Introduction to HLSL Shaders in XNA

Game Design Experience

Modified from Jim Whitehead’s slides
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

(Modified with information on HLSL and GLSL)
• 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
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>Global variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data structure definitions</td>
</tr>
<tr>
<td>Vertex shading functions</td>
</tr>
<tr>
<td>Pixel shading functions</td>
</tr>
<tr>
<td>Techniques</td>
</tr>
<tr>
<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.

```c
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;
}
```

**Vertex Shader**

Computes final output position \((x,y,z,w)\) from input position.
Example Shader (cont’d)

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

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

An output semantic

Pixel Shader function

Makes every pixel red.

Define a technique combining the vertex and pixel shaders

Contains a single pass

Compile Vertex and Pixel shaders using Shader version 1.1
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();
Lighting

- In games, often want to have parts of a scene that are more lit than other parts
  - Helps create the mood of a scene
    - Dark and mysterious, bright and cheerful
  - Increase realism
    - Streetlights are brighter under the light

- Lighting is a complex subject
  - Many ways to create lights, shadows
  - Physical materials interact with light in different ways
  - Dull surface, shiny surface, skin: all different
Ambient Light

• Ambient light
  ► When a scene has a uniform level of lighting
  ► All surfaces of all objects have the same amount of light

• In code
  ► Brighter lighting
    • RGB values that are closer to 1
    • As lights get brighter, everything seems more and more white
  ► Dimmer lighting
    • RGB values that are closer to 0
    • As lights get dimmer, everything seems more dark

• Ambient light is not very realistic
Point Light

- Represents lights that are similar to a bare light bulb
- Light radiates uniformly in all directions
- Light modeled with a location (lightPos) and an intensity (xPower, values between 0 and 3 work well)

http://exploreankit.files.wordpress.com/2007/05/lightbulb1.jpg

http://www.gamasutra.com/features/20030418/pointlight.gif
Point lighting on a model

• To determine point lighting on a model
  ► Determine lightDir vector
    • Direction from point light to location on surface of model
      – lightDir = inPos – lightPos
      – Normalize to make next step easier
  ► Compute angle between lightDir and surface normal
    • This gives the percentage of the light’s value to apply to surface
    • Determine using dot product
      – a dot b = |a||b| \cos (angle)
      – If a & b are normalized, a dot b is \cos (angle)
      – \cos(0) = 1, \cos(pi/2) = 0
      – If light overhead (angle = 0), get full intensity
      – If light parallel to surface, get no lighting
• Compute final color as follows
  ► Calculate a base color
    • Grab a color value from a texture by applying texture coordinates
    • Or, apply a uniform base color
  ► Compute the fraction of the light’s intensity that reaches model
    • Model intensity = light intensity (xPower) * cos(angle)
  ► Add the ambient light and the light from the point light to the base color
    • Final color = base color * (model intensity + ambient)
Some important details

- To compute lighting, Vertex shader needs normal vectors as input
  - Normals come into the Shader via the NORMAL0 semantic
  - These need to be supplied from C#/XNA, since they are part of the model
  - This occurs by default if you draw meshes
    - mesh.Draw sends normal information
  - If drawing triangles, need to tell XNA to send normal information
    - Do this by using the VertexPositionNormalTexture class to define vertices of triangles
      - Each point has (x,y,z) position, (x,y,z) normal, and (u,v) texture coordinate
    - Then, must
      - GraphicsDevice.VertexDeclaration = new VertexDeclaration(GraphicsDevice, VertexPositionNormalTexture.VertexElements);
      - This determines the kind of input data that is passed to the vertex shader
Using your own shader with a mesh

- By default, each part of a mesh has a shader associated with it
  - Each ModelMeshPart has an associated Effect
  - An Effect is a shader
- To use your own shader, need to replace model effects with your own
  ```csharp
  for (int i = 0; i < mesh.MeshParts.Count; i++)
  {
    // Set this MeshParts effect to our pixel lighting effect
    mesh.MeshParts[i].Effect = effect;
  }
  ```
- Overrides effects present in model originally