Getting the Most Out of OpenGL® ES

Daniele Di Donato, Tom Olson, and Dave Shreiner
ARM

What This Talk is About

What's new in OpenGL® ES

- Introducing OpenGL ES 3.1!
- Fun with compute shaders and DrawIndirect

What's new in ASTC texture compression

- What ASTC is and why you should care
- Using ASTC: porting the Seemore demo
- Fun with ASTC 3D textures

ARM

OpenGL® ES 3.1

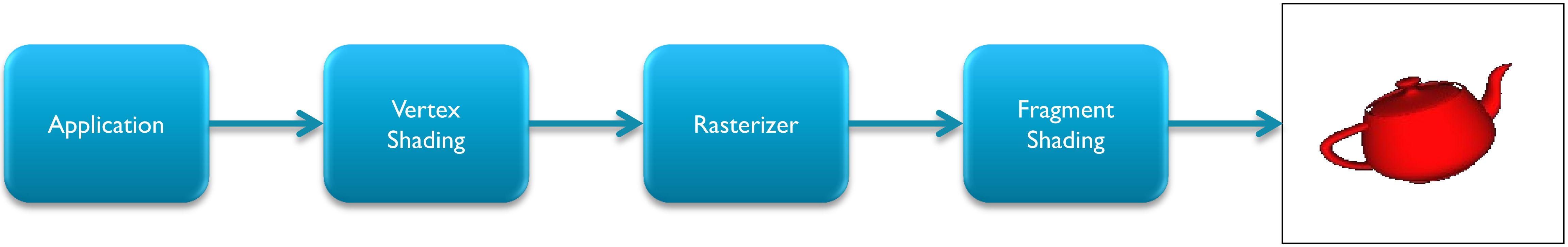
More, Better, Faster

Headline Features of OpenGL 3.1

- Backwards compatible with 2.0 & 3.0
- Compute shaders
  - Atomics
  - Shader load/store
- Separate shader objects
- Shader storage buffer objects
- Draw indirect rendering
- Enhanced texturing
  - Texture gather
  - Multi-sample textures
  - Stencil textures
- Shading Language Enhancements
  - Arrays of arrays
  - Explicit uniform location
  - Shader bitfield operations
  - Shader helper operations
  - Shader load/store operations
  - Layout bindings
- Vertex attribute binding

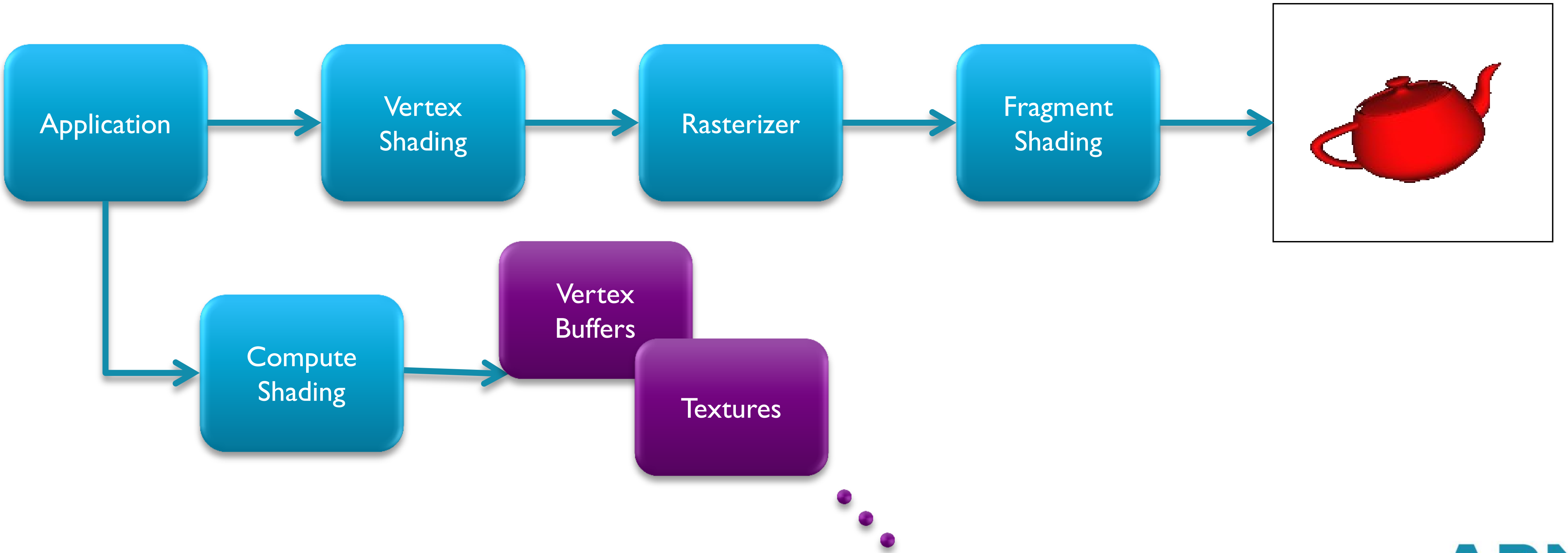
The Canonical Immediate-mode Rasterization Pipeline

OpenGL® ES 2.0 and 3.0 versions



The OpenGL® ES 3.1 Pipeline

OpenGL ES 2.0 version



Separate Shader Objects (SSOs)

More efficient use of shader resources

- In prior versions of OpenGL® ES, a *shader program* had to contain every shader stage
- This caused a lot of redundancy
  - For example, if you had two fragment shaders that used the same vertex shader, you would need two separate shader programs
- SSOs allow you to bind combinations of shaders to form a rendering pipeline
  - Shader interfaces still need to match
  - Each SSO must be marked *separable*

```
// … create & compile like normal, but
//  before linking the shader

glProgramParameteriv( program,
  GL_PROGRAM_SEPERABLE, GL_TRUE );
glLinkProgram( program );
```


ARM

One-Step Shader Object Creation

Sanity returns just in time …

- All this:

```
const GLuint shader = glCreateShader(type);

if (!shader) {
    return 0;
}

glShaderSource(shader, count, strings, NULL);
glCompileShader(shader);

const GLuint program = CreateProgram();

if (program) {
    GLint compiled;
    glGetShaderiv(shader, GL_COMPILE_STATUS, &compiled);
    glProgramParameteri(program, GL_PROGRAM_SEPARABLE, GL_TRUE);

    if (compiled) {
        glAttachShader(program, shader);
        glLinkProgram(program);
        glDetachShader(program, shader);
    }
}

glDeleteShader(shader);
return program;
```

- Becomes:

```
glCreateShaderProgramv(type,
    count, strings);
```

- Compiles, links and cleans up
- Marks the program *separable*
  - Suitable for use with SSOs
- Just add to a shader pipeline

Shader Pipelines

- Compose rendering pipeline from SSOs
  - Think `glUseProgram` but in pieces
  - `glUseProgram` overrides a pipeline

- Uses a new object: *program pipeline*
  - Same GL object semantics: gen, bind, …
    - `gl*ProgramPipeline`
  - Bind-to-edit & bind-to-use

```
// create SSOs:
//   vProgram – vertex shader
//   fProgram[] – fragment shaders

enum {Flat, Gourard, NumPipelines };
GLuint pipeline[NumPipelines];

glGenProgramPipelines( pipelines, NumPipelines );
glBindProgramPipeline( pipelines[Flat] );
glUseProgramStages( pipelines[Flat],
   GL_VERTEX_SHADER_BIT, vProgram);
glUseProgramStages( pipelines[Flat],
   GL_FRAGMENT_SHADER_BIT, fProgram[Flat] );
glBindProgramPipeline( pipelines[Gouraud] );
   …
glBindProgramPipeline( renderingMode );
//  render
```

Indirect Rendering

Storing rendering commands in buffers

```
glDrawArraysIndirect( mode, cmd );
```

where *cmd* is a pointer to a structure containing

```
struct {
  GLuint count;
  GLuint instanceCount;
  GLuint first;
  GLuint mustBezero;  // necessary for alignment
};
```

```
glDrawElementsIndirect(
   mode, type, cmd );
```

where *cmd* is a pointer to a structure containing

```
struct {
  GLuint count;
  GLuint instanceCount;
  GLuint first;
  GLuint base;
  GLuint mustBezero;  // necessary for alignment
};
```


ARM

Indirect Rendering (cont'd)

Subtle details

- Those command structures must be stored in buffer objects
  - `GL_DRAW_INDIRECT_BUFFER` object types, to be precise
- Command structures must be tightly-packed 32-bit unsigned integers (`GLuint`)
- Currently, no `glMultiDraw*Indirect` like in OpenGL®

Compute Shaders

And the great reasons to use them …

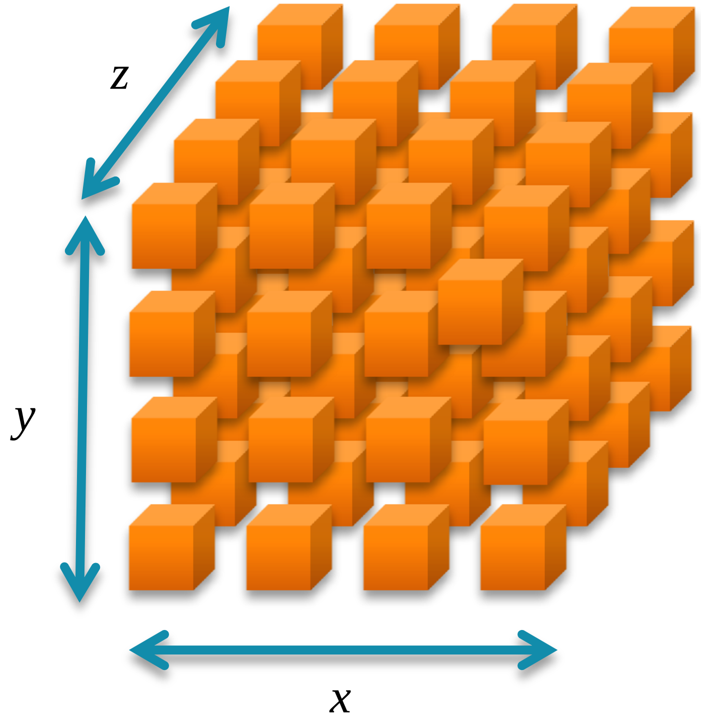
- Compute shaders perform generic computation
  - As compared to vertex and fragment shaders, which work on graphics primitives and pixels
- Integrated into OpenGL® ES
  - No need to include another API
  - Smaller app footprint
- Keeps data local to the GPU
  - Minimizes system bandwidth
  - Saves bandwidth (which conserves power)
- Combined with draw indirect, minimizes CPU activity
  - Reduces context switching and cache flushing
  - Helps conserve power

ARM

Compute Shader Basics

Thinking in parallel …

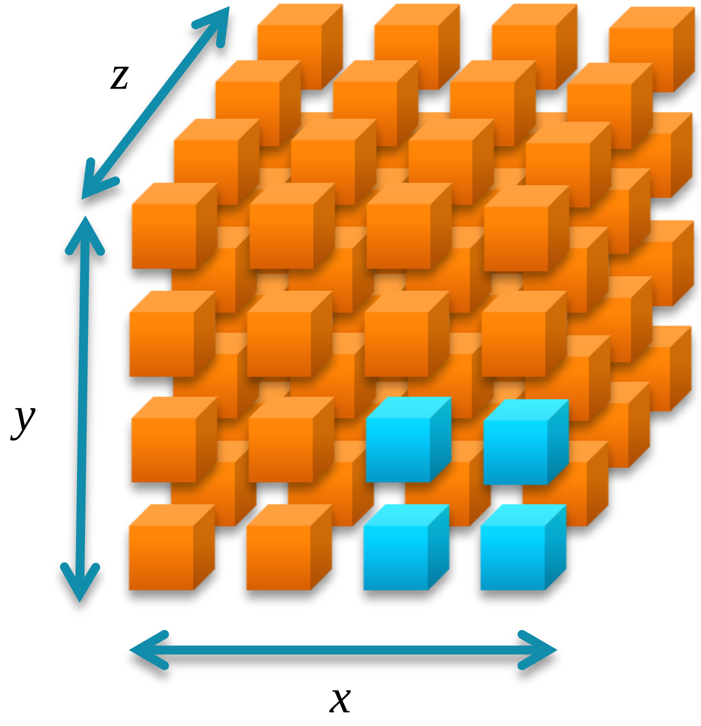
- Each compute thread processes on a *work item*
  - Each work item has a unique ID
- *Work items* are combined into *local work groups*
  - These are used mostly for scheduling
  - Compute jobs are launched in increments of local work group size
- Local work groups from *global work group*
  - Contains all the data for a single compute shader job



Compute Shader Basics (cont'd)

- Work is dispatched across a 3D domain
  - Schedule across lower domains by setting the local size to one

```
// create compute shader program
glUseProgram( compute );
glDispatchCompute( 2, 2, 4 );
glMemoryBarrier( GL_SHADER_STROAGE_BARRIER_BIT );
```



Global work size:	(4, 4, 4)
Local work group size:	(2, 2, 1)
Job dispatch size:	(2, 2, 4)

ARM

Compute Shaders

Data and computation

- Yet another GLSL ES shader ☺
  - Nothing surprising here
- Slight programming differences
  - No “rendering” data access – attributes and varyings
- Data transactions through either
  - Shader Storage Objects
  - Images (textures)
- Local work group size declared using `layout` qualifier in shader source

```
layout ( local_x_size = 2, local_y_size = 2,
         local_z_size = 1 ) in;

#define NumElems <n>

layout ( shared, binding = 1 ) buffer Data {
    vec4 data[NumElems][NumElems];
};
layout ( rgba32f ) uniform image2D image;

void main()
{
   uvec idx = gl_GlobalInvocationID.xy;

   vec4 color = c(idx);
   imageStore( image, idx, color );
   data[idx.x][idx.y] = f(idx);
}
```

Shader Storage Objects

Just another OpenGL® buffer type

- Another buffer to run through the gen-bind-bind-(maybe)delete cycle
- Shader storage is only available for compute shaders
  - Bind to update for compute stage
  - Rebind to *another buffer type* to use in the pipeline
- Otherwise, think more like C++ than graphics
  - read-write and random access

```
// Setup
glGenBuffers( … );
glBindBufferBase( GL_SHADER_STORAGE_BUFFER,
    index, bufferId );
glBufferData( GL_SHADER_STORAGE_BUFFER,
    sizeof(data), data, GL_DYNAMIC_READ );

// Update
glUseProgram( compute );
glDispatchCompute( … );
glMemoryBarrier( GL_SHADER_STROAGE_BARRIER_BIT );

// Use
glBindBufferBase( GL_VERTEX_ARRAY_BUFFER,
    index, bufferId );
glUseProgram( render );
glDrawArrays( … );
```


ARM

Images and Compute

Another way to update textures

- Compute shaders mandate image load/store operations
  - These have been optional in other shader stages
- Allow random read/write access to a texture bound as an *image sampler*
  - Use `image*D` as shader sampler type
- Layer parameters control if an single image, or an entire level is made accessible
  - Think texture array or 3D textures

```
// Setup
glGenTextures( … );
glBindTexture( GL_TEXTURE_2D, texId );
glTextureStorage2D( GL_TEXTURE_2D, levels,
   format, width, height );
glBindImageTexture( unit, texId, layered,
   layer, GL_READ_WRITE, GL_RGBA32F );

// Update
glUseProgram( compute );
glDispatchCompute( … );
glMemoryBarrier( GL_SHADER_STROAGE_BARRIER_BIT );

// Use
glUseProgram( render );
glDrawArrays( … );
```

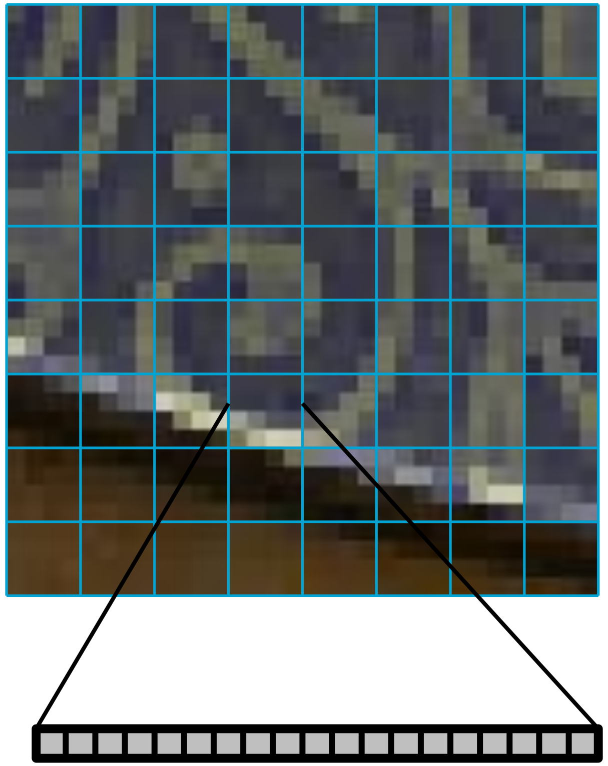
Compute Shaders and Atomics

Synchronization

- Compute threads run asynchronously
- Data is shared between threads
  - May need to wait for threads to update shared data before using it in other threads
- Shader atomic operations supported across all shader stages
  - Useful for counting and other data recording

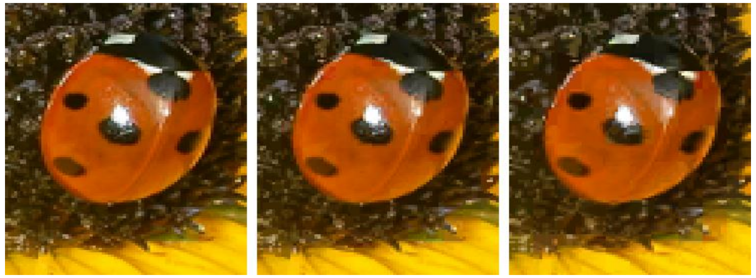
- GLSL ES memory barrier functions
  - `memoryBarrier*`
  - `groupMemoryBarrier` (compute only)
- GLSL ES atomic operations
  - `atomicCounter*`
  - `atomicAdd`
  - `atomic{Min,Max}`
  - `atomic{And,Or,Xor}`
  - `atomicExchange`
  - `atomicCompSwap`

What is ASTC?



Adaptive Scalable Texture Compression

- YABBCTF*
- Developed by ARM for an industry competition
- "The last compressed texture format you'll ever need"

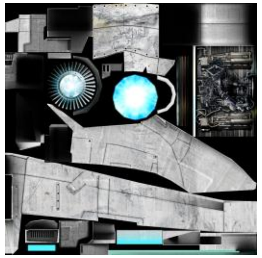


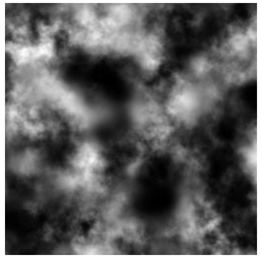
ASTC Compression
8bpp 3.56bpp 2bpp

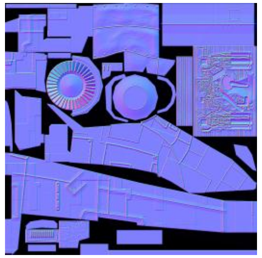
*Yet another block-based compressed texture format

Why Was It Needed?

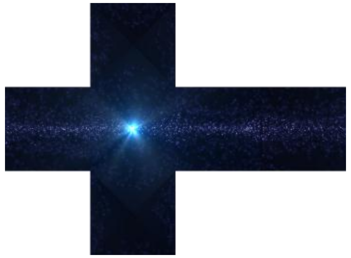
- Textures are used for many different things:

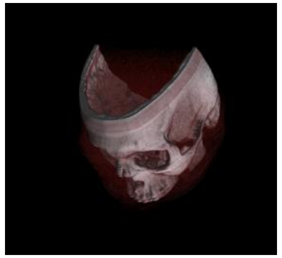

Reflectance


Gloss, Height, etc


Normals


Illuminance


Lighting environment

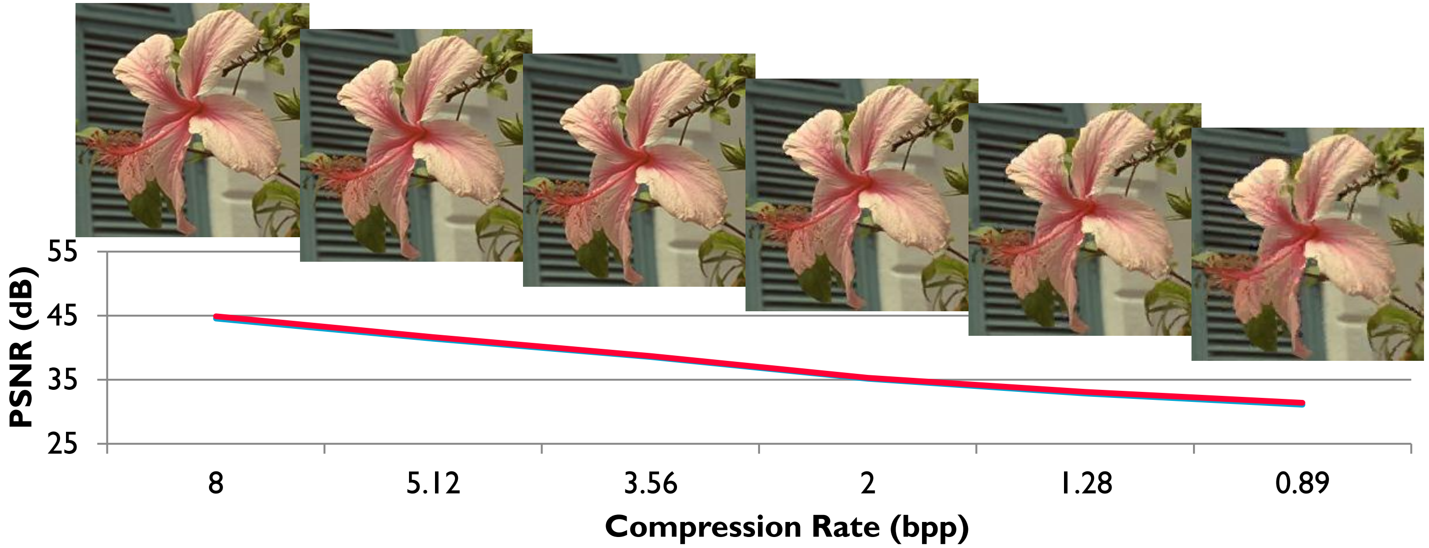

3D Properties

- Each use has its own requirements
  - Number of color components
  - Dynamic range (LDR vs HDR)
  - Dimensionality (2D vs 3D)
  - Quality (≈ bit rate)

No existing format addressed all of these use cases

Adaptive Scalable Texture Compression

- Goals
  - Cover all the key use cases
  - Provide excellent quality



- Key properties
  - Scalable bit rate: 8bpp down to <1bpp in fine steps
  - Any number of components at any bit rate
  - Both LDR and HDR pixel formats
  - Both 2D and 3D textures
  - Significant quality improvement over existing formats

ARM

How It Works

- Global image properties
  - Dimensionality
  - Bit rate
  - sRGB-ness

2D Bit Rates				3D Bit Rates			
4x4	8.00 bpp	10x5	2.56 bpp	3x3x3	4.74 bpp	5x5x4	1.28 bpp
5x4	6.40 bpp	10x6	2.13 bpp	4x3x3	3.56 bpp	5x5x5	1.02 bpp
5x5	5.12 bpp	8x8	2.00 bpp	4x4x3	2.67 bpp	6x5x5	0.85 bpp
6x5	4.27 bpp	10x8	1.60 bpp	4x4x4	2.00 bpp	6x6x5	0.71 bpp
6x6	3.56 bpp	10x10	1.28 bpp	5x4x4	1.60 bpp	6x6x6	0.59 bpp
8x5	3.20 bpp	12x10	1.07 bpp				
8x6	2.67 bpp	12x12	0.89 bpp				

- Per-block (partition) properties
  - Number of color channels
  - Dynamic range

# color channels	Sampler return value
one	(L, L, L, 1.0)
two	(L, L, L, A)
three	(R, G, B, 1.0)
four	(R, G, B, A)

ASTC in Standards

- Khronos ASTC 2D-LDR extension
  - KHR_texture_compression_astc_ldr
  - Released at SIGGRAPH 2012
- Now available with HDR…
  - KHR_texture_compression_astc_hdr
- …and 3D!
  - OES_texture_compression_astc

Full functionality of ASTC is now available as ratified Khronos standards

What's New – ASTC in Products

- ARM® Mali™ GPUs
  - Full profile supported in all Midgard family GPUs starting with Mali-T624
  - Mali-T624, T628, T760, T720

- Support coming from many other GPU vendors
  - Imagination Technologies: PowerVR™ Series6XT GPU IP
  - NVIDIA: Tegra® K1GPU
  - Qualcomm: Snapdragon™ 805 processor / Adreno™ 420 GPU

ASTC is going to be everywhere, very soon!

Resources

- Evaluation codec (source)
  - http://malideveloper.arm.com/develop-for-mali/tools/astc-evaluation-codec/
- Tools
  - ARM® Mali™ Texture Compression Tool
  - Mali OpenGL® ES 3.0 Emulator
- Practical advice for the developer
  - Whitepapers and blogs by Stacy Smith (ARM) on ASTC
- How and why it works
  - Nystad et al, *Adaptive Scalable Texture Compression*, Proc. HPG 2012

ASTC In Action: The SeeMore Files

Porting SeeMore to ASTC

Goal

- Apply ASTC to the SeeMore demo
- Provide a visual comparison of compression formats
- See https://www.youtube.com/watch?v=jEv-UvNYRpk

Process

- Define a way to compare the image
  - Split Screen: Diff-map, PSNR
- Find the best compression rate / settings
- Showcase advantages of ASTC compared to other formats

Conversion Process

- Easy if your engine/tools already use a block compressed format

- ASTC 5x5 block size gives compared to ETC2+EAC (RGBA):
  - Same quality
  - ~24% smaller texture memory footprint
  - ~11% less memory read bandwidth per sec
  - ~10% less energy consumption per frame

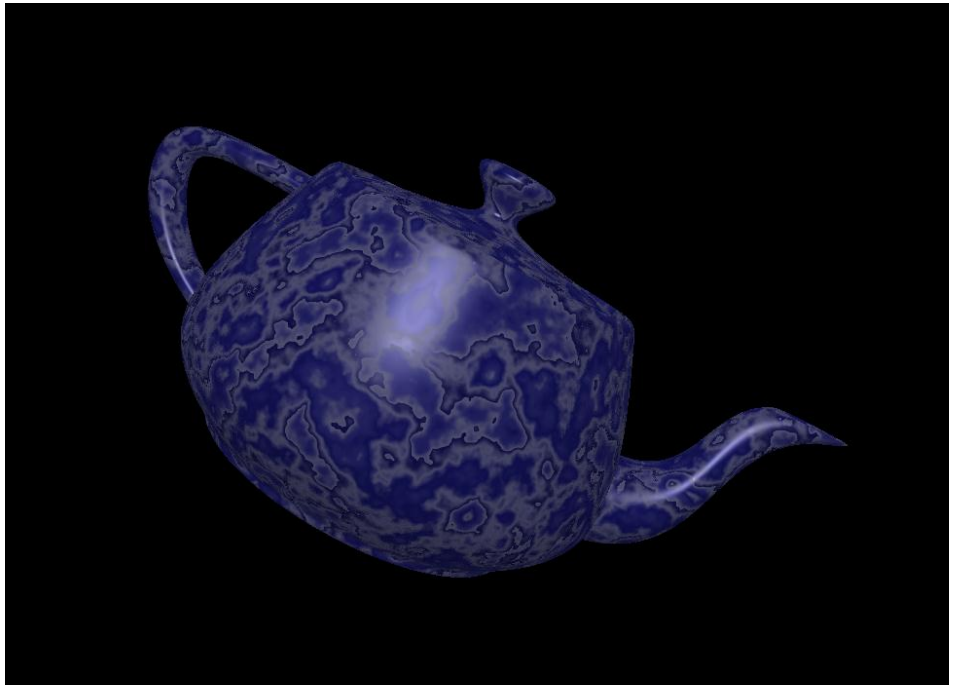
- Improved Normal Map
  - Remember to swizzle the green and alpha channel in the shader!!!!

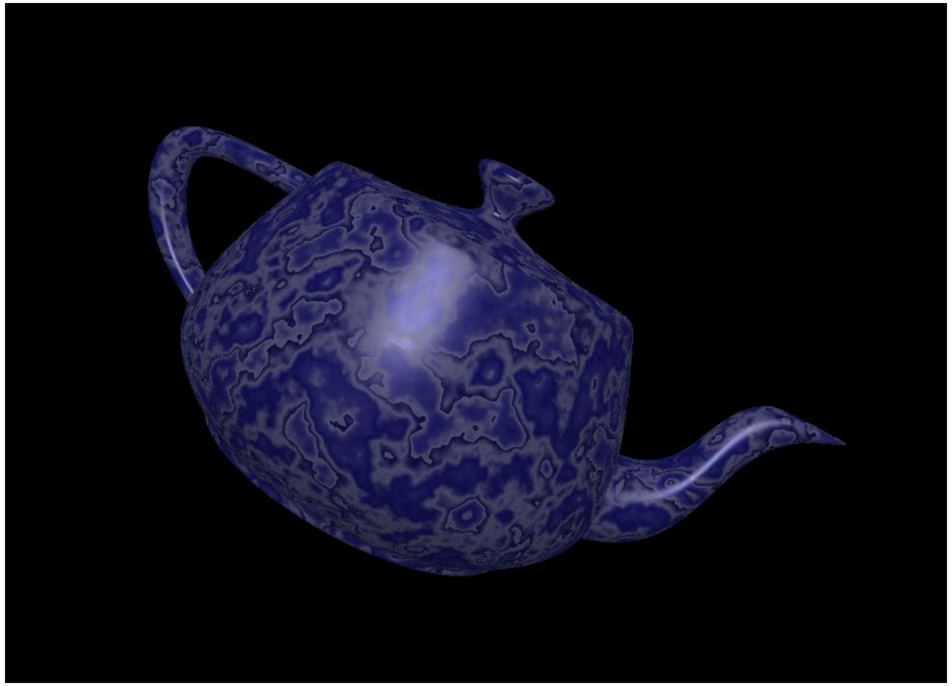
Fun with ASTC 3D

3D Textures

- They're cool, right?
  - How would I know? I can't afford to use them.
  - 128x128x128 RGB texture is (probably) 4MB – pretty big for mobile
- ASTC to the rescue!
  - 128x128x128 at 0.59 bpp is ~150KB!
- *Low bit-rate 3D compression changes the game*
- *What can you do if 3D textures are cheap?*

Procedural Texture Demo


Using original 128x128x128 texture (2MB)


Using ASTC 3D texture (150KB)

Procedural texture

- Points on object surface map to a 3D noise texture
- Noise value used to sample a color gradient

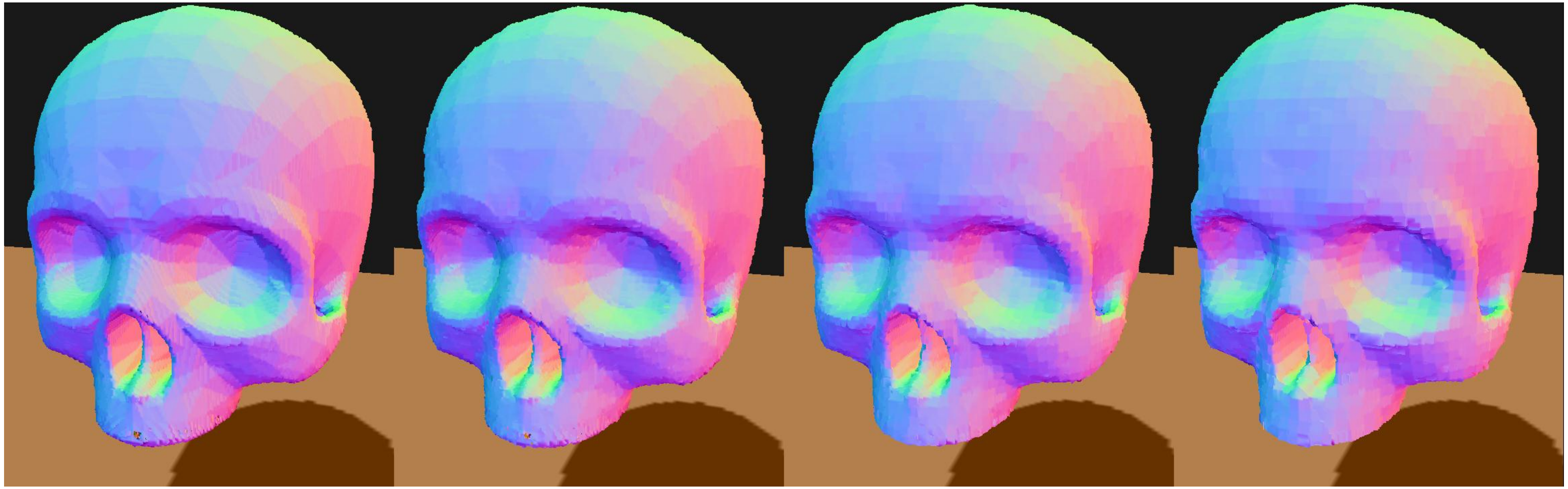
Procedural Texture Demo

<Cut to RenderMonkey and show compression artifacts>

Particle System Demo

Goal

- Use 3D HDR texture compression in an innovative way
- Add some cool OpenGL® ES 3.1 features
- Show the effect of various compression rates



Particle System Demo

- Up to ~90% less memory using the lowest compression rate!!
- Particles in the vertex shader look up collision data stored in a 3D texture.
- Transform feedback allows for physics simulation entirely on the GPU…and much more
- Instancing - because nobody wants replicated static geometry.

ARM

Summary

1. OpenGL® ES 3.1 is here!
   - Learn to use compute shaders
   - Think about what you want to do with them
2. ASTC texture compression will be everywhere soon
   - You ***can*** use it as a plug-in substitute for DXT*n* / PVRTC / ETC*…
   - … but why stop there?
   - Consider what you can do with cheap, small HDR and volume textures
3. Have fun!

Questions?

For more or OpenGL® ES 3.1…

Come to the Khronos OpenGL ES DevU

- Moscone Center, West Mezzanine (access from South Lobby, above rooms ABC)
- Meeting room #262

Thank You