Starting out with OpenGL® ES 3.0
Games for Mobile

Rich Su
Graphics Support Manager APAC
Agenda

- Some foundational work
- Instanced geometry rendering
- Transform feedback
- Occlusion Queries
What’s New in OpenGL ES 3.0

- Updated shading language – GLSL ES 3.00
- Updated vertex shading using *transform feedback mode*
- Lots of new object types
  - shader uniform buffers
  - vertex array objects
  - sampler objects
  - sync objects
  - pixel buffer objects (PBOs)
- Occlusion queries
  - that work efficiently with tiled renderers
- Instanced rendering
- New texture formats and features
  - texture swizzles
  - (sized) integer formats
  - ETC2 texture compression
- Primitive restart
- ... and a whole lot more
A Quick Review …

- OpenGL ES 3.0 is a shader-based API

- The pipeline has two shading stages:
  
<table>
<thead>
<tr>
<th>Stage</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex Shader</td>
<td>Transformation of 3D world data to 2D screen coordinates.</td>
</tr>
<tr>
<td>Fragment Shader</td>
<td>Shading (coloring) of potential pixels on the screen</td>
</tr>
</tbody>
</table>
Preparing Geometric Data for OpenGL ES

- All data sent to OpenGL ES must be passed through a buffer
  
<table>
<thead>
<tr>
<th>Buffer Type</th>
<th>Description</th>
<th>Usage Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>client-side arrays</td>
<td>CPU-based memory like you get from malloc()</td>
<td>Evil and bandwidth unfriendly</td>
</tr>
<tr>
<td>vertex-buffer objects (VBOs)</td>
<td>GPU-based memory that the graphics driver allocates on your behalf</td>
<td>Fast and GPU friendly</td>
</tr>
</tbody>
</table>

- OpenGL ES 3.0 supports both varieties, but only use VBOs
- We’ll see more uses for buffers in a bit
Rendering in OpenGL ES 3.0

- In OpenGL ES 2.0, you could render in two ways:

<table>
<thead>
<tr>
<th>Rendering Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>glDrawArrays</td>
<td>Pass vertex data to vertex shader sequentially</td>
</tr>
<tr>
<td>glDrawElements</td>
<td>Pass vertex data to vertex shader indexed by element list</td>
</tr>
</tbody>
</table>

- Rendering the same model multiple times was inconvenient
- In OpenGL ES 3.0, we can *instance rendering*
  - one draw call replaces entire loop from above
  - Shader Inputs: gl_InstanceID

<table>
<thead>
<tr>
<th>Rendering Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>glDrawArraysInstanced</td>
<td>Repeatedly pass vertex data to vertex shader sequentially</td>
</tr>
<tr>
<td>glDrawElementsInstanced</td>
<td>Repeatedly pass vertex data to vertex shader indexed by element list</td>
</tr>
</tbody>
</table>
Converting to Instanced Rendering

- Less code, more performance

```c
GLfloat xform[NumInstances][3] = {
    { x0, y0, z0 },
    { x1, y1, z1 },
    ...
};

for (int i = 0; i < NumInstances; ++i) {
    glUniform3fv(xform, 1, xform[i]);
    glDrawArrays(GL_TRIANGLES, 0, NumTris);
}

glUniform3fv(xform, NumInstances, xform);
glDrawArraysInstanced(GL_TRIANGLES, 0, NumTris, NumInstances);
```
Converting to Instanced Rendering

```glsl
in vec4 position;
uniform vec4 xform;
void main()
{
    gl_Position = position + xform[gl_InstanceID];
}
```
Instance Rendering Demo
Optimally Storing Data Using Uniform Buffers

- **Uniforms** are like constant global variables for a shader
  - their value stays the same for all primitives in a draw call
- Loading large numbers of uniform variables is tedious
  - there is a struct packaging mechanism, but it's not widely used
- **Uniform Buffer Objects** let you load many uniforms easily

```cpp
uniform vec4 position[NumObjects];
uniform vec4 velocity[NumObjects];
uniform float drag[NumObjects];

void main() { ... }
```

```cpp
uniform ObjectData {
    vec4 position[NumObjects];
    vec4 velocity[NumObjects];
    float drag[NumObjects];
};

void main() { ... }
```
Initializing Uniforms: a comparison

```cpp
struct {
    GLfloat position[NumObjects][4];
    GLfloat velocity[NumObjects][4];
    GLfloat drag[NumObjects];
} data;

GLuint positionLoc = glGetUniformLocation( program, "position" );
GLuint velocityLoc = glGetUniformLocation( program, "velocity" );
GLuint dragLoc = glGetUniformLocation( program, "drag" );

if( positionLoc < 0 || velocityLoc < 0 || dragLoc < 0 ) {
    throw UniformLocationError();
}

glUniform4fv( positionLoc, NumObjects, data.position );
glUniform4fv( velocityLoc, NumObjects, data.velocity );
glUniform4fv( dragLoc, NumObjects, data.drag );
```

(application code)
Instance Tessellation Demo
Recall that every vertex is processed by a vertex shader.

For complex vertex shaders executing the shader could take a long time.

- Could result in this being a performance bottleneck.

**Transform feedback** allows the results of vertex shading to be captured in a buffer, and rendered later.

- Very useful if the object doesn’t change between frames.
During transform feedback, the outputs of the vertex shader are written to a buffer, which can then become a new data buffer for vertex shading.
Configuring Transform Feedback

1. Compile and link transform feedback shader program
2. Determine the outputs of your transform feedback buffer
   - the order of varying names specify their output index
3. Associate transform feedback buffer with output streams

```c
const GLchar* varyings = { "location", "velocity" };
glTransformFeedbackVaryings( program, 2, varyings,
   GL_SEPARATE_ATTRIBS );

GLuint index  = 0; // for "location"
GLintptr offset = 0; // "location" starts at the beginning of the buffer
GLsizeptr size = 4 * NumVertices * sizeof(GLfloat);
glBindBufferRange( GL_TRANSFORM_FEEDBACK_BUFFER, index, xfbID, offset, size );

index  = 1;   // for "velocity"
offset = size; // data starts immediately after previous entries
glBindBufferRange( GL_TRANSFORM_FEEDBACK_BUFFER, index, xfbID, offset, size );
```
Generating Data with Transform Feedback

```c
glEnable( GL_RASTERIZER_DISCARD );

glUseProgram( xfbProgram );

 glBeginTransformFeedback( GL_POINTS );
    glDrawArrays( GL_POINTS, 0, NumVertices );
 glEndTransformFeedback();

 glDisable( GL_RASTERIZER_DISCARD );
```

Specify that we're not going to engage the rasterizer to generate any fragments.
Generating Data with Transform Feedback

```c
glEnable( GL_RASTERIZER_DISCARD );

glUseProgram( xfbProgram );

glBeginTransformFeedback( GL_POINTS );

glDrawArrays( GL_POINTS, 0, NumVertices );

glEndTransformFeedback();

glDisable( GL_RASTERIZER_DISCARD );
```

Switch to our transform feedback shader program (this is the one with our xfb varyings in it)
Generating Data with Transform Feedback

```c
glEnable( GL_RASTERIZER_DISCARD );

glUseProgram( xfbProgram );

glBeginTransformFeedback( GL_POINTS );

glDrawArrays( GL_POINTS, 0, NumVertices );

glEndTransformFeedback();

glDisable( GL_RASTERIZER_DISCARD );
```

Switch into transform feedback mode, requesting that points are generated.
Generating Data with Transform Feedback

```c
glEnable( GL_RASTERIZER_DISCARD );

glUseProgram( xfbProgram );

glBeginTransformFeedback( GL_POINTS );

glDrawArrays( GL_POINTS, 0, NumVertices );

glEndTransformFeedback();

glDisable( GL_RASTERIZER_DISCARD );
```

Send our input data through our transform feedback shader, which will output into our vertex buffer.
Generating Data with Transform Feedback

```c
glEnable( GL_RASTERIZER_DISCARD );

glUseProgram( xfbProgram );

glBeginTransformFeedback( GL_POINTS );

    glDrawArrays( GL_POINTS, 0, NumVertices );

glEndTransformFeedback();

Return to normal rendering (i.e., vertex shader output isn’t sent to an xfb buffer)

glDisable( GL_RASTERIZER_DISCARD );
```
Generating Data with Transform Feedback

```glsl
glEnable( GL_RASTERIZER_DISCARD );
glUseProgram( xfbProgram );
glBeginTransformFeedback( GL_POINTS );
glDrawArrays( GL_POINTS, 0, NumVertices );
glEndTransformFeedback();
glDisable( GL_RASTERIZER_DISCARD );
```

Disable the rasterizer sending fragments to the bit-bucket.
Transform Feedback Demo
Occlusion Queries

- OpenGL shaded before determining visibility
  - the fragment shader is executed before depth testing
- For complex fragment shading, this can be wasteful
  - lots of work for naught
- Occlusion Queries help determine if the fragments from a rendered object will pass the depth test
- Fundamental Idea: render a simply shaded, low-resolution version of your object to determine if any of it is visible
  - constant color, object-aligned bounding-boxes are a nice choice
Using Occlusion Queries

- Queries need to be allocated
  
  ```c
  GLuint queries[NumQueries];
  glGenQueries( NumQueries, queries );
  ```

- Render in query mode
  
  ```c
  glBeginQuery( GL_ANY_SAMPLES_PASSED_CONSERVATIVE, queries[i] );
  glDrawArrays( ... );
  glEndQuery( GL_ANY_SAMPLES_PASSED_CONSERVATIVE );
  ```
Using Occlusion Queries (cont’d.)

- Check if query computation is completed
  ```
  GLboolean ready;
  GLboolean visible;
  do {
    glGetQueryObjectiv(queries[i],
        GL_QUERY_RESULT_AVAILABLE, &ready );
  } while ( !ready );
  glGetQueryObjective(queries[i], GL_QUERY_RESULT, visible );
  if ( visible ) {
    // render
  }
  ```
Occlusion Query Demo
Mobile Gaming and OpenCL™

Rich Su
Graphics Support Manager APAC
GPU Compute

- CPU is not designed for parallel workloads
- GPU is massively parallel – historically used only for graphics
- Enter GPU Compute
What is OpenCL?

- Khronos API
- Implemented in desktop GPU and CPUs
- Similar structure to OpenGL® ES
- Allows access to the compute potential of the GPU
- High performance for parallel tasks
- Can share data with OpenGL ES
OpenCL

for (int i = 0; i < arraySize; i++)
{
    output[i] = inputA[i] + inputB[i];
}

__kernel void kernel_name(__global int* inputA,
                          __global int* inputB,
                          __global int* output)
{
    int i = get_global_id(0);
    output[i] = inputA[i] + inputB[i];
}

cEnqueueNDRangeKernel(..., kernel, ..., arraySize, ...)

The Architecture for the Digital World®
OpenCL Vectors

for (int i = 0; i < arraySize; i++)
{
    output[i] = inputA[i] + inputB[i];
}

__kernel void kernel_name(__global int* inputA,
                          __global int* inputB,
                          __global int* output)
{
    int i = get_global_id(0);
    int4 a = vload4(i, inputA);
    int4 b = vload4(i, inputB);
    vstore4(a + b, i, output);
}

clEnqueueNDRangeKernel(..., kernel, ..., arraySize / 4, ...)
inPutA, inputB
OpenCL on Mali™ GPUs

- Hardware design for GPU Compute
- Vector capable ALUs
- Unified memory
- Full Profile
OpenCL Gaming Use Cases

- Physics
- AI
- Voice recognition
- Gesture recognition
- AR
- Multimedia post-processing

These images are still going to change (JK)

Anand Patel is working on this.
Physics Demo
Physics Demo

- Spring model with 6,000 vertices

- OpenCL version:
  - 8x times faster and twice the number of vertices
  - Single digit CPU load

- Multithreaded C version:
  - 100% CPU load
OpenCL Face Detection
Face Detection Case Study

- Initial investigation focused on face detection application accelerated using OpenCL
Mali OpenCL SDK

- Simplify writing, porting and optimizing OpenCL 1.1 code for Mali GPU based platforms

- Demonstrate key differentiating features to developers and programmers
OpenCL Performance Analysis

ARM DS-5™ and Streamline™ Performance Analyzer

- Support for graphics and GPU compute performance analysis on Mali-T604/T658
- Timeline profiling of hardware counters for detailed analysis
- Software counter support for OpenCL 1.1
  - Custom counters
  - Per-core/thread/process granularity
Summary

- GPU Compute in a familiar style
- Available on Mali GPU platforms
- OpenCL Resources and tools available from ARM
  - [http://malideveloper.arm.com](http://malideveloper.arm.com)
- Potential for OpenCL in mobile gaming
Thank you!