Introduction to the OpenGL Shading Language

Randi Rost
Director of Developer Relations, 3Dlabs

Why use graphics programmability?
- Graphics hardware has changed radically
- Fixed functionality is too limiting
- Never-before-seen effects are possible
- Now, APPLICATIONS can take control over the processing that occurs on the graphics hardware

Think of yourself as a prisoner (to fixed functionality) that has been set free! Anything is possible!

Use Programmability For…
- Rendering increasingly more realistic materials
  - Metals
  - Stone
  - Wood
  - Paints
  - Etc.
- Rendering natural phenomena
  - Fire
  - Clouds
  - Smoke
  - Water
  - Etc.
Use Programmability For...

- Procedural texturing
  - Stripes
  - Polka dots
  - Bricks
  - Stars
  - Etc.

Use Programmability For...

- Non-photorealistic (NPR) effects
  - Painterly
  - Hatch/stroke/pen and ink
  - Technical illustration
  - Cartoon
  - Etc.

Use Programmability For...

- Animation
  - On/off based on threshold
  - Translation/rotation/scaling of any shader parameter
  - Key-frame interpolation
  - Particle systems
  - Etc.

Use Programmability For...

- Doing new things with texture maps (or doing old things more easily)
  - Polynomial texture maps
  - BRDFs
  - Bump maps
  - Gloss maps
  - Irradiance maps
  - Environment maps
  - Etc.
Use Programmability For...

- More realistic lighting effects
  - Global illumination
  - Spherical harmonics lighting
  - Image based lighting
  
  ScoutWalker model courtesy of Christophe Desse

Use Programmability For...

- More realistic shadow effects
  - Ambient occlusion
  - Shadow mapping
  - Volume shadows
  
  Orc model courtesy of Christophe Desse

Use Programmability For...

- More realistic surface effects
  - Refraction
  - Diffraction
  - Anisotropic reflection
  - BRDFs
  
  MascotAngst model courtesy of Christophe Desse

Use Programmability For...

- Imaging operations
  - Color correction/transformation
  - Noise removal
  - Sharpening
  - Complex blending
Use Programmability For…

- Better antialiasing
  - Stochastic sampling
  - Adaptive prefiltering
  - Analytic integration
  - Frequency clamping
  - Etc.

Use Programmability for…

- Highly parallel computation
  - Visualization of complex functions
  - Numerical simulation
  - Etc.

Shading Languages

- Key to making visual programmability accessible to ISVs
  - Need to get out of the assembler dark ages
- Graphics vendors busy building compiler expertise
  - Soon will be as important to performance as drivers are today
- Same industry API dynamics as fixed function APIs
  - Just the programming level has changed

Market Creation With API Standards

3Dlabs – Initiated OpenGL ES development and is chairing Khronos and the OpenGL ES Working Group

The foundation of programmable, cross-platform, professional graphics

The standard for embedded 3D graphics – launched at SIGGRAPH 2003

3Dlabs – chaired Khronos Graphics Working Group

OpenGL 2.0 was launched at Siggraph 2004

The standard for dynamic media authoring – launched at SIGGRAPH 2001

3Dlabs – initiated OpenGL 2.0 development and is Permanent ARB Member

HLSL
Direct3D
Microsoft

Cg
Glide

OpenGL 2.0
Proprietary
Open Standard
Visual Processing Revolution

- Visual processing is changing the face of hardware, APIs and tools
- Innovation is required at all three levels

OpenGL Shading Language is part of the OpenGL standard as of OpenGL 2.0 – Sept. 2004. 3Dlabs released compiler front-end as open source

GLSL and 3Dlabs

- 3Dlabs shipped industry's first OpenGL Shading Language drivers
  - Running on complete family of Wildcat VP boards
- 3Dlabs has placed compiler front-end into open source
  - To catalyze industry adoption
  - To encourage cross-vendor consistency to error-checking
- 3Dlabs has placed various development tools into open source
  - GLSLdemo, GLSLparsertest, GLSLvalidate, ShaderGen
- Already in use by leading-edge Toolkit Providers
  - Lightwork Design
- Already in use by leading-edge ISVs
  - Solidworks
  - Pandromeda
  - Many others that have not yet announced products

GLSL Background and Current Status

- OpenGL 2.0 is here!
  - Specification approved in September 2004
  - OpenGL Shading Language is part of core
  - API for shading language is part of core
  - Spec is available at OpenGL.org
  - Still backwards compatible with previous versions
GLSL Book

- First edition released by Addison-Wesley in Feb. 2004
- Second edition due out early January 2006
- Contains more detailed information
  - Introduction and overview
  - Complete reference
  - Dozens of detailed examples
- Companion web site
  - http://3dshaders.com

Shading Language Differences

- GLSL compiles directly from high level source to machine code inside of OpenGL
- HLSL translates high level source to Direct3D source outside of DirectX

Vertex Processor Capabilities

- Lighting, material and geometry flexibility
- Vertex processor can do general processing, including things like:
  - Vertex transformation
  - Normal transformation, normalization and rescaling
  - Lighting
  - Color material application
  - Clamping of colors
  - Texture coordinate generation
  - Texture coordinate transformation
**Vertex Processor Capabilities**

- **The vertex shader does NOT replace:**
  - Perspective divide and viewport mapping
  - Frustum and user clipping
  - Backface culling
  - Primitive assembly
  - Two sided lighting selection
  - Polygon offset
  - Polygon mode

**Fragment Processor Capabilities**

- **Flexibility for texturing and per-pixel operations**
- **Fragment processor can do general processing, including things like:**
  - Operations on interpolated values
  - Texture access
  - Texture application
  - Fog
  - Color sum
  - Pixel zoom
  - Scale and bias
  - Color table lookup
  - Convolution
  - Color matrix

**Vertex Processor Overview**

- **Standard OpenGL attributes**
  - `gl_color`
  - `gl_normal`
  - etc.

- **Generic attributes**
  - `0, 1, 2, ...`

- **User-defined uniforms:**
  - `epsilon`, `myLightPos`, `surfColor`, etc.

- **Built-in uniforms:**
  - `gl_FogColor`, `gl_ModelViewMatrix`, etc.

**Fragment Processor Capabilities**

- **The fragment shader does NOT replace:**
  - Shading model
  - Coverage
  - Pixel ownership test
  - Scissor
  - Stipple
  - Alpha test
  - Depth test
  - Stencil test
  - Alpha blending
  - Logical ops
  - Dithering
  - Plane masking
### Fragment Processor Overview

- **User-defined uniforms:** 
  - \( \varepsilon \), \( \text{myLightPos} \), \( \text{surfColor} \), etc.
- **Built-in uniforms:** 
  - \( \text{gl_FogColor} \), \( \text{gl_ModelViewMatrix} \), etc.

### Vertex Processor Input

- **Vertex shader is executed once each time a vertex position is specified**
  - Via `glVertex` or `glDrawArrays` or other vertex array calls
- **Per-vertex input values are called attributes**
  - Change every vertex
  - Passed through normal OpenGL mechanisms (per-vertex API or vertex arrays)
- **More persistent input values are called uniforms**
  - Can come from OpenGL state or from the application
  - Constant across at least one primitive, typically constant for many primitives
  - Passed through new OpenGL API calls

### Vertex Processor Output

- **Vertex shader uses input values to compute output values**
- **Vertex shader must compute \( \text{gl\_Position} \)**
  - Mandatory, needed by the rasterizer
  - Can use built-in function \( \text{frtransform()} \) to get invariance with fixed functionality
- **Vertex shader may compute:**
  - \( \text{gl\_ClipVertex} \) (if user clipping is to be performed)
  - \( \text{gl\_PointSize} \) (if point parameters are to be used)

- **Other output values are called varying variables**
  - E.g., color, texture coordinates, arbitrary data
  - Will be interpolated in a perspective-correct fashion across the primitives
  - Defined by the vertex shader
  - Can be of type float, \( \text{vec2} \), \( \text{vec3} \), \( \text{vec4} \), \( \text{mat2} \), \( \text{mat3} \), \( \text{mat4} \), or arrays of these

- **Output of vertex processor feeds into OpenGL fixed functionality**
  - If a fragment shader is active, output of vertex shader must match input of fragment shader
  - If no fragment shader is active, output of vertex shader must match the needs of fixed functionality fragment processing
**Vertex Processor Definition**
- The vertex processor executes the vertex shader
- The vertex processor has knowledge of only the current vertex
- An implementation may have multiple vertex processors operating in parallel

**When the vertex processor is active, the following fixed functionality is disabled:**
- The modelview matrix is not applied to vertex coordinates
- The projection matrix is not applied to vertex coordinates
- The texture matrices are not applied to texture coordinates
- Normals are not transformed to eye coordinates
- Normals are not rescaled or normalized
- Normalization of GL_AUTO_NORMAL evaluated normals is not performed
- Texture coordinates are not generated automatically
- Per vertex lighting is not performed
- Color material computations are not performed
- Color index lighting is not performed
- Point size distance attenuation is not performed
- All of the above applies when setting the current raster position

**Intervening Fixed Functionality**
- Results from vertex processing undergo:
  - Color clamping or masking (for built-in varying variables that deal with color, but not user-defined varying variables)
  - Perspective division on clip coordinates
  - Viewport mapping
  - Depth range
  - Clipping, including user clipping
  - Front face determination
  - Flat-shading
  - Color, texture coordinate, fog, point-size and user-defined varying clipping
  - Final color processing

**Fragment Processor Input**
- Output of vertex shader is the input to the fragment shader
  - Compatibility is checked when linking occurs
  - Compatibility between the two is based on varying variables that are defined in both shaders and that match in type and name
- Fragment shader is executed for each fragment produced by rasterization
- For each fragment, the fragment shader has access to the interpolated value for each varying variable
  - Color, normal, texture coordinates, arbitrary values
Fragment Processor Input

- Fragment shader may access:
  - `gl_FrontFacing` – contains direction (front or back) of primitive that produced the fragment
  - `gl_FragCoord` – contains computed window relative coordinates x, y, z, 1/w
- Uniform variables are also available
  - OpenGL state or supplied by the application, same as for vertex shader
- If no vertex shader is active, fragment shader get the results of OpenGL fixed functionality

Fragment Processor Definition

- The fragment processor executes the fragment shader
- The fragment processor has knowledge of only the current fragment
- An implementation may have multiple fragment processors operating in parallel
- When the fragment processor is active, the following fixed functionality is disabled:
  - The texture environments and texture functions are not applied
  - Texture application is not applied
  - Color sum is not applied
  - Fog is not applied

Fragment Processor Output

- Output of the fragment processor goes on to the fixed function fragment operations and frame buffer operations using built-in variables
  - `gl_FragColor` – computed R, G, B, A for the fragment
  - `gl_FragDepth` – computed depth value for the fragment
  - `gl_FragData[n]` – arbitrary data per fragment, stored in multiple render targets
  - Values are destined for writing into the frame buffer if back end tests all pass
- Clamping or format conversion to the target buffer is done automatically outside of the fragment shader

Fragment Processor Definition

- The fragment processor does not affect the behavior of the following:
  - Texture image specification
  - Alternate texture image specification
  - Compressed texture image specification
  - Texture parameters behave as specified even when a texture is accessed from within a fragment shader
  - Texture state and proxy state
  - Texture object specification
  - Texture comparison modes