High-Level Shading Languages

Daniel Weiskopf

Institute of Visualization and Interactive Systems
University of Stuttgart

Why?

- Avoids programming, debugging, and maintenance of long assembly shaders
- Easy to read
- Easier to modify existing shaders
- Automatic code optimization
- Wide range of platforms
- Shaders often inspired RenderMan shading language

Assembly vs. High-Level Language

Assembly

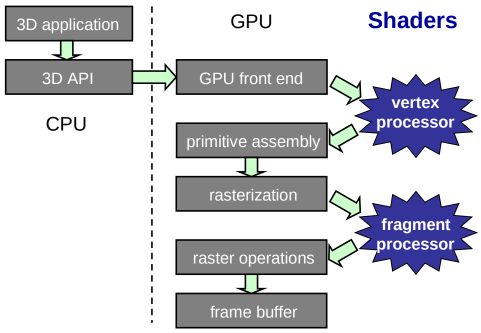
```
...
dp3 r0, r0, r1
max r1.x, c5.x, r0.x
pow r0.x, r1.x, c4.x
mul  r0, c3.x, r0.x
mov r1, c2
add  r1, c1, r1
mad r0, c0.x, r1, r0
...
```

High-level language

```
...
float4 cSpec = pow(max(0, dot(Nf, H)), phongExp).xxx;
float4 cPlastic = Cd * (cAmbi + cDiff) + Cs * cSpec;
...
```

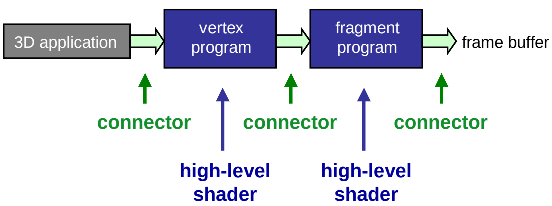
Blinn-Phong shader expressed in both assembly and high-level language

Graphics Pipeline



Data Flow through Pipeline

- Vertex shader program
- Fragment shader program
- Connectors



High-Level Shading Languages

- Cg
  - "C for Graphics"
  - By nVidia
- HLSL
  - "High-level shading language"
  - Part of DirectX 9 (Microsoft)
- OpenGL 2.0 Shading Language
  - Proposal by 3D Labs

Cg as an Example for a Shading Language

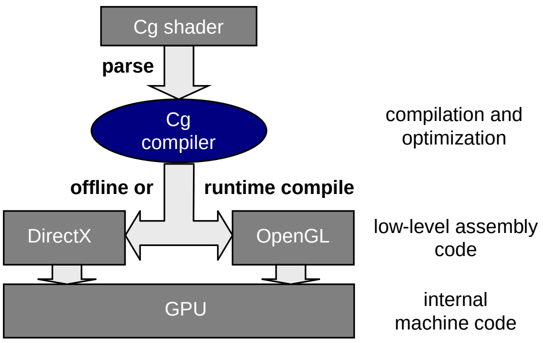
- Typical concepts for a high-level shading language
- Language is (almost) identical to DirectX HLSL
- Supported by
  - Tools
  - Documentation on the Web and a book
- Wide platform support:
  - DirectX and OpenGL
  - Windows, Linux, Mac OS X

General Features in Cg

- Syntax, operators, functions from C/C++
- Conditionals and flow control
- Backends according to hardware profiles
- Support for GPU-specific features (compare to low-level functionality):
  - Vector and matrix operations
  - Hardware data types for maximum performance
  - Access to GPU functions: mul, sqrt, dot, …
  - Mathematical functions for graphics, e.g. reflect
  - Profiles for particular hardware feature sets

Tutorial T7: Programming Graphics Hardware | High-Level Shading Languages, Daniel Weiskopf | VIS Group, University of Stuttgart

Workflow in Cg



Tutorial T7: Programming Graphics Hardware | High-Level Shading Languages, Daniel Weiskopf | VIS Group, University of Stuttgart

Hardware Profiles

- Platform-independent basic Cg programs
- Adaptation to latest hardware features and APIs
- Large number of profiles:
  - DX8/9 Vertex Shader (vs1.1) and Pixel Shader (ps1.1, ps1.2, ps1.3)
  - DX9 Vertex Shader (vs2.0, vs2.x) and Pixel Shader (ps2.0, ps2.x)
  - OpenGL Vertex Program (ARB_vertex_program, ARB_fragment_program)
  - OpenGL NV20 and NV30 profiles for vertex programs and fragment programs
  - More to come … (?)

Tutorial T7: Programming Graphics Hardware | High-Level Shading Languages, Daniel Weiskopf | VIS Group, University of Stuttgart

Data Types

- `float` = 32-bit IEEE floating point
- `half` = 16-bit IEEE-like floating point
- `fixed` = 12-bit fixed [-2,2) clamping (*OpenGL only*)
- `bool` = Boolean
- `sampler*` = Handle to a texture sampler

Tutorial T7: Programming Graphics Hardware | High-Level Shading Languages, Daniel Weiskopf | VIS Group, University of Stuttgart

Vector, Matrix, and Arrays

- Built-in, first-class vector, matrix, and array data types
  - No pointers
- Up to four components for vectors:

```
float4    mypoint;
```

- Matrices are up to size 4x4, e.g.

```
float3x3 mymatrix;
```

- More general arrays:

```
float     myarray[4];
```

Tutorial T7: Programming Graphics Hardware | High-Level Shading Languages, Daniel Weiskopf | VIS Group, University of Stuttgart

Vector and Matrix Operations

- Component-wise `+ - * /` for vectors
- Extract elements from a vector by swizzle

  `a = b.xxyy;`
- Vector constructor

  `a = float4(1.0, 0.5, 0.3, 0.0);`
- Dot product

  `dot(v1,v2);` // returns a scalar
- Matrix multiplications:
  - matrix-vector: `mul(M, v);` // returns a vector
  - vector-matrix: `mul(v, M);` // returns a vector
  - matrix-matrix: `mul(M, N);` // returns a matrix

Program Flow Control and Functions

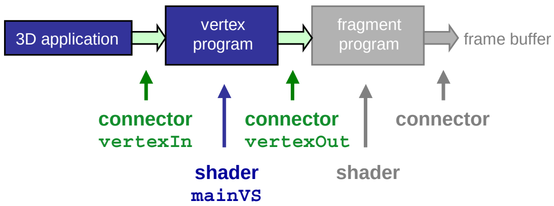
- Controls: `if-then-else`, `while-for`
- Loops via `while-for`:
  - Unrolling by the compiler
  - No dynamic loops
  - Iteration has to terminate during compile time
- Function calls and functions
  - Functions end with `return` statement
  - No recursion

Cg Example: Phong Shading

- Phong shading
  - Computation of illumination with interpolated normal vector on a per-fragment basis
  - More complex than Gouraud shading (illumination computation per-vertex with subsequent interpolation of colors)
- Blinn-Phong illumination model
  - Halfway vector
  - Ambient, diffuse, and specular terms
- Both vertex and pixel shaders are needed

Phong Shading in Cg: Vertex Shader

- First part of pipeline
- Connectors: what kind of data is transferred to / from vertex program?
- Actual vertex shader



Phong Shading in Cg: Connectors

- Describe input and output
- Varying data
- Specified as `struct`
- Extensible
- Adapted to respective implementation
- Only important data is transferred
- Pre-defined registers: `POSITION, NORMAL,` ...

Phong Shading in Cg: Connectors

```
// data from application to vertex program
struct vertexIn {
    float3 Position : POSITION;     // pre-defined registers
    float4 UV :       TEXCOORD0;
    float4 Normal :   NORMAL;
};
// vertex shader to pixel shader
struct vertexOut {
    float4 HPosition :   POSITION;
    float4 TexCoord :    TEXCOORD0;
    float3 LightVec :    TEXCOORD1;
    float3 WorldNormal : TEXCOORD2;
    float3 WorldPos :    TEXCOORD3;
    float3 WorldView :   TEXCOORD4;
};
```

Phong Shading in Cg: Vertex Shader

- Vertex shader is a function with required
  - Varying application-to-vertex input parameter
  - Vertex-to-fragment output structure
- Optional uniform input parameters
  - Constant for a larger number of primitives
  - Passed to the Cg program by the application through the runtime library
- Vertex shader for Phong shading:
  - Compute position, normal vector, viewing vector, and light vector in world coordinates

Phong Shading in Cg: Vertex Shader

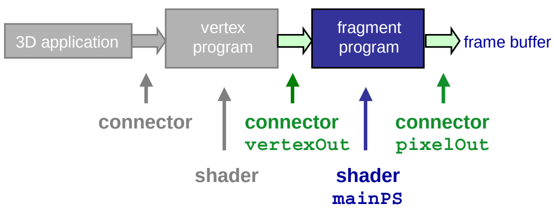
```
// vertex shader
vertexOut mainVS(vertexIn IN,       // vertex input from app
    uniform float4x4 WorldViewProj,// constant parameters
    uniform float4x4 WorldIT,       // from app: various
    uniform float4x4 World,         // transformation
    uniform float4x4 ViewIT,        // matrices and a
    uniform float3 LightPos         // point-light source
)
{
    vertexOut OUT;  // output of the vertex shader
    OUT.WorldNormal = mul(WorldIT, IN.Normal).xyz;
    // position in object coords:
    float4 Po = float4(IN.Position.x,IN.Position.y,
                       IN.Position.z,1.0);
    float3 Pw = mul(World, Po).xyz; // pos in world coords
```


Phong Shading in Cg: Vertex Shader

```
    OUT.WorldPos = Pw;            // pos in world coords
    OUT.LightVec = LightPos - Pw; // light vector
    OUT.TexCoord = IN.UV;         // original tex coords
    // view vector in world coords:
    OUT.WorldView = normalize(ViewIT[3].xyz - Pw);
    // pos in clip coords:
    OUT.HPosition = mul(WorldViewProj, Po);
    return OUT;                 // output of vertex shader
}
```


Phong Shading in Cg: Pixel Shader

- Second part of pipeline
- Connectors: from vertex to pixel shader, from pixel shader to frame buffer
- Actual pixel shader



Phong Shading in Cg: Pixel Shader

- Pixel shader is a function with required
  - Varying vertex-to-fragment input parameter
  - Fragment-to-pixel output structure
- Optional uniform input parameters
  - Constant for a larger number of fragments
  - Passed to the Cg program by the application through the runtime library
- Pixel shader for Phong shading:
  - Normalize light, viewing, and normal vectors
  - Compute halfway vector
  - Add specular, diffuse, and ambient contributions

Phong Shading in Cg: Pixel Shader

```
// final pixel output:
// data from pixel shader to frame buffer
struct pixelOut {
  float4 col : COLOR;
};
// pixel shader
pixelOut mainPS(vertexOut IN,   // input from vertex shader
    uniform float SpecExpon,    // constant parameters from
    uniform float4 AmbiColor,   // application
    uniform float4 SurfColor,
    uniform float4 LightColor
) {
    pixelOut OUT;   // output of the pixel shader
    float3 Ln = normalize(IN.LightVec);
    float3 Nn = normalize(IN.WorldNormal);
    float3 Vn = normalize(IN.WorldView);
    float3 Hn = normalize(Vn + Ln);
```

Phong Shading in Cg: Pixel Shader

```
   // scalar product between light and normal vectors:
   float ldn = dot(Ln,Nn);
   // scalar product between halfway and normal vectors:
   float hdn = dot(Hn,Nn);
   // specialized "lit" function computes weights for
   // diffuse and specular parts:
   float4 litV = lit(ldn,hdn,SpecExpon);
   float4 diffContrib =
         SurfColor * ( litV.y * LightColor + AmbiColor);
   float4 specContrib = litV.y*litV.z * LightColor;
   // sum of diffuse and specular contributions:
   float4 result = diffContrib + specContrib;
   OUT.col = result;
   return OUT;          // output of pixel shader
}
```


Cg Runtime

- Compilation offline or at runtime
- Offline
  - Preprocessing: generate assembler code by Cg compiler
  - Load assembler code at runtime
- At runtime:
  - Load Cg program
  - Compiled at runtime by using the Cg runtime

Tutorial T7: Programming Graphics Hardware | High-Level Shading Languages Daniel Weiskopf | VIS Group, University of Stuttgart

Cg Runtime

- Benefits
  - Future compatibility
  - Parameter management is easy
  - Works with effects (see later)
- Cons
  - Loading is slower because of compilation
  - No hand optimization of the result of the compilation

Tutorial T7: Programming Graphics Hardware | High-Level Shading Languages Daniel Weiskopf | VIS Group, University of Stuttgart

Core Cg Runtime

- Create a context: **`cgCreateContext()`**
- Compile a program: **`cgCreateProgram()`**, **`cgGetProgramString()`**, etc...
- Set program parameters
  - Iteration: **`cgGetFirstParameter()`**, **`cgGetNextParameter()`**, etc...
- Handle errors: **`cgGetError()`**, **`cgSetErrorCallback()`**, etc...

Tutorial T7: Programming Graphics Hardware | High-Level Shading Languages Daniel Weiskopf | VIS Group, University of Stuttgart

DX Cg Runtime: Minimal Interface

- Does not make any DX calls
- Translate information from Core Runtime to DX
  - DX vertex declaration from the Cg program: **`cgD3D9GetVertexDeclaration()`**
  - Validate a DX vertex declaration: **`cgD3D9ValidateVertexDeclaration()`**
- Generate a DX vertex or pixel shader from the Cg program

Tutorial T7: Programming Graphics Hardware | High-Level Shading Languages Daniel Weiskopf | VIS Group, University of Stuttgart

DX Cg Runtime: Expanded Interface

- Automatically performs required DX calls
- Set the DX device: **`cgD3D9SetDevice()`**
- Load a program: **`cgD3D9LoadProgram()`**
- Bind a program: **`cgD3D9BindProgram()`**
- Set parameter values: **`cgD3D9SetUniform()`**, **`cgD3D9SetUniformArray()`**, **`cgD3D9SetTexture()`**, etc...

Tutorial T7: Programming Graphics Hardware | High-Level Shading Languages Daniel Weiskopf | VIS Group, University of Stuttgart

OpenGL Cg Runtime

- Automatically performs required OpenGL calls
- Load a program: `cgGLLoadProgram()`
- Enable a profile: `cgGLEnableProfile()`
- Bind a program: `cgGLBindProgram()`
- Set parameter values:
  `cgGLSetParameter{1234}{fd}{v}()`,
  `cgGLSetParameterArray{1234}{fd}()`,
  `cgGLSetTextureParameter()`, etc...

Effect Files

- Problem: Shaders are just one building block
  - Only one part of the rendering pipeline (vertex or pixel shader)
  - One rendering pass only
  - No shading context
- An *effect* is the solution
  - Entire collection of render states, texture states, …
  - Representation of the same shading idea independent of hardware and API

FX Files

- .fx file format and effects runtime API
  - Introduced in DX8, extended in DX9
- Constituents
  - HLSL and/or assembly vertex and pixel shaders
  - Parameters that can be exposed/tweaked
  - Render / texture / fixed-function states etc.
  - Single or multi-pass rendering
  - Multiple implementations (techniques) for targeting different hardware: fallback for less powerful GPUs

CgFX

- Supports DirectX .fx files
  - Fully compatible with DX9 D3DXEffects
- Combines features of Cg and DirectX effects
- CgFX runtime
  - API enables creation of effects from .fx files
  - Compiled into state blocks
  - Enumerates techniques, parameters, …

Structure of an Effect File

- Multiple rendering algorithms (techniques)
- One or multiple passes per technique
- Outside parameters and tweakables
- File structure

```
Variable declarations
Technique 1
  Pass 1
  …
  Pass n
…
Technique n
  Pass 1
  …
  Pass n
```


Effect File: Example

```
string description = "Simple color map texture example";
// tweakable
texture colorTexture : DiffuseMap
<
  string File = "color.dds";
  string TextureType = "2D";
>;

sampler2D map = sampler_state
{
    Texture = <colorTexture>;
    MinFilter = Linear;
    MagFilter = Linear;
    MipFilter = None;
};
```

Effect File: Example

```
// vertex output structure
struct vertexOut {
   float4 texCoord0 : TEXCOORD0;
};
// fragment output structure
struct fragment {
  float4 col : COLOR;
};

fragment texturePS(vertexOut I, uniform sampler2D
   colorMap)
{
     fragment O;
     // Lookup the color map texture
     float4 texColor = tex2D(colorMap);
     O.col = texColor;
     return O;
}
```


Effect File: Example

```
// just a single technique with a pixel shader
technique simpleTextured
{
  pass p0
  {
   ZEnable = true;
   ZWriteEnable = true;
   CullMode = None;
   // compile high-level shader to ps1.0 backend:
   PixelShader = compile ps_1_1 texturePS(map);
  }
}
```


Using Effects

- Load .fx file
- Validate technique for hardware
- Detect parameters for technique
- Render loop for all scene objects
  - Set technique and parameters
  - For each pass in technique
    - Set state for pass
    - Draw object

Using Effects: Render Loop

```
// Render the scene with the chosen technique
for (unsigned int m = 0; m < numObjects; ++m) {
    // Begin effect
    UINT numPasses;
    HRESULT hr = Effect->Begin(&numPasses, 0);
    // Set parameters
    SetParameters();
    // Render the model with the effect
    for (unsigned int pass = 0; pass < numPasses; ++pass) {
        // Setup the render states for this pass
        hr = Effect->Pass(pass);
        // Render the geometry of the object
        DrawGeometry();
    }
    // End effect: cleanup
    hr = Effect->End();
}
```

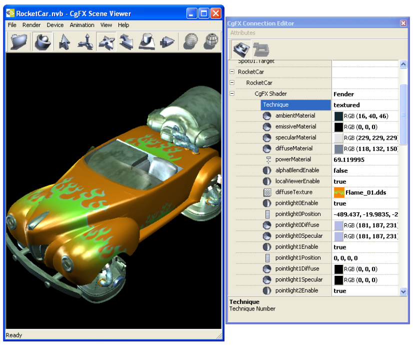

Tools

- What is still missing in effect files?
  - Scene objects
- Integrated authoring in animation and modeling tools
  - 3ds MAX
  - MAYA
  - XSI
- Support (in parts or completely) for
  - Shaders
  - Materials
  - Scenes

Tools

- Editors and viewers
  - Cg: CgFX Viewer
  - ATI's RenderMonkey
  - DirectX: EffectEdit
- Advantages
  - Rapid prototyping
  - Compile error messages
  - No C / C++ programming required
  - Instant result

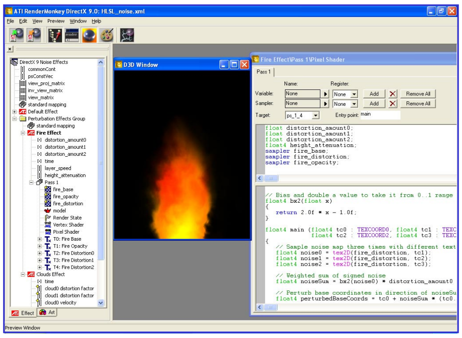
CgFX Viewer



CgFX Viewer

- NVB files created by exporter for 3ds MAX
  - Contains scene objects
- Connects scene objects to effects / techniques
- Only changes to .fx files can be saved in CgFX Viewer
- Other parameters are saved by 3ds MAX exporter only
- OpenGL, DX8, and DX9 backends
  - Switch between devices at any point
- Included in full Cg SDK

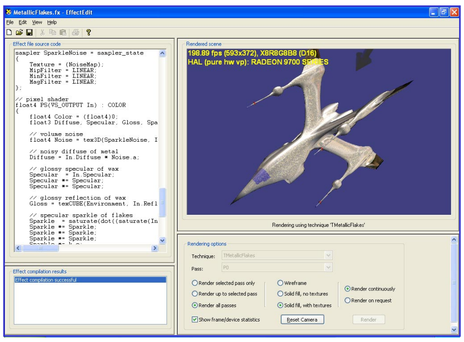
RenderMonkey



RenderMonkey

- Currently for DirectX
  - Assembler or HLSL
  - Extension to OpenGL possible
- XML file format
  - Render states
  - Model and texture information
  - Vertex and fragment shaders
  - Instead of .fx files
- Render-to-texture functionality
- Import / export plug-ins

DirectX EffectEdit



DirectX EffectEdit

- .fx files
- ASCII description
- Vertex and fragment programs in assembler and HLSL
- Manages several techniques
- Error messages
- Included in DirectX SDK

Summary

- High-level shading languages for
  - Vertex programs
  - Fragment programs
- Cg as an example
  - C-like language
  - Special support for GPU and CG programming
  - Platform independency
- Effects
  - Information beyond vertex and fragment shaders
  - Complete render states
- Tools for rapid prototyping

Documentation and Links

- Cg SDK:
  - Cg User Manual
  - Cg Specification
- DirectX 9 SDK
- Links:
  - http://developer.nvidia.com/cg
  - http://www.microsoft.com/directx

Further Reading

[Fernando & Kilgard 03] R. Fernando, M. J. Kilgard. The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics. Addison-Wesley, 2003.

[Hanrahan & Lawson 90] P. Hanrahan, J. Lawson. A Language for Shading and Lighting Calculations. In *SIGGRAPH 1990 Conference Proceedings*, pages 289-298.

[Mark et al. 03] W. R. Mark, R. S. Glanville, K. Akeley, M. J. Kilgard. Cg: A System for Programming Graphics Hardware in a C-like Language. In *SIGGRPAPH 2003 Conference Proceedings.*

[McCool et al. 02] M. D. McCool, Z. Qin, T. S. Popa. Shader Metaprogramming. In *Graphics Hardware Workshop 2002*, pages 57-68.

[Proudfoot et al. 01] K. Proudfoot, W. R. Mark, S. Tzvetkov, P. Hanrahan. A Real-Time Procedural Shading System for Programmable Graphics Hardware. In *SIGGRAPH 2001 Conference Proceedings*, pages 159-170.

Further Reading

[Olano & Lastra 98] M. Olano, A. Lastra. A Shading Language on Graphics Hardware: The PixelFlow Shading System. In *SIGGRAPH 1998 Conference Proceedings*, pages 159-168.

[Upstill 90] S. Upstill. The Renderman Companion: A Programmer's Guide to Realistic Computer Graphics. Addison-Wesley, 1990.