARRIER_BITS

memoryBarrier behaves the same way than glMemoryBarrier except that it doesn't expose a fine grain memory barrier and ensure instead that all memory accesses are performed prior to the synchronization point.

This set of feature is inherited from NV_texture_barrier which has demonstrated effective use cases outside the scope of OpenGL images, for example for ping-pong rendering and it provides significative performance benefices. I believe that this extension should have been exposed in a separated ARB extension to benefit OpenGL 3 hardware.

2.3. Early fragment tests

Early z-test is a GPU optimisation of the OpenGL execution pipeline which performs the depth test before running the fragment shader. With OpenGL 3 hardware, it is possible to determine the cases when the early z-test can't be perform, the cases when the fragment shader has to be executed. For example, if the depth test is enabled and the OpenGL programmer is changing the gl_FragDepth value.

With OpenGL 4 hardware and shader load and store, things became more fuzzy to determine if early fragment tests can be enable. The approach chosen is to delegate to the OpenGL user the duty of determining if early z-test can be perform or not.

Enabling the early fragment tests is done in the fragment shader simply as follow:

```
layout(early_fragment_tests) in;
```

I remain unsure whether this feature should be part of this extension or a separated one to benefit OpenGL 3 hardware.

3. GLSL improvements

3.1. Packing functions: GL_ARB_shading_language_packing

This extension provides a set of "pack" and "unpack" functions which allow loading and storing 16 bits floating values as an unsigned integer in a similar fashion than packDouble2x32 and unpackDouble2x32 using the new GLSL functions packHalf2x16 and unpackHalf2x16.

This extension also gathers some packing functions from ARB_gpu_shader5 so that the packing functions can be exposed by OpenGL 3 hardware. This includes `[un]packUnorm4x8`, `[un]packSnorm4x8`, `[un]packUnorm2x16` and the previously missing `[un]packSnorm2x16`. These functions are essential as they expose the normalization mechanisms of the hardware.

The feature set exposed by this extension is taking a new light with the release of OpenGL 4.2 and `ARB_shader_image_load_store` because atomic operations on image data can only been perform on signed and unsigned 32 bits integers.

3.2. Binding point initialization: GL_ARB_shading_language_420pack

OpenGL 4.2 provides two new opaque objects for both image types and atomic counter types. To associate variables declared with these types, GLSL provides the layout qualifier `<binding>`. The extra goodness is that this qualifier has been extended to sampler and uniform buffer objects, bye bye `glUniform1i` and `glUniformBlockBinding`, I won't miss you!

3.3. Shared uniform buffer array

With OpenGL 4.1, we can attach multiple uniform buffers to a shader stage (12 on NVIDIA, 15 on AMD). For each uniform block we can associate a different uniform buffer object using `glBindBufferBase` or share the use of uniform buffers using `glBindBufferRange`. However, until OpenGL 4.2, it wasn't possible to share a single uniform buffer with multiple element of an uniform block array… This specification's bug as been fixed.

```
// Bind a separate buffer object per block array element
// or share a single buffer object for the entire block array
uniform block
{
	...
} Block[4];
```

3.4. Various GLSL improvements: GL_ARB_shading_language_420pack

GLSL 4.20 brings a lot of clever little improvement which it listed below:

- ".length()" for vector and matrix types
- Order restriction on qualifiers is removed
- "const" can be used for variables declared in a function
- '\' as line-continuation character
- UTF8 character set support for shaders
- Implicit cast for values returned by a function
- Swizzle operators on scalar (float f; …; f.xxx)

4. Texture

4.1. Immutable textures: GL_ARB_texture_storage

The new `ARB_texture_storage` extension decouples the allocation and the initialisation of a texture object to provide immutable textures. After calling the new commands `glTexStorage*D`, the three commands `glTexImage*D`, `glCopyTexImage*D` and `glCompressedTexImage*D` can't be call anymore on this texture object without generating an invalid operation error.

Immutable texture objects are initialized with the commands glTexSubImage*D, glCompressedTexSubImage*D, glCopyTexSubImage*D and actually any command or set of commands that won't reallocate the memory for this object.

The purpose of immutable texture objects is mainly on the drivers side to avoid continuous complex completeness checking. Hence, it should provide some performance improvement without really affecting the user programmability. If a user really needs to change the format, the target or the size of a texture, he can always delete and create a new texture object which is pretty much what is happening anyway on mutable texture object.

Bonus of this extension: it provides some good interactions with EXT_direct_state_access. However, this extension doesn't provide any interaction with multisample textures so that it's not possible to create immutable multisample texture objects. This type of texture doesn't really have completeness checking as it doesn't hold mipmaps so the functionality itself isn't needed and we could only enjoy such command for consistency.

```
// OpenGL 4.1, mutable texture creation
glGenTextures(1, &Texture);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, Texture);
glTexParameteri(GL_TEXTURE_2D, GL_*, …);
for(std::size_t Level = 0; Level < Levels; ++Level)
	glTexImage2D(GL_TEXTURE_2D, …); // Allocation and initialisation

// OpenGL 4.2, immutable texture creation
glGenTextures(1, &Texture);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, Texture);
glTexParameteri(GL_TEXTURE_2D, GL_*, …);
glTexStorage2D(GL_TEXTURE_2D, ...); // Allocation
for(std::size_t Level = 0; Level < Levels; ++Level)
	glTexSubImage*D(GL_TEXTURE_2D, ...); // Initialisation

// OpenGL 4.2 + EXT_direct_state_access, immutable texture creation
glGenTextures(1, &Texture);
glTextureParameteriEXT(GL_TEXTURE_2D, GL_*, ... );
glTextureStorage2DEXT(GL_TEXTURE_2D, ...); // Allocation
for(std::size_t Level = 0; Level < Levels; ++Level)
	glTextureSubImage*DEXT(GL_TEXTURE_2D, ...); // Initialisation
```

4.2. Partial copy of compressed storage: GL_ARB_compressed_texture_pixel_storage

OpenGL provides a functionality which allows uploading a subset of a texture to graphics memory without creating temporary buffers. This is accomplished using glPixelStorei with the arguments GL_UNPACK_ROW_LENGTH, GL_UNPACK_SKIP_PIXELS and GL_UNPACK_SKIP_ROWS. It also allows downloading to global memory a subset of a texture store in graphics memory using the arguments `GL_PACK_ROW_LENGTH`, `GL_PACK_SKIP_PIXELS` and `GL_PACK_SKIP_ROWS`.

The following code uploads a subset of the image `<TEXTURE_DIFFUSE_RGB8>` which is half the size and the centre of original picture.

```
gli::texture2D Image = gli::load(TEXTURE_DIFFUSE_RGB8);

glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
glPixelStorei(GL_UNPACK_ROW_LENGTH, Image[0].dimensions().x);
glPixelStorei(GL_UNPACK_SKIP_PIXELS, GLsizei(Image[0].dimensions().x) / 4);
glPixelStorei(GL_UNPACK_SKIP_ROWS, GLsizei(Image[0].dimensions().y) / 4);

glTexImage2D(
	GL_TEXTURE_2D,
	GLint(0),
	GL_RGBA8,
	GLsizei(Image[0].dimensions().x) / 2,
	GLsizei(Image[0].dimensions().y) / 2,
	0,
	GL_BGR,
	GL_UNSIGNED_BYTE,
	Image[0].data());
```

However, this is designed to work on pixels. It is not good for compressed texture formats which are typically packed blocks of N by M pixels. Thus, this isn’t available for compressed texture format in OpenGL 4.1.

OpenGL 4.2 and `ARB_compressed_texture_pixel_storage` remove this limitation of compressed textures allowing `glPixelStorei` to control the way compressed texture are uploaded and downloaded from the GPU memory. To make this possible, arguments for `glPixelStorei` has been added:

New tokens for partial compressed texture data copy:

- `GL_UNPACK_COMPRESSED_BLOCK_WIDTH`
- `GL_UNPACK_COMPRESSED_BLOCK_HEIGHT`
- `GL_UNPACK_COMPRESSED_BLOCK_DEPTH`
- `GL_UNPACK_COMPRESSED_BLOCK_SIZE`
- `GL_PACK_COMPRESSED_BLOCK_WIDTH`
- `GL_PACK_COMPRESSED_BLOCK_HEIGHT`
- `GL_PACK_COMPRESSED_BLOCK_DEPTH`
- `GL_PACK_COMPRESSED_BLOCK_SIZE`

These arguments are used to specify the size of a block in pixels and in bytes giving to OpenGL implementations enough information to read and write subset of compressed images without splitting compression blocks.

The following code uploads a subset of the compressed texture `<TEXTURE_DIFFUSE_DXT1>` which is half the size and the centre of original picture.

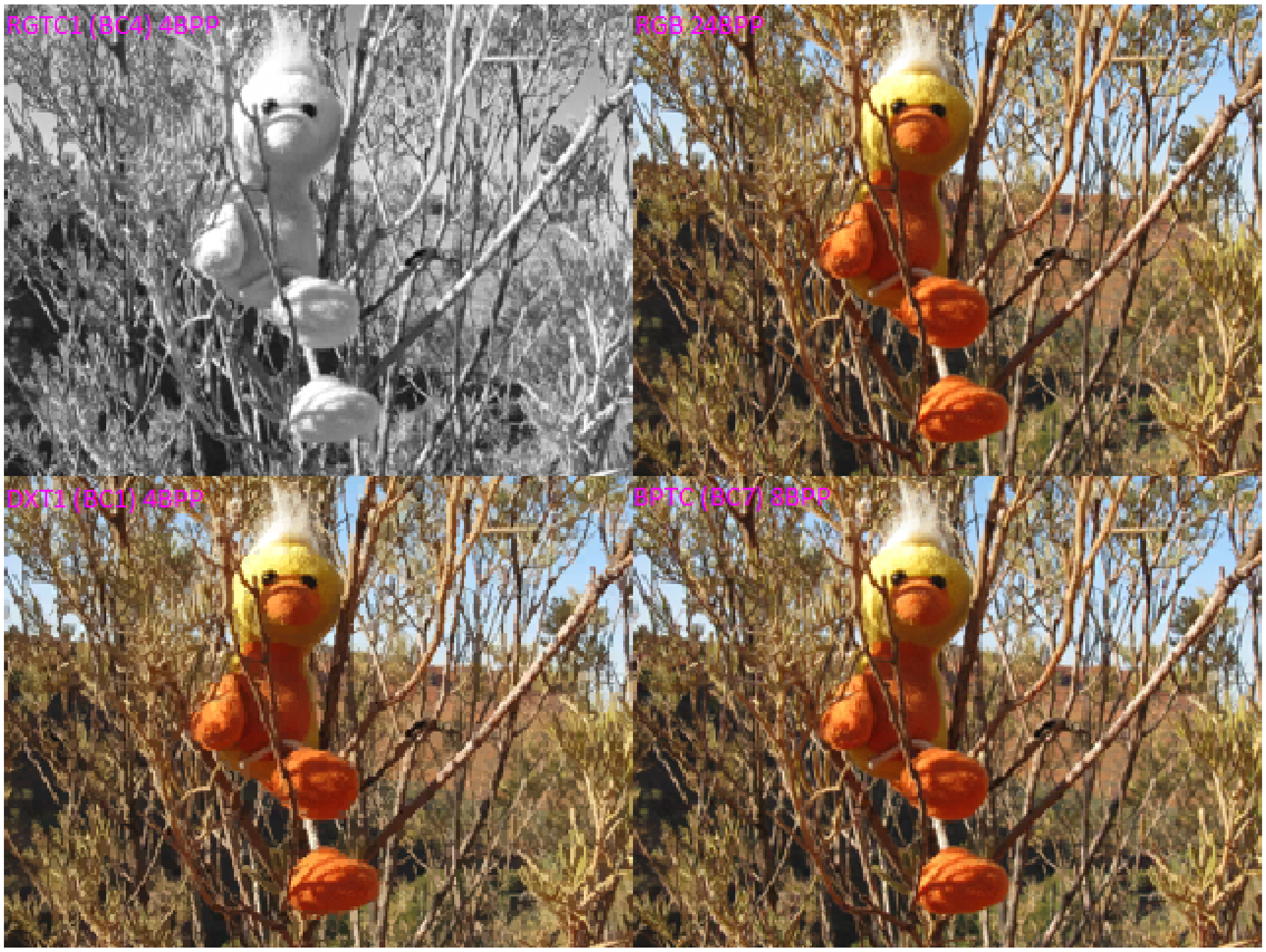
```
gli::texture2D Image = gli::load(TEXTURE_DIFFUSE_DXT1);

glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
glPixelStorei(GL_UNPACK_COMPRESSED_BLOCK_WIDTH, 4);
glPixelStorei(GL_UNPACK_COMPRESSED_BLOCK_HEIGHT, 4);
glPixelStorei(GL_UNPACK_COMPRESSED_BLOCK_DEPTH, 1);
glPixelStorei(GL_UNPACK_COMPRESSED_BLOCK_SIZE, 8);
glPixelStorei(GL_UNPACK_SKIP_PIXELS, GLsizei(Image[0].dimensions().x) / 4);
glPixelStorei(GL_UNPACK_SKIP_ROWS, GLsizei(Image[0].dimensions().y) / 4);
glPixelStorei(GL_UNPACK_ROW_LENGTH, Image[0].dimensions().x);

glCompressedTexImage2D(
      GL_TEXTURE_2D,
      GLint(0),
      GL_COMPRESSED_RGB_S3TC_DXT1_EXT,
      GLsizei(Image[0].dimensions().x) / 2,
      GLsizei(Image[0].dimensions().y) / 2,
      0,
      GLsizei(Image[0].capacity()),
      Image[0].data());
```

4.3. BPTC texture formats: GL_ARB_texture_compression_bptc

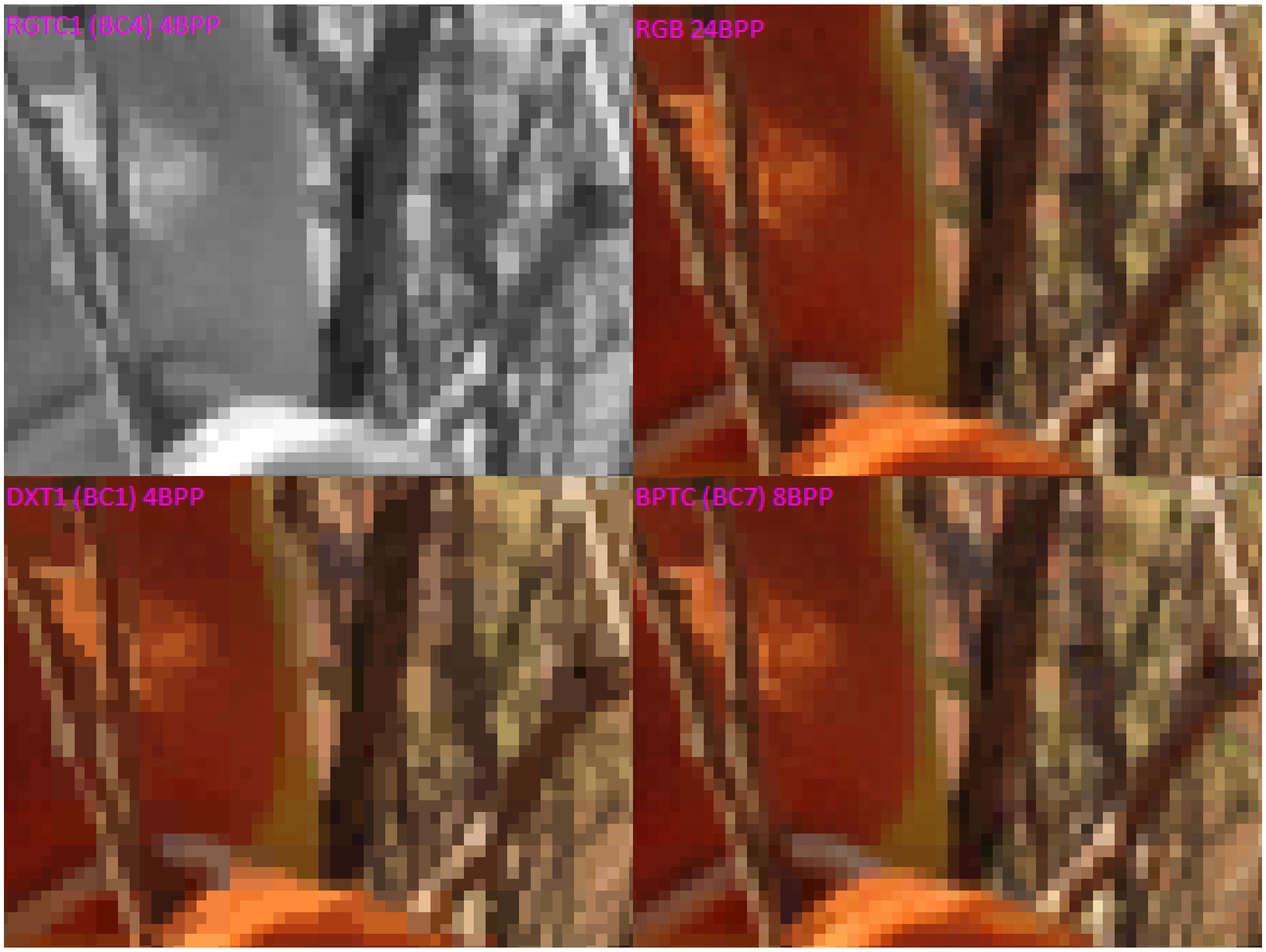
Released alongside with OpenGL 4.0, `ARB_texture_compression_bptc` extension finally reach OpenGL core specification. It provides Direct3D 11 compressed formats known as BC6H and BC7 and called respectively `GL_BPTC_FLOAT` and `GL_BPTC` with OpenGL. They aim high dynamic range, low dynamic range texture compression and high quality compression of sharp edges. The compression ratio for `GL_BPTC_FLOAT` and `GL_BPTC` are 6:1 and 3:1.



Four screenshots of the same image compressed in four different formats: RGTC1, RGB8, DXT1 and BPTC. Texture rendered with point sampling.

- BPTC is a 3:1 compression format, RGB, 8BPP
- DXT1 is a 6:1 compression format, RGB, 4BPP
- DXT5 is a 4:1 compression format, RGBA, 8BPP
- RGTC1 is a 2:1 compression format, 1 channel only, 4BPP

When I had a look at my first BPTC/BC7 texture, I had been really impressed by the visual quality. Like DXT5, BPTC is 8 bits per pixel format but it doesn't contain an alpha channel. Effectively, half of the bits of DXT5 are dedicated to the alpha channel. Hence, it's actually more fair to compare BPTC with DXT1 which is a RGB 4 bits per pixel format with an optional 1 alpha bit. To get a better quality, it's possible to use the alpha channel of DXT5 to store another channel which provides 4 bits per pixel for each channel. This format is usually called DXT5 RXBG and it provides a significantly higher quality alternative to DXT1 despite a feeling of red and blue noise. However for the same bit-rate, BC7 is by far a higher quality compression format.



Four screenshots of the same image compressed in four different formats: RGTC1, RGB8, DXT1 and BPTC. Texture rendered with point sampling and zoomed in

For the previous captures, I used the RGB mode of DXT1 and generated it with the amazing AMD The Compressonator (which could enjoy to be maintain!). I generated the RGTC1 texture with the same tool, taking the red channel only. The BPTC texture has been generated with the Direct3D 11 SDK encoder with the CPU version of this encoder. A compressed texture is as good as the compressor is good so maybe there are tools out there which would give better results but generally I find The Compressonator really good.

The result given by BPTC is absolutely stunning. This picture is actually challenging which explains the ugly result given by DXT1: It's blocky, smooth parts become stairs and noisy parts become fat pixels. BPTC remains very close to the uncompressed texture. BPTC even manage to provide more details than RGTC1 in many cases.

BPTC is a great texture format for visual quality so that for this property I expect to see it used instead of DXT1 and DXT5 in many cases in the future. However, the BPTC format has a main drawback: It's slow, very slow to generate! Chances are that all the real-time compression use cases of DXT5 will stick to this format for a while. I even think that this is such a limitation that it will prevent a wide adoption in real application for a while.

4.4. Format query: GL_ARB_internalformat_query

ARB_internalformat_query is an extension with a promising name which adds a new command with a promising name glGetInternalformativ but which functionalities are very limited in OpenGL 4.2. This extension adds a query mechanism that allows the user to determine which sample counts are available for a specific internal target and format in the context of multisampling.

5. Draw

5.1. Base instance: GL_ARB_base_instance

OpenGL 3.2 brought a very interesting feature on the regard of buffer management through the extension ARB_draw_elements_base_vertex which allows to run a draw call on a subset of the array buffers attached to the VAO for what could be understood as sparse rendering of large buffers, multiple rendering of different meshes with the same VAO.

Effectively, this extension provides a parameter call `<basevertex>` which is an offset from the beginning of the array buffer. This offset is really interesting but it implies that each draw call must use the same series of index read from the array element buffer bound to the VAO. A workaround to this behaviour is to bind different array element buffer for each draw call but we are losing a part of the purpose of glDrawElementInstancedBaseVertex and the interaction with instanced arrays is not so good either. Instanced arrays can be small, building larger instanced arrays by packing multiple of then could have benefits for performance. This wasn't possible until the release of OpenGL 4.2 and ARB_base_instance.

To remove these restrictions, the new command:
glDrawElementsInstancedBaseVertexBaseInstance (*ouf*!)

It adds a new parameter call `<baseinstance>` which adds an offset to the element index using the following equation:
```
<element> = floor(<gl_InstanceID> / <divisor>) + <baseinstance>
```

The `<baseinstance>` parameter has no effect if the value of the divisor is 0.

5.2. Transform feedback instanced: GL_ARB_transform_feedback_instanced

OpenGL 4.0 brought a lot of improvement for transform feedback including a transform feedback object, transform feedback streams, the capability to pause the transform feedback or to draw directly without querying the number of primitives recorded in the buffer.

However an interaction with an essential extension has been forgotten on the way: ARB_draw_instanced which is part of OpenGL 3.2 core specification. It implies that with OpenGL 4.1, it isn't possible to draw instances from transform feedback buffers without querying the number of primitives written in these buffers. A penalty because of the CPU - GPU synchronisation.

Fortunately, OpenGL 4.2 and ARB_transform_feedback_instanced fixed this issue by providing the following new commands:

- glDrawTransformFeedbackInstanced

- glDrawTransformFeedbackStreamInstanced

Including these new commands, I don't think it's enough features for transform feedback but it might be all OpenGL 4 hardware got.

6. Mics features

6.1. Conservative depth: GL_ARB_conservative_depth

AMD_conservative_depth has been promoted to ARB_conservative_depth and OpenGL 4.2 core specification. It enables some hardware optimisations according criteria defined by the OpenGL users.

For forward renderers, a typical practice is to start by rendering a fast depth buffer pass only and then render the colorbuffer so that only the visible fragments are processed by the fragment shader and write to the framebuffer, saving both compute and bandwidth.

This extension provides a level of programmability for at least some of the optimisations involved at this level using some layout qualifiers to specify how to expect the value of the gl_FragDepth.

```
// assume it may be modified in any way
layout (depth_any) out float gl_FragDepth;

// assume it may be modified such that its value will only increase
layout (depth_greater) out float gl_FragDepth;

// assume it may be modified such that its value will only decrease
layout (depth_less) out float gl_FragDepth;

// assume it will not be modified
layout (depth_unchanged) out float gl_FragDepth;
```

6.2. Map buffer alignment: GL_ARB_map_buffer_alignment

This is maybe a minor feature but it is also one of my favorite as performance often depends on how we feed the GPU with our data. This extension aims to make sure it's as fast as possible!

glMapBufferRange is one of the best OpenGL command ever made which allows an application to update buffer object data in a very advanced manner: using multiple CPU core and with some control of the synchronisation between the CPU and GPU. However, with OpenGL 4.1 the pointer returned by this command is unaligned which makes the update of buffer object data using SSE or AVX instructions more difficult and less efficient than it could be.

OpenGL 4.2 and ARB_map_buffer_alignment specify a minimum alignment requirement for the pointer returned by glMapBufferRange. It is giving by the value GL_MIN_MAP_BUFFER_ALIGNMENT passed to the command glGetIntegerv and the minimum value is 64 bytes for OpenGL 4.2 implementation. This is enough for SSE and AVX requirements (16 bytes and 32 bytes respectively) and leave opportunities for AMD or Intel to come up with a new instruction set registers able to store entire 4 by 4 single precision floating-point matrices.

Conclusions

Since the release of `EXT_direct_state_access`, a lot of desire has been formulate by the community for this extension to be improve and promoted. Since, no news about it which is big disappointment indeed. This extension can't be fully used with something else than an OpenGL 2.1 program due to some major interaction issues with later specifications. OpenGL programmers will be also disappointed that some elements of weakness like the GLSL binaries (just a binary cache in OpenGL 4.1) or the shader interface matching (I dare you to tell me you fully understand it) haven't been improved or fixed. Those with many more possible improvements give enough room for an OpenGL 4.3 release.

Beyond these elements, OpenGL 4.2 is a lot of goodness, a lot of work on details which proves that the OpenGL 4 API is reaching a certain level of maturity. The atomic counter and the shader load and store are bringing a lot of paths to explore including programmable vertex and voxel pulling, programmable blending and more generally it unleashes the true potential of OpenGL 4 hardware.

OpenGL 4 hardware done. Now... what about OpenGL 5 hardware? :D

Download: OpenGL 4.2 core specification
Download: GLSL 4.2 specification
Link: OpenGL 4.2 man pages
Link: My OpenGL 4.2+ wish-list
Link: OpenGL 4.1 review
Link: OpenGL 4.0 review
Link: OpenGL 3.3 review