Cg 2.0

Mark Kilgard

What is Cg?

- Cg is a GPU shading language
  - C/C++ like language
  - Write vertex-, geometry-, and fragment-processing kernels that execute on massively parallel GPUs
  - Productivity through a high-level language
  - Supports NVIDIA, ATI, and Intel graphics
  - Supports OpenGL and Direct3D
- Cg also run-time system for shaders
  - Run-time makes best use of available GPU
  - Use OpenGL or Direct3D
  - Effect system for meta-shading

Why Cg?

- Cg = cross-platform shaders
  - Same Cg shader source compiles to:
    - Multi-vendor OpenGL extensions
      - ARB_vertex_program & ARB_fragment_program
    - NVIDIA-specific OpenGL extensions
      - GeForce 8's NV_gpu_program4
    - DirectX 9 assembly shaders
      - Shader Models 1.x, 2.x, and 3.x
    - OpenGL Shading Language (GLSL) cross-compile!
    - DirectX 9 HLSL cross-compile!
    - Sony's support for Cg for PlayStation 3
  - Multi-OS: Vista, XP, 2000, MacOS X, Linux, Solaris
- Sophisticated CgFX effects system
  - Compatible with Microsoft's FX in DirectX 9
- Abstraction no other GPU standard shading language has
  - Interfaces and un-sized arrays

Why Cg 2.0?

- Keeps current with DirectX 10-class functionality
  - New profiles for GeForce 8
  - Geometry shaders
  - Bind-able uniform buffers, a.k.a. constant buffers
  - Texture arrays
- New HLSL 9 cross-compile profiles
- Performance improvements
- Compiler improvements
- New examples show of Cg 2.0 and GeForce 8
- Greatly expanded documentation

Primary Cg 2.0 Features

- 100% compatibility with Cg 1.5
- New GeForce 8 (G80) OpenGL profiles
  - gp4vp (*vertex*), gp4gp (*geometry*), gp4fp (*fragment*)
  - Per-primitive (geometry) programs
    - Vertex attribute arrays
    - Primitive types: point, line, line adjacency, triangle, triangle adjacency
  - Bind-able buffers for uniform parameters
  - Texture arrays & texture buffer objects
  - Interpolation modifiers (flat, centroid, non-perspective)
  - True 32-bit integer variables and operators
- New HLSL9 profiles
  - hlslv (*vertex*), hlslf (*fragment*)
  - Run-time or compile-time translation of Cg to optimized HLSL

Other Cg 2.0 Features

- New compiler back-end for DX10-class unified, scalar GPU architecture
- Improved FX compatibility for CgFX
- More efficient parameter update API via buffers
- Updated documentation
  - New Cg language specification
  - New CgFX standard state manual pages
  - New Cg standard library manual pages
  - New Cg runtime API manual pages
- Updated examples
  - Geometry shaders, uniform buffers, interpolation modifiers, etc.

Cg 2.0 Support for GeForce 8 OpenGL

- New G80 profiles
  - gp4vp: NV_gpu_program4 vertex program
  - gp4gp: NV_gpu_program4 geometry program
  - gp4fp: NV_gpu_program4 fragment program

New programmable domain

- New Cg language support
  - int variables really are integers now
  - Temporaries dynamically index-able now
  - All G80 texturing operations exposed
    - New samplers, new standard library functions
  - New semantics
    - Instance ID, vertex ID, bind-able buffers, viewport ID, layer
  - Geometry shader support
    - Attrib arrays, emitVertex & restartStrip library routines
    - Profile modifiers for primitive input and output type

Geometry Pass Through Example

Length of attribute arrays depends on the input primitive mode, 3 for TRIANGLE

Semantic ties uniform parameter to a buffer, compiler assigns offset

```
uniform float4 flatColor : BUFFER[0] ;

TRIANGLE void passthru(AttribArray<float4> position : POSITION,
                       AttribArray<float4> texCoord : TEXCOORD0)
{
  flatAttrib(flatColor:COLOR);
  for (int i=0; i<position.length; i++) {
   emitVertex(position[i], texCoord[i],float3(1,0,0):TEXCOORD1);
  }
}
```

Makes sure flat attributes are associated with the proper provoking vertex convention

Bundles a vertex based on parameter values and semantics

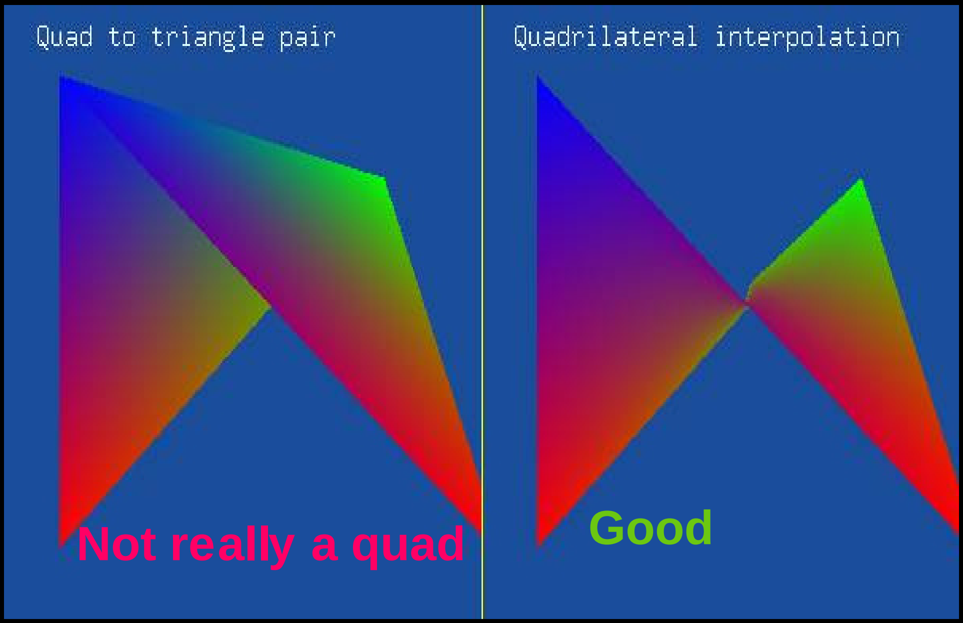
Hermite Curve Tessellation

```
void LINE hermiteCurve(AttribArray<float4> position : POSITION,
                       AttribArray<float4> tangent  : TEXCOORD0,

               uniform float steps)  // # line segments to approx. curve
{
  emitVertex(position[0]);
  for (int t=1; t<steps; t++) {
    float s        = t / steps;
    float ssquared = s*s;
    float scubed   = s*s*s;

    float h1 =  2*scubed - 3*ssquared  + 1; // calculate basis function 1
    float h2 = -2*scubed + 3*ssquared;      // calculate basis function 2
    float h3 =    scubed - 2*ssquared + s;  // calculate basis function 3
    float h4 =    scubed -   ssquared;      // calculate basis function 4

    float4 p : POSITION = h1*position[0] +  // multiply and sum all functions
                          h2*position[1] +  // together to build interpolated
                          h3*tangent[0]  +  // point along the curve
                          h4*tangent[1];
    emitVertex(p);
  }
  emitVertex(position[1]);
}
```

(Geometry shaders not really ideal for tessellation.)

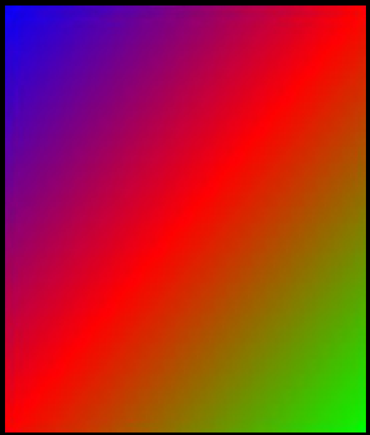
True Quadrilateral Rasterization & Interpolation (1)

- The world is not all triangles
- Quads exist in real-world meshes
- Fully continuous interpolation over quads not linear
  - Mean value coordinate interpolation [Floater, Hormann & Tarini]
- Quads can “bow tie”

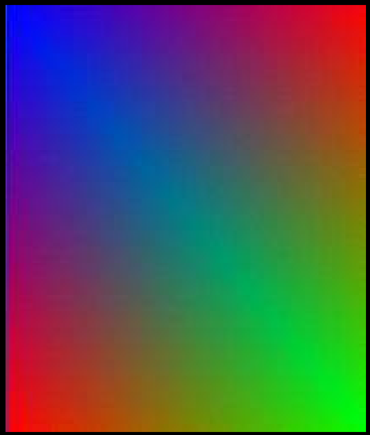


True Quadrilateral Rasterization & Interpolation (2)

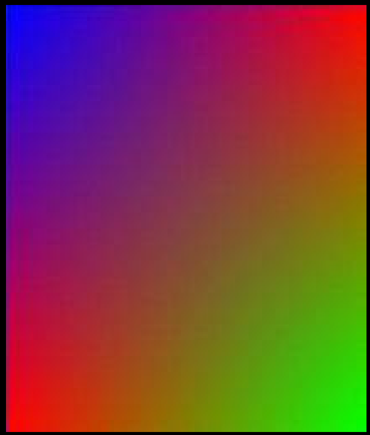
- **Conventional hardware:** **How you split quad to triangles can greatly alter interpolation**
  - **Both ways to split introduce interpolation discontinuities**



“Slash” split



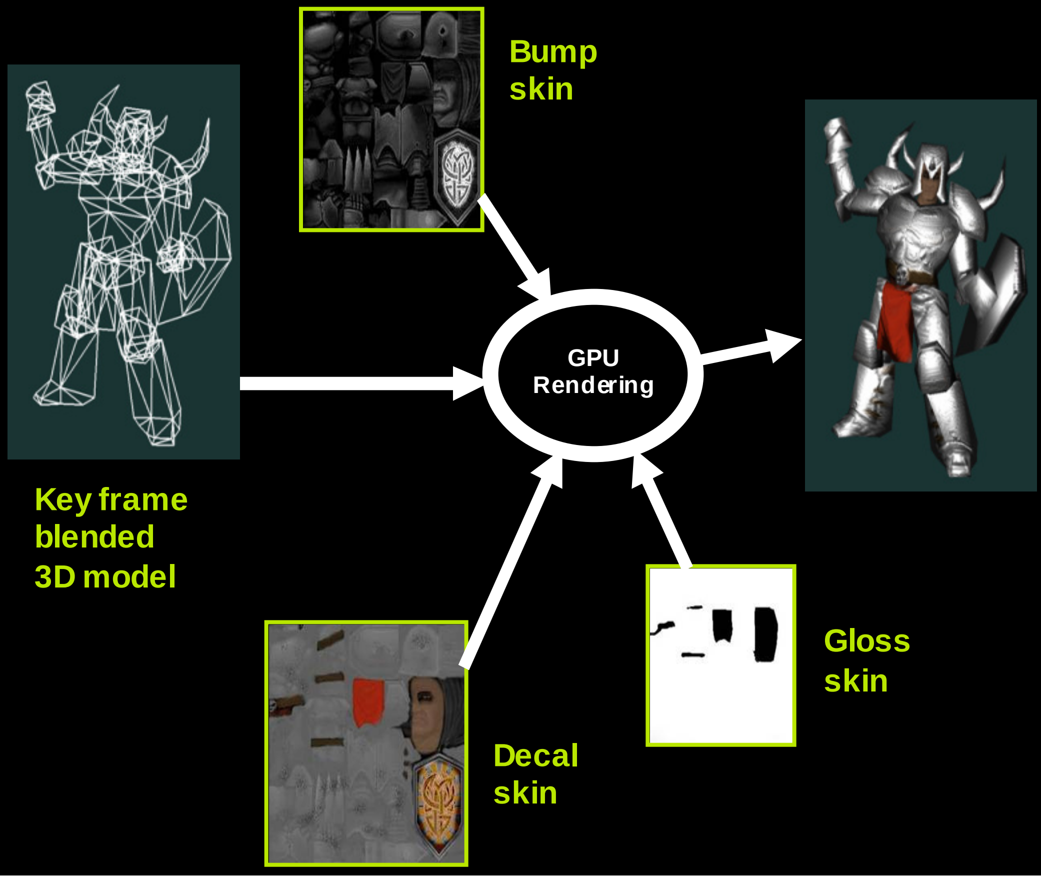
“Backslash” split



Mean value coordinate interpolation via Cg geometry and fragment shaders

Bump Map Skinned Characters (1)

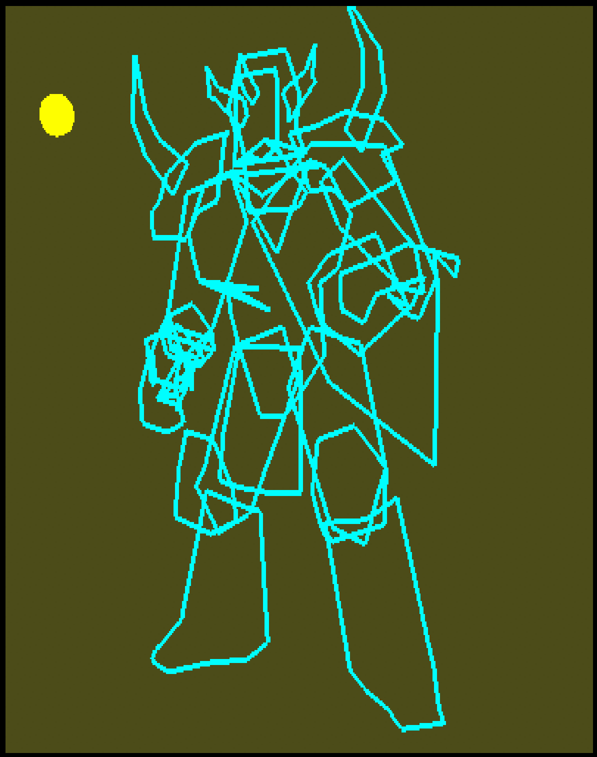
- **Pre-geometry shader approach:** CPU computes texture-space basis per skinned triangle to transform lighting vectors properly
  - **Problem:** Meant skinning was done on the CPU, not GPU



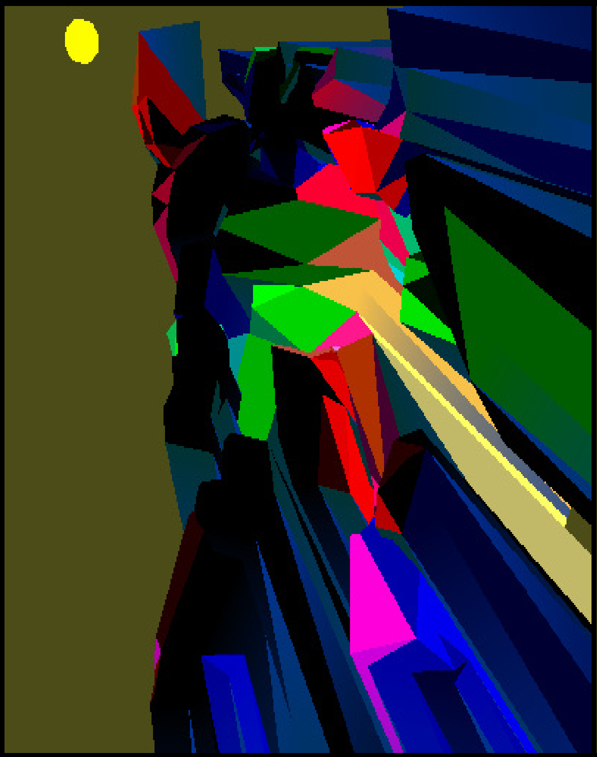
Bump Map Skinned Characters (2)

- Cg **vertex** shader does skinning
- Cg **geometry** shader computes transform from object- to texture-space based on each triangle
- Cg **geometry** shader then transforms skinned object-space vectors (light and view) to texture space
- Cg **fragment** shader computes bump mapping using texture-space normal map
- *Computations all stay on the GPU*

Next, Geometry Shader-Generated Shadows with Stenciled Shadow Volumes



Cg geometry shader computes possible silhouette edges from triangle adjacency
(visualization)

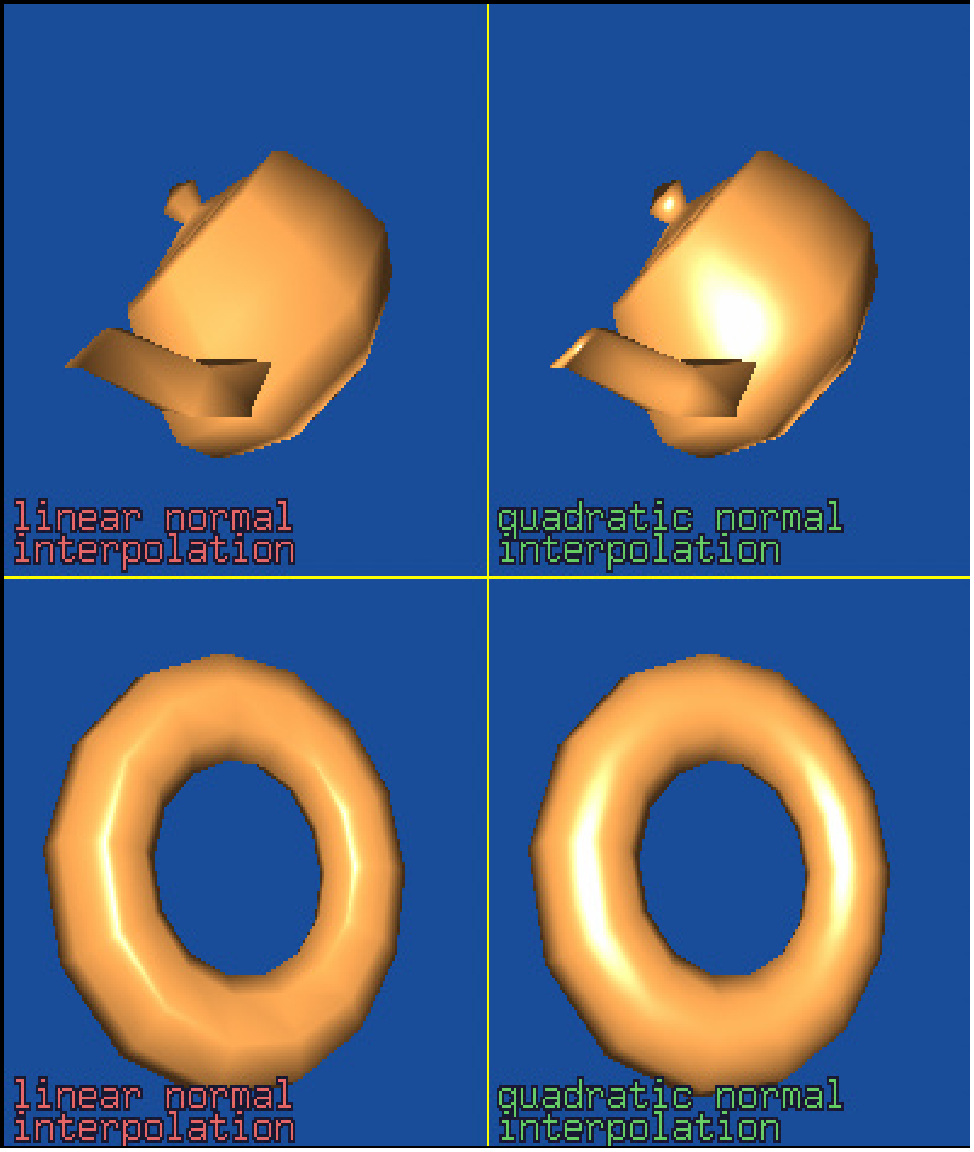


Extrude shadow volumes based on triangle facing-ness and silhouette edges
(visualization)

Add bump mapped lighting based on stenciled shadow volume rendering
(complete effect)

Geometry Shader Setup for Quadratic Normal Interpolation

- Linear interpolation of surface normals don't match real surfaces (except for flat surfaces)
- Quadratic normal interpolation [van Overveld & Wyvill]
  - Better Phong lighting, even at low tessellation
- Approach
  - Geometry shader sets up linear parameters
  - Fragment shader combines them for quadratic result
- Best exploits GPU's linear interpolation resources



Cg 2.0 Bind-able Buffer API

- Cg API modeled after OpenGL buffer object API
- cgCreateBuffer—creates bindable uniform buffer
  - CGbuffer cgBuffer = cgCreateBuffer(cgContext, sizeInBytes, NULL, CG_BUFFER_USAGE_xxx)
- cgSetBufferSubData—copies bytes into buffer
  - cgSetBufferSubData(cgBuffer, offset, sizeInBytes, data);
  - Also cgSetBufferData—redefines entire buffer with new size
  - Also cgMapBuffer & cgUnmapBuffer—gives pointer to buffer data
- cgSetProgramBuffer—associates buffer object to program's buffer index
  - Cg program maps uniforms to buffers with BUFFER semantic:
    - uniform float4 someUniform[20] : BUFFER[5];
  - cgGetParameterBufferOffset & cgGetParameterIndex
  - cgSetProgramBuffer(cgProgram, cgGetParameterBufferIndex(cgParam, cgGetNamedParameter("someUniform")), cgBuffer);

Cg 2.0 API-specific Buffers

- cgCreateBuffer creates API-independent buffers
  - Cg runtime creates API-dependent buffers as needed
  - Cg runtime “fakes” bind-able buffers for pre-DirectX 10-class (pre-G80) profiles
  - Allows runtime to perform efficient parameter update into the API-dependent buffers
- cgGLCreateBuffer creates API-dependent buffers for OpenGL
  - Cg runtime creates OpenGL buffer
  - Cg runtime will provide GLuint handle to the buffer
  - All buffer interactions by Cg require immediate 3D API-dependent execution
- Expected usage
  - Use “cg” buffers for batching conventional uniforms more efficiently
  - Use “cgGL” buffers for transform feedback, pixel buffer object read-backs, etc. when GPU is writing data into buffers

Updated Documentation

- New *CgReferenceManual.pdf* includes
  - New Cg language specification
  - Updated run-time API documentation
  - Full Cg standard library
  - CgFX states documented
  - Command-line cgc compiler documentation
- Reference manual also available as
  - Unix-style man pages
  - Microsoft's indexed & search-able Compiled HTML *CgReferenceManual.chm*
  - Raw HTML pages
- Includes tutorial white papers on Cg and CgFX

Greatly Expanded Examples

- Examples from *The Cg Tutorial*
  - Twenty-two OpenGL-based examples with both C and Cg source code
    - Using OpenGL Utility Toolkit (GLUT)
  - Seven also available as Direct3D-based examples
    - Using miniDXUT
- Advanced examples
  - Vertex texturing for GeForce 6 and up
    - vertex_texture
  - Interfaces and un-sized arrays
    - cgfx_interfaces
  - Geometry shader examples for GeForce 8
    - Simple (gs_simple, gs_shrinky), texture-space bump mapping setup (gs_md2bump), shadow volume generation (gs_md2shadow, gs_md2shadowvol), quadrilateral rasterization (gs_interp_quad), quadratic normal interpolation (gs_quadnormal)
  - Buffer example for GeForce 8
    - buffer_lighting
  - Other GeForce 8 features
    - Texture arrays (cgfx_texture_array, texture_array), interpolation_modifiers
- Examples packaged with all operating systems