What’s New in OpenGL ES

Tom Olson
Director of Graphics Research, ARM
Chair, OpenGL ES Working Group
Introducing OpenGL ES 3.1

• Designed for rapid adoption
  - Should run on most OpenGL ES 3.0 hardware
  - Backward compatible with ES 2.0/3.0 for easy software porting

• Brings the most requested features of OpenGL 4.x to mobile
  - Significant, even game-changing upgrade in functionality

• Continued improvement in portability - tighter specs, less undefined behavior
Key Working Group Participants

- GPU Designers
- SoC Vendors
- Platform Owners
- End Equipment Makers
- Middleware ISVs
- Tool and Game Engine Developers
Outline

• Introduction and Goals
  - Tom Olson, ARM / ES WG Chair

• OpenGL ES 3.1 Compute Features
  - Daniel Koch, NVIDIA

• OpenGL ES 3.1 API Features
  - Slawek Grajewski, Intel

• GLSL ES 3.1 Shading Language
  - Bill Licea-Kane, Qualcomm

• EGL 1.5 Features
  - Jon Leech, Khronos / EGL 1.5 and OpenGL ES 3.1 Specification Editor

• Wrap-up / Questions / Demos
OpenGL ES 3.1 Compute Features

Daniel Koch, NVIDIA
dkoch@nvidia.com
Compute Shader Motivation

• A new way of accessing the computation power in the GPU
  - Execute general purpose algorithms
  - New single-stage pipeline, separate from the graphics pipeline
• Uses concepts familiar to graphics programs
  - Written in same GLSL ES language as other shaders
• Standard part of all OpenGL ES 3.1 implementations
  - Similar to OpenGL 4.3 and DirectX 11 functionality
Compute Shaders

- New shader type: GL_COMPUTE_SHADER
- Behaves as other shader types you know and love (or hate...)
  - glShaderSource, glCompileShader, glAttachShader, and glLinkProgram
  - Or glCreateShaderProgram()
- Operates on the same resources as the graphics pipeline
  - buffers, textures
- No predefined inputs or outputs; only side effects are on memory
- New types of memory-backed resources
  - images, buffers, atomic counters (more about these later)
- Conceptually a grid of work items
  - Communicate with each other via shared memory
- To launch work: glDispatchCompute*()
in uvec3 gl_NumWorkGroups;   // Number of workgroups dispatched
const uvec3 gl_WorkGroupSize; // Size of each work group for current shader
in uvec3 gl_WorkGroupID;     // Index of current work group being executed
in uvec3 gl_LocalInvocationID; // index of current invocation in a work group
in uvec3 gl_GlobalInvocationID; // Unique ID across all work groups and invocations
Use void barrier() to synchronize invocations in a work group

Use memory barriers to order reads/writes accessible to other invocations

void memoryBarrier();
void memoryBarrierAtomicCounter();
void memoryBarrierBuffer();
void memoryBarrierImage();
void memoryBarrierShared(); // Only for compute shaders
void groupMemoryBarrier(); // Only for compute shaders
Compute Shader Example

```glsl
#version 310 es
layout (local_size_x = 4, local_size_y = 2) in; // z defaults to 1
float varA;    // per invocation variable
// per workgroup shared memory [4][2]
shared float shmem[gl_WorkGroupSize.x][gl_WorkGroupSize.y];
layout(std430, binding = 0) buffer b {
  float result;
};
void main(void) {
  varA = ....;
  shmem[gl_LocalInvocationID.x][gl_LocalInvocationID.y] = varA;
  barrier();
  b.result = ....; // calculation on any/all locations in shmem
}
```
Shader Storage Buffer Objects (SSBO)

- Read/write and atomic operations on the variables stored in a buffer object
  - Essentially writeable UBOs
- Support for very large buffers
  - Required minimum is 128 MB
- New buffer binding point `SHADER_STORAGE_BUFFER`
  - Limited number available per shader type: `MAX_<STAGE>_STORAGE_BLOCKS`
  - Required for compute (min 4), optional in vertex and fragment
- New `std430` memory layout
  - More efficient packing of vector and scalar arrays, structures
  - `std140` is also supported
- Can use C-style code in a shader to read and write the buffer
- Ideal for getting data in/out of a compute shader
SSBO Example

```cpp
struct MyVertex {
    vec2 tex[2]; // tightly packed array in std430
    vec3 pos;
    int materialIdx;
}
layout(std430, binding = 2) buffer b {
    MyVertex vertices[ ]; // unsized array allowed at end of buffer
};

... // compute data to store in Vertices[]
b.vertices[i].materialIdx = idx; // directly write to buffer content
```
Shader Image Load Store

- Read/Write and atomic operations on the data in an image
  - Essentially writeable textures

- Adds *image units* where a single texture level is bound, similar to texture units

- New GLSL *image* uniform data types, similar to samplers
  - Limited number available per shader type: `MAX_<STAGE>_IMAGE_UNIFORMS`
  - Required for compute (min 4), optional for vertex and fragment

- Built-in `imageLoad()`, `imageStore()`, `imageSize()` functions
  - Integer coordinates used to identify texels accessed

- Optional ability to perform atomic read-modify-write operations
  - `imageAtomic*()` functions on subset of formats: `r32i`, `r32ui`, `r32f`
  - OES_shader_image_atomic

- Ability to explicitly enable “early” per-fragment tests
  - Depth and stencil operations can occur before fragment shader execution
  - Must be opted in when fragment shaders can have side effects
Image Load Store Example

```glsl
layout(rgba32f, binding = 3) readonly highp uniform image2D myImage;
ivec2 size = imageSize(myImage); // get image dimensions
int i, j;
for (i = 0; i < size.x; i++) {
    for (j = 0; j < size.y; j++) {
        vec4 color = imageLoad(myImage, ivec2(i, j));
        // do something with color
    }
}
```
Shader Atomic Counters

- Provides a set of buffer-backed atomic counters
  - The contents of the atomic counters are stored in the buffer object
- Can be set and queried with normal buffer access functions
  - `glBufferSubData()`, `glMapBufferRange()`, etc
- Enables a shader to read or write from unique offsets
  - Append and consume buffers
- New buffer binding point `ATOMIC_COUNTER_BUFFER`
  - Limited number per shader type: `MAX_<SHADER>_ATOMIC_COUNTER_BUFFERS`
  - Required for compute (8), optional for vertex and fragment
- New GLSL `atomic_counter` opaque uniform data type
  - Limited number available per shader type: `MAX_<STAGE>_ATOMIC_COUNTERS`
  - Required for compute (8), optional for vertex and fragment
- Built-in GLSL functions to atomically query/increment/decrement the counters
  - `atomicCounter()`, `atomicCounterIncrement()`, `atomicCounterDecrement()`
Shader Memory Access

• Updates to buffer or texture data
  - API commands: the GL implementation takes care of synchronization
  - Written by shaders: the **app is responsible for synchronization**

• Order of shader memory accesses is largely undefined
  - Order (and even number!) of shader types and invocations may vary

• **Use MemoryBarrier() API**
  - Memory operations issued by rendering command **before** the MB
  - Are ordered relative to commands coming **after** the MB

• **barriers parameter**: specify how the data will be used **after** the MB

• Eg. DispatchCompute - launch compute shader which updates a buffer
  - **glMemoryBarrier(GL_BUFFER_UPDATE_BARRIER_BIT)**
    - buffer can be read/updated using glMapBufferRange(), etc.
  - **glMemoryBarrier(GL_VERTEX_ATTRIB_ARRAY_BARRIER_BIT)**
    - Buffer can be used as vertex data source for subsequent Draw* command
Shader Memory Control Functions

- Multiple invocations can simultaneously access the same memory
- Memory access qualifiers (for image and buffer variables)
  - coherent, volatile, restrict, readonly and writeonly
- Atomic memory functions (for buffer and shared variables)
  - Atomic read/modify/write operations that return the original value
  - Atomic{Add, Min, Max, And, Or, Xor, Exchange, CompSwap}
- Order accesses to selected types of resources shared between invocations
  - memoryBarrier{AtomicCounter, Buffer, Image, Shared}
- Order access to all types of memory resources between invocations
  - memoryBarrier() // for all invocations
  - groupMemoryBarrier() // for all invocations in the same work group
- Used with coherent variables
Differences from GL

• Coherence guarantees altered (no inter-stage coherence within a draw call)
• Lower minimum maxima (4 vs 8) [buffers, images]
• Support is optional in fragment shaders [buffers, images, atomic counters]
• Compile-time constant expressions to index into arrays of buffers, images
  - GL allows dynamically uniform expressions
• Binding points can only be set in the shader
  - no glShaderStorageBlockBinding API to change bind point after linking [buffers]
  - glUniform1i() cannot be used to change location after linking [images]
• Compute Shaders
  - Added precision qualifiers, same default precisions as vertex shaders in ES 3.0
    - Precision of new types and resources must be specified
  - Reduced workgroup size (x, y) dimensions and invocations (128 vs 1024)
• Minimum SSBO size is 128MB instead of 16MB
Differences from GL

• Images
  - Only supported for immutable textures (use TexStorage* API!)
  - Format **must** be specified in shader for all image declarations
  - Image variables must be **readonly** or **writeonly** except r32f, r32i, and r32ui
  - Image formats supported have been subset (check your formats!)
    - Different default format as well (R32UI vs R8)
  - Support for **imageAtomic** functions is optional (OES_shader_image_atomic)
  - No support for multisample images

• Didn’t add queries that would be duplicated by program interface query
  - GetActiveAtomicCounterBufferiv
  - UNIFORM_BLOCK_REFERENCED_BY_COMPUTE_SHADER
  - ATOMIC_COUNTER_BUFFER_REFERENCED_BY_COMPUTE_SHADER
OpenGL ES3.1 API Features

Slawomir (Slawek) Grajewski, Intel
slawomir.grajewski@intel.com
Agenda

New ES3.1 features

- Framebuffer no attachments
- New draw indirect commands
- Separate Shader Objects (aka SSO)
- Program Interface Query
- Vertex Attrib Binding
- Texture gather
- Texture multisample
- Stencil texturing
- OES_texture_stencil8
Framebuffer No Attachments

- Shaders can operate on
  - Images
  - Shader storage buffer objects
  - Atomic counter buffers

- Sometimes framebuffer attachments not needed at all

- Specify default framebuffer parameters and don’t waste memory
  - FRAMEBUFFER_DEFAULT_WIDTH
  - FRAMEBUFFER_DEFAULT_HEIGHT
  - FRAMEBUFFER_DEFAULT_SAMPLES
  - FRAMEBUFFER_DEFAULT_FIXED_SAMPLE_LOCATIONS

[framebuffer diagram]
New Draw Indirect Commands

- ES3.1 Introduces two new DrawIndirect commands
  - glDrawArraysIndirect()
  - glDrawElementsIndirect()

- And one for dispatching indirectly Compute Shaders
  - glDispatchComputeIndirect()

- Useful for draw parameters established by GPU
  - Compute Shader for example
  - Eliminate GPU/CPU round trip

- Important differences from GL
  - Cannot be used with transform feedback active
  - Cannot be used with default VAO
  - Parameters cannot be fetched from client memory
glDrawArraysIndirect() glDrawArraysIndirect(mode, indirect);

struct {
    uint    count;
    uint    instanceCount;
    uint    first;
    uint    mustBeZero;
} DrawArraysIndirectCommand;

glBindBuffer(DRAW_INDIRECT_BUFFER, ...);
The `glDrawElementsIndirect()` function is used to draw elements from a buffer using indirect parameters.

```
glBindBuffer(DRAW_INDIRECT_BUFFER, ...);
glDrawElementsIndirect(mode, type, indirect);
```

The function takes a `mode` parameter to specify the drawing operation, a `type` parameter to specify the data type, and an `indirect` parameter which contains the indirect parameters.

The indirect parameters are stored in a structure that includes:

- `count`: The number of elements in the buffer.
- `instanceCount`: The number of geometry instances.
- `firstIndex`: The starting element in the element array.
- `baseVertex`: The base vertex for the buffer.
- `mustBeZero`: A boolean indicating whether the `firstIndex` must be zero.

```
struct {
    uint    count;
    uint    instanceCount;
    uint    firstIndex;
    uint    baseVertex;
    uint    mustBeZero;
} DrawElementsIndirectCommand;
```
glDispatchComputeIndirect()(

```c
uint num_groups_x;
uint num_groups_y;
uint num_groups_z;
```

glBindBuffer(DISPATCH_INDIRECT_BUFFER, ...);

glDispatchComputeIndirect (indirect);

Dispatch dimensions

Offset
Separate Shader Objects

- Separate Shaders
  - Eliminate (expensive) program linking operations (glLinkProgram())
    - Needed for every VS/FS pair; in extreme case $N \times M$
  - Replaces with mix-and-match approach
    - Compiled shaders can be mixed-and-matched at "zero" cost.
Monolithic Programs

For every Vertex Shader

- `glCreateShader()`
- `glShaderSource()`
- `glCompileShader()`

For every Fragment Shader

- `glCreateShader()`
- `glShaderSource()`
- `glCompileShader()`

For every VS/FS pair (program)

- `glCreateProgram()`
- `glAttachShader()`
- `glLinkProgram()`

Can be expensive
Pipeline Objects

• "Replacement" for monolithic programs

• A container for collection of separate shaders

• Where to get separate shaders from?
  - Stand-alone shader programs containing single shader

  - Monolithic programs linked with SEPARABLE_PROGRAM parameter

• Pipeline objects can be used instead of programs

Mix-and-match

glGenProgramPipelines()

glUseProgramStages()

glUseProgramStages()

glCreateShaderProgram()

.glProgramParameter(...,
  PROGRAM_SEPARABLE);

.glLinkProgram();

.glUseProgram(0);

.glBindProgramPipeline();
Separate Shaders

For every Vertex Shader

1. glCreateShaderProgram()
2. glGenProgramPipelines()
   glUseProgramStages(pipeline, VERTEX_SHADER_BIT, program)
3. Lite operations
   // for bound pipeline object
   glUseProgramStages(); // VS
   glUseProgramStages(); // FS

For every Fragment Shader

1. glCreateShaderProgram()
2. glGenProgramPipelines()
   glUseProgramStages(pipeline, FRAGMENT_SHADER_BIT, program)
3. Lite operations
   // for bound pipeline object
   glUseProgramStages(); // FS

For every VS/FS pair (pipeline)

1. glCreateShaderProgram()
2. glUseProgramStages(pipeline, VERTEX_SHADER_BIT, program)
3. glUseProgramStages(pipeline, FRAGMENT_SHADER_BIT, program)
4. Lite operations
   // for bound pipeline object
   glUseProgramStages(); // VS
   glUseProgramStages(); // FS

Or at draw time
Putting This All Together

- **Monolithic program**
  - VS<sub>1</sub>
  - FS<sub>1</sub>
  - uniforms
  - separable

- **Monolithic program**
  - VS<sub>2</sub>
  - FS<sub>2</sub>
  - uniforms
  - separable

- **Stand-alone program**
  - VS<sub>3</sub>
  - FS<sub>3</sub>
  - uniforms
  - separable

- **Actual GPU configuration**

- **Monolithic program**
  - `glUseProgram()`
  - `glUseProgramStages()`

- **Stand-alone program**
  - `glUseProgramStages()`
  - `glBindProgramPipeline()`

- **Effective after**
  - `glUseProgram(0);`
Uniforms

- Constant data per program

- If VS and FS come from pipeline object each come from different program

  - glActiveShaderProgram() selects where glUniform*() calls are directed
    - Redirects only if pipeline object defines VS/FS

- To make things easier DSA versions of glUniform*() entry points are introduced
  - glProgramUniform*() (program, location, value);
Matching Rules

- Each vertex shader output has matching fragment shader input and vice versa
  - No dangling outputs/inputs

- Input matches output if:
  - Both are declared with the same location qualifiers OR with the same name
  - Have the same type
  - Variables defined as structures have to match
  - Variables defined as arrays have to match
  - Precision has to match!

- Some things can differ
  - Obvious: storage qualifier (in/out)
  - Interpolation qualification (flat/smooth)
  - Auxiliary qualification (centroid)
  - Qualifier *sample*

- Draw time error in case of mismatch
Pros/Cons

• Separate shaders eliminate expensive linking of vertex and fragment shaders

• Vertex and fragment shaders can be mixed-and-matched at "zero" cost

• "Legacy" GLSL monolithic approach allows for more advanced optimizations
  - The extend of such optimizations is IHV dependent
Program Interface Query

• Why do we need Program Interface Query?
  - New queries required for new features
    - Shader storage buffer objects
    - Atomic counter buffers

• ES3.0 has separate set of query commands for different program interfaces and resources
  - Some existing ES queries have limited capabilities
    - Fragment shader outputs cannot be queried

• Design decision for ES3.1
  - New program interface query that
    - Addresses the new functionality
    - Covers missing gaps
    - Allows for easy expansion
Program Interface Query

- All program interfaces can be queried
- Only ACTIVE resources within these interfaces are reported
  - That have observable effect
- What are resources?
  - Variables
  - Interface blocks
  - Atomic counter binding points
Program Interface Query API

- UNIFORM
- UNIFORM_BLOCK
- PROGRAM_INPUT
- PROGRAM_OUTPUT
- BUFFER_VARIABLE
- SHADER_STORAGE_BLOCK
- ATOMIC_COUNTER_BUFFER
- TRANSFORM_FEEDBACK_VARYING

`glGetProgramResourceiv(program, programInterface, index, propCount, *props, bufSize, *length, *params);`

Index of a resource in the interface
# Program Interface Query API

<table>
<thead>
<tr>
<th>Property</th>
<th>Supported Interfaces</th>
<th>Property</th>
<th>Supported Interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCATION</td>
<td>UNIFORM, PROGRAM_INPUT, PROGRAM_OUTPUT</td>
<td>TYPE</td>
<td>UNIFORM, PROGRAM_INPUT, PROGRAM_OUTPUT, TRANSFORM_FEEDBACK_VARYING, BUFFER_VARIABLE</td>
</tr>
<tr>
<td>OFFSET</td>
<td>UNIFORM, BUFFER_VARIABLE</td>
<td>ATOMIC_COUNTER_BUFFER_INDEX</td>
<td>UNIFORM</td>
</tr>
<tr>
<td>REFERENCED_BY_VERTEX_SHADER</td>
<td>UNIFORM, UNIFORM_BLOCK, ATOMIC_COUNTER_BUFFER, BUFFER_VARIABLE, PROGRAM_INPUT, PROGRAM_OUTPUT</td>
<td>BLOCK_INDEX</td>
<td>UNIFORM, BUFFER_VARIABLE</td>
</tr>
<tr>
<td>REFERENCED_BY_FRAGMENT_SHADER</td>
<td>UNIFORM, UNIFORM_BLOCK, ATOMIC_COUNTER_BUFFER, BUFFER_VARIABLE, PROGRAM_INPUT, PROGRAM_OUTPUT</td>
<td>NAME_LENGTH</td>
<td>all but ATOMIC_COUNTER_BUFFER</td>
</tr>
<tr>
<td>REFERENCED_BY_COMPUTE_SHADER</td>
<td>UNIFORM, UNIFORM_BLOCK, ATOMIC_COUNTER_BUFFER, BUFFER_VARIABLE, PROGRAM_INPUT, PROGRAM_OUTPUT</td>
<td>ARRAY_STRIDE</td>
<td>UNIFORM, BUFFER_VARIABLE</td>
</tr>
<tr>
<td>BUFFER_BINDING</td>
<td>UNIFORM_BLOCK, ATOMIC_COUNTER_BUFFER, SHADER_STORAGE_BLOCK</td>
<td>BUFFER_DATA_SIZE</td>
<td>UNIFORM_BLOCK, ATOMIC_COUNTER_BUFFER, SHADER_STORAGE_BLOCK</td>
</tr>
<tr>
<td>NUM_ACTIVE_VARIABLES</td>
<td>UNIFORM_BLOCK, ATOMIC_COUNTER_BUFFER, SHADER_STORAGE_BLOCK</td>
<td>ACTIVE_VARIABLES</td>
<td>UNIFORM_BLOCK, ATOMIC_COUNTER_BUFFER, SHADER_STORAGE_BLOCK</td>
</tr>
<tr>
<td>ARRAY_SIZE</td>
<td>UNIFORM, BUFFER_VARIABLE, PROGRAM_INPUT, PROGRAM_OUTPUT, TRANSFORM_FEEDBACK_VARYING</td>
<td>IS_ROW_MAJOR</td>
<td>UNIFORM, BUFFER_VARIABLE</td>
</tr>
</tbody>
</table>
Vertex Attrib Binding

• Why do we need new vertex attrib API?
  - To be efficient for interleaved vertex buffers
  - To make independent updates of
    - Vertex buffer
    - Vertex format
"Legacy" Vertex Arrays

Buffer Objects

Vertex Arrays

glVertexAttribPointer() establishes buffer binding and provides vertex format definition

Multiple calls needed: one for each vertex array

Data for Vertex Shader

VA₀
VA₁
VA₂
VA₃
VA₄
VA₅

glVertexAttribPointer() establishes buffer binding and provides vertex format definition

Data for Vertex Shader

VA₀
VA₁
VA₂
VA₃
VA₄
VA₅

Vertex Array Object
Vertex Attrib Bindings

Buffer Objects
- Vertex Data
- Vertex Data
- Vertex Data

Vertex Arrays
- glVertexAttribBinding()
- glVertexAttribDivisor()
- glVertexAttribFormat()
- glVertexAttribBinding()

Vertex Buffer Binding Points

Data for Vertex Shader

Less/simpler calls
- buffer binding updates

glBindVertexBuffer(bindingIndex, buffer, ...)

glVertexAttribDivisor()

relative_offset
enabled
size
stride
type
normalized
integer
divisor
pointer
buffer_binding
binding_index
textureGather

- Fetch 2x2 footprint used in linear filtering
- Selected component is returned
- Supported on samplers
  - 2D, 2DArray, Cube, 2DShadow, 2DArrayShadow, CubeShadow
- textureGatherOffset()
  - Run-time offsets \((x,y)\) common for all channels
- Differences from GL
  - Offset must be a constant expression
  - No textureGatherOffsets()
Multisample Textures

• New type of immutable textures
  - TEXTURE_2D_MULTISAMPLE and TEXTURE_2D_MULTISAMPLE_ARRAY
  - No LOD
  - Can be attached to framebuffer
  - New GLSL sampler types
    - \(\{i|u\}\)sampler2DMS
    - \(\{i|u\}\)sampler2DMSArray
  - New texelFetch function
    - Returns requested sample for given coordinates

\[
\begin{align*}
gvec4 \text{texelFetch}(\text{gsampler2DMS sampler}, \text{ivec2 } P, \text{int sample}) \\
gvec4 \text{texelFetch}(\text{gsampler2DMSArray sampler}, \text{ivec3 } P, \text{int sample})
\end{align*}
\]

\[
\begin{align*}
\text{ivec2 textureSize}(\text{gsampler2DMS sampler}) \\
\text{ivec3 textureSize}(\text{gsampler2DMSArray sampler})
\end{align*}
\]
Stencil Texturing

• Enables sampling stencil component from packed depth stencil texture
  - As a unsigned integer
• Per texture switch
• Either
  - DEPTH_COMPONENT or
  - STENCIL_INDEX

![Stencil Texturing Diagram]

- Stencil Texturing Diagram
  - Depth component
  - Stencil component
  - DEPTH_COMPONENT
  - STENCIL_INDEX
  - DEPTH_STENCIL_TEXTURE_MODE
  - glTexParameter*()
OES_texture_stencil8

- **STENCIL_INDEX8** accepted as `<internalformat>` by `glTexImage{2|3}D`, `glTexStorage{2|3}D`, `glTexStorage{2|3}Dmultisample` for `TEXTURE_2D`, `TEXTURE_2D_ARRAY`, `TEXTURE_CUBE_MAP`, `TEXTURE_2D_MULITSAMPLE`, `TEXTURE_2D_MULITSAMPLE_ARRAY`.

- **STENCIL_INDEX** accepted as `<format>` by `glTexImage{2|3}D`, `glTexSubImage{2|3}D` specifying image for `TEXTURE_2D`, `TEXTURE_2D_ARRAY`, `TEXTURE_CUBE_MAP`.

- Render buffers no longer needed if stencil-only format is required.
Open GL ES Version 3.1
Shading Language

Bill Licea-Kane
Qualcomm Technologies, Inc
GDC, San Francisco, March 2014
Overview

- `explicit_uniform_location`
- `shader_helper_invocation`
- `shader_bitfield_operations`
- `arrays_of_arrays`
- `shader_integer_mix`
explicit_uniform_location

- Shader writer may specify default-block uniform location table
- Similar to shader input, shader output location tables
- Goal - portable compile time locations, bindings, offsets
explicit_uniform_location

// vertex shader example
layout(location=0) in vec4 position;
layout(location=1) in vec3 normal;
layout(location=0,binding=1) uniform sampler2D lookupTable;
layout(std140,binding=0) uniform xformBlock {
    mat4 mvpMatrix;
    mat3 normalMatrix;
};

// fragment shader example
layout(location=0) out vec4 color;
layout(location=0,binding=0) uniform sampler2D baseMap;
layout(location=1,binding=2) uniform samplerCubeMap cubeMap;
shader_helper_invocation

- New! Why?
- dFdx(p), dFdy(p)
- fWidth(p)
- implicit derivatives

Forward differencing:

\[ F(x+dx) - F(x) \sim dFdx(x) \cdot dx \quad 1a \]
\[ dFdx(x) \sim \frac{F(x+dx) - F(x)}{dx} \quad 1b \]

Backward differencing:

\[ F(x-dx) - F(x) \sim -dFdx(x) \cdot dx \quad 2a \]
\[ dFdx(x) \sim \frac{F(x) - F(x-dx)}{dx} \quad 2b \]
shader_helper_invocation
shader_helper_invocation
shader_helper_invocation
shader_helper_invocation
shader_helper_invocation
shader_helper_invocation
shader_helper_invocation
shader_helper_invocation
shader_helper_invocation
shader_helper_invocation
shader_helper_invocation
Two types of shader_helper_invocations:
- derivatives and extrapolation
- derivatives and early fragment tests
shader_helper_invocation

- Why now?
- Side effects!
  - atomic_counters
  - shader_image_load_store
  - shader_storage_buffer_objects
- “Atomic operations to... atomic counter variables performed by helper invocations have no effect on the underlying... buffer memory. The values returned by such atomic operations are undefined.”
shader_helper_invocation

layout(location=0) out uint data;
layout(binding=0,offset=0) uniform atomic_uint counter;
void main()
{

    uint limit = atomicDecrement( counter );
    uint count=0;
    for ( int i=0; i<limit; i++ )
    {
        count++;
    }
    data = count;
}

shader_helper_invocation

layout(location=0) out uint data;
layout(binding=0,offset=0) uniform atomic_uint counter;
void main()
{

    uint limit = atomicDecrement( counter );
    uint count=0;
    for ( int i=0; i<limit; i++ )
    {
        count++;
    }
    data = count;
}
shader_helper_invocation

layout(location=0) out uint data;
layout(binding=0,offset=0) uniform atomic_uint counter;
void main()
{
    if (!gl_HelperInvocation)
    {
        uint limit = atomicDecrement(counter);
        uint count=0;
        for (int i=0; i<limit; i++)
        {
            count++;
        }
        data = count;
    }
}
shader_bitfield_operations

- Bitfield operations from GPU_shader5
- added precision qualifiers
shader_bitfield_operations

highp genType frexp(highp genType x, out highp genIType exp);
highp genType ldexp(highp genType x, in highp genIType exp);
shader_bitfield_operations

highp uint packUnorm4x8(mediump vec4 v);
highp uint packSnorm4x8(mediump vec4 v);

mediump vec4 unpackUnorm4x8(highp uint v);
mediump vec4 unpackSnorm4x8(highp uint v);
shader_bitfield_operations

geniType bitfieldExtract(geniType value,
                        int offset, int bits);

genuType bitfieldExtract(genuType value,
                        int offset, int bits);

geniType bitfieldInsert(geniType base, geniType insert,
                        int offset, int bits);

genuType bitfieldInsert(genuType base, genuType insert,
                        int offset, int bits);
shader_bitfield_operations

highp genIType bitfieldReverse(highp genIType value);
highp genUType bitfieldReverse(highp genUType value);
shader_bitfield_operations

lowp genIType bitCount(genIType value);
lowp genIType bitCount(genUType value);

lowp genIType findLSB(genIType value);
lowp genIType findLSB(genUType value);

lowp genIType findMSB(highp genIType value);
lowp genIType findMSB(highp genUType value);
shader_bitfield_operations

highp genUType uaddCarry(highp genUType x,
                         highp genUType y,
                         out lowp genUType carry);

highp genUType usubBorrow(highp genUType x,
                          highp genUType y,
                          out lowp genUType borrow);
shader_bitfield_operations

void umulExtended(highp genUType x, highp genUType y,
   out highp genUType msb,
   out highp genUType lsb);
void imulExtended(highp genIType x, highp genIType y,
   out highp genIType msb,
   out highp genIType lsb);
arrays_of_arrays

- You can have arrays of arrays
- Subset
  - Can not use arrays of arrays for inputs and outputs
  - Can not use c-style initializers, {}, use constructors
  - Can not use all the declaration styles
arrays_of_arrays

int a[2][3][4][5];

// ok
arrays_of_arrays

int a[2][3][4][5];
int [2][3][4][5] b;

// ok
// error
arrays_of_arrays

int a[2][3][4][5];
int[2][3][4][5] b;  // ok
int[3][4][5] c[2];  // error
int[4][5] d[2][3];  // error
arrays_of_arrays

```c
int a[2][3][4][5]; // ok
int[2][3][4][5] b; // error
int[3][4][5] c[2]; // error
int[4][5] d[2][3]; // error
int[5] e[2][3][4]; // ok
```
arrays_of_arrays

int a[2][3][4][5];  // ok
int[2][3][4][5] b;  // error
int[3][4][5] c[2];  // error
int[4][5] d[2][3];  // error
int[5] e[2][3][4];  // ok
int f[2][2] = int[2][2](int[2](1,2),int[2](3,4));  // error
arrays_of_arrays

```c
int a[2][3][4][5];  // ok
int[2][3][4][5] b;  // error
int[3][4][5] c[2];  // error
int[4][5] d[2][3];  // error
int[5] e[2][3][4];  // ok
int f[2][2] = int[2][2](int[2](1,2),int[2](3,4)); // error
int g[2][2] = int[][](int[2](1,2),int[2](3,4)); // ok
int h[2][2] = int[][](int[](1,2),int[](3,4)); // ok
```
shader_integer_mix

genIType mix(genIType x,
             genIType y,
             genBType a)

genUType mix(genUType x,
             genUType y,
             genBType a)

genBType mix(genBType x,
             genBType y,
             genBType a)
developer.qualcomm.com - Thank You!
EGL 1.5 Features

Jon Leech
EGL and OpenGL ES 3.1 Spec Editor
First update since 2008

- Lots of EGL functionality was developed in extensions
- Traditional EGL
  - Interface to underlying platform (aka window system)
  - Graphics context and surface management
- Current EGL - API “hub”
  - Client API interop - sharing with GL ES, GL, CL, VG
  - EGLImage - share images (textures, video, etc.)
  - EGLSync - cross-API GPU-level fences
Platform Interfaces

- **EGL can now support multiple platforms in one runtime**
  - Traditional window systems or full-screen or headless
  - Android, GBM, Wayland, X11 defined today, more coming
  - Different ways to create EGLDisplays
  - Additional control possible, such as specifying an X11 Screen
  - Distinguish platform extensions from client extensions

- **Cleaned up 64-bit support**
  - EGLint was supposed to contain all possible attribute values but this didn’t always happen in practice
  - New EGLAttrib type explicitly large enough to contain pointers and handles
  - All commands taking attribute lists will henceforth be defined to use EGLAttrib
Contexts and Surfaces

• Make context current without a surface
  - Offscreen render-to-texture and compute
  - Graphics on compute servers without displays

• Robust context creation
  - Important for WebGL in particular
  - Restricts contexts to guard against malicious attacks on GPU, by enabling extensions like GL_EXT_robustness
  - Provides guarantees against buffer overruns
  - Control of and detection of graphics reset notification so single context / app failure won’t cause wider takedown of the graphics stack

• Create EGLSurfaces which GL / GL ES will render to in sRGB color space
Image and Event Sharing

- EGLImage allows sharing textures / renderbuffers / image objects between client APIs
  - Promote a client API resource to an EGLImage
  - Bind an EGLImage to a client image
  - Explicit handoff between APIs (Acquire/Release)
  - Supported as KHR extensions for years, now EGL core

- EGLSync allows finer-grained synchronization between client APIs
  - Like GL fence sync objects, but the resulting EGLSync object can be converted to other forms (e.g. link EGLSync to OpenCL event)
  - Server-side waiting allows implementations to perform cross-API synchronization inside the GPU with no round trips to the CPU
Potential EGL Future Directions

• These are interesting possibilities, not commitments
• EGLImageStream extensions are very powerful today
  - But need wider implementation in drivers
  - Would like to stream other types of data - unformatted buffers for metadata and more
  - GPU-to-GPU streaming and invoking client API activities directly from other client APIs without CPU intervention
• Separation of traditional context/surface functionality from “hub” functionality
• Support for new Khronos APIs where appropriate
  - Streaming video + image processing + display use case
Wrap-up
OpenGL ES 3.1 Status

• Specifications and manual pages published
  - [http://www.khronos.org/registry/gles/](http://www.khronos.org/registry/gles/)

• Conformance test is code complete and undergoing test
  - Expecting to start certifying implementations by mid-June
  - Note: major upgrade to ES 3.0 test released in January

• Khronos reference compiler in progress
  - ‘Oracle of correctness’ for shader programs
Khronos wants you!

- Specs can always be better
  - Tell us when you find an ambiguity or a hole in the specs

- Conformance tests can always be better
  - Tell us when you find out-of-spec behavior in conformant implementations

- Man pages aren’t perfect
  - Tell us if there is missing / wrong information

- https://www.khronos.org/bugzilla/
Have Fun!