OpenGL® ES 3.0 and Beyond
How To Deliver Desktop Graphics on Mobile Platforms

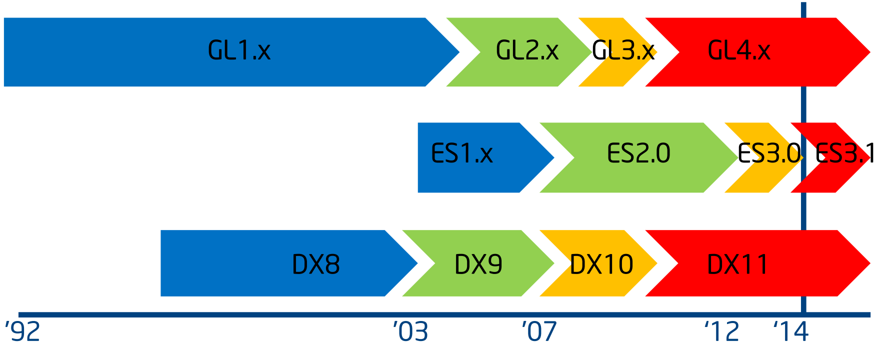
Chris Kirkpatrick, Jon Kennedy

Legal

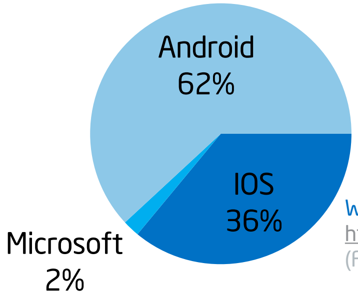
Why is OpenGL ES 3.0 and Beyond Important?

OpenGL ES 3.1 specification is released at GDC 2014.
http://www.khronos.org/registry/gles/

OpenGL ES 3.1 is reaching parity with desktop

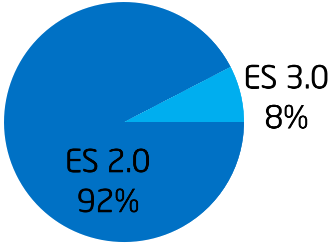


Android is dominant in the market



Worldwide Tablet Sales to End Users 2013
http://www.gartner.com/newsroom/id/2674215
(Feb 2014)

OpenGL ES 3.0 is gaining market share



Android OpenGL ES Version Support Distribution
https://developer.android.com/about/dashboards/index.html
(Feb 2014)

New Features for OpenGL ES 3.0

Main new features

- Multiple Render Targets
- Occlusion Queries
- Instanced rendering
- Uniform Buffer Objects (UBO) and Uniform Blocks
- Transform feedback
- Primitive restart
- Program Binary

Enhanced texturing functionality

- Swizzles, 3D textures, 2D array textures, LOD/MIP level clamps, seamless cube maps, immutable textures, NPOT textures, sampler objects

New renderbuffer and texture formats

- Floating point formats
- Shared exponent RGB formats
- ETC/EAC texture compression
- Depth and depth/stencil formats
- Single and dual channel texture
  - (R and RG)

ES Shading Language Version 3.00

- Full support for 32 bit integer/floating point data types (IEEE754)
- In/out storage qualifier
  - value copied to/from subsequent/previous pipeline stage
- Array constructors and operations
- New built-in functions

OpenGL ES 3.0 - Multi-Render Targets

What is it?

- Enables writing to multiple framebuffer color buffer attachment points with a single pass

Why is it useful?

- Techniques requiring multiple passes can be condensed into a single pass to save redundant execution of the vertex shader
- Useful for Deferred Shading and Screen Space Ambient Occlusion

OpenGL ES 3.0 - Multi-Render Targets

Enabled by attaching framebuffer-attachable images to GL_COLOR_ATTACHMENTi of a created FBO

- Support for at least 4 attachment points
  - Intel supports 8
- Maximum specified by GL_MAX_COLOR_ATTACHMENTS

Most often used in deferred shading i.e.

- 1 colour buffer for the surface colours
- 1 colour buffer for the surface normals
- 1 colour buffer for the depth values
- 1 colour buffer for extra lighting information, such as specular or ambient occlusion

OpenGL ES 3.0 - Multi-Render Targets Sample

GL ES API Code Snippet

```
// Create FBO and bind it
glGenFramebuffers(1, &fbo);
glBindFramebuffer(GL_FRAMEBUFFER, fbo);

// Create 2 textures, allocate storage and attach to FBO
glGenTextures(2, texBuf);
…
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, texBuf[0], 0);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1, texBuf[1], 0);

// Set the list of draw buffers.
Glenum drawBuffers[2] = {GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1};
glDrawBuffers(2, drawBuffers);
```

GLSL Fragment Shader Snippet

```
out vec4 my_FragData[2];

void main(void)
{
    my_FragData[0] = vec4(1.0, 0.0, 0.0, 1.0);
    my_FragData[1] = vec4(0.0, 1.0, 0.0, 1.0);
}
```

OpenGL ES 3.0 - Occlusion Queries

What is it?

- A hardware method for detecting whether an object is visible
  - Works by testing if samples pass the depth test
- Queries are asynchronous, but blocking call available if required

Why is it useful?

- Remove complex scene geometry by culling large batches of geometry via bounding box tests
  - Best on large scenes with large nearby occluders

OpenGL ES 3.0 - Occlusion Queries Sample

GL ES API Code Snippet

```
glGenQueries(1,  &query);
glBeginQuery(GL_ANY_SAMPLES_PASSED, query);
// Draw some primitives
…
glEndQuery(query);

// Check if the result is available
glGetQueryObjectuiv(query, GL_QUERY_RESULT_AVAILABLE, &result);
if (result == GL_TRUE)
{
    // This is a blocking call
    glGetQueryObjectuiv(query, GL_QUERY_RESULT, &anyPassed);
}
```

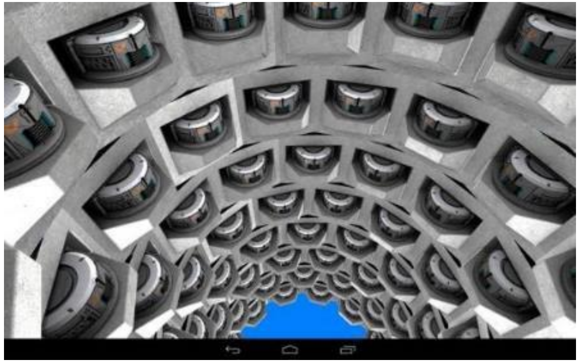
OpenGL ES 3.0 - Instanced Rendering

What is it?

- Enables rendering multiple geometry instances with a single draw call
- Instances may be provided with unique attributes (transformation, bones, etc.)

Why is it useful?

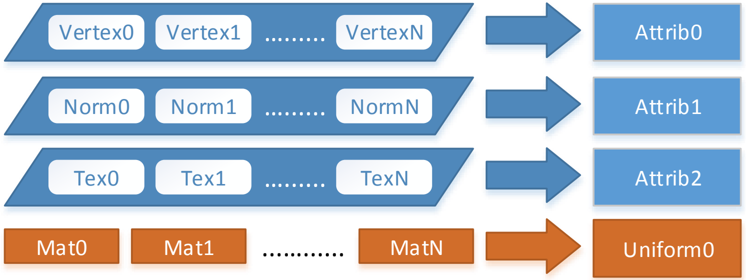
- Reduces API call overhead when rendering duplicate meshes



OpenGL ES 3.0 - Instanced Rendering

Non-instanced

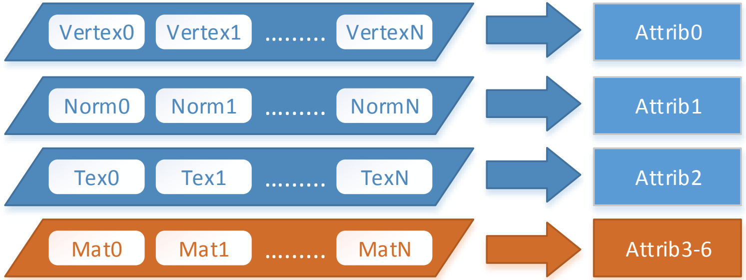
- BO's store Vertex, Normal, Tex data
- Transformations stored as uniform data
  - Set per-instance with glUniformMatrix*




```
for (int i = 0; i < NumInstances; ++i) {
    …
    glUniformMatrix4fv(…);
    ….
    glDrawElements(…);
}
```

Instanced

- BO's store Vertex, Normal, Tex data
- Transformations stored in a BO
  - glVertexAttribDivisor handles the creation of "instanced attributes"




```
glVertexAttribDivisor(3, 1);
…
glDrawElementsInstanced(…, NumInstances);
```

OpenGL ES 3.0 - Instanced Rendering Sample

GL ES API Code Snippet

```
// Attrib 0 (vertex information) changes per vertex
glVertexAttribDivisor(0,0)

// Attrib 1 (matrix data) changes per instance
glVertexAttribDivisor(1,1)
…

// When rendering
glDrawArraysInstanced(Mode, First, Count,
                      NumberOfInstances);
// or
glDrawElementsInstanced(Mode, Count, IndType, Indicies,
                        NumOfInstances);
```

GLES Vertex Shader Code Snippet

```
// By default attributes have a divisor of zero—advancing per vertex
// Attributes with a positive divisor will advance every divisor instances

// The built-in variable gl_InstanceID holds the current instance
// Default value is zero; safe to reference when not using instanced draw
// calls
in vec3 Position;

// Takes attribute positions 1,2,3,4
in mat4 WorldPosition;

// Pass the instance id on to the pixel shader
flat out int InstanceID;

void main()
{
   gl_Position = vec4(WorldPosition + Position, 1.0);
   InstanceID = gl_InstanceID;
};
```

OpenGL® ES 3.1

OpenGL ES 3.1

Intel announced support for the OpenGL ES 3.1 specification on the Bay Trail platform for Android.

http://blogs.intel.com/technology/2014/03/open-gl-es-gdc-2104-sweet-spot-mobile-graphics-evolution/

"Product is based on a published Khronos Specification, and is expected to pass the Khronos Conformance Testing Process when available. Current conformance status can be found at www.khronos.org/conformance."

Intel has extended support beyond the core specification to include Geometry Shaders, Tesselation and Intel Pixel Sync Technology.

OpenGL ES 3.1 Specification and header files can be found here :

http://www.khronos.org/registry/gles/

OpenGL ES 3.1 on Intel's Bay Trail Platform

Shader Stages
- Compute Shaders
- EXT: Tesselation
- EXT: Geometry Shaders

Shader I/O
- Shader Image Load Store
- Shader Storage Buffer Objects

Shader Synchronisation
- Atomic Counters
- OES: Shader Image Atomics
- INTEL: Fragment Shader Ordering

API Helpers
- Indirect Draw Commands
- Framebuffer No Attachments
- Program Interface Query
- Separate Shader Objects
- Vertex Attribute Binding

Textures
- Texture Storage Multisample
- OES: Texture Stencil 8

Blending
- KHR: Blend Equation Advanced

GLSL
- Array of Arrays
- Explicit Uniform Locations
- Shader Bitfield Operation
- Shader Layout Binding
- Texture Gather
- Stencil Texturing

GLSL Extensions
- OES: Shader Multisample Interpolation
- OES: Sample Shading
- OES: Sample Variables

Core

Extension

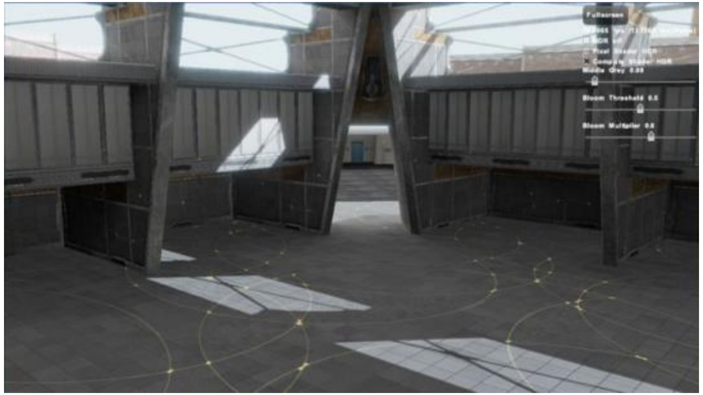
OpenGL ES 3.1 - Compute Shaders

What are they?

- A compute shader is used for general compute on shader defined inputs with shader defined outputs.
- Run logically independent of the 3D pipeline.
  - Although well pipelined with 3D primitives.
- Run at a user defined frequency.
- Similar to OpenCL® Kernels.
  - Allow better integration into 3D applications.
  - Can directly access OpenGL ES textures, images and buffer objects.
  - Can be efficiently pipelined with 3D primitives.
  - Lightweight.

Why are they useful?

- Compute shaders are frequently used on the desktop for image post-processing, deferred rendering, visibility culling, computer vision, particle physics, etc...



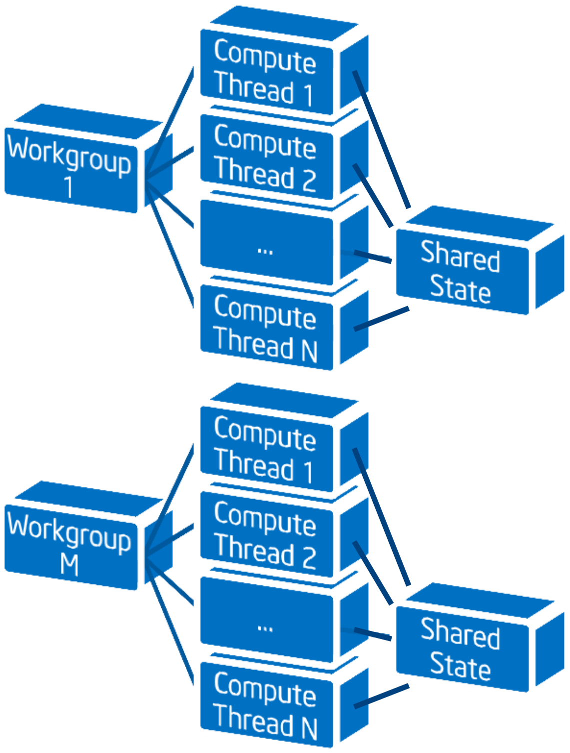
HDR using compute shaders

Cloth using compute shaders

OpenGL ES 3.1 - Compute Shaders

Compute shaders work on:

- Workgroups
  - Each workgroups consists of a number of compute shader threads,
  - The user defines the workgroup size and number of workgroups. Both parameters are in 3 dimensions.
    - The workgroup size is fixed at compilation time,
    - The number of workgroups is specified at dispatch time.
- Compute Shader Threads
  - Each thread can share data with other members of the workgroup via special shared variables,
  - Each thread can issue memory and control barriers to synchronise with other members of the workgroup,
  - Data can **not** be effectively shared between workgroups, unless via images, buffer objects or atomic counters,
  - Each thread can uniquely identify itself within a workgroup and globally with builtin variables. This is the only method for a thread to determine where to get its input and where to write its output.



OpenGL ES 3.1 - Compute Shaders

Compute shaders also bring:

- Shader Image Load Store
  - Random read/write access to a single level of a texture map
  - Atomic operations
- Shader Storage Buffer Objects
  - Random read/write access to variables stored within a buffer object
  - Atomic operations
- Shader Atomic Counters
  - Backed by buffer object memory
  - They allow the proper sequencing of memory accesses between workgroups

These are also available to other shader stages.

GL ES API Code Snippet

```
glGenTextures(1, &texHandle);
glBindTexture(GL_TEXTURE_2D, texHandle);
glTexImage2D(GL_TEXTURE_2D, 0, GL_R32F, 512, 512, 0, GL_RED, GL_FLOAT, NULL);

// Bind the texture to an image so it can be written to
glBindImageTexture(0, texHandle, 0, GL_FALSE, 0, GL_WRITE_ONLY, GL_R32F);

glUseProgram(computeHandle);
GLuint loc = glGetUniformLocation(computeHandle, "roll");
glUniform1f(loc, frame*0.01f);
// 512^2 threads in blocks of 16^2
glDispatchCompute(512/16, 512/16, 1);
```

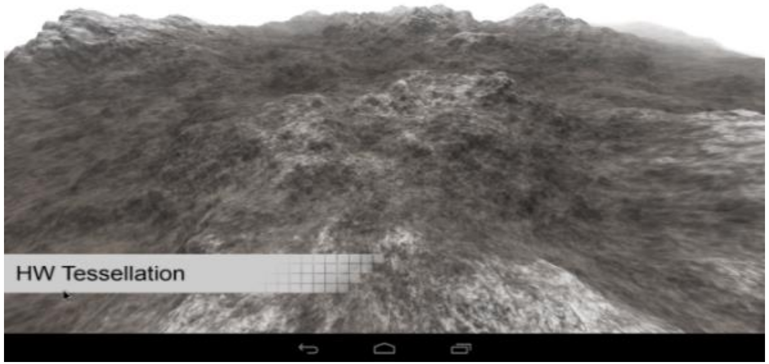
GLSL Compute Shader Code Snippet

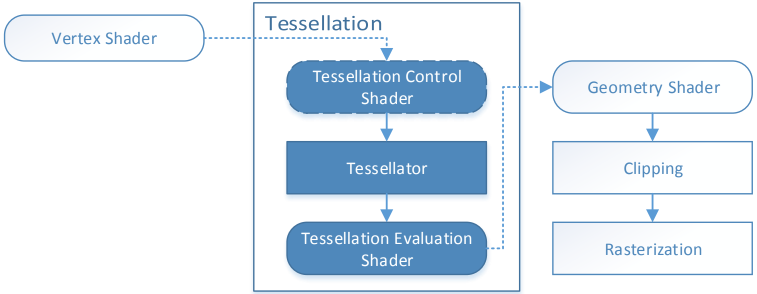
```
uniform float roll;
uniform image2D destTex;
layout (local_size_x = 16, local_size_y = 16) in; // 16x16 threads per workgroup
void main()
{
    ivec2 storePos = ivec2(gl_GlobalInvocationID.xy);
    float localCoef = length(vec2(ivec2(gl_LocalInvocationID.xy)-8)/8.0);
    float globalCoef = sin(float(gl_WorkGroupID.x+gl_WorkGroupID.y)*0.1 + roll)*0.5;
    imageStore(destTex, storePos, vec4(1.0-globalCoef*localCoef, 0.0, 0.0, 0.0));
}
```

OpenGL ES 3.1 EXT Extensions –Tessellation Shaders

What is it?

- An optional stage in the rendering pipeline that is capable of generating additional geometry
    - More efficient than geometry shaders for high levels of geometry expansion; tessellation can not be used for culling patches.
- The **control shader** operates on control points and is responsible for specifying tessellation levels, per-control point position and per patch varyings for the evaluation shader.
- The **evaluation shader** outputs the positions/normal/etc. using abstract coordinates from the tesselator
    - Each invocation operates on a single vertex within the tessellated patch





Why do you want it?

- Reduces memory bandwidth/footprint

What can you do with it?

- Progressive LOD, Displacement mapping, Sub-D surfaces, Complex hair modelling

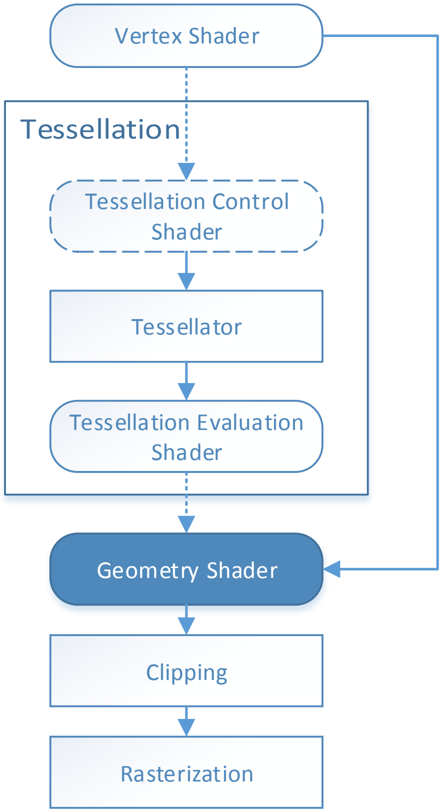
OpenGL ES 3.1 EXT Extensions - Geometry Shaders

What are they?

- A shader which processes the output of the primitive assembler (or the tessellation evaluation shader)
  - Full access to the assembled primitive (points, lines, lines with adjacency, triangles, triangles with adjacency)
  - Output new geometry (points, line strips, triangle strips)—does not have to match the input stage

Why are they useful?

- Impostors, Wireframe rendering, NPR, Procedural Geometry, Shadow Volume Extrusion, Geometry Culling
- Layered rendering(with the appropriate extensions)—rendering a single primitive to multiple images without changing render targets



OpenGL ES 3.1 Intel Extensions – Pixel Sync

What is it?

- An Intel OpenGL|ES Extension:
  `GL_INTEL_fragment_shader_ordering`
- Allows synchronisation to unordered memory accesses from within a shader
- Add a single builtin to your shader at the point of synchronization

```
beginFragmentShaderOrderingINTEL();
```

Why do you want it?

- Fragments mapping to the same pixel using unordered memory accesses can cause data races
- Fragments can be shaded out-of-order

What can you do with it?

- Order independent transparency
- Programmable blending
- Adaptive volumetric shadow maps
- Etc

Adaptive Volumetric Shadow Maps (AVSM)

No AVSM

With AVSM

Codemasters Grid 2 in-game screenshots

http://software.intel.com/en-us/articles/pixel-synchronization-solving-old-graphics-problems-with-new-data-structures

OpenGL ES 3.1 – More Information

- More demos can be seen at the Intel Booth (#1016) in the South Hall.
- You can hear more about OpenGL ES 3.1 and its use in real games by visiting further Intel talks entitled:
  - “SSX: Bringing a PS3 game to Android”
    – Thursday 10-11AM
  - “Adding High-end Graphical Effects to GT Racing 2 on Android x86”
    – Thursday 2:30-3:30
  - “Rendering in Codemasters’ GRID2 and beyond: Achieving the ultimate graphics on both PC and tablet”
    – Thursday 4-5PM

intel®

Ready for More? Look Inside™.

Keep in touch with us at GDC and beyond:

- Game Developer Conference
  Visit our Intel® booth #1016 in Moscone South
- Intel University Games Showcase
  Marriott Marquis Salon 7, Thursday 5:30pm
  RSVP at **bit.ly/intelgame**
- Intel Developer Forum, San Francisco
  September 9-11, 2014
  intel.com/idf14
- Intel Software Adrenaline
  @inteladrenaline
- Intel Developer Zone
  software.intel.com
  @intelsoftware

Up Next…

3:30 – 4:30

Multi-player, multi-touch game development: Developing games for the fastest growing segment in desktop!

Presented by:

Alex Guo - Symbio Games & Faisal Habib - Intel