Programming the GPU: High-Level Shading Languages

Randy Fernando
Developer Technology Group
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

- **C** (AT&T, 1970s)
- **C++** (AT&T, 1983)
- **Java** (Sun, 1994)
- **IRIS GL** (SGI, 1982)
- **OpenGL** (ARB, 1992)
- **Reality Lab** (RenderMorphics, 1994)
- **Direct3D** (Microsoft, 1995)
- **RenderMan** (Pixar, 1988)
- **PixelFlow Shading Language** (UNC, 1998)
- **Real-Time Shading Language** (Stanford, 2001)

- **HLSL** (Microsoft, 2002)
- **Cg** (NVIDIA, 2002)
- **GLSL** (ARB, 2003)
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
  - NVIDIA compiler team working on GLSL compiler
- If you find bugs, send them to us and we’ll get them fixed
The Need for Programmability

- **Virtua Fighter** (SEGA Corporation)
  - NV1
  - 50K triangles/sec
  - 1M pixel ops/sec
  - 1M transistors
  - 1995

- **Dead or Alive 3** (Tecmo Corporation)
  - Xbox (NV2A)
  - 100M triangles/sec
  - 1G pixel ops/sec
  - 20M transistors
  - 2001

- **Dawn** (NVIDIA Corporation)
  - GeForce FX (NV30)
  - 200M triangles/sec
  - 2G pixel ops/sec
  - 120M transistors
  - 2003
The Need for Programmability

Virtua Fighter (SEGA Corporation)
- NV1
- 16-bit color
- 640 x 480
- Nearest filtering
- 1995

Dead or Alive 3 (Tecmo Corporation)
- Xbox (NV2A)
- 32-bit color
- 640 x 480
- Trilinear filtering
- 2001

Dawn (NVIDIA Corporation)
- GeForce FX (NV30)
- 128-bit color
- 1024 x 768
- 8:1 Aniso filtering
- 2003

XY
Where We Are Now

- 222M Transistors
- 660M tris/second
- 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

Assembly

```
... 
RSQ R0, R0.x;
MUL R0, R0.x, c[11].xyzx;
MOV R1, c[3];
MUL R1, R1.x, c[0].xyzx;
DP3 R2, R1.xyzx, R1.xyzx;
RSQ R2, R2.x;
MUL R1, R2.x, R1.xyzx;
ADD R2, R0.xyzx, R1.xyzx;
DP3 R3, R2.xyzx, R2.xyzx;
RSQ R3, R3.x;
MUL R2, R3.x, R2.xyzx;
DP3 R2, R1.xyzx, R2.xyzx;
MAX R2, c[3].z, R2.x;
MOV R2.z, c[3].y;
MOV R2.w, c[3].y;
LIT R2, R2;
... 
```

High-Level Language

```
... 
float3 cSpecular = pow(max(0, dot(Nf, H)), phongExp).xxx;
float3 cPlastic = Cd * (cAmbient + cDiffuse) + Cs * cSpecular;
... 
```
GPU Programming Languages and the Graphics Pipeline
The Graphics Pipeline

Colored Vertices After Vertex Transformation → Primitive Assembly → Rasterization → Interpolation, Texturing, and Coloring
The Graphics Pipeline

- Colored Vertices After Vertex Transformation
- Primitive Assembly
- Rasterization
- Interpolation, Texturing, and Coloring

- Vertex Program
  - Executed Once Per Vertex

- Fragment Program
  - Executed Once Per Fragment
Shaders and the Graphics Pipeline

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

3D Application
- Direct3D
- OpenGL
- HLSL
- Cg
- GLSL

3D Graphics API
- Shading Language

GPU
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:
- Cg: The Cg Tutorial ([http://developer.nvidia.com/CgTutorial](http://developer.nvidia.com/CgTutorial))
- GLSL: The OpenGL Shading Language ([http://www.opengl.org](http://www.opengl.org))
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):

```c
float4    mycolor;
float3x3  mymatrix;
```

Declare more general arrays exactly as in C:

```c
float lightpower[8];
```

But, arrays are first-class types, not pointers

```c
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

```c
half x, y;
x = y * 2.0;  // Multiply is at
              // float precision!
```

Not in HLSL

```c
x = y * 2.0;  // Multiply is at
              // half precision (from y)
```

Unless you want to

```c
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**
  
  \[ a = float4(1.0, 0.0, 0.0, 1.0); \]
Examples
Sample Shaders
Looking Through a Shader

Demonstration in FX Composer
HLSSL FX Framework
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 may 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
An Example: SimpleTexPs.fx

/******* TWEAKABLES **********/

float4x4 WorldIT : WorldInverseTranspose < string UIWidget="None"; >;
float4x4 WorldViewProj : WorldViewProjection < string UIWidget="None"; >;
float4x4 World : World < string UIWidget="None"; >;
float4x4 ViewI : ViewInverseTranspose < string UIWidget="None"; >;

////////////////

float3 LightPos : Position
<
    string Object = "PointLight";
    string Space = "World";
> = {-10.0f, 10.0f, -10.0f};

float3 AmbiColor : Ambient = {0.1f, 0.1f, 0.1f};
An Example: SimpleTexPs.fx (Cont’d)

texture ColorTexture : DIFFUSE

<
  string ResourceName = "default_color.dds";
  string TextureType = "2D";
>

sampler2D cmap = sampler_state
{
  Texture = <ColorTexture>;
  MinFilter = Linear;
  MagFilter = Linear;
  MipFilter = None;
};
An Example: SimpleTexPs.fx (Cont’d)

/* data from application vertex buffer */
struct appdata {
    float3 Position : POSITION;
    float4 UV : TEXCOORD0;
    float4 Normal : NORMAL;
};

/* data passed from vertex shader to pixel shader */
struct vertexOutput {
    float4 HPosition : POSITION;
    float2 TexCoord0 : TEXCOORD0;
    float4 diffCol : COLOR0;
};
/*********** vertex shader *****/

vertexOutput lambVS(appdata IN)
{
    vertexOutput OUT;
    float3 Nn = normalize(mul(IN.Normal, WorldIT).xyz);
    float4 Po = float4(IN.Position.xyz,1);
    OUT.HPosition = mul(Po, WorldViewProj);
    float3 Pw = mul(Po, World).xyz;
    float3 Ln = normalize(LightPos - Pw);
    float ldn = dot(Ln,Nn);
    float diffComp = max(0,ldn);
    OUT.diffCol = float4((diffComp.xxx + AmbiColor),1);
    OUT.TexCoord0 = IN.UV.xy;
    return OUT;
}

An Example: SimpleTexPs.fx (Cont’d)
An Example: SimpleTexPs.fx (Cont’d)

/******* pixel shader *****/**

float4 myps(vertexOutput IN) : COLOR {
    float4 texColor = tex2D(cmap, IN.TexCoord0);
    float4 result = texColor * IN.diffCol;
    return result;
}

Prototype: float4 myps(vertexOutput IN) : COLOR;

The pixel shader includes:
- A fragment that is to be rendered.
- A texture coordinate and a color value.
- A texture mapping.
- The pixel shader function itself, which takes the texture coordinate and color value, and returns a result according to the pixel shader logic.
An Example: SimpleTexPs.fx (Cont’d)

technique t0
{
  pass p0
  {
    VertexShader = compile vs_1_1 lambVS();
    ZEnable = true;
    ZWriteEnable = true;
    CullMode = None;
    PixelShader = compile ps_1_1 myps();
  }
}
HLSL .fx Example

Demonstrations in FX Composer
Questions?
developer.nvidia.com
The Source for GPU Programming

- Latest documentation
- SDKs
- Cutting-edge tools
  - Performance analysis tools
  - Content creation tools
- Hundreds of effects
- Video presentations and tutorials
- Libraries and utilities
- News and newsletter archives

- Practical real-time graphics techniques from experts at leading corporations and universities

- Great value:
  - Full color (300+ diagrams and screenshots)
  - Hard cover
  - 816 pages
  - CD-ROM with demos and sample code

For more, visit: http://developer.nvidia.com/GPUGems

“GPU Gems is a cool toolbox of advanced graphics techniques. Novice programmers and graphics gurus alike will find the gems practical, intriguing, and useful.”

Tim Sweeney
Lead programmer of Unreal at Epic Games

“This collection of articles is particularly impressive for its depth and breadth. The book includes product-oriented case studies, previously unpublished state-of-the-art research, comprehensive tutorials, and extensive code samples and demos throughout.”

Eric Haines
Author of Real-Time Rendering