Mark Kilgarrd

• Principal System Software Engineer, NVIDIA, Austin, Texas
  - Developed original OpenGL driver for 1st GeForce GPU
  - Specified many key OpenGL extensions
  - Works on Cg for portable programmable shading
  - NVIDIA Distinguished Inventor

• Before NVIDIA, worked at Silicon Graphics
  - Worked on X Window System integration for OpenGL
  - Developed popular OpenGL Utility Toolkit (GLUT)

• Wrote book on OpenGL and X, co-authored Cg Tutorial
Outline

• OpenGL 3.0 specification overview
  - What does OpenGL 3.0 add?

• Deprecation and the OpenGL roadmap
  - OpenGL going forward

• Synergy between OpenCL and OpenGL
  - Sharing graphics resources with compute
OpenGL 3.0 Status

• **Announced at SIGGRAPH August 2008**
  - Jointly by AMD, Apple, Intel, NVIDIA, S3, Blizzard, TransGaming, and others
  - NVIDIA provided beta implementation the same week

• **Provides rough feature parity with Direct3D 10**
  - 97% of OpenGL 3.0 integrates proven, shipping extensions

• **First 3.0 implementation available now**
  - NVIDIA providing production OpenGL 3.0 driver this week
  - Estimated 100 million OpenGL 3.0-capable GPUs shipped
OpenGL Version Evolution

• Now OpenGL is part of Khronos
  - Previously OpenGL’s evolution was governed by the OpenGL Architectural Review Board (ARB)
  - Now officially a Khronos working group

• How OpenGL version updates happen
  - OpenGL participants proposing extensions
  - Successful extensions are polished and incorporated into core
  - OpenGL 3.0 is great example of this process
    - Roughly 20 extensions folded into “core”
    - Just 3 of those previously unimplemented
Areas of Functionality Improvement

• **Programmability**
  - Shader Model 4.0 features
  - OpenGL Shading Language (GLSL) 1.30

• **Texturing**
  - New texture representations and formats

• **Framebuffer operations**
  - Framebuffer objects
  - New formats
  - New blit, clear, blend, and masking operations

• **Buffer management**
  - Non-blocking and fine-grain update of buffer object data stores

• **Vertex processing**
  - Vertex array configuration objects
  - Conditional rendering for occlusion culling
  - New half-precision vertex attribute formats

• **Pixel processing**
  - New half-precision external pixel formats
OpenGL 3.0 Programmability

• Shader Model 4.0 additions
  - True signed & unsigned integer values
  - True integer operators: ^, &, |, <<, >>, %, ~
  - Texture additions
    - Texture arrays
    - Base texture size queries
    - Texel offsets to fetches
    - Explicit LOD and derivative control
  - Integer samplers
  - Interpolation modifiers: centroid, noperspective, and flat
  - Vertex array element number: gl_VertexID

• Open GL Shading Language (GLSL) improvements
  - ## concatenation in pre-processor for macros
  - switch/case/default statements
OpenGL 3.0 Texturing Functionality

• **Texture representation**
  - Texture arrays: indexed access to a set of 1D or 2D texture images

• **Texture formats**
  - Floating-point texture formats
    - Single-precision (32-bit, IEEE s23e8)
    - Half-precision (16-bit, s10e5)
  - Red & red/green texture formats
    - Intended as FBO framebuffer formats too
  - Compressed red & red/green texture formats
  - Shared exponent texture formats
  - Packed floating-point texture formats
Texture Arrays

• Conventional texture
  - One logical pre-filtered image

• Texture array
  - An array of mipmap sets, a plurality of pre-filtered images
  - No filtering between mipmap sets in a texture array
  - All mipmap sets in array share same format/border & base dimensions
  - Both 1D and 2D texture arrays
  - Require shaders, no fixed-function support

• Texture image specification
  - Use glTexImage3D, glTexSubImage3D, etc. to load 2D texture arrays
    - No new OpenGL commands for texture arrays
  - 3rd dimension specifies integer array index
    - No halving in 3rd dimension for mipmaps
    - So 64×128×17 reduces to 32×64×17 all the way to 1×1×17
Texture Arrays Example

- Multiple skins packed in texture array
  - **Motivation:** binding to one multi-skin texture array avoids texture bind per object
Compact Floating-point Textures

• Shared exponent & packed float representations are ideal of High Dynamic Range (HDR) applications
Compact Floating-point Texture Formats

• **Packed float format**
  - No sign bit, independent exponents

  ![Diagram of packed float format]

  - bit 31: Exponent
  - bit 0: Mantissa

• **Shared exponent format**
  - No sign bit, shared exponent, no implied leading 1

  ![Diagram of shared exponent format]

  - bit 31: Shared exponent
  - bit 0: Mantissa
1- and 2-component Block Compression Scheme

• Basic 1-component block compression format
  - Borrowed from alpha compression scheme of S3TC 5

2 min/max values + 16 bits

64 bits total per block

16 pixels x 8-bit/componet = 128 bits decoded
so effectively 2:1 compression

© Copyright Khronos Group, 2008 - Page 13
Framebuffer Operations

• **Framebuffer objects**
  - Standardized framebuffer objects (FBOs) for rendering to textures and renderbuffers
    - Render-to-texture
    - Multisample renderbuffers for FBOs

• **Framebuffer operations**
  - Copies from one FBO to another, including multisample data
  - Per-color attachment color clears, blending, and write masking

• **Framebuffer formats**
  - Floating-point color buffers
  - Floating-point depth buffers
  - Rendering into framebuffer format with 3 small unsigned floating-point values packed in a 32-bit value
  - Rendering into sRGB color space framebuffers
Delicate Color Fidelity with sRGB

• **Problem:** Display devices have non-linear (sRGB) display gamut—delicate color shading looks wrong

Conventional rendering (uncorrected color)

Gamma correct (sRGB rendered)

Unnaturally deep facial shadows

Softer and more natural

NVIDIA’s Adriana GeForce 8 Launch Demo
What is sRGB?

- **Color space standard**
  - Intended for monitors, printers, and the Internet
  - Created cooperatively by HP and Microsoft
  - Non-linear, roughly gamma of 2.2
  - Intuitively “encodes more dark values”

- **OpenGL 2.1 already added sRGB texture formats**
  - Texture fetch converts sRGB to linear RGB, then filters
  - Result takes more than 8-bit fixed-point to represent in shader

- **3.0 adds complementary sRGB framebuffer support**
  - “sRGB correct blending” converts framebuffer sRGB to linear, blend with linear color from shader, then convert back to sRGB
  - Works with FrameBuffer Objects (FBOs)
Vertex Processing

• **Vertex array configuration**
  - Objects to manage vertex array configuration client state
  - Half-precision floating-point vertex array formats

• **Vertex output streaming**
  - Stream transformed vertex results into buffer object data stores

• **Occlusion culling**
  - Skip rendering based on occlusion query result
Miscellaneous

• **Pixel Processing**
  - Half-precision floating-point pixel external formats

• **Buffer Management**
  - Non-blocking and fine-grain update of buffer object data stores
Beyond OpenGL 3.0

**OpenGL 3.0**
- 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_depthStencil
- EXT_draw_buffers2
- EXT_texture_compression_rgtc
- EXT_transform_feedback
- APPLE_vertex_array_object
- EXT_framebuffer_sRGB
- APPLE_flush_buffer_range (modified)

**In GeForce 8, 9, & 2xx Series**
*but not yet core*
- EXT_geometry_shader4 (now ARB)
- EXT_bindable_uniform
- NV_gpu_program4
- NV_parameter_buffer_object
- EXT_texture_compression_latc
- EXT_texture_buffer_object (now ARB)
- NV_framebuffer_multisample_coverage
- NV_transform_feedback2
- NV_explicit_multisample
- NV_multisample_coverage
- EXT_draw_instanced (now ARB)
- EXT_direct_state_access
- EXT_vertex_array_bgra
- EXT_texture_swizzle

Plenty of proven OpenGL extensions for OpenGL Working Group to draw upon for OpenGL 3.1
ARB Extensions to OpenGL 3.0

• OpenGL 3.0 standard provides new ARB extensions
  - Extensions go beyond OpenGL 3.0
    - Standardized at same time as OpenGL 3.0
    - Support features in hardware today
  
• Specifically
  - ARBGeometryShader4—provides per-primitive programmable processing
  - ARBDrawInstanced—gives shader access to instance ID
  - ARBTextureBufferObject—allows buffer object to be sampled as a huge 1D unfiltered texture

• Shipping today
  - NVIDIA driver provides all three
Transform Feedback for Terrain Generation by Recursive Subdivision

- Geometry shaders + transform feedback

1. Render quads (use 4-vertex line adjacency primitive) from vertex buffer object
2. Fetch height field
3. Stream subdivided positions and normals to transform feedback “other” buffer object
4. Use buffer object as vertex buffer
5. Repeat, ping-pong buffer objects

Computation and data all stays on the GPU!
Skin Deformation

• Capture & re-use geometric deformations

Transform feedback allows the GPU to calculate the interactive, deforming elastic skin of the frog
Silhouette Edge Rendering

- Uses geometry shader

Complete mesh

Useful for non-photorealistic rendering

*Looks like human sketching*

Silhouette edges
More Geometry Shader Examples

- Shimmering point sprites
- Generate fins for lines
- Generate shells for fur rendering
- Shimmering point sprites
Improved Interpolation Techniques

- Using geometry shader functionality

Quadratic normal interpolation

Quad to triangle pair
Quadrilateral interpolation

True quadrilateral rendering with mean value coordinate interpolation
“Fair” Quadrilateral Interpolation

- `glBegin(GL_QUADS);`
- `glColor3fv(red);
glVertex3fv(lowerLeft);`
- `glColor3fv(green);
glVertex3fv(lowerRight);`
- `glColor3fv(red);
glVertex3fv(upperRight);`
- `glColor3fv(blue);
glVertex3fv(upperLeft);`
- `glEnd();`

- Geometry shader actually operates on 4-vertex GL_LINE_ADJACENCY primitives instead of quads

Wrong, backslash triangle split
Wrong, slash triangle split
Better: Mean value coordinates
Many OpenGL 3.0 extensions have corresponding ARB extensions for OpenGL 2.1 implementations to advertise
- Helps get 3.0 functionality out sooner, rather than later

New ARB extensions for 3.0 functionality
- ARB_framebuffer_object—framebuffer objects (FBOs) for render-to-texture
- ARB_texture_rg—red and red/green texture formats
- ARB_map_buffer_region—non-blocking and fine-grain update of buffer object data stores
- ARB_instanced_arrays—instance ID available to shaders
- ARB_half_float_vertex—half-precision floating-point vertex array formats
- ARB_framebuffer_sRGB—rendering into sRGB color space framebuffers
- ARB_texture_compression_rgtx—compressed red and red/green texture formats
- ARB_depth_buffer_float—floating-point depth buffers
- ARB_vertex_array_object—objects to manage vertex array configuration client state
Deprecation

• OpenGL 3.0 specification lists “deprecated” functionality
  - Don’t panic—everything in OpenGL 3.0 still works

• Plan is a future OpenGL 3.x revision might remove this functionality
  - Ask vendors about their policy for actually removing OpenGL functionality
Synergy between OpenGL and OpenCL

• Complimentary capabilities
  - OpenGL 3.0 = 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

• Appendix B of OpenCL specification
  - Details with sharing objects between OpenGL and OpenCL
    - Called “GL” and “CL” from here on…
Four Kinds of Shared Objects

**OpenGL**

- **OpenGL buffer object**
  - GLuint `bufferobj`

- **OpenGL texture 2D object**
  - GLenum `target`
  - GLuint `texture`
  - GLint `miplevel`

- **OpenGL texture 3D object**
  - GLenum `target`
  - GLuint `texture`
  - GLint

- **OpenGL renderbuffer object**
  - GLuint `renderbuffer`

**OpenCL**

- **OpenCL buffer object**
  - cl_mem

- **OpenCL 2D image object**
  - cl_mem

- **OpenCL 3D image object**
  - cl_mem

- **2D image object**
  - cl_mem

---

© Copyright Khronos Group, 2008 - Page 30
OpenGL / OpenCL Sharing

• Requirements for GL object sharing with CL
  - CL context must be created with an OpenGL context
  - Each platform-specific API will provide its appropriate way to create an OpenGL-compatible CL context
    - For WGL (Windows), CGL (OS X), GLX (X11/Linux), EGL (OpenGL ES), etc.

• Creating cl_mem for GL Objects does two things
  1. Ensures CL has a reference to the GL objects
  2. Provides cl_mem handle to acquire GL object for CL’s use

• clRetainMemObject & clReleaseMemObject can create counted references to cl_mem objects
Acquiring GL Objects for Compute Access

• Still must “enqueue acquire” GL objects for compute kernels to use them
  - Otherwise reading or writing GL objects with CL is undefined
  - Enqueue acquire and release provide sequential consistency with GL command processing

• Enqueue commands for GL objects
  - clEnqueueAcquireGLObjects
    - Takes list of cl_mem objects for GL objects & list of cl_events that must complete before acquire
    - Returns a cl_event for this acquire operation
  - clEnqueueReleaseGLObjects
    - Takes list of cl_mem objects for GL objects & list of cl_events that must complete before release
    - Returns a cl_event for this release operation
Buffers in the context of the OpenGL pipeline
Summary

• **OpenGL 3.0 is now here**
  - Big step forward in core functionality
    - New Programmability, Texturing, Framebuffer Operations, Vertex Processing, Pixel Processing, and Buffer Management functionality
    - Standardizes “ARB” extensions for geometry and texture buffers

• **OpenGL 3.0 anticipates deprecation**

• **OpenGL & OpenCL integrate compute & graphics**
  - Read and write to GL textures, buffers, and renderbuffers from OpenCL
Plug for “Modern OpenGL” Course

• Want to know more about OpenGL
  - How it got to where it is?
  - Insights into where it is going?

• Attend “Modern OpenGL: Its Design and Evolution”
  - Co-presenters: Kurt Akeley (SGI co-founder & co-author of the original OpenGL specification) & myself
  - Finishes with conversation about OpenGL facilitated by Dr. Marc Levoy (Stanford University)

• When & where
  - Saturday, 13 December, 13:45-17:30
  - Room 311/312
  - Last course of the conference!