Agenda

- **OpenGL 3.1 announcement and OpenGL 3 overview**
  - Barthold Lichtenbelt, NVIDIA

- **OpenGL 2 vs OpenGL 3**
  - Jeremy Sandmel, OpenGL-next TSG chair

- **Blizzard perspective**
  - Rob Barris, Blizzard

- **TransGaming perspective**
  - Gavriel State, TransGaming

- **gDEBugger demo**
  - Avi Shapira, Graphic Remedy
OpenGL 3

The train has left!
Structure

- Overview of OpenGL 3.0 and GLSL 1.30
- The new deprecation model
- OpenGL 3.1 and GLSL 1.40
- OpenGL and OpenCL
- Future plans
- OpenGL 3 IHV support statements
OpenGL 3 – Moving OpenGL forward

- Expose all available hardware features asap
- Keep innovating where it makes sense
- Increase ease of porting from DX9 and DX10 to OpenGL
- Introduce mechanism to remove features
- Introduce mechanism to provide market specific features
- Enable interoperability with compute (OpenCL)
- Become a true superset of OpenGL ES

This is done incrementally, as a series of point releases, schedule driven
OpenGL 3.0 and GLSL 1.30

- Support for latest generations of Programmable Hardware
  - Installed base > 100 Million units

- Announced at Siggraph 2008

- Drivers now shipping from AMD, NVIDIA and S3 Graphics
  - gDEBugger support also available

- Introduced a ton of new features

- No removal of any feature, fully backwards compatible

- Full interoperability with OpenCL
  - Access to compute

- Collaboration among hardware vendors and software vendors
  - Solving real needs

- Cross platform
  - Windows XP and Vista, Linux, Mac OS, ...
OpenGL 3.0 new features

• Forward-looking context
• Greater VBO flexibility
• FBO and related extensions
  - EXT_framebuffer_object, EXT_framebuffer_blit, EXT_framebuffer_multisample, EXT_packed_depthStencil
• Conditional rendering
• Transform feedback
• Floating point internal formats for textures and renderbuffers
• Half-float (16-bit) vertex and pixel data formats
• One and two-channel (R and RG) internal formats for textures and renderbuffers
• RGTC internal compressed texture formats, packed float and texture shared exponent
• sRGB framebuffer support
**GLSL 1.30 new features**

- **Native integer support**
  - bitwise operators, texture return values, uniforms, shader input/outputs

- **Expanded texturing support**
  - Size queries, offsets, explicit LOD and derivative control, texture arrays, integer support

- **Switch statements**

- **Several new built-in functions**
  - Hyperbolic trig functions
  - `trunc()`, `round()`, `roundEven()`, `isnan()`, `isinf()`, `modf()`
  - Integer related: `sign()`, `min/max()`, `abs()`, ….

- **Pre-processor token pasting (##)**

- **User-defined fragment outputs**

- **Non-perspective interpolation of varying variables**

- **gl_VertexID vertex shader input**

- **Follows the same deprecation model as the API**
OpenGL 3.0 based on:

- EXT_gpu_shader4
- NV_conditional_render
- ARB_color_buffer_float
- NV_depth_buffer_float
- ARB_texture_float
- EXT_packed_float
- EXT_texture_shared_exponent
- NV_half_float
- ARB_half_float_pixel
- EXT_framebuffer_object
- EXT_framebuffer_multisample
- EXT_framebuffer_blit
- EXT_texture_integer
- EXT_texture_array
- EXT_packed_depth_stencil
- EXT_draw_buffers2
- EXT_texture_compression_rgtc
- EXT_transform_feedback
- APPLE_vertex_array_object
- EXT_framebuffer_sRGB
- APPLE_flush_buffer_range
- ARB_texture_RG
# Extensions for OpenGL 3.0

<table>
<thead>
<tr>
<th>Feature</th>
<th>Extension for OpenGL 3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform extension support for managing OpenGL 3.0 contexts</td>
<td>`{WGL</td>
</tr>
<tr>
<td>Geometry shaders to modify vertices and/or generate new vertices and primitives</td>
<td><code>ARB_geometry_shader4</code></td>
</tr>
<tr>
<td>Large 1D table lookups for GLSL</td>
<td><code>ARB_texture_buffer_object</code></td>
</tr>
<tr>
<td>Instanced primitive rendering for OpenGL 3.0 capable hardware</td>
<td><code>ARB_draw_instanced</code></td>
</tr>
</tbody>
</table>
## Extensions for OpenGL 2.x

<table>
<thead>
<tr>
<th>Feature from OpenGL 3.0</th>
<th>Extension for OpenGL 2.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>All framebuffer object functionality</td>
<td>ARB_framebuffer_object</td>
</tr>
<tr>
<td>16-bit floating point vertex formats</td>
<td>ARB_half_float_vertex</td>
</tr>
<tr>
<td>sRGB color space rendering</td>
<td>ARB_framebuffer_sRGB</td>
</tr>
<tr>
<td>More efficient buffer mapping</td>
<td>ARB_map_buffer_range</td>
</tr>
<tr>
<td>1 and 2 component texture compression</td>
<td>ARB_texture_compression_rgtc</td>
</tr>
<tr>
<td>Efficient vertex array state management</td>
<td>ARB_vertex_array_object</td>
</tr>
<tr>
<td>1 and 2 component render-to-texture</td>
<td>ARB_texture_rg</td>
</tr>
<tr>
<td>Vertex array instancing for OpenGL 2.x capable hardware</td>
<td>ARB_instanced_arrays</td>
</tr>
</tbody>
</table>
The Deprecation Model
Removing features

- OpenGL has never removed features
  - Commitment to backwards compatibility is one of OpenGL’s strengths
  - After 15+ years, defining new features to work with old features becomes increasingly difficult
- OpenGL 3.0 did not remove any features
- OpenGL 3.0 did mark certain features as deprecated
  - Redundant, Legacy and obsolete features
  - Parts of OpenGL unlikely to be accelerated
- Future OpenGL revisions will remove these deprecated features
  - Guidance to developers to prepare for future revisions
  - Plan to remove these features sooner, rather than later.
Deprecated features

- Fixed-function vertex and fragment processing
- Color-index mode
- Display lists, and Selection and Feedback modes
- GLSL 1.10 and 1.20
- Begin/End based rendering
- Application-generated object names
- Quads and polygon primitives
- Polygon and Line Stipple
- Pixel transfer modes
- Bitmaps, DrawPixels, PixelZoom
- and quite a few others...
  - See Appendix E of OpenGL 3.0 specification for the list
Deprecation mechanism

• **Step 1 Core feature**
  - In core, fully supported. *Will* be in the next API version

• **Step 2 Core (Deprecated feature)**
  - In core, marked as deprecated
  - *May* be fully or partly removed in a later version
  - New features need not define interactions with deprecated ones

• **Step 3 ARB approved Extension**
  - *Removed* from core -> an ARB extension (no suffix)
  - Extension spec identifies the removed functionality
  - Vendors may support the extension if markets require it

• **Step 4 Removed from ARB extension list**
  - Could be an EXT or vendor extension, if vendor markets still require it (still no suffixes required)
Deprecation mechanism

- Features will be deprecated for at least one spec release (step 2) before being removed

- **Extension Path:** Vendor/EXT→ARB→Core
  - With possible API / functionality changes as we learn from experience

- **Deprecation Path:** Core→ARB→EXT/Vendor
  - No API or functionality changes
Feature Evolution Model - Deprecation

Core Specification

- New Functionality Before Adoption into Core
- Incoming Extensions that may be integrated into Core in future

- Old Functionality Removed from Core
- Outgoing Extensions that may be dropped completely in future
OpenGL 3.1 and GLSL 1.40

Released 3/24/09
Announcing OpenGL 3.1

• More Texturing
  - Texture Buffer Objects
  - SNORM Texture format support
  - Rectangle Textures

• Additional Buffer management
  - Copy data between buffers
  - Uniform buffer objects

• Better Vertex Processing
  - Primitive Restart (NV_primitive_restart)
  - Instancing (ARB_draw_instanced)

• Removal of features
  - Everything on the deprecated list in OpenGL 3.0

• ARB_compatibility extension
  - Optional. Encapsulates removed functionality

• New Programmability
  - GLSL 1.40
  - Uniform Buffer Objects
Announcing GLSL 1.40

• Uniform blocks to be backed by buffer objects
  - Major new feature

• Texture buffers

• gl_InstanceID for instance drawing

• Don't require writing to gl_Position

• Rectangular textures
# New Extensions for OpenGL 2.x

<table>
<thead>
<tr>
<th>Feature from OpenGL 3.0/3.1</th>
<th>Extension for OpenGL 2.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>All framebuffer object functionality</td>
<td>ARB_framebuffer_object</td>
</tr>
<tr>
<td>16-bit floating point vertex formats</td>
<td>ARB_half_float_vertex</td>
</tr>
<tr>
<td>sRGB color space rendering</td>
<td>ARB_framebuffer_sRGB</td>
</tr>
<tr>
<td>More efficient buffer mapping</td>
<td>ARB_map_buffer_range</td>
</tr>
<tr>
<td>1 and 2 component texture compression</td>
<td>ARB_texture_compression_rgtc</td>
</tr>
<tr>
<td>Efficient vertex array state management</td>
<td>ARB_vertex_array_object</td>
</tr>
<tr>
<td>1 and 2 component render-to-texture</td>
<td>ARB_texture_rg</td>
</tr>
<tr>
<td>Vertex array instancing for OpenGL 2.x capable hardware</td>
<td>ARB_instance_arrays</td>
</tr>
<tr>
<td>Store uniform values in buffer objects</td>
<td>ARB_uniform_buffer</td>
</tr>
<tr>
<td>Copy data between buffer objects</td>
<td>ARB_copy_buffer</td>
</tr>
</tbody>
</table>
OpenGL 3.1 based on

- ARB_copy_buffer
- NV_primitive_restart
- ARB_draw_instanced
- ARB_texture_buffer_object
- ARB_texture_rectangle
- ARB_uniform_buffer_object
Uniform Buffer Objects

- **Introduction of uniform blocks**
  - Group of uniforms declared in a shader

- **Storage for values in uniform blocks is provided by a buffer object**

- **Defines standard (portable) and optimized layouts**
  - Portable across OpenGL implementations
  - Portable across program objects and shader stages
  - Or fully optimized, non-portable

- **Uniform data is loaded with existing buffer object API**

- **A buffer object is bound to an element of an array of uniform block binding points**
  - This is context state

- **A (program, uniform block) pair is associated with an element in the same array**
Advantages

• Sharing of uniform data between program objects and program stages

• Rapid switching between sets of uniform data
  - Buffer objects stored on the server
  - Eliminate calling glGetUniformLocation* many times over

• Rapid updates of uniform data
  - Using the existing buffer object commands. BufferData(), MapBufferRange() etc.

• Can store arbitrarily complex structures of data
  - Not limited to arrays of uniforms anymore

• Standard layout of data in memory, even across OpenGL vendors
  - Determined by a set of packing rules. Inspection of GLSL source code conveys layout

• Can store large amounts of data
  - Storage provided by a buffer object
Uniform buffer object example

```cpp
#extension GL_ARB_uniform_buffer_object : enable

// Define a uniform block, using std140 layout
layout(std140) uniform colors0 {
    float DiffuseCool;
    float DiffuseWarm;
    vec3 SurfaceColor;
    vec3 WarmColor;
    vec3 CoolColor;
};

void main (void)
{
    vec3 kcool = min(CoolColor + DiffuseCool * SurfaceColor, 1.0);
    ...
    gl_FragColor = ...
}
```
Program initialization (1/2)

// There's only one uniform block, the 'colors0' uniform block.
uniformBlockIndex = glGetUniformLocation(prog_id, "colors0");

// associate the uniform block to binding point 0
glUniformBlockBinding(prog_id, uniformBlockIndex, 0);

// Get the uniform block's size
glGetActiveUniformBlockiv(prog_id, uniformBlockIndex,
    GL_UNIFORM_BLOCK_DATA_SIZE_ARB,
    &uniformBlockSize);
Program initialization (2/2)

//SurfaceColor might change, so we'll query its offset/size.
const char *name = "SurfaceColor";

//First, get the index for the uniform
glGetUniformLocation(prog_id, 1, &name, &index);

//Use the index to query offset and size
glGetActiveUniformsiv(prog_id, 1, &index,
   GL_UNIFORM_OFFSET_ARB, &offset);

glGetActiveUniformsiv(prog_id, 1, &index,
   GL_UNIFORM_SIZE_ARB, &singleSize);

//Because this is std140 layout, we know the answer already
assert(offset == 16 && singleSize == 12);
Buffer initialization

//Create UBO
glBindBuffer(GL_UNIFORM_BUFFER_ARB, buffer_id);

//We can useBufferData to upload our data to the shader,
//since we know it's in the std140 layout
glBufferData(GL_UNIFORM_BUFFER_ARB, uniformBlockSize, NULL,
             GL_DYNAMIC_DRAW);
foreach (object) {
    // Set state
    // Bind vertex buffers

    // Bind constants to UBO binding point 0
    glBindBufferBase(GL_UNIFORM_BUFFER_ARB, 0, buffer_id);

    if (surfacecolor has changed) {
        glBufferSubData(GL_UNIFORM_BUFFER_ARB, offset,
            singleSize, &newcolor);
    }
    Draw();
}
OpenGL 3 Modern Buffer-centric Processing Model

- Array Element Buffer
- Vertex Array Buffer Object (VAO)
- Transform Feedback Buffer
- Uniform Buffer Object (UBO)
- Texture Buffer Object (TexBO)
- Pixel Unpack Buffer
- Pixel Pack Buffer
- Framebuffer

- glBegin, glEnd, glDrawElements, etc.
- vertex data
- texel data
- parameter data
- fragment data
- pixel data
- pixel pipeline
- Texture

- Vertex Puller
- Vertex Shading
- Geometry Shading
- Fragment Shading
- Pixel Pipeline
- Framebuffer

© Copyright Khronos Group, 2009 - Page 31
OpenGL and Compute
OpenGL and OpenCL synergy

• Complimentary capabilities
  - OpenGL 3.x = state-of-the-art, cross-platform graphics
  - OpenCL 1.0 = state-of-the-art, cross-platform compute

• Computation & Graphics should work together
  - Most natural way to intuit compute results is with graphics
  - When Compute is done on a GPU, there’s no need to “copy” the data to see it visualized

=> Use OpenCL for compute!
OpenGL and OpenCL interop

• Interop – the ability to efficiently transfer buffers or textures between OpenGL and OpenCL
• Enables application to use the API that makes most sense for their problem domain
  - No square peg in a round hole gymnastics
• Works on single GPU and multi-GPU systems
Four Kinds of Shared Objects

**OpenGL**

- OpenGL buffer object
  - GLuint bufferobj

- OpenGL texture 2D object
  - GLenum target
  - GLuint texture
  - GLint mipmap

- OpenGL texture 3D object
  - GLenum target
  - GLuint texture
  - GLint mipmap

- OpenGL renderbuffer object
  - GLuint renderbuffer

**OpenCL**

- OpenCL buffer object
  - cl_mem

- OpenCL 2D image object
  - cl_mem

- OpenCL 3D image object
  - cl_mem

- Renderbuffer object
  - 2D image object
  - cl_mem

Methods:

- clCreateFromGLBuffer
- clCreateFromGLTexture2D
- clCreateFromGLTexture3D
- clCreateFromGLRenderbuffer
What we said at Siggraph 2008

- Schedule driven
- ARB extensions are candidates for folding into a future core
  - ARB_draw_instanced
  - ARB_geometry_shader
  - ARB_texture_buffer_object
- Backing uniform variables with buffer objects
- `#include` mechanism for GLSL
- Attribute index offsets
- Remove deprecated features
- Profiles
- Object model improvements
- Other functionality you need?
Future versions

- ARB just started discussion on the next version - release likely within a year
- Close look at what remains to be done to increase ease of DX portability
- ARB extensions: Geometry shaders and copy buffer
- Finish making GLSL a true superset of ES
- Using program objects without linking
- Direct State Access
- Sampler objects - Splitting a texture object into image and sampler object
- Support for loading shader binaries
- Fences
- User specified UBO packing
- Explicit MSAA control
- Cube map arrays, MRT blending, Tessellation, Programmable blending
OpenGL 3.1 Specification Download

http://www.opengl.org/registry

Three new specs approved and available today

1) OpenGL 3.1 specification
2) OpenGL 3.1 + ARB_compatibility compatibility extension
3) GLSL 1.40 specification
OpenGL 3.1 IHV Statements
AMD and OpenGL 3.0 / OpenGL 3.1

• AMD already ships OpenGL 3.0 today
  - Full context
  - Forward compatible context
  - Support for Radeon and FirePro products

• AMD will add support for OpenGL 3.1 in the next few months

• AMD will support for ARB_compatibility extension which enables existing application to more easily use the latest features

• Contact AMD for details: pierre.boudier@amd.com
Intel on OpenGL 3.1

• “Intel is excited about OpenGL 3.1, the continuing evolution of OpenGL, and our future product support of OpenGL 3.x”
NVIDIA on OpenGL 3.0 / 3.1

• Have been shipping OpenGL 3.0 drivers since Siggraph 2008

• Announcing *immediate* availability of OpenGL 3.1 beta drivers
  - On both Windows and Linux

• OpenGL 3.1 drivers DO support the ARB_compatibility extension

• Download and release notes at
Trivia Questions

• How good is your knowledge of OpenGL and GLSL?