# Do you wish you could hear the audio and read the transcription of this session?

Then come to JavaOne℠ Online where this session is available in a multimedia tool with full audio and transcription synced with the slide presentation.

JavaOne Online offers much more than just multimedia sessions. Here are just a few benefits:

- 2003 and 2002 Multimedia JavaOne conference sessions
- Monthly webinars with industry luminaries
- Exclusive web-only multimedia sessions on Java technology
- Birds-of-a-Feather sessions online
- Classified Ads: Find a new job, view upcoming events, buy or sell cool stuff and much more!
- Feature articles on industry leaders, Q&A with speakers, etc.

# Advanced OpenGL® for the Java™ Platform

**Kenneth Russell**
Sun Microsystems, Inc.

**Christopher Kline**
Irrational Games

**Gerard Ziemski**
Apple Computer

# Presentation Goal

Demonstrate the latest 3D graphics techniques available through the OpenGL® API and the Java™ programming language

# Speakers' Qualifications

- **Kenneth Russell** works on the Java HotSpot™ Virtual Machine at Sun Microsystems and has nine years of 3D graphics experience
- **Christopher Kline** is a lead programmer for Irrational Games, makers of System Shock II and Freedom Force, and has over six years of 3D graphics experience
- **Gerard Ziemski** works on the graphics libraries for the Java™ platform at Apple Computer and has over four years of 3D graphics experience

# Real-time Graphics in Transition

We are finally leaving behind the stone age of real-time 3D graphics programming.

# Agenda

- What's new in real-time graphics?
- OpenGL interfaces for the Java™ platform
- Demos and Tutorials
  - Fixed-function pipeline
  - Programmable pipeline
  - Shadows
  - High-level shading languages

# Real-time 3D Graphics Timeline

- Early 1990s: SGI and E&S pioneer dedicated (and expensive!) graphics hardware
- Late 1990s: VGA controllers make way for more powerful, mass-market GPUs
- GPU Generation 1 (< 1998): basic rasterization and texturing
- GPU Generation 2 (1999–2000): hardware T&L, better blending and texturing options
- GPU Generation 3 (2001): programmable (but limited) vertex and pixel shaders
- GPU Generation 4 (2002): floating point framebuffers, lengthy vertex and pixel shaders

# Trend: Increasing Programmability

- Trend from *configurability* to *programmability*:
  - Fixed blending modes: limited configurability
  - Register combiners: more configurable
  - Vertex and fragment programs: finally, assembly-level control of transformation and shading
  - Now high-level languages and compilers
  - Soon: a unified data model; hardware support for loops and conditionals

# What Does This Mean for Programmers?

- In the future, graphics programming will focus less on data management and configuration
- Innovation will be in the area of sophisticated visual effects algorithms
- Pixar and ILM-caliber effects are within the reach of the desktop
- Latest features are now available to the Java™ platform

# OpenGL Interfaces for the Java™ Platform

- Several bindings available
  - “ OpenGL, for Java™ Technology” (abbreviated “gl4java”)
  - LWJGL (Lightweight Java™ Game Library)
  - Magician
  - Jungle
- Brief discussion of each

# OpenGL Interfaces for the Java™ Platform

- " OpenGL, for Java™ Technology" (abbr. "gl4java")
  - One of the oldest and most popular bindings
  - Runs on nearly every platform
  - Integrates with AWT and Swing
  - Supports, but not designed for, New I/O
  - Open source
  - Supports only up to OpenGL 1.3, but exposes vendor extensions
  - API is complex
  - Difficult to maintain and enhance

# OpenGL Interfaces for the Java™ Platform

- LWJGL (Lightweight Java™ Game Library)
  - Supports latest features (OpenGL 1.4 with vendor extensions)
    - Innovative organization of extensions
  - Designed for New I/O
  - Additional support for audio (OpenAL) and game input devices
  - Supports full-screen rendering
  - Open source
  - Does not support AWT and Swing integration
  - Exposes pointers as longs
    - Destroys type safety

# OpenGL Interfaces for the Java™ Platform

- Magician
  - Clean API
  - Integrated with AWT and Swing
  - Innovative composable pipeline (e.g., DebugGL)
  - Did not support New I/O
  - Defunct (no longer being developed or shipped)
  - Was never open source

# OpenGL Interfaces for the Java™ Platform

- Jungle
  - New OpenGL interface for the Java™ platform
  - Supports OpenGL 1.4 and vendor extensions
  - Integrates with AWT and Swing
  - Designed for New I/O
  - Clean, minimalist API
  - Supports composable pipeline (e.g., DebugGL)
  - Open source
  - Written almost entirely in Java™ programming language
    - AWT Native Interface, WGL and GLX bound into Java™ programming language using GlueGen

# OpenGL Interfaces for the Java™ Platform

- GlueGen
  - Parses C header files using ANTLR
  - Generates intermediate representation expressing primitive types, function prototypes, structs, unions and function pointers
  - Autogenerates Java™ programming language and JNI code
  - Powerful enough to bind AWT Native Interface back into Java programming language
    - Enabled Jungle to be written in Java programming language instead of C
  - Open source; part of Jungle package

# OpenGL Interfaces for the Java™ Platform

- Jungle
  - Working in collaboration with Java™ Gaming Initiative
  - Has been adopted as JGI's OpenGL binding
  - Now named "Jogl"
  - Open source (modified BSD license)
  - Available from http://jogl.dev.java.net/

# Demos and Techniques

- Illustrations of latest techniques
  - Demonstrations borrowed from several sources
  - Ported where necessary to Java™ programming language
  - Utilizing Jungle OpenGL interface

# Overview of Demos and Tutorials

- Fixed-function pipeline
- Programmable pipeline
- High-level languages
- Larger demos

# Fixed-function Pipeline

- Basically a “black box” that generates images according to a standard set of algorithms
- You supply the input data
  - Vertex attributes, connectivity, textures
- You configure the algorithm parameters
  - Transform matrices, blending modes, light colors, data formats, etc.
- No programmability, only configurability

# Fixed-function Pipeline

- Why use the fixed-function pipeline?
  - Easy to understand
  - Best availability
  - Only option on legacy hardware
- Core OpenGL 1.3 and earlier
- Still powerful!

# Example: The Virtual Fishtank

- Developed by Nearlife, Inc.
  http://www.nearlife.com/
- Developed in 1998; now at the Boston Museum of Science, with a second installation in the St. Louis Science Center
- Museum exhibit designed to teach children about emergent self-organizing behavior within decentralized rule-based systems

# Example: The Virtual Fishtank

- Distributed simulation running 15 networked machines, rendered on 13 large projection screens, simulating a 24,000 gallon aquarium
- Fish migrate from server to server as they swim from screen to screen
- Written entirely in Java™ programming language; Originally used Java™ 3D software, later ported to custom OpenGL-based renderer

# Example: The Virtual Fishtank

**DEMO**

# Programmable Pipeline

- What is the programmable pipeline?
  - Allows you to replace “black box” components of FF-pipeline with your own implementation
- What does it replace?
  - Vertex shaders
    - Transformation and lighting of vertices
  - Fragment shaders
    - Texturing, fog, color sum

# Programmable Pipeline

- *Program* the rendering process instead of *configuring* it
- Wow, I can do anything I want to?
  - Yes, but if you choose to replace *anything*, you have to implement *everything*
  - Great power at the cost of great responsibility

# Programmable Pipeline

- Why use the programmable pipeline?
  - Can be more efficient
    - Higher-quality results with less detailed geometry
    - Don’t need multi-pass to accumulate intermediate results
    - Cut corners or customize to your needs
  - Do things that aren’t possible with FF pipeline
    - Non-standard lighting models
  - Humans perceive detail by observing how light interacts with a surface
    - More control over light means more impressive graphics

# Vertex Shaders

- Calculate all attributes of one particular vertex
  - No access to other vertices!
  - No hand holding: you must code all calculations yourself
  - Vertex position, normal, colors, texture coords, fog depth
- Additional input registers for arbitrary constants:
  - Transform matrices, light information, time, etc.
  - Parameters to your VS “function”

# Vertex Shaders

- Output is used as input to fragment shader
  - Interpolated
- Assembly language syntax
  - Can be compiled from high-level language
    - Nvidia Cg
    - OpenGL GLSL
    - Microsoft DX9 HLSL

# Vertex Shaders

- Example: 3-Component Normalize

```
#
# Assume R1 = (nx,ny,nz)
#
# Calculate:
# R0.xyz = normalize(R1)
# R0.w   = 1/sqrt(nx*nx + ny*ny + nz*nz)
#
DP3 R0.w, R1, R1;
RSQ R0.w, R0.w;
MUL R0.xyz, R1, R0.w;
```

# Vertex Shaders

- Can arbitrarily swizzle components of registers
  - No additional cost
  - Good for vector math operations
  - Save instructions, render faster
  - Impress your friends

# Vertex Shaders

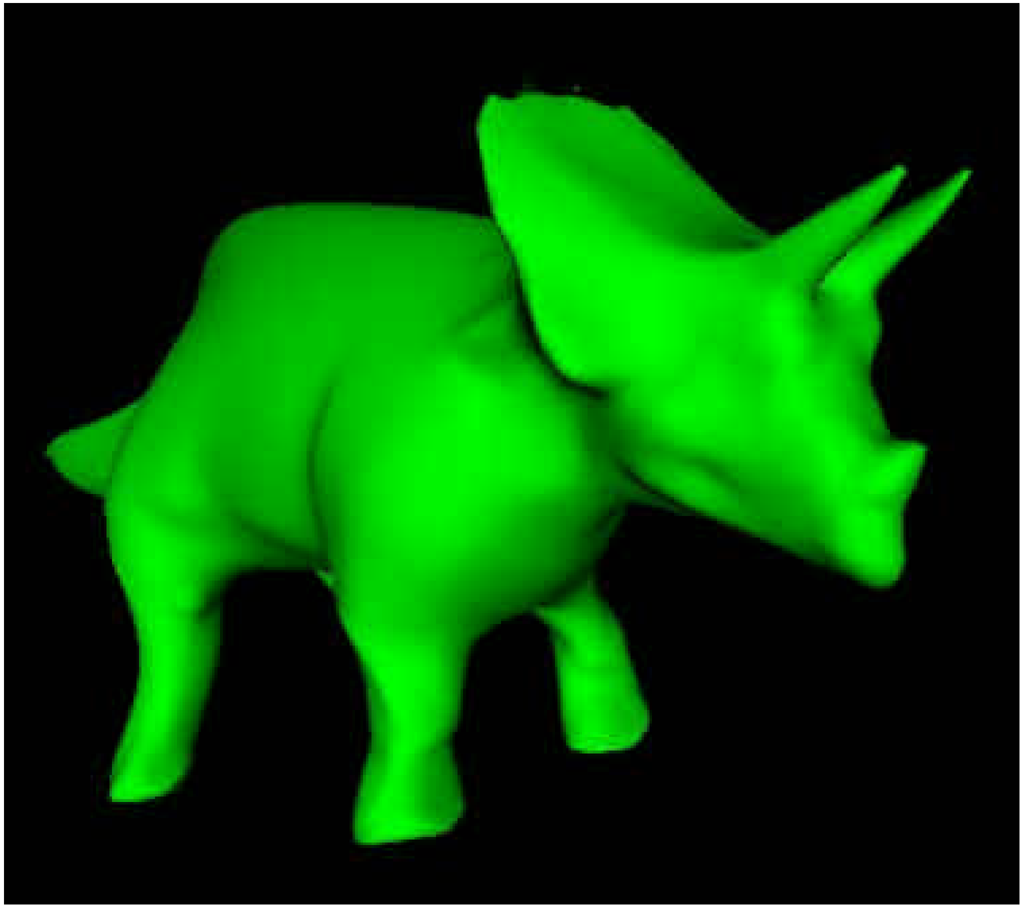
- Example: 3-Component Cross Product

```
# Calculate R2 = R0.cross(R1)
#
# Cross product | i     j     k    | into R2.
#               | R0.x  R0.y  R0.z |
#               | R1.x  R1.y  R1.z |
#
#   R2.x = (R0.y*R1.z – R0.z*R1.y)
#   R2.y = (R0.z*R1.x – R0.x*R1.z)
#   R2.z = (R0.x*R1.y – R0.y*R1.x)
#
MUL R2,  R0.yzxw, R1.zxyw;      # Swizzle
MAD R2, -R1.yzxw, R0.zxyw, R2;  # Swizzle again
```

# Vertex Shaders: vtxprog_warp

## DEMO

Nvidia vtxprog_warp



# Vertex Shaders: vtxprog_warp

- Several per-vertex distortion effects
  - Wave, fisheye, spherize, ripple, twist
- Static effects compute vertex's distance from center point and scale according to function
- Dynamic effects based mostly on sine waves
  - Computed ***on the GPU*** via Taylor series approximation to sin(x)
- All effects' programs contain small snippet of code implementing diffuse lighting

# Vertex Shaders: vtxprog_refract

## DEMO

Nvidia vtxprog_refract

# Vertex Shaders: vtxprog_refract

- Implements chromatic aberration through multipass rendering
  - Fresnel term determines fraction of light transmitted as opposed to reflected
  - Renders three times with fresnel terms modified for differing wavelengths of red, green and blue light
    - Causes slightly different distortion for each

# Vertex Shaders: vtxprog_refract

- Vertex program computes approximation to reflection/refraction based on vertex's relative position and normal to eye
  - Approximation: only takes into account forward-facing triangles, not the depth of the surface
- Resulting rays are transformed into texture coordinates into surrounding cube map
- Provides blended reflection and refraction effects even in single pass and without fragment shaders

# Vertex Shaders: ProceduralTexturePhysics

## DEMO

Nvidia ProceduralTexturePhysics

# Vertex Shaders: ProceduralTexturePhysics

- Performs physical simulation of water entirely on graphics card using texture maps as units of computation
- Every pixel affects its nearest neighbors
- Vertex program transforms vertices and produces initial sets of texture coordinates
- Offset texture coordinates used in conjunction with register combiners to perform approximation to integration of water forces
- Blur (convolution) smooths result

# Fragment Shaders

- Calculate final visual appearance of one fragment
  - Operates on a rasterized pixel (a *fragment*)
    - Sometimes called *pixel shaders*
- Input:
  - Interpolated color, tex/fog coords, window position
    - Note: no world-space position, no normal!
  - Additional registers for arbitrary constants
- Output:
  - Color and depth of pixel

# Fragment Shaders

- Similar to vertex shaders
  - No access to other pixels
  - Must roll your own shading code
  - Assembly syntax
- But different from vertex shaders
  - Texture sampler assembly instructions
  - No knowledge of geometry

# Fragment Shaders

Example:
Modulate diffuse color by texture color

```
# sample texture color and load into R0
TEX R0, fragment.texcoord[0], texture[0], 2D;
# load diffuse color into R1
MOV R1, fragment.color.secondary;
# final color = diffuse * texture
MAD result.color,fragment.color.primary, R0, R1;
```

# Fragment Shaders

## Why No Standalone FS Demo?

- FS of limited utility without VS support
  - Remember, no knowledge of geometry
  - Can do tricks in normalized device coord space
    - Position-based fades and masks
    - Depth-based color (e.g., fake heat-vision)
  - To do really interesting things, need geometric information
    - Use VS to smuggle geometry data into FS

# Combining Vertex and Fragment Shaders

- Work together in unison
  - VS writes geometry data into attributes that PS can access (secondary color, tex/fog coords)
  - PS reads this data to get geometry info
- Share the computational burden
  - VS calculates low-frequency (per vertex) data
  - PS calculates high-frequency (per pixel) data
- Good way to optimize performance

# VS + FS Example: Phong Lighting

- Ubiquitous model in computer graphics
  - If it looks like plastic, it's probably Phong
- Simple idea
  - Surface should look shiniest where incident light is reflecting directly into your face
  - Less shiny as angle between reflected light and observer direction increases
  - Easy and efficient to implement
- OpenGL FF-pipeline vertex lighting is Phong variant

# VS + FS Example: Phong Lighting

## DEMO:

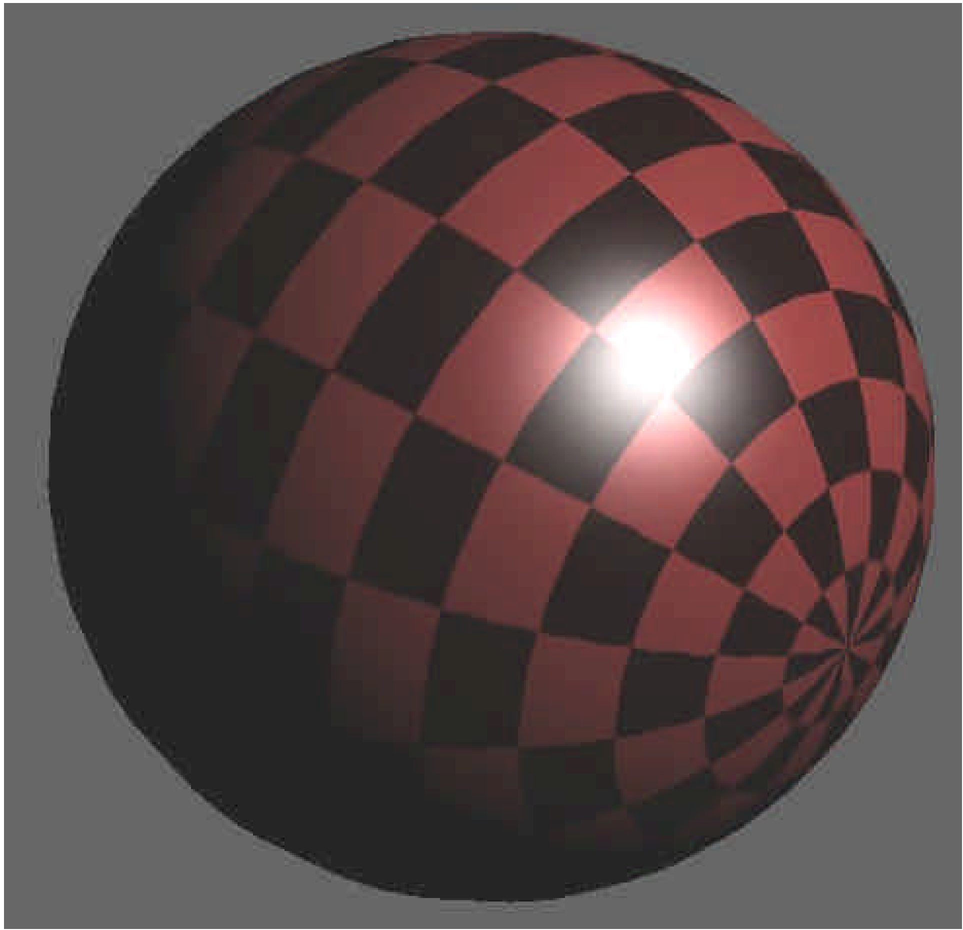
### Cg Toolkit OpenGL Phong Lighting

- Vertex shader
  - Calculates vertex position and normal in eye space, stores in texture coordinate sets 0 and 1
- Fragment shader
  - Reads texture coordinates to retrieve (interpolated) eye-space position and normal of fragment
  - Reads light position passed in by program as “arbitrary constant”
  - Compares fragment position and normal with light position to calculate specular highlight intensity

# VS + FS Example: Phong Lighting

## DEMO

NVidia Cg Toolkit OpenGL
Phong Lighting



# Shadows

- Why do we need shadows?
  1) Humans use shadows to infer spatial relationships
     - Relative positions of objects
     - Locations of light sources
     - Shape of an object
  2) Scene looks natural
  3) Scene is easier to understand

# Shadows

- Why do we need shadows?

*4) Technically speaking, shadows are “groovy”*

# Shadows

- Two basic categories
  - Render-to-texture
    - Image-space technique
  - Volumetric
    - Geometric technique

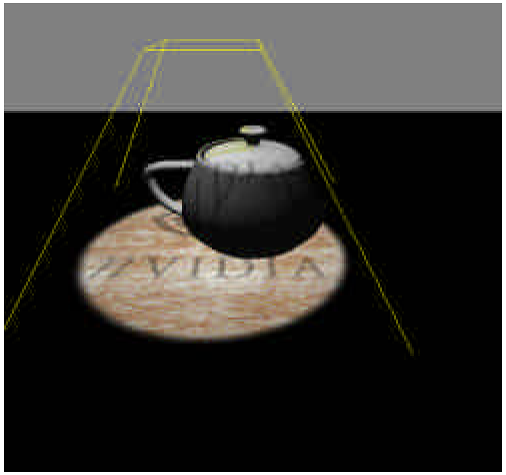
# Render-to-texture Shadows

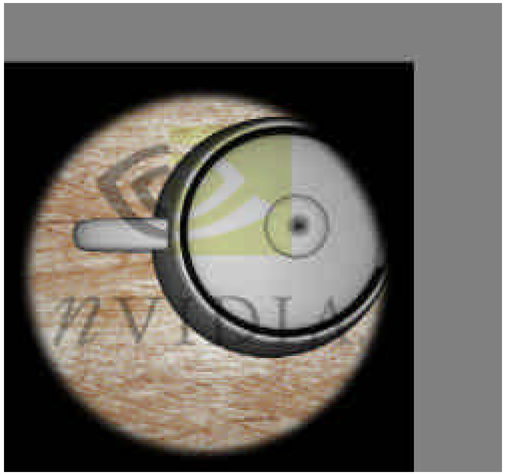
- Render the scene from the light’s perspective
- Store depth of rendered scene as texture
- Render scene from the viewer’s perspective
- Render the depth texture onto the scene
  - Careful setup of texture transform and texture-coord generation
    - Object’s position maps to correct u-v texture coords in depth texture
    - Object’s r texture coord maps to distance from the object to the light source
  - If r-value is greater than texture value, pixel is in shadow

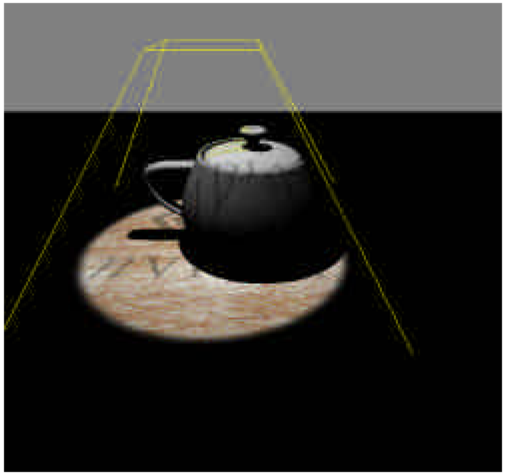
# Render-to-texture Shadows

## DEMO

### NVidia Hardware Shadow Mapping







# Render-to-texture Shadows

Advantages

- Performance independent of geometric complexity
- No additional cost for animated geometry
- Can take into account alpha-masked geometry (example: a chain-link fence)

# Render-to-texture Shadows

Disadvantages:

- Dependent on texture resolution (aliasing)
  - Not good for long projections
- Need special tricks to get self-shadowing to work well
- Older hardware may not support render-to-texture in hardware
  - Fall back to slow framebuffer->texture copy

# Volumetric Shadows

Basic idea: Use geometry to calculate volume of space that is in shadow

- Calculate silhouette edge of object, from light's perspective
- Extrude the silhouette away from the light
- Objects inside this volume are in shadow from the light

# Volumetric Shadows

Uses stencil buffer for per-pixel in/out test

- Render scene, ambient light only
  - Sets the depth buffer
- Render shadow volumes w/ stencil enabled
  - Render front/back faces separately
  - If pixel passes depth test, adjust stencil value
    - Many adjustment heuristics (z-pass, z-fail)
- If stencil value is 0 afterwards, pixel is not in shadow

# Volumetric Shadows

## DEMO:

NVidia Infinite Shadow Volumes

# Volumetric Shadows

Advantages

- Self-shadowing “just works”
- No aliasing problems
  - Crisp shadows, even at infinite projection distances
  - Good for wide-open spaces

# Volumetric Shadows

Disadvantages:

- Performance depends on scene
  - Expensive for complex objects, many lights, or many shadow receivers
    - N lights = N+1 render passes per shadowed object
  - Slow for non-static geometry/non-static lights
    - Silhouettes must be recalculated each frame
- Incorrect shadows cast from alpha-masked geometry
  - Purely geometric technique
- Many subtleties to make it work correctly for all intersections of light, viewer, and shadow volume

# Shading Languages

- What is a shading language?
  - High-level language for programming vertex and fragment operations
  - Compiles down to low-level hardware representation (assembly)
  - Analogous to the relationship between C and Assembly

# Shading Languages

- Why use a shading language?
  - Create and re-use code libraries
    - Borrow snippets from others
  - Can be platform-independent
    - Compile at run-time for target hardware
    - Cross-platform development, easier porting
  - Compiler is probably better at optimizing than you are

# Shading Languages

- Why use a shading language?

**It’s just plain easier!**

# Shading Languages

- Many shading languages available today
  - NVidia Cg
  - Microsoft DirectX9 HLSL
  - OpenGL GLSL (soon)
- Derive from lots of prior art
  - Pixar RenderMan
  - Stanford Real-Time Shading Language
  - UNC PixelFlow

# Shading Languages: Cg

- What is Cg?
  - Product of NVidia corporation
  - C-like language
  - Hardware-independent
  - Compiles to various forms of assembly for vertex and pixel shaders

# Shading Languages: Cg

- Cg example: Phong lighting vertex shader

```
void main(float4 Pobject       : POSITION,
       float3 Nobject       : NORMAL,
       float2 TexUV         : TEXCOORD0,
       float3 diffuse       : TEXCOORD1,
       float3 specular      : TEXCOORD2,
       uniform float4x4 ModelViewProj,
       uniform float4x4 ModelView,
       uniform float4x4 ModelViewIT,

       out float4 HPosition     : POSITION,
       out float3 Peye          : TEXCOORD0,
       out float3 Neye          : TEXCOORD1,
       out float2 uv            : TEXCOORD2,
       out float3 Kd            : COLOR0,
       out float3 Ks            : COLOR1) {
 // compute homogeneous position of vertex for rasterizer
 HPosition = mul(ModelViewProj, Pobject);
```

(Cont.)

# Shading Languages: Cg

- Cg example: Phong lighting vertex shader

```
    // transform position and normal from model-space
    // to view-space
    Peye = mul(ModelView, Pobject).xyz;
    Neye = mul(ModelViewIT, float4(Nobject, 0)).xyz;

    // pass uv, Kd, and Ks through unchanged;
    // if they are varying per-vertex, however,
    // they'll be interpolated before being
    // passed to the fragment program.
    uv = TexUV;
    Kd = diffuse;
    Ks = specular;
}
```

# Shading Languages: Cg

- Cg Phong vertex shader, compiled:

```
!!ARBvp1.0
# ARB_vertex_program generated by NVIDIA Cg compiler
TEMP R0;
ATTRIB v26 = vertex.texcoord[2];
ATTRIB v25 = vertex.texcoord[1];
ATTRIB v24 = vertex.texcoord[0];
ATTRIB v18 = vertex.normal;
ATTRIB v16 = vertex.position;
PARAM c8[4] = { program.local[8..11] };
PARAM c4[4] = { program.local[4..7] };
PARAM c0[4] = { program.local[0..3] };
        MOV result.texcoord[2].xy, v24;
        MOV result.color.front.primary.xyz, v25;
        MOV result.color.front.secondary.xyz, v26;
        DP4 result.position.x, c0[0], v16;
        DP4 result.position.y, c0[1], v16;
        DP4 result.position.z, c0[2], v16;
        DP4 result.position.w, c0[3], v16;
```

(Cont.)

# Shading Languages: Cg

- Cg Phong vertex shader, compiled:

```
        DP4 result.texcoord[0].x, c4[0], v16;
        DP4 result.texcoord[0].y, c4[1], v16;
        DP4 result.texcoord[0].z, c4[2], v16;
        MOV R0.xyz, v18.xyzz;
        MOV R0.w, c12.x;
        DP4 result.texcoord[1].x, c8[0], R0;
        DP4 result.texcoord[1].y, c8[1], R0;
        DP4 result.texcoord[1].z, c8[2], R0;
END
```

# Shading Languages: Cg

- Why use Cg?
  - OpenGL GLSL not yet available
  - Cg compiles for many different backends
    - OpenGL
      - Both ARB and vendor-specific shader extensions
    - DirectX 8 and 9
  - Cg comes with the Cg Runtime Library
    - Easy to load, compile, and set up your vertex and fragment shaders