Lecture 13 of 41

Surface Detail 4 of 5: Pixel & Vertex Shaders
Lab 2b: Shading in Direct3D

William H. Hsu
Department of Computing and Information Sciences, KSU

Public mirror web site: http://www.kddresearch.org/Courses/KS636
Instructor home page: http://www.cis.ksu.edu/~bhsu

Readings:
Next class: Section 3.2 – 3.4, Eberly 2e: Direct3D handout
  here article #21 (NB: not an old lesson): http://bit.ly/1hygGd7

2

Lecture Outline

- Reading for Last Class: §20.5 – 20.13, Eberly 2e
- Reading for Today: §3.1, Eberly 2e
- Reading for Next Class: §3.2 – 3.4, Eberly 2e; Direct3D handout

- Last Time: Mappings, OpenGL Texturing
  Shadow, reflection/environment, transparency, bump, displacement
  Other mappings: gloss, volumetric fog, skins, rainbows, water
  OpenGL texture mapping how-to

- Previously: Classical Fixed-Function Pipeline, Drawing in Direct3D
- Today: Shaders in Modern Pipeline
  Vertex shaders: vertex attributes to illumination at vertices
  Pixel shaders: per-vertex to pixel colors, transparency
  Hardware Rendering: Application Programmer Interfaces (APIs)
- Next: Shader Languages – (O)GLSL, HLSL / Direct3D, Renderman

Where We Are

<table>
<thead>
<tr>
<th>Lecture</th>
<th>Topic</th>
<th>PRIMARY SHADERS</th>
<th>FIXED SHADERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Course Introduction</td>
<td>Vertex Shaders</td>
<td>FFP: Per-Vertex Lighting, MVT + VT</td>
</tr>
<tr>
<td>2</td>
<td>Transition, Lesson 2</td>
<td>Vertex Shaders</td>
<td>Still used in mobile computing</td>
</tr>
<tr>
<td>3</td>
<td>Lesson 3</td>
<td>Vertex Shaders</td>
<td>Modern Programmable Pipeline: Per-Pixel Lighting</td>
</tr>
<tr>
<td>4</td>
<td>Lesson 4</td>
<td>Vertex Shaders</td>
<td>Vertex Shaders (FFP and Programmable)</td>
</tr>
<tr>
<td>5</td>
<td>Lesson 5</td>
<td>Vertex Shaders</td>
<td>Pixel Shaders (Programmable Only)</td>
</tr>
<tr>
<td>6</td>
<td>Lesson 6</td>
<td>Vertex Shaders</td>
<td>Input: per-vertex attributes (P, object space position, normal)</td>
</tr>
<tr>
<td>7</td>
<td>Lesson 7</td>
<td>Vertex Shaders</td>
<td>Output: lighting model terms (e.g. diffuse, specular, etc.)</td>
</tr>
<tr>
<td>8</td>
<td>Lesson 8</td>
<td>Vertex Shaders</td>
<td>Input: output of vertex shaders (lighting aka illumination)</td>
</tr>
<tr>
<td>9</td>
<td>Lesson 9</td>
<td>Vertex Shaders</td>
<td>Output: pixel color, transparency (R, G, B, A)</td>
</tr>
<tr>
<td>10</td>
<td>Lesson 10</td>
<td>Vertex Shaders</td>
<td>Brief Digression</td>
</tr>
<tr>
<td>11</td>
<td>Lesson 11</td>
<td>Vertex Shaders</td>
<td>Note: vertices are lit, pixels are shaded</td>
</tr>
<tr>
<td>12</td>
<td>Lesson 12</td>
<td>Vertex Shaders</td>
<td>“Pixel shader”: well-defined (iff “pixel” is)</td>
</tr>
<tr>
<td>13</td>
<td>Lesson 13</td>
<td>Vertex Shaders</td>
<td>“Vertex shader”: misnomer (somewhat)</td>
</tr>
<tr>
<td>14</td>
<td>Lesson 14</td>
<td>Vertex Shaders</td>
<td>Most people refer to both as “shaders”</td>
</tr>
<tr>
<td>15</td>
<td>Lesson 15</td>
<td>Vertex Shaders</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Lesson 16</td>
<td>Vertex Shaders</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Lesson 17</td>
<td>Vertex Shaders</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Lesson 18</td>
<td>Vertex Shaders</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Lesson 19</td>
<td>Vertex Shaders</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Lesson 20</td>
<td>Vertex Shaders</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>Lesson 21</td>
<td>Vertex Shaders</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Lesson 22</td>
<td>Vertex Shaders</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>Lesson 23</td>
<td>Vertex Shaders</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>Lesson 24</td>
<td>Vertex Shaders</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>Lesson 25</td>
<td>Vertex Shaders</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>Lesson 26</td>
<td>Vertex Shaders</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>Lesson 27</td>
<td>Vertex Shaders</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>Lesson 28</td>
<td>Vertex Shaders</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>Lesson 29</td>
<td>Vertex Shaders</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>Lesson 30</td>
<td>Vertex Shaders</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Lesson 31</td>
<td>Vertex Shaders</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>Lesson 32</td>
<td>Vertex Shaders</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>Lesson 33</td>
<td>Vertex Shaders</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>Lesson 34</td>
<td>Vertex Shaders</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>Lesson 35</td>
<td>Vertex Shaders</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>Lesson 36</td>
<td>Vertex Shaders</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>Lesson 37</td>
<td>Vertex Shaders</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>Lesson 38</td>
<td>Vertex Shaders</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>Lesson 39</td>
<td>Vertex Shaders</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>Lesson 40</td>
<td>Vertex Shaders</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>Lecture 41</td>
<td>Vertex Shaders</td>
<td></td>
</tr>
</tbody>
</table>

Review:

Vertex Shaders vs. Pixel Shaders

- Classical Fixed-Function Pipeline (FFP): Per-Vertex Lighting, MVT + VT
  - Largely superseded on desktop by programmable pipeline
  - Still used in mobile computing

- Modern Programmable Pipeline: Per-Pixel Lighting

- Vertex Shaders (FFP and Programmable)
  - Input: per-vertex attributes (P, object space position, normal)
  - Output: lighting model terms (e.g. diffuse, specular, etc.)

- Pixel Shaders (Programmable Only)
  - Input: output of vertex shaders (lighting aka illumination)
  - Output: pixel color, transparency (R, G, B, A)

Brief Digression

- Note: vertices are lit, pixels are shaded
- “Pixel shader”: well-defined (iff “pixel” is)
- “Vertex shader”: misnomer (somewhat)
- Most people refer to both as “shaders”

Review [1]:

Shadow Mapping

- Ways to Handle Shadows
  - Projected planar shadows: works well on flat surfaces only
  - Shadow stencil buffer: powerful, excellent results possible; hard!

- Open GL Shadow Mapping Tutorials

Adapted from “Shadow Mapping” © 2001 C. Everett, vtdia

Acknowledgements:

Many Mappings

Stefan Jeschke
Research Assistant
http://user.tuwien.ac.at/sjeschke

Eduard Gröller
Associate Professor
Director, Visualization Working Group
http://kddresearch.org/Courses/KS636

Institute of Computer Graphics and Algorithms
Technical University of Vienna

Thanking students from earlier: © 2002 E. Gröller & S. Jeschke, Vienna University of Technology

Institut für Computergraphik und Algorithmen
Technische Universität Wien

Published material from earlier: © 1995 – 2008 P. Hanrahan, Stanford University

Computer graphics imaging Symposium (SIGGRAPH)
Review [2]: Reflection/Environment Mapping
- How To Create Direction Maps
  - Latitude-Longitude (Map Projections) - paint
  - Gazing Ball - photograph reflective sphere
  - Fish eye Lens - standard (wide-angle) camera lens
  - Cubical Environment Map - rendering program or photography
    - Easy to produce
    - "Uniform" resolution
    - Simple texture coordinates calculation
- Old NeHe OpenGL Mapping Tutorials (2000)
  - #6 (texture map onto cube) – Beginner (Molofee): http://bit.ly/2s2Nh
- Issues: Non-Linear Mapping, Area Distortion, Converting Between Maps

Review [3]: Transparency Mapping
- Screen Door Transparency
  - Use glPolygonStipple(), glEnable(GL_POLYGON_STIPPLE)
  - See http://bit.ly/g1hQpJ
- Glass-Like Transparency using Alpha Blending
  - Use glEnable(GL_BLEND), glBlendFunc(…)
  - See http://bit.ly/hs82Za

Review [4]: Bump Mapping
- Goal: Create Illusion of Textured Surface
- Idea
  - Start with regular smooth object
  - Make height map (by hand and/or using program, i.e., procedurally)
  - Use map to perturb surface normals
  - Plug new normals into illumination equation

Review [5]: Displacement Mapping
- Displacement Map: Similar to Bump Map – Contains Delta Values
- Displacement Mapping: Uses OpenGL Shading Language (GLSL)

Review [6]: OpenGL Shading (Overview)
- Set Up Point Light Sources
  - Directional light given by "constant" vector
  - Point light" vector, dot product
- Set Up Materials, Turn Lights On
- Start Drawing (glBegin)
Review [8]: OpenGL Texturing

In initialization:
- gConTextures(...);
- gBlendTexture(...);
- gTexParameteri(...);
- gTexImage2D(...);
- gEnable(GL_TEXTURE_2D);

In display:
- Call the required `DrawTexture(...)` function.
- Activate the texture defined in initialization.
- gBlendControl() for blend parameter.
- gTexCoord(...);
- gVertex3f(...);
- gTexCoordL(...);
- gTexCoordS(...);

More Special Effects (SFX)

- Lighting (T&L)
- Enclosing Volumes
- Material Effects: Gloss
- Shadows
- Reflection
- Indirect Illumination
- Environment
- Iridescence
- Water
- Shadow Maps
- Projective Textures
- Reflections
- Refractions
- Environment
- Cube
- Sphere
- Projective Textures
- Environment
- Iridescence
- Water

History

- 1992 - id's Wolfenstein 3D video game rocks gaming world, all objects are billboards (flat planes) and rendered in software
- 1996 - id's Quake introduces a full 3D polygonal game, lighting vertices and shading pixels is still done in software
- 1999 - GeForce 256 graphics card released, now transform and lighting (T&L) of vertices is done in hardware as well (uses the fixed function pipeline)
- 2001 - GeForce 256 graphics card released, now transform and lighting (T&L) of vertices is done in hardware as well (uses the fixed function pipeline)

Fixed Function Pipeline

- Starting in 1999 some graphics cards began to do the standard lighting model and transformations in hardware (T&L). CPUs everywhere sighed in relief.
- Hardware T&L existed in the 60s and 70s, but was slow and really expensive.
- Implementing the pipeline in hardware made processing polygons much faster, but the developer could not modify the pipeline (hence "fixed function pipeline"). The fixed function pipeline dates back to the first SGI workstations.
- New programmable hardware allows programmers to write vertex and pixel programs to change the pipeline
- Vertex and pixel programs aren't necessarily slower than the fixed function alternative.
- Note that the common term "vertex shader" to describe a vertex program is misleading: vertices are 3D and pixels are shaded
Cg is a C-like language that the graphics card compiles into a program. The program is run once per-vertex and/or per-pixel on the graphics card. Cg does not have all the functionality of C. Different type systems, no malloc. http://www.cgshaders.org/articles/ has the technical documentation for Cg. Cg is actually an abstraction of the more primitive assembly language that the programmable hardware originally supported.

Quick Review: Interpolative Shading in OpenGL

By default, GL will do the following:

1. Take as input various per-vertex quantities (color, light source, eye point, texture coordinates, etc.)
2. Calculate a final color for each vertex using a basic lighting model (OpenGL uses Phong lighting)
3. For each pixel, linearly interpolate the three surrounding vertex colors to shade the pixel (OpenGL uses Gouraud shading)
4. Write the pixel color value to the frame buffer

Example: Cartoon Shader & Utah Teapot

- Cartoon shading is a cheap and neat looking effect used in video games such as Jet Set Radio Future
- Instead of using traditional methods to light a vertex, use the dot product of the light vector and the normal of the vertex to index into a 1 dimensional "texture"
- A texture is simply a lookup function for colors – nothing more and nothing less.
- Instead of a smooth transition from low intensity light (small dot product) to high intensity light (large dot product) make the 1 dimensional texture have sharp transitions.
- Textures aren’t just for “wrapping” 2D images on 3D geometry!
- Viola! Cartoon Teapot

Cg Tips

- Understand the different spaces your vertices may exist in:
  - model space: the space in which your input vertex positions exist, in this space the center of the model is at the origin
  - world space: the space in which you do most of your calculations
  - clip space: the space in which your output vertex positions must exist, this space represents the canonical view volume
- If you want a vector to have length 1 make sure to normalize the vector, this often happens when you want to use a vector to represent a direction.
- When writing a Cg program try to go one step at a time, one sequence of steps might be
  - Make sure the model vertex positions are being calculated correctly
  - set the color or texture coordinates to an arbitrary value, verify that you are changing the surface color
  - Calculate the color or texture coordinates correctly
- Check out http://cgshaders.org/articles/ for some helpful documents.

Programmable Hardware Pipeline

- clip space refers to the space of the canonical view volume
- 1 without model space vertex
- 1 with clip space vertex
- Standard HLSL
- Vertices
- Standard
- Program
- Vertex
- Texture
- Fragment
- Color
- Interpolative Shading in OpenGL
- Jet Set Radio Future
Cg: Big Picture

- Write a .cg file. This will invariably take some sort of information as a parameter to its "main()" function.
  - Note that this main() is not compiled by gcc (or any C/C++ compiler).
  - That would generate a symbol conflict, among other things. It is only processed by Nvidia's Cg compiler.
- Write a class that extends CGEffect. This is cs123's object-oriented wrapper around the basic C interface provided by Nvidia.
  - The CGEffect subclass allows you to bind data from your .C files to variables in your cg vertex program.
- Make CGEffect the IScene's current CGEffect by calling IScene: : setCGEffect(). IScene will take ownership of the CGEffect at this point, so you will not be leaky the memory you allocated yourself. Rendering will now be done using your vertex shader.
- Call IScene: : removCGEffect() if you want to turn vertex shaders off again.

HLSL [1]

- High-Level Shader Language (HLSL) is Microsoft's language for programming GPUs.
- Looks like C.
- Example vertex and pixel shader for projective texturing (texture should appear to be projected onto the scene, as if from a slide projector):

```c
VS_OUTPUTPROJTEX Out = (VS_OUTPUTPROJTEX)0;
{
    float4 Tex : TEXCOORD0;
    float4 Pos : POSITION;
    {
        struct VS_OUTPUTPROJTEX // output structure
        {
            float4 PSProjTexture(float4 Tex: TEXCOORD0) : COLOR
            {
                return tex2Dproj(ProjTexMapSampler, Tex);
            }
        }
    }
}
```

Q: What Is Cg?

- A1: Nvidia's high-level shading language.
- A2: OpenGL, Direct3D, RenderMan descendant…
- Polygons-to-Pixels Pipeline (Fixed-Function & Programmable) in Action
  - Programmable Graphics Pipeline

HLSL [2]: Code Example

- This is used by IScene to load the current passes:
  ```c
  CGEffect Pass = "TestFrag";
  CGEffect CGEffect = CGEffectFromPass(Pass);
  
  CGEffect* CGEffect = CGEffectFromPass(Pass);
  
  CGEffect CGEffect = CGEffectFromPass(Pass);
  
  CGEffect CGEffect = CGEffectFromPass(Pass);
  ```

Cg [1]

- The Cg Tutorial.
- OpenGL State Machine (Simplified) from Wikipedia:

```
Shader
Pass
P0

draw

draw

draw
```

Cg [2]

- OpenGL State Machine (Simplified) from Wikipedia:

```
Shader
Pass
P0

draw

draw

draw
```

Cg Tutorial

- Intro Slides for Cg Tutorial (Online Book): http://bl.ly/59ffSR
  - © 2009 Koen Samyn
  - GLSL: Shader language and API developed by the OpenGL consortium and usable from within an OpenGL application.
  - HLSL: Shader language and API developed by Microsoft, only usable from within a DirectX and OpenGL application. Cg has a close resemblance to HLSL.

HLSL [2]: Code Example

- This is used by IScene to load the current passes:
  ```c
  CGEffect Pass = "TestFrag";
  CGEffect CGEffect = CGEffectFromPass(Pass);
  
  CGEffect* CGEffect = CGEffectFromPass(Pass);
  
  CGEffect CGEffect = CGEffectFromPass(Pass);
  ```

HLSL [1]
GLSL [1]:
**Building on Top of OpenGL**

- New Function: Fragment (Pixel-Level) Shaders
  - Programmable pipeline – like HLSL, Cg
  - Compiles to shader objects
  - Runs on hardware: ATI Radeon 9x00+, nVidia GeForce 5x00+

GLSL [2]:
**Hybrid Shader Example – Color Cube**

Vertex Shader
```c
varying float xpos;
varying float ypos;
varying float zpos;
void main(void)
{
xpos = clamp(gl_Vertex.x,0.0,1.0);
ypos = clamp(gl_Vertex.y,0.0,1.0);
zpos = clamp(gl_Vertex.z,0.0,1.0);
gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Fragment Shader
```c
varying float xpos;
varying float ypos;
varying float zpos;
void main (void)
{
  gl_FragColor = vec4 (xpos, ypos, zpos, 1.0);
}
```

GLSL [3]:
**Cg/HLSL-Style Pipeline**

- Diffuse Shader (NeHe GLSL Example)
- Machine Problems, Projects: Will Use Combination of Shaders

Pixel & Fragment Shaders [1]

- Fragments: Pixels Plus Properties
  - Everything needed to shade pixel
  - Coordinates
  - Normals
  - Object colors: diffuse, specular
  - Other properties
  - May involve local computation of lighting (local to pixel)
  - Typical example: Phong-like shading (normal interpolation)
- Programmable
  - Use fragment data
  - Combine it with lights, textures, etc.
- Hybridization: Combine Vertex and Pixel Shading
  - Varying attributes for vertex (basis for fragment shading)
  - Interpolating value from vertex shader across fragments
Consider BRDF (Especially $N \cdot L$) From Last Lecture

Result: Diffuse Term for Phong Shading (One Light, No Specular)

Pixel & Fragment Shaders [3]: GLSL Example

```
shading2.tex

vertexDemo.sm:

void main() {
    vec3 eye = normalize(eye_position);
    vec3 half = normalize(eye + light_position);
    float shading = max(dot(eye, normal), 0.0);
    gl_FragColor = vec4(shading, 0.0, 0.0, 0.0);
}
```

© 2004 F. Rudolf, NeHe Productions

http://bit.ly/gi9g47

Summary

- Last Time: Mappings, OpenGL Texturing
- Other mappings: gloss, volumetric fog, skins, rainbows, water
- OpenGL texture mapping how-to
- Previously: Basic Drawing in Direct3D, Shading & Texturing in OpenGL
- Today: Shaders in Modern Pipeline
- Vertex shaders
  - Input: per-vertex attributes (e.g., object space position, normal)
  - Output: lighting model terms (e.g., diffuse, specular, etc.)
- Pixel shaders
  - Input: output of vertex shaders (lighting aka illumination)
  - Output: pixel color, transparency (R, G, B, A)
- Shader Languages – (O)GLSL, HLSL / Direct3D, Renderman
- Next: Using Shader Languages

Terminology

- Mappings
  - Shadow, reflection/environment, transparency, bump, displacement
  - Other mappings: gloss, volumetric fog, skins, rainbows, water
- Classical Fixed-Function Pipeline (FFP): Per-Vertex Lighting, MVT + VT
- Modern Programmable Pipeline: Per-Pixel Lighting
- Shader Languages (SLs)
  - Domain-specific programming languages
  - Geared towards hardware rendering
- Specific SLs Covered
  - HLSL / Direct3D – Microsoft’s programmable pipeline; overview
  - Cg – Nvidia’s OpenGL / Direct3D descendant
  - OGLSL – OpenGL shading language, covered in more detail
- Other Shader Languages Beyond Scope of This Course
  - Gelato – Nvidia’s production render farm SL
  - Renderman – Pixar’s specification, renderer, or SL