Introduction to Shader Programming

Game Design Experience
Professor Jim Whitehead
March 9, 2009
Announcements

• Days until Final Project Due: 7
  ▶ Due Monday, March 16
  ▶ Few students have been attending help sessions
  ▶ We will not be able to help you as well at the last minute

• 3D modeling homework
  ▶ Due Today, by 5PM
  ▶ Submit code via Homework submission website
What is a Shader?

• Recall that all 3D drawing in XNA uses a Shader
  ► Have been using BasicEffect shader so far
• But, more generally, what is a shader?
  ► Today, gaming computers have both a CPU, and a GPU
    • CPU is on motherboard, GPU is on graphics card
      – CPU is an unspecialized computer
    • GPU is a computer specialized for 3D graphics
      – Advantage: faster 3D graphics, more effects, larger scenes
  ► A Shader is a small program that runs on the GPU
    • Written in a Shader language (HLSL, Cg, GLSL)
    • XNA supports only the HLSL shader language
Shader Languages

- Currently 3 major shader languages
  - Cg (Nvidia)
  - HLSL (Microsoft)
    - Derived from Cg
  - GLSL (OpenGL)
- Main influences are
  - C language
  - pre-existing Shader languages developed in university and industry

Source: [http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter01.html](http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter01.html) (Modified with information on HLSL and GLSL)
Brief history

• Initially, computers did not have specialized graphics hardware
  ▶ In mid-90’s 3D acceleration hardware appeared
    • OpenGL typically provided better support
  ▶ DirectX 7 (1999) introduced support for hardware T&L
    • Transform and lighting
    • Moved vertex transformations and lighting computations from CPU to GPU
    • Improved game graphics, but at a cost: lighting and display calculations hard-wired into cards
    • Led to games having similar look
  ▶ In 2002, first consumer-level programmable GPUs became available
    • Led to development of Cg, HLSL, and GLSL shader languages
    • Benefit: can have game-specific custom graphics programs running on GPU
    • Games can have very distinctive visuals
Types of Shaders

- Shaders (GPU programs) are specialized into 3 different types:
  - Vertex shaders
    - Executed once per vertex in a scene.
    - Transforms 3D position in space to 2D coordinate on screen
    - Can manipulate position, color, texture coordinates
    - Cannot add new vertices
  - Geometry shaders
    - Can add/remove vertices from a mesh
    - Can procedurally generate geometry, or add detail to shapes
  - Pixel shaders (fragment shaders)
    - Calculates the color of individual pixels
    - Used for lighting, texturing, bump mapping, etc.
    - Executed once per pixel per geometric primitive
Shader control flow

- C#/XNA program sends vertices and textures to the GPU
  - These are the input for the vertex and pixel shader
- Shader executes vertex shader
  - Once per vertex
- Shader executes pixel shader
  - Once per pixel in each primitive object

Diagram:

- **CPU**
- **GPU**
  - C#/XNA program → vertices, textures
  - Vertex Shader
  - Pixel Shader
  - display
Anatomy of a Shader in HLSL

- Shader is a program written in textual form in HLSL
- Programs tend to have these parts
  - Global variables
    - Variables used by multiple functions
    - Way to pass arbitrary data from C#/XNA to Shader
  - Data structure definitions
    - Data structures used within the shader functions
  - Vertex and Pixel shaders
    - Functions written in HLSL
  - Techniques
    - Describe grouping of vertex and pixel shaders
    - Describe ordering of same

<table>
<thead>
<tr>
<th></th>
<th>Global variables</th>
<th>Data structure definitions</th>
<th>Vertex shading functions</th>
<th>Pixel shading functions</th>
<th>Techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(calls to vertex and pixel shading functions)</td>
</tr>
</tbody>
</table>
Common data types in HLSL

- HLSL has well known data types
  - int, float, bool, string, void
- Vectors
  - float3, float4 – 3/4 item floating point vector
    - float4 color = float4(1, 0, 0, 1);
    - Red, in RGBA (red, green, blue, alpha) color space
    - Used to represent vertices, colors
- Matrices
  - floatRxC – creates matrix with R rows, C cols
    - Float4x4 – a 4x4 matrix
    - Used to represent transformation matrices
- Structures
  struct structname {
    variable declarations of members
  }
  Example:
  struct myStruct {
    float4 position;
  }

There are two ways information is passed into a Shader:

- Directly set global variables
  - In C#/XNA:
    - `effect.Parameters[“global variable name”].SetValue(value)`
  - Example:
    - HLSL: `float4x4 World;` ▲ The global variable
    - C#/XNA: `effect.Parameters[“World”].SetValue(Matrix.Identity);`

- Semantics
  - “Magic” variables
  - Names and meaning are hard-wired by HLSL language specification
  - Examples:
    - `POSITION0: a float4 representing the current vertex`
      - When the HLSL program is executing, before each Vertex shader is called, `POSITION0` is updated with the next vertex
    - `COLOR0: a float4 representing the current pixel color`
Example Shader

- Example is Shader from Chapter 13 of *Learning XNA 3.0*, Aaron Reed, O'Reilly, 2009.

**Vertex Shader**

Computes final output position \((x,y,z,w)\) from input position

```cpp
float4x4 World;
float4x4 View;
float4x4 Projection;

struct VertexShaderInput {
    float4 Position : POSITION0;
};

struct VertexShaderOutput {
    float4 Position : POSITION0;
};

VertexShaderOutput VertexShaderFunction(VertexShaderInput input) {
    VertexShaderOutput output;
    float4 worldPosition = mul(input.Position, World);
    float4 viewPosition = mul(worldPosition, View);
    output.Position = mul(viewPosition, Projection);
    return output;
}
```
Example Shader (cont’d)

An *output* semantic

```cpp
float4 PixelShaderFunction() : COLOR0
{
    return float4(1, 0, 0, 1);
}
```

Pixel Shader function
Makes every pixel red.

Define a technique combining the vertex and pixel shaders
Contains a single pass

```cpp
Technique Technique1
{
    pass Pass1
    {
        VertexShader = compile vs_1_1 VertexShaderFunction();
        PixelShader = compile ps_1_1 PixelShaderFunction();
    }
}
```

Compile Vertex and Pixel shaders using Shader version 1.1
Connecting Shader to C#/XNA

Four main steps in using a Shader from XNA

1. **Load the Shader via the Content manager**
   - Creates Effect variable using the loaded shader
   - Add shader under Content directory
     - Move .fx file in file system to Content directory
     - On Content, right-click, then Add … Existing Item to add to project
   - Content.Load<Effect>("@"name of effect")

2. **Identify current technique to use**
   - effect.CurrentTechnique = effect.Techniques["technique name from HLSL source code"]

3. **Set global variables**
   - effect.Parameters["global variable name"]').SetValue(value)

4. **Iterate through passes (techniques) in the shader**
Connecting sample shader to C#/XNA

Effect effect;
effect = Content.Load<Effect>(@"red");
effect.CurrentTechnique = effect.Techniques["Technique1"]; effect.Parameters["World"].SetValue(Matrix.Identity);
effect.Parameters["View"].SetValue(camera.view);
effect.Parameters["Projection"].SetValue(camera.projection);
effect.Begin();
foreach (EffectPass pass in effect.CurrentTechnique.Passes) {
    pass.Begin();
    GraphicsDevice.DrawUserPrimitives<VertexPositionTexture>(PrimitiveType.TriangleStrip, verts, 0, 2);
    pass.End();
}
effect.End();

Create effect, load it via Content manager
Set current technique
Set global variables in HLSL code
Iterate through passes inside current technique