Programming the GPU: High-Level Shading Languages

Randy Fernando
Developer Technology Group

NVIDIA

Talk Overview
- The Evolution of GPU Programming Languages
- GPU Programming Languages and the Graphics Pipeline
- Syntax
- Examples
- HLSL FX framework

The Evolution of GPU Programming Languages

<table>
<thead>
<tr>
<th>Language</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1970</td>
</tr>
<tr>
<td>C++</td>
<td>1983</td>
</tr>
<tr>
<td>Java</td>
<td>1994</td>
</tr>
<tr>
<td>Reality Lab</td>
<td>1994</td>
</tr>
<tr>
<td>Direct3D</td>
<td>1995</td>
</tr>
<tr>
<td>HLSL</td>
<td>2002</td>
</tr>
<tr>
<td>Cg</td>
<td>2002</td>
</tr>
<tr>
<td>GLSL</td>
<td>2003</td>
</tr>
</tbody>
</table>

NVIDIA’s Position on GPU Shading Languages

- Bottom line: please take advantage of all the transistors we pack into our GPUs!
- Use whatever language you like
- We will support you
  - Working with Microsoft on HLSL compiler
  - NVIDIA compiler team working on Cg compiler
  - Working with OpenGL ARB on GLSL compiler
- If you find bugs, send them to us and we’ll get them fixed

The Need for Programmability

<table>
<thead>
<tr>
<th>Game</th>
<th>Year</th>
<th>Frames/sec</th>
<th>Resolution</th>
<th>Quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtua Fighter</td>
<td>1995</td>
<td>50K</td>
<td>640 x 480</td>
<td>Nearest filtering</td>
</tr>
<tr>
<td>Dead or Alive 3</td>
<td>2001</td>
<td>16-bit color</td>
<td>640 x 480</td>
<td>Trilinear filtering</td>
</tr>
<tr>
<td>Dawn</td>
<td>2003</td>
<td>128-bit color</td>
<td>1024 x 768</td>
<td>8x1 Aniso filtering</td>
</tr>
</tbody>
</table>

The Need for Programmability

<table>
<thead>
<tr>
<th>Game</th>
<th>Year</th>
<th>Frames/sec</th>
<th>Resolution</th>
<th>Quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtua Fighter</td>
<td>1995</td>
<td>50K</td>
<td>640 x 480</td>
<td>Nearest filtering</td>
</tr>
<tr>
<td>Dead or Alive 3</td>
<td>2001</td>
<td>16-bit color</td>
<td>640 x 480</td>
<td>Trilinear filtering</td>
</tr>
<tr>
<td>Dawn</td>
<td>2003</td>
<td>128-bit color</td>
<td>1024 x 768</td>
<td>8x1 Aniso filtering</td>
</tr>
</tbody>
</table>
**Where We Are Now**

- 222M Transistors
- 660M tri/sec
- 64 Gflops
- 128-bit color
- 1600 x 1200
- 16:1 aniso filtering

**The Motivation for High-Level Shading Languages**

- Graphics hardware has become increasingly powerful
- Programming powerful hardware with assembly code is hard
- GeForce FX and GeForce 6 Series GPUs support programs that are thousands of assembly instructions long
- Programmers need the benefits of a high-level language:
  - Easier programming
  - Easier code reuse
  - Easier debugging

**GPU Programming Languages and the Graphics Pipeline**

**The Graphics Pipeline**

- **Application**
- **Vertex Shader**
- **Fragment Shader**
- **Frame Buffer**

**Shaders and the Graphics Pipeline**

In the future, other parts of the graphics pipeline may become programmable through high-level languages.
Compilation

Using GPU Programming Languages

- Use 3D API calls to specify vertex and fragment shaders
- Enable vertex and fragment shaders
- Load/enable textures as usual
- Draw geometry as usual
- Set blend state as usual
- Vertex shader will execute for each vertex
- Fragment shader will execute for each fragment

Compilation Targets

- Code can be compiled for specific hardware
- Optimizes performance
- Takes advantage of extra hardware functionality
- May limit language constructs for less capable hardware
- Examples of compilation targets:
  - vs_1_1, vs_2_0, vs_3_0
  - ps_1_1, ps_2_0, ps_2_x, ps_2_a, ps_3_0
  - vs_3_0 and ps_3_0 are the most capable profiles, supported only by GeForce 6 Series GPUs

Shader Creation

- Shaders are created (from scratch, from a common repository, authoring tools, or modified from other shaders)
- These shaders are used for modeling in Digital Content Creation (DCC) applications or rendering in other applications
- A shading language compiler compiles the shaders to a variety of target platforms, including APIs, OSes, and GPUs

Language Syntax
Let's Pick a Language

- HLSL, Cg, and GLSL have much in common
- But all are different (HLSL and Cg are much more similar to each other than they are to GLSL)
- Let’s focus on just one language (HLSL) to illustrate the key concepts of shading language syntax

General References:
- HLSL: DirectX Documentation (http://www.microsoft.com/DirectX)
- Cg: The Cg Tutorial (http://developer.nvidia.com/CgTutorial)
- GLSL: The OpenGL Shading Language

Data Types

- float 32-bit IEEE floating point
- half 16-bit IEEE-like floating point
- bool Boolean
- sampler Handle to a texture sampler
- struct Structure as in C/C++
- No pointers... yet.

Array / Vector / Matrix Declarations

- Native support for vectors (up to length 4) and matrices (up to size 4x4):
  - float4 mycolor;
  - float3x3 mymatrix;
- Declare more general arrays exactly as in C:
  - float lightpower[8];
- But, arrays are first-class types, not pointers:
  - float v[4] != float4 v
- Implementations may subset array capabilities to match HW restrictions

Function Overloading

- Examples:
  - float myfuncA(float3 x);
  - float myfuncA(half3 x);
  - float myfuncB(float2 a, float2 b);
  - float myfuncB(float3 a, float3 b);
  - float myfuncB(float4 a, float4 b);
- Very useful with so many data types.

Different Constant-Typing Rules

- In C, it’s easy to accidentally use high precision:
  - half x, y;
  - x = y * 2.0; // Multiply is at half precision!
- Not in HLSL:
  - x = y * 2.0; // Multiply is at float precision (from y)
- Unless you want to:
  - x = y * 2.0f; // Multiply is at float precision

Support for Vectors and Matrices

- Component-wise + - * / for vectors
- Dot product:
  - dot(v1, v2); // returns a scalar
- Matrix multiplications:
  - assuming a float4x4 M and a float4 v
  - matrix-vector:mul(M, v); // returns a vector
  - vector-matrix:mul(v, M); // returns a vector
  - matrix-matrix:mul(M, N); // returns a matrix
New Operators

- Swizzle operator extracts elements from vector or matrix
  \[ a = b.xxyy; \]

  Examples:
  ```
  float4 vec1 = float4(4.0, -2.0, 5.0, 3.0);
  float2 vec2 = vec1.yx;     // vec2 = (-2.0,4.0)
  float scalar = vec1.w;     // scalar = 3.0
  float3 vec3 = scalar.xxx;  // vec3 = (3.0, 3.0, 3.0)
  float4x4 myMatrix;
  // Set myFloatScalar to myMatrix[3][2]
  float myFloatScalar = myMatrix._m32;
  ```

- Vector constructor builds vector
  \[ a = \text{float4}(1.0, 0.0, 0.0, 1.0); \]

Sample Shaders

Looking Through a Shader

- Demonstration in FX Composer

The Problem with Just a Shading Language

- A shading language describes how the vertex or fragment processor should behave
- But how about:
  - Texture state?
  - Blending state?
  - Depth test?
  - Alpha test?
- All are necessary to really encapsulate the notion of an “effect”
- Need to be able to apply an “effect” to any arbitrary set of geometry and textures
- Solution: .fx file format
**HLSL FX**

- Powerful shader specification and interchange format
- Provides several key benefits:
  - Encapsulation of multiple shader versions
  - Level of detail
  - Functionality
  - Performance
- Editable parameters and GUI descriptions
- Multipass shaders
- Render state and texture state specification
- FX shaders use HLSL to describe shading algorithms
- For OpenGL, similar functionality is available in the form of CgFX (shader code is written in Cg)
- No GLSL effect format yet, but will appear eventually

**Using Techniques**

- Each .fx file typically represents an effect
- Techniques describe how to achieve the effect
- Can have different techniques for:
  - Level of detail
  - Graphics hardware with different capabilities
  - Performance
- A technique is specified using the `technique` keyword
- Curly braces delimit the technique’s contents

**Multipass**

- Each technique may contain one or more passes
- A pass is defined by the `pass` keyword
- Curly braces delimit the pass contents
- You can set different graphics API state in each pass

**HLSL .fx Example**

- Demonstration in FX Composer

**Questions?**