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/CIS636
Instructor home page: http://www.cis.ksu.edu/~bhsu

Readings:

Today: Section 3.1, Eberly 2º – see http://bit.ly/ieUq45
Next class: Section 3.2 – 3.4, Eberly 2º; Direct3D handout

NeHe article #21 (NB: not an old lesson): http://bit.ly/gi9g47

Lecture Outline

Reading for Last Class: §20.5 – 20.13, Eberly 2º (Many Mappings)
Reading for Today: §3.1, Eberly 2º
Reading for Next Class: §3.2 – 3.4, Eberly 2º; 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: lit vertices 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 Source(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Course Overview</td>
<td>Chapter 1. Eberly Z.</td>
</tr>
<tr>
<td>1</td>
<td>CG Basics; Transformation Matrices; Lab 0</td>
<td>Sections (§) 2.1, 2.2</td>
</tr>
<tr>
<td>2</td>
<td>Viewing 1: Overview, Projections</td>
<td>§ 2.3 esp. 2.4, 2.8</td>
</tr>
<tr>
<td>3</td>
<td>Viewing 2: Viewing Transformation</td>
<td>§ 2.3 esp. 2.4, 2.8</td>
</tr>
<tr>
<td>4</td>
<td>Lab 1a: Flash &amp; OpenGL Basics</td>
<td>Ch. 2, 18. Angel Primer</td>
</tr>
<tr>
<td>5</td>
<td>Viewing 3: Graphics Pipeline</td>
<td>§ 2.3 esp. 2.3, 2.6, 2.7</td>
</tr>
<tr>
<td>6</td>
<td>Scan Conversion 1: Lines, Midpoint Algorithm</td>
<td>§ 2.1.3, 2.1.4, FPVH slides</td>
</tr>
<tr>
<td>7</td>
<td>Viewing 4: Clipping &amp; Culling; Lab 1b</td>
<td>§ 2.2.3, 2.3, 2.1.3</td>
</tr>
<tr>
<td>8</td>
<td>Scan Conversion 2: Polygons, Clipping Intro</td>
<td>§ 2.4, 2.5 esp. 2.4, 3.1.6</td>
</tr>
<tr>
<td>9</td>
<td>Surface Detail 1: Illumination &amp; Shading</td>
<td>§ 2.5, 2.6.1, 2.6.2, 4.3.2, 20.2</td>
</tr>
<tr>
<td>10</td>
<td>Lab 2a: Direct3D / DirectX Intro</td>
<td>§ 2.7, Direct3D handout</td>
</tr>
<tr>
<td>11</td>
<td>Surface Detail 2: Textures, OpenGL Shading</td>
<td>§ 2.6.3, 20.3–20.4, Primer</td>
</tr>
<tr>
<td>12</td>
<td>Surface Detail 3: Mapping, OpenGL Textures</td>
<td>§ 20.5–20.13</td>
</tr>
<tr>
<td>13</td>
<td>Surface Detail 4: Pixels/Vertex Shaders, Lab 2b</td>
<td>§ 2.7</td>
</tr>
<tr>
<td>14</td>
<td>Surface Detail 5: Direct3D Shading, OGLSL</td>
<td>§ 2.7–2.8, Direct3D handout</td>
</tr>
<tr>
<td>15</td>
<td>Demos 1: CGA, Fun Scene Graphics, State</td>
<td>§ 4.1–4.3, CGA handout</td>
</tr>
<tr>
<td>16</td>
<td>Lab 3a: Shading &amp; Transparency</td>
<td>§ 2.6, 20.1, Primer</td>
</tr>
<tr>
<td>17</td>
<td>Animation 1: Basics, Keyframes, HW/Exam</td>
<td>§ 5.1–5.2</td>
</tr>
<tr>
<td>18</td>
<td>Scene Graphs: Rendering; Lab 3b: Shader</td>
<td>§§ 4.4–4.7</td>
</tr>
<tr>
<td>19</td>
<td>Demos 2: SFX, Skinning, Morphing</td>
<td>§ 8.3–8.6, CGA handout</td>
</tr>
<tr>
<td>20</td>
<td>Demos 3: Surfaces, B-reps/Volume Graphics</td>
<td>§ 10.4, 12.7, Mesh handout</td>
</tr>
</tbody>
</table>

Lightly-shaded entries denote the due dates of a written problem set, heavily-shaded entries that of a machine problem (programming assignment), blue shaded entries that of a paper review, and the green shaded entry that of the term project.

Green, blue and red letters denote exam review, exam, and exam solution review dates.

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 (e.g., 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!

- OpenGL Shadow Mapping Tutorials

Adapted from “Shadow Mapping” © 2001 C. Everitt, nVidia
Review [2]:
Reflection/Environment Mapping

- How To Create Direction Maps
  - Latitude-Longitude (Map Projections) - paint
  - Gazing Ball - photograph reflective sphere
  - Fisheye 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)
- Issues: Non-Linear Mapping, Area Distortion, Converting Between Maps

Adapted from slides © 1995 – 2009 P. Hanrahan, Stanford University

Review [3]:
Transparency Mapping

- Screen Door Transparency
  - Use `glPolygonStipple()`, `glEnable(GL_POLYGON_STIPPLE)`
- Glass-Like Transparency using Alpha Blending
  - Use `glEnable(GL_BLEND)`, `glBlendFunc(…)`

Technical University of Vienna, IEEE Vis 2004

Goon Creative, Maya Transparency Tutorial
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 [4]: Bump Mapping

Review [5]: Displacement Mapping
Review [6]:
OpenGL Shading (Overview)

- Set Up Point Light Sources
  - Directional light given by "position" vector
    ```
    GLfloat light_position[] = {-1.0, 1.0, -1.6, 0.0};
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    ```
  - Point source given by "position" point
    ```
    GLfloat light_position[] = {-1.0, 1.0, -1.0};
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    ```

- Set Up Materials, Turn Lights On
  ```
  GLfloat mat_specular[] = {0.0, 0.0, 0.0, 1.0};
  GLfloat mat_diffuse[] = {0.8, 0.6, 0.4, 1.0};
  GLfloat mat_ambient[] = {0.0, 0.0, 0.4, 1.0};
  GLfloat mat_shininess = 20.0;
  glEnable(GL_FRONT, GL_SPECULAR, mat_specular);
  glEnable(GL_FRONT, GL_DIFFUSE, mat_diffuse);
  glEnable(GL_FRONT, GL_AMBIENT, mat_ambient);
  glEnable(GL_FRONT, GL_SHININESS, mat_shininess);
  ```
  ```
  glShadeModel(GL_SMOOTH); /* enable smooth shading */
  glEnable(GL_LIGHTING); /* enable lighting */
  glEnable(GL_LIGHT0); /* enable light 0 */
```

- Start Drawing (glBegin ... glEnd)

See also: OpenGL: A Primer, 3rd (Angel)

Review [7]:
Texturing – Object Center Method

- When we treat the object intersection point as a point on a sphere, our “sphere” won’t always have the same radius

![Diagram of spheres through house/eye intersection point]

- What radius to use?
  - Compute the radius as the distance from the center of the complex object to the intersection point. Use that as the radius for the (u, v) mapping.

Adapted from slides © 2010 van Dam et al, Brown University
Review [8]:
OpenGL Texturing

In initialization:
```
glGenTextures(...);
glBindTexture( ... );
gTexParameter(...); glTexParameter(...); ... 
gTexImage2D(...);
glEnable(GL_TEXTURE_2D);
```

In display:
```
glBindTexture( ... ); // Activate the texture defined in initialization
glBegin(GL_TRIANGLES);
glTexCoord2f(...); glVertex3f(...);
```

Review [9]:
Mappings, Eberly 2e

- Fine Surface Detail: Bump ($\S$20.5 Eberly 2e)
- Material Effects: Gloss ($\S$20.6)
- Enclosing Volumes
  - Sphere ($\S$20.7)
  - Cube ($\S$20.8)
- Light
  - Refraction for Transparency ($\S$20.9)
  - Reflection aka Environment ($\S$20.10)
- Shadow
  - Shadow Maps ($\S$20.11, 20.13)
  - Projective Textures ($\S$20.12)
- More Special Effects (SFX)
  - Fog ($\S$20.14)
  - Skinning ($\S$20.15)
  - Iridescence ($\S$20.16), Water ($\S$20.17)

Adapted from slides
© 2007 Jacobs, D. W., University of Maryland

Babylon 5
**Review: Drawing in Direct3D**

- Specify the material we wish to use for the following triangles
- Specify the texture we wish to use (if we want one or NULL if not)
- Set the stream source to our vertex buffer
- Set the FVF we will be using
- Set the index buffer we will be using
- Call the required `DrawPrimitive` function

```cpp
void CGfxEntityCube::Render()
{
    gD3dDevice->SetMaterial( &m_material );
gD3dDevice->SetTexture(0,NULL);
gD3dDevice->SetStreamSource( 0, m_vb,0, sizeof(CUBEVERTEX) );
gD3dDevice->SetFVF( D3DFVF_CUBEVERTEX );
gD3dDevice->SetIndices( m_ib );

    // draw a triangle list using 24 vertices and 12 triangles
    gD3dDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST,0,0,24,0,12);
}
```
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
- 1996 - Voodoo 3Dfx graphics card released, does shading operations (such as texturing) in hardware. QuakeWorld brings hardware acceleration to Quake
- 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 3 graphics card lets programmers download assembly programs to control vertex lighting and pixel shading keeping the speed of the fixed function pipeline with none of the restrictions
- **Future** – Expanded features and high level API’s for vertex and pixel shaders, increased lighting and shadowing effects such as bump mapping and shadowing. QuakeWorld usher in a new era of realism.

Adapted from slides © 2002 – 2003 van Dam et al., Brown University

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, it was just really 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 lit and pixels are shaded.

Adapted from slides © 2002 – 2003 van Dam et al., Brown University
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

Adapted from slides © 2002 – 2003 van Dam et al., Brown University
Programmable Hardware Pipeline

- clip space refers to the space of the canonical view volume
- New graphics cards can use either the fixed function pipeline or vertex/pixel programs

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
What is Cg?

- 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
  - Can’t include standard system headers
  - No malloc
  - [http://www.cgshaders.org/articles/](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

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 will 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/](http://cgshaders.org/articles/) for some helpful documents
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 that 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 deleting the memory you allocated yourself. Rendering will now be done using your vertex shader
- Call IScene::removeCGEffect() if you want to turn vertex shaders off again

Shader Languages Overview

- **HLSL**: Shader language and API developed by Microsoft, only usable from within a DirectX application.
- **Cg**: Shader language and API developed by Nvidia, usable from within a DirectX and OpenGL application. Cg has a stark resemblance to HLSL.
- **GLSL**: Shader language and API developed by the OpenGL consortium and usable from within an OpenGL application.
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
struct VS_OUTPUTPROJTEX // output structure
{
    float4 Pos : POSITION;
    float4 Tex : TEXCOORD0;
};

VS_OUTPUTPROJTEX VSProjTexture(float4 Pos : POSITION, float3 Normal : NORMAL)
{
    VS_OUTPUTPROJTEX Out = (VS_OUTPUTPROJTEX)0;
    Out.Pos = mul(Pos, matWorldViewProj); // transform Position
    Out.Tex = mul(ProjTextureMatrix, Pos); // project texture coordinates
    return Out;
}

float4 PSProjTexture(float4 Tex : TEXCOORD0) : COLOR
{
    return tex2Dproj(ProjTexMapSampler, Tex);
}
```

Adapted from slide 2003 Wolfgang Engel, http://www.wolfgang-engel.info

HLSL [2]: Code Example

```c
// This is used by 3dsmax to load the correct parser
string ParamID = "0x0";
// DxMaterial specific
float4x4 wvp : WORLDVIEWPROJ;
struct VS_OUTPUT
{
    float4 Pos : POSITION;
    float4 Col : COLOR0;
};

VS_OUTPUT VS(float3 Pos : POSITION)
{
    VS_OUTPUT Out = (VS_OUTPUT)0;
    float4 hPos = float4( Pos, 1);
    Out.Pos = mul( hPos, wvp);
    Out.Col = float4( 1, 1, 1, 1);
    return Out;
}

technique Default
{
    pass P0
    // shaders
    CullMode = None;
    VertexShader = compile vs_2_0 VS();
}
```

© 2009 Koen Samyn
http://knol.google.com/k/hlsl-shaders
Q: What is Cg?

- A1: Nvidia's high-level shading language
- A2: An OpenGL, Direct3D, RenderMan descendant...


---

Polygons-to-Pixels Pipeline (Fixed-Function & Programmable) in Action

Programmable Graphics Pipeline

---

GLSL [1]: Building on Top of OpenGL

- How It Used to Be (in 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);
}
```

© 2003 – 2005 M. Christen, ClockworkCoders.com
Vertex Shaders [1]

- **“Fixed-Function”**
  - Standard pipeline
  - Typical examples include simple “my first shader” definitions
    - Constant (flat) shading
    - Smooth (Gouraud) shading
- **Brief Digression: Sample-Based vs. Geometry-Based Graphics**
  - Sample-based: image manipulation
    - Not rendered from (3-D) model
    - Examples: Photoshop, GIMP
  - Geometry-based: transform and render representations of objects
- **Programmable**
  - Fixed material properties selected by user
  - Procedural material properties
  - Gradients
  - Lots of other things you see in sample-based graphics

---

Vertex Shaders [2]:
Cg/HLSL-Style Pipeline

© 2003 R. Fernando & M. Kilgard. The Cg Tutorial.
**Vertex Shaders [3]: GLSL Example**

- Diffuse Shader (NeHe GLSL Example)

```glsl
varying vec3 normal;
varying vec3 vertex_to_light_vector;

void main()
{
    // Transforming The Vertex
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    // Transforming The Normal To ModelView-Space
    normal = gl_NormalMatrix * gl_Normal;
    // Transforming The Vertex Position To ModelView-Space
    vec3 vertex_in_modelview_space = gl_ModelViewMatrix * gl_Vertex;
    // Calculating The Vector From The Vertex Position To The Light Position
    vertex_to_light_vector = vec3(gl_LightSource[0].position - vertex_in_modelview_space);
}
```

- Machine Problems, Projects: Will Use Combination of Shaders

© 2004 F. Rudolf, NeHe Productions

http://bit.ly/gi9g47

---

**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 $\mathbf{N} \cdot \mathbf{L}$) From Last Lecture

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

- 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: 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
  - (O)GLSL – 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