Mali & OpenGL ES 3.0

Dave Shreiner
Jon Kirkham
ARM

Game Developers' Conference
27 March 2013

Agenda

- Some foundational work
- Instanced geometry rendering
- Transform feedback
- Occlusion Queries

What's New in OpenGL ES 3.0

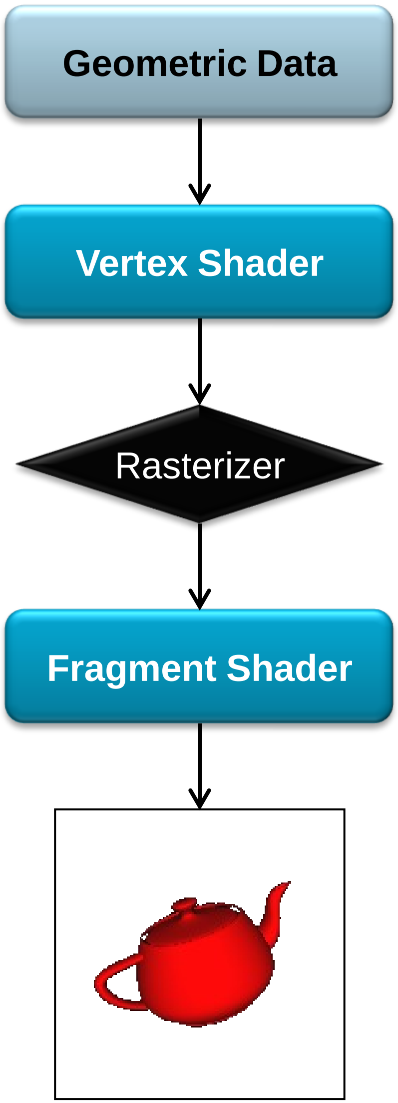
- Updated shading language – GLSL ES 3.00
- Updated vertex shading using *transform feedback mode*
- Lots of new object types
  - shader uniform buffers
  - vertex array objects
  - sampler objects
  - sync objects
  - pixel buffer objects (PBOs)
- Occlusion queries
  - that work efficiently with tiled 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:

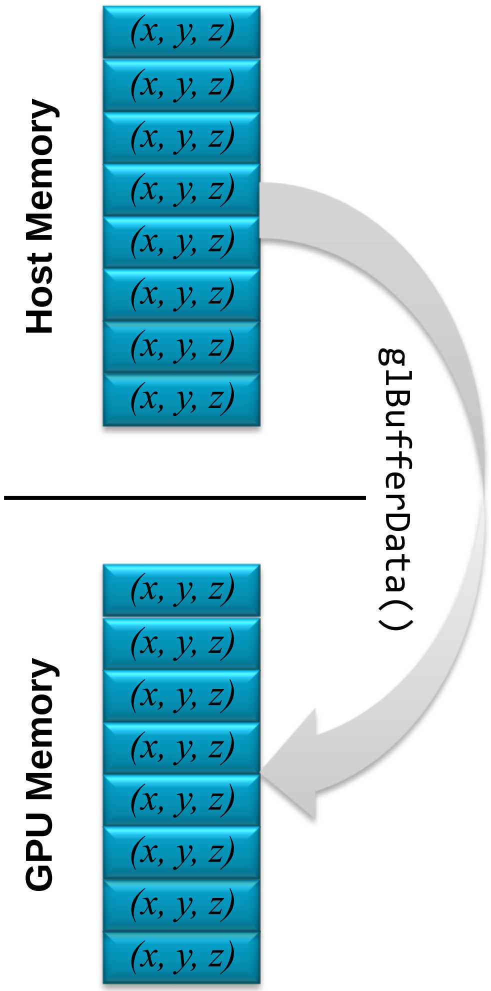
Stage	Operation
Vertex Shader	Transformation of 3D world data to 2D screen coordinates.
Fragment Shader	Shading (coloring) of *potential* pixels on the screen



Preparing Geometric Data for OpenGL ES

- All data sent to OpenGL ES must be passed through a buffer

Buffer Type	Description	Usage Characteristics
client-side arrays	CPU-based memory like you get from `malloc()`	Evil and bandwidth unfriendly
vertex-buffer objects (VBOs)	GPU-based memory that the graphics driver allocates on your behalf	Fast and GPU friendly



- 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 ES 2.0, you could render in two ways:

Rendering Command	Description
`glDrawArrays`	Pass vertex data to vertex shader sequentially
`glDrawElements`	Pass vertex data to vertex shader indexed by element list

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

Rendering Command	Description
`glDrawArraysInstanced`	Repeatedly pass vertex data to vertex shader sequentially
`glDrawElementsInstanced`	Repeatedly pass vertex data to vertex shader indexed by element list

Converting to Instanced Rendering

- Less code, more performance

(application code)

```
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 );
}
```

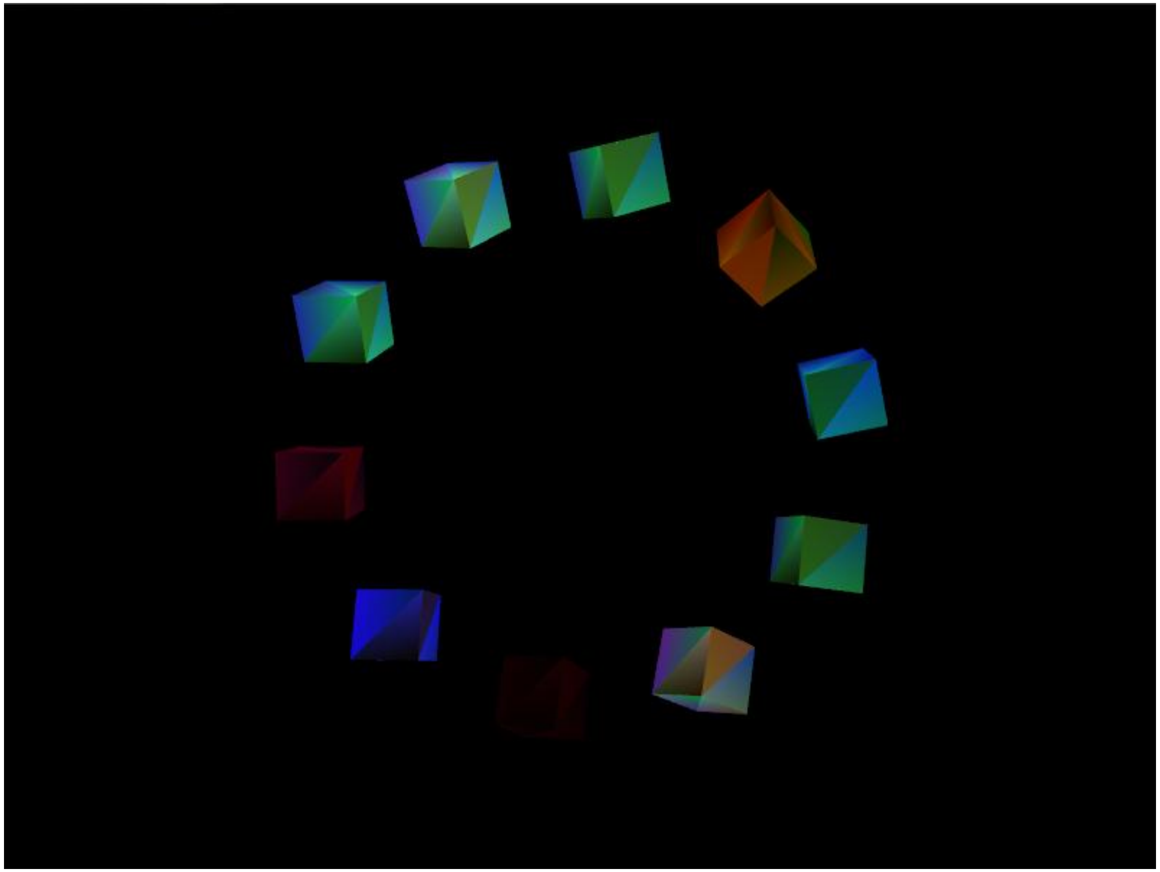
```
glUniform3fv( xform, NumInstances, xform );
glDrawArraysInstanced( GL_TRIANGLES, 0, NumTris, NumInstances );
```

Converting to Instanced Rendering

(shader code)

```
in vec4 position;

uniform vec4  xform;

void main()
{
    gl_Position = position + xform;
}
```

```
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

(shader code)

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

void main() { … }
```

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

void main() { … }
```

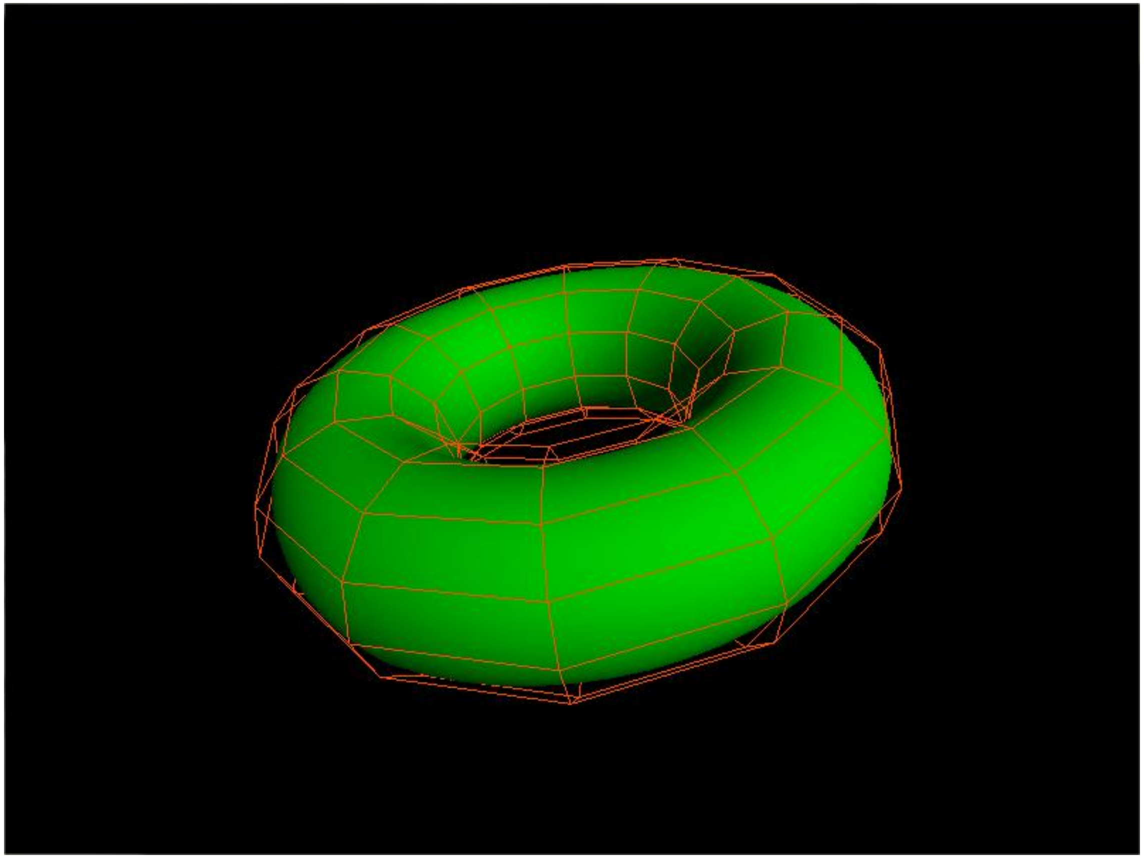
Initializing Uniforms: A Comparison

(application code)

```
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 );
```

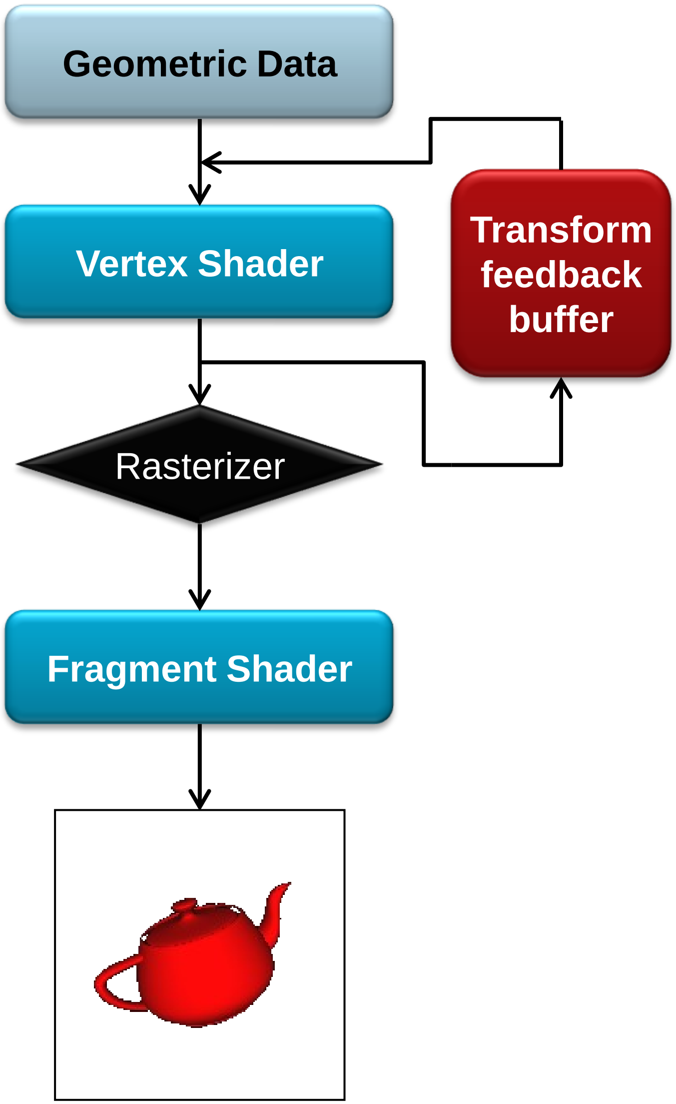
```
glGenBuffer( 1, &uniformBuffer );
glBufferData( GL_UNIFORM_BUFFER, sizeof(data), data, GL_STATIC_DRAW );
GLuint uniformIndex = glGetUniformBlockIndex( program, "ObjectData" );
glUniformBlockBinding( program, uniformIndex, n );
glBindBufferBase( GL_UNIFORM_BUFFER, 0, uniformBuffer );
```



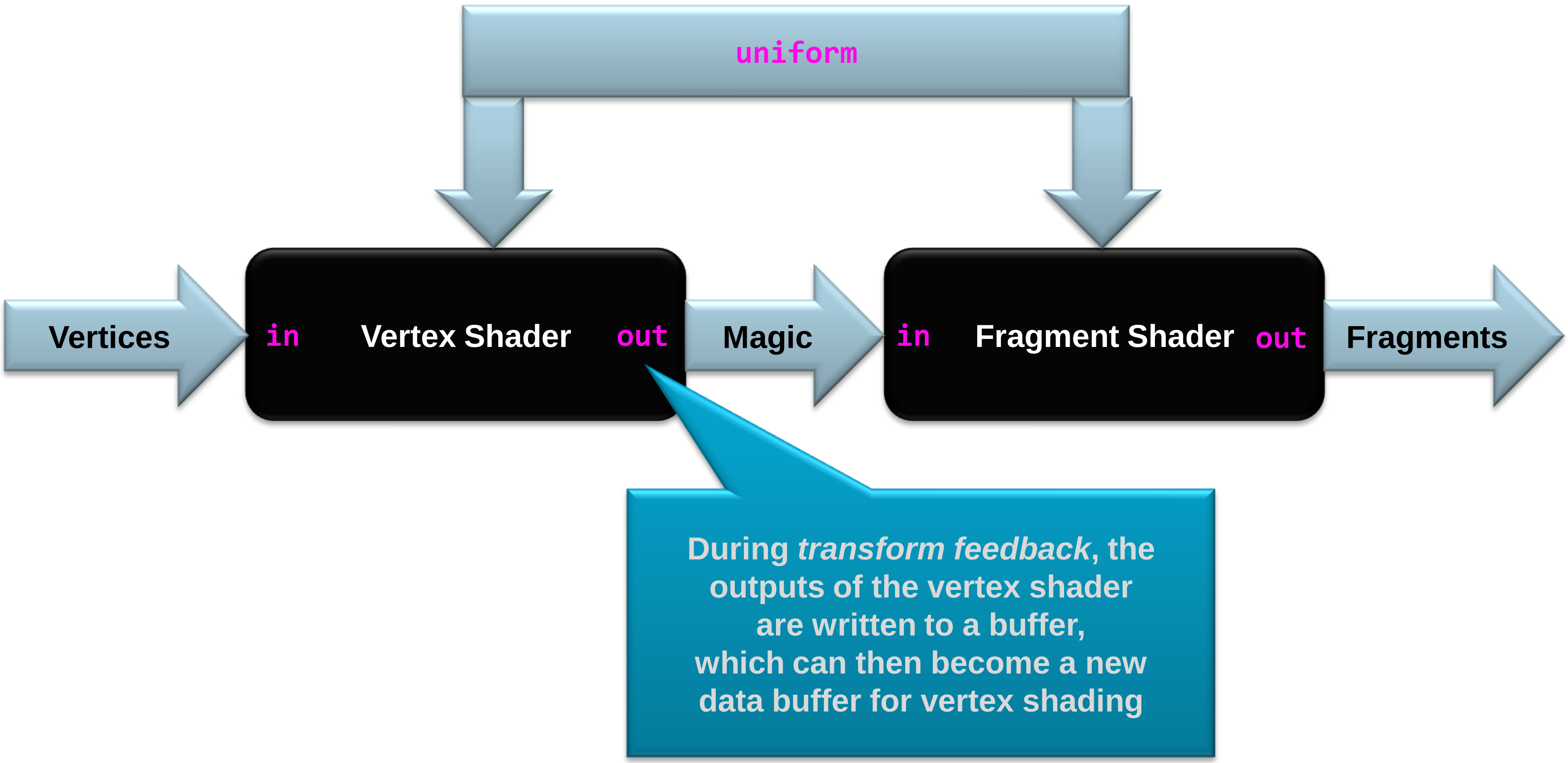
Instanced Tessellation Demo

Transform Feedback

- 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



Data Flow in Shaders



Configuring Transform Feedback

1. Compile and link transform feedback shader program
2. Determine the outputs of your transform feedback buffer

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

- the order of varying names specify their output index

3. Associate transform feedback buffer with output streams

```
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 “veclocity”
offset = size; // data starts immediately after previous entries
glBindBufferRange( GL_TRANSFORM_FEEDBACK_BUFFER, index, xfbID, offset, size);
```

Generating Data with Transform Feedback

```
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

```
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

```
glEnable( GL_RASTERIZER_DISCARD );
 glUseProgram( xfbProgram );
  glBeginTranformFeedback( 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

```
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

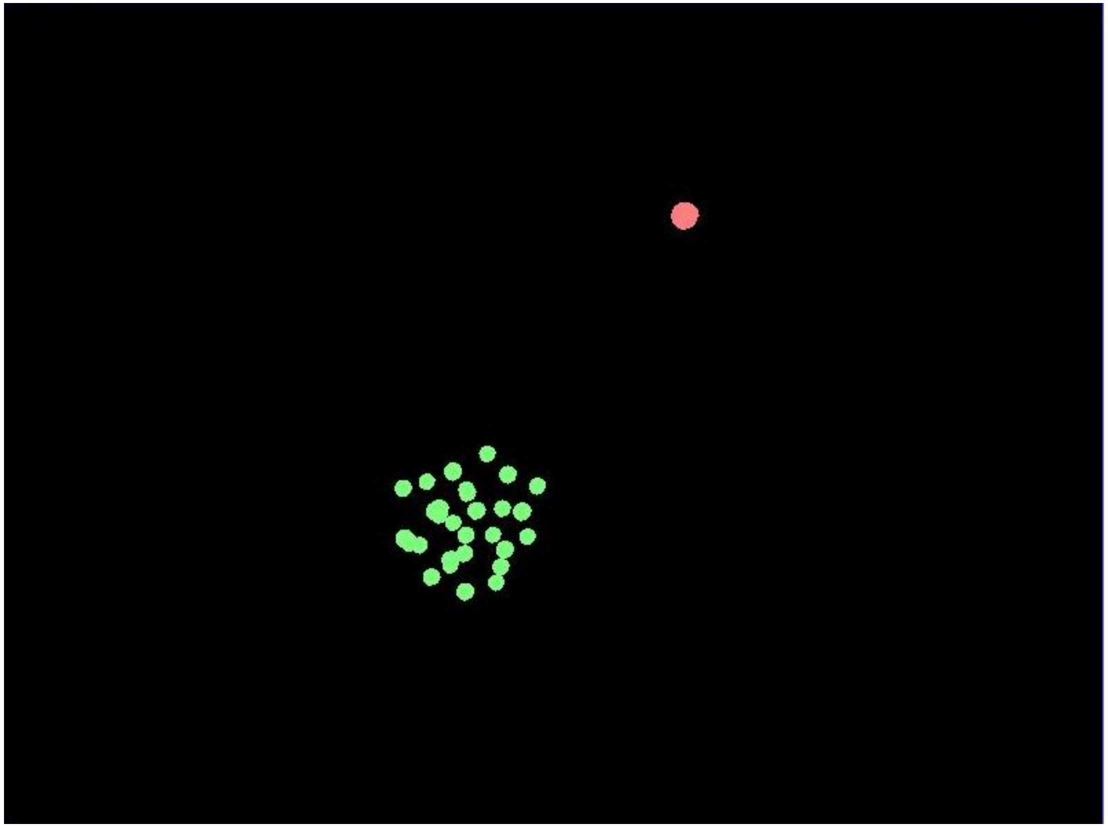
```
glEnable( GL_RASTERIZER_DISCARD );
 glUseProgram( xfbProgram );
  glBeginTransformFeedback( GL_POINTS );
    glDrawArrays( GL_POINTS, 0, NumVertices );
  glEndTransformFeedback();
glDisable( GL_RASTERIZER_DISCARD );
```

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

Generating Data with Transform Feedback

```
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 *shades* 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

```
GLuint queries[NumQueries];
glGenQueries( NumQueries, queries );
```

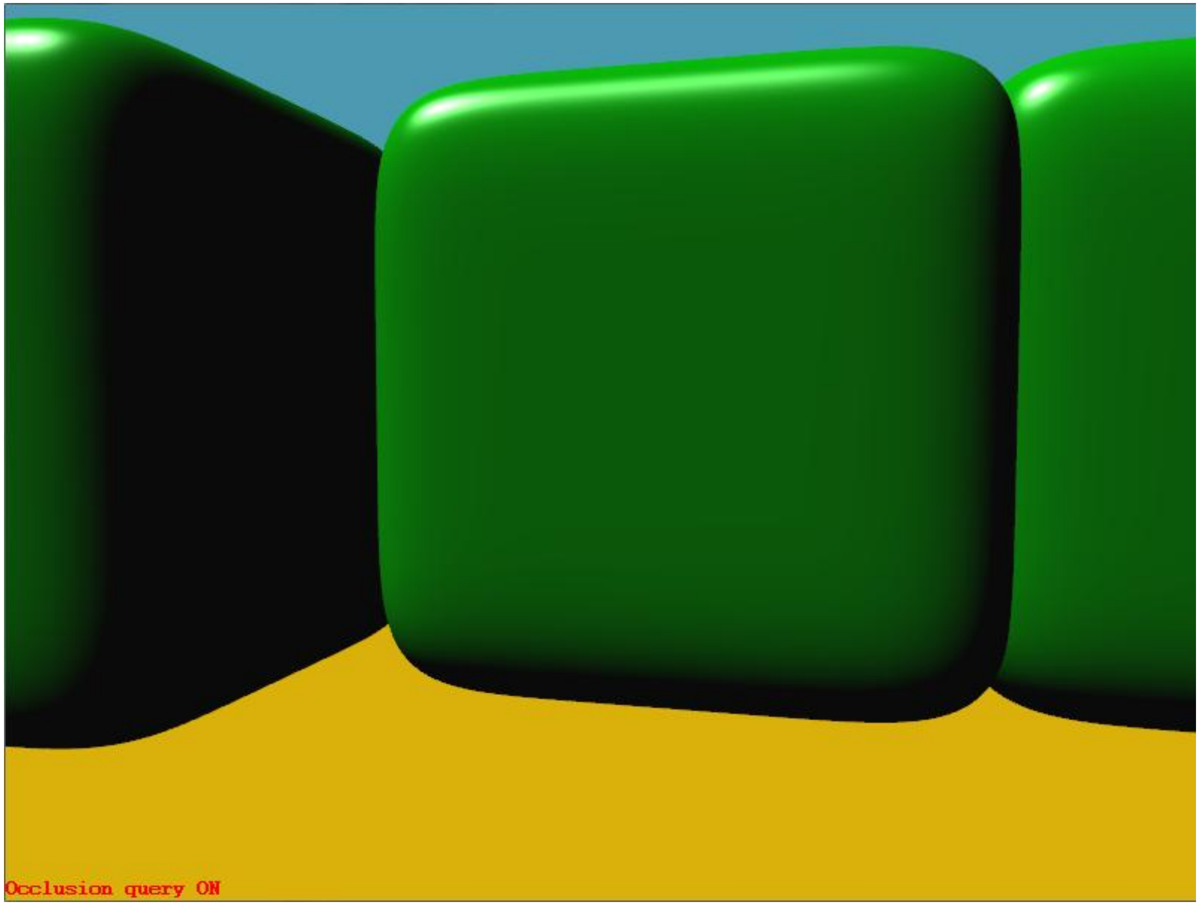
- Render in query mode

```
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( GL_ANY_SAMPLES_PASSED_CONSERVATIVE,
        GL_QUERY_RESULT_AVAILABLE, &ready );
} while ( !ready );

glGetQueryObjective( GL_ANY_SAMPLES_PASSED_CONSERVATIVE,
    GL_QUERY_RESULT, visible );

if ( visible ) {
    // render
};
```



Occlusion Query Demo

END