

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

KSOL course pages: <http://bit.ly/hGvXIH> / <http://bit.ly/eVizrE>

Public mirror web site: <http://www.kddresearch.org/Courses/CIS636>

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

Readings:

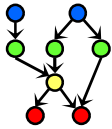
Today: Section 3.1, Eberly 2^e – see <http://bit.ly/ieUq45>

Next class: Section 3.2 – 3.4, Eberly 2^e; **Direct3D handout** *

* **Toymaker tutorials, K. Ditchburn**: <http://bit.ly/hMqxMI>

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

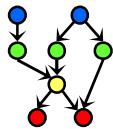




Lecture Outline

- Reading for Last Class: §20.5 – 20.13, Eberly 2^e (Many Mappings)
- Reading for Today: §3.1, Eberly 2^e
- Reading for Next Class: §3.2 – 3.4, Eberly 2^e; 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





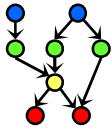
Where We Are

Lecture	Topic	Primary Source(s)
0	Course Overview	Chapter 1, Eberly 2 ^e
1	CG Basics: Transformation Matrices; Lab 0	Sections (§) 2.1, 2.2
2	Viewing 1: Overview, Projections	§ 2.2.3 – 2.2.4, 2.8
3	Viewing 2: Viewing Transformation	§ 2.3 esp. 2.3.4; FVFH slides
4	Lab 1a: Flash & OpenGL Basics	Ch. 2, 16¹, Angel Primer
5	Viewing 3: Graphics Pipeline	§ 2.3 esp. 2.3.7; 2.6, 2.7
6	Scan Conversion 1: Lines, Midpoint Algorithm	§ 2.5.1, 3.1; FVFH slides
7	Viewing 4: Clipping & Culling; Lab 1b	§ 2.3.5, 2.4, 3.1.3
8	Scan Conversion 2: Polygons, Clipping Intro	§ 2.4, 2.5 esp. 2.5.4, 3.1.6
9	Surface Detail 1: Illumination & Shading	§ 2.5, 2.6.1 – 2.6.2, 4.3.2, 20.2
10	Lab 2a: Direct3D / DirectX Intro	§ 2.7, Direct3D handout
11	Surface Detail 2: Textures; OpenGL Shading	§ 2.6.3, 20.3 – 20.4, Primer
12	Surface Detail 3: Mappings; OpenGL Textures	§ 20.5 – 20.13
13	Surface Detail 4: Pixel/Vertex Shad.; Lab 2b	§ 3.1
14	Surface Detail 5: Direct3D Shading; OGLSL	§ 3.2 – 3.4, Direct3D handout
15	Demos 1: CGA, Fun; Scene Graphs: State	§ 4.1 – 4.3, CGA handout
16	Lab 3a: Shading & Transparency	§ 2.6, 20.1, Primer
17	Animation 1: Basics, Keyframes; HW/Exam	§ 5.1 – 5.2
	Exam 1 review; Hour Exam 1 (evening)	Chapters 1 – 4, 20
18	Scene Graphs: Rendering; Lab 3b: Shader	§ 4.4 – 4.7
19	Demos 2: SFX; Skinning, Morphing	§ 5.3 – 5.5, CGA handout
20	Demos 3: Surfaces; B-reps/Volume Graphics	§ 10.4, 12.7, Mesh handout

Lightly-shaded entries denote the due date 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.

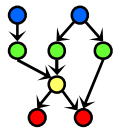




Review: Vertex Shaders vs. Pixel Shaders

- **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”





Acknowledgements: Many Mappings



Stefan Jeschke

Research Assistant
<http://bit.ly/hUUM94>

Eduard Gröller

Associate Professor
Director, Visualization Working Group
<http://bit.ly/hUUM94>

Institute of Computer Graphics and Algorithms
Technical University of Vienna

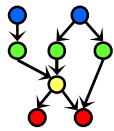


Texturing material from slides © 2002 E. Gröller & S. Jeschke, Vienna University of Technology
<http://bit.ly/dJFYq9>



Mapping material from slides © 1995 – 2009 P. Hanrahan, Stanford University
<http://bit.ly/hZfsjZ> (CS 348B, Computer Graphics: Image Synthesis Techniques)





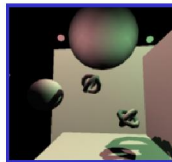
Review [1]: Shadow Mapping

● Ways to Handle Shadows

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



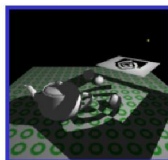
Projected
planar
shadows



Shadow
volumes



Light maps



Hybrid
approaches



Shadow Stencil Buffer

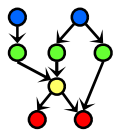


● OpenGL Shadow Mapping Tutorials

- ✦ Beginner/Intermediate (Baker, 2003): <http://bit.ly/e1LA2N>
- ✦ Advanced (Octavian et al., 2000): <http://bit.ly/f1iRYB> (old NeHe #27)

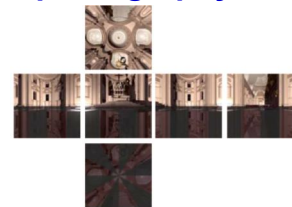
Adapted from "Shadow Mapping" © 2001 C. Everitt, nVidia
http://developer.nvidia.com/object/shadow_mapping.html





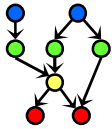
Review [2]: Reflection/Environment Mapping

- **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)**
 - ✱ #6 (texture map onto cube) – Beginner (Molofee): <http://bit.ly/gKj2Nb>
 - ✱ #23 (sphere) – Intermediate (Schmick & Molofee): <http://bit.ly/e3Zb8h>
- **nVidia Tutorial: OpenGL Sphere Map (1999):** <http://bit.ly/eJEdAM>
- **Issues: Non-Linear Mapping, Area Distortion, Converting Between Maps**



Adapted from slides © 1995 – 2009 P. Hanrahan, Stanford University
<http://bit.ly/hZfsjZ> (CS 348B)



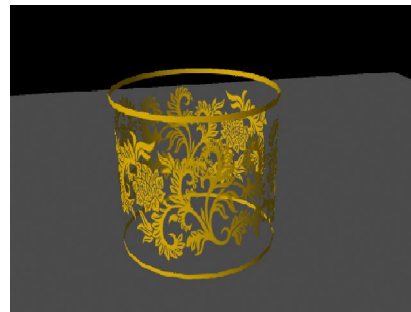


Review [3]: Transparency Mapping

- OpenGL Transparency How-To at [OpenGL.org](http://bit.ly/hRaQgk): <http://bit.ly/hRaQgk>
- 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>

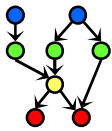


Viola et al. (2004), <http://bit.ly/dVEa7I>
Technical University of Vienna, IEEE Vis 2004



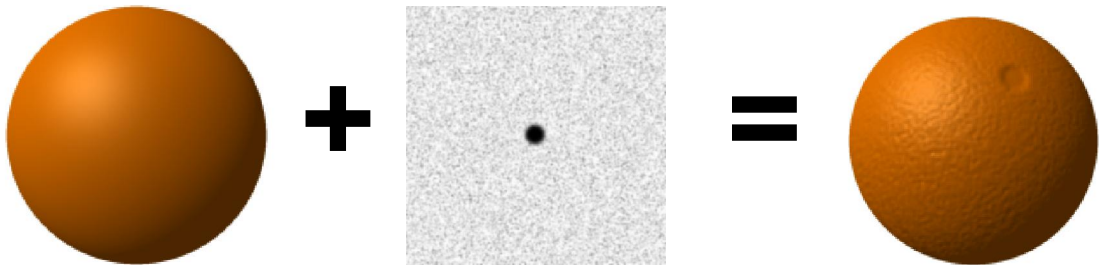
Alpha blending: Lim (2010), <http://bit.ly/6TsJrb>
Goon Creative, Maya Transparency Tutorial





Review [4]: Bump Mapping

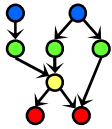
- Goal: Create Illusion of Textured Surface



Bump Mapping © 2010 Wikipedia
http://en.wikipedia.org/wiki/Bump_mapping

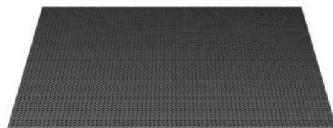
- Idea
 - * Start with regular smooth object
 - * Make height map (by hand and/or using program, *i.e.*, procedurally)
 - * Use map to perturb surface normals
 - * Plug new normals into illumination equation
- Tutorial for OpenGL (Baker, 2003): <http://bit.ly/fun4a5>





Review [5]: Displacement Mapping

- **Displacement Map: Similar to Bump Map – Contains Delta Values**



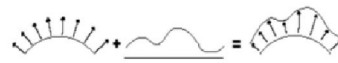
ORIGINAL MESH



DISPLACEMENT MAP



MESH WITH DISPLACEMENT



$$\mathbf{P}(u, v)$$

$$\mathbf{S}(u, v) = \frac{\partial \mathbf{P}(u, v)}{\partial u} \quad \mathbf{T}(u, v) = \frac{\partial \mathbf{P}(u, v)}{\partial v}$$

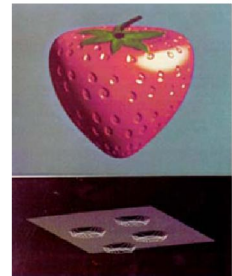
$$\mathbf{N}(u, v) = \mathbf{S} \times \mathbf{T}$$

■ Displacement

$$\mathbf{P}'(u, v) = \mathbf{P}(u, v) + h(u, v)\mathbf{N}(u, v)$$

■ Perturbed normal

$$\begin{aligned} \mathbf{N}'(u, v) &= \mathbf{P}'_u \times \mathbf{P}'_v \\ &= \mathbf{N} + h_u(\mathbf{T} \times \mathbf{N}) + h_v(\mathbf{S} \times \mathbf{N}) \end{aligned}$$



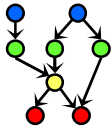
From Blinn 1976

Displacement Mapping © 2005 Wikipedia
http://en.wikipedia.org/wiki/Displacement_mapping

Adapted from slides © 1995 – 2009 P. Hanrahan, Stanford University
<http://bit.ly/hZfsjZ> (CS 348B)

- **Displacement Mapping: Uses Open GL Shading Language (GLSL)**
- **Tutorial using GLSL (Guinot, 2006): <http://bit.ly/dWXNya>**





Frank Pfenning

Professor of Computer Science
School of Computer Science
Carnegie Mellon University
<http://www.cs.cmu.edu/~fp/>

See also: *OpenGL: A Primer, 3^e (Angel)*
<http://bit.ly/hVcVWN>

Adapted from slides © 2003 F. Pfenning, Carnegie Mellon University
<http://bit.ly/g1J2nj>

Review [6]: OpenGL Shading (Overview)

● Set Up Point Light Sources

- Directional light given by "position" **vector**

```
GLfloat light_position[] = {-1.0, 1.0, -1.0, 0.0};
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```
- Point source given by "position" **point**

```
GLfloat light_position[] = {-1.0, 1.0, -1.0, 1.0};
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

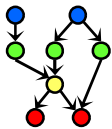
● Set Up Materials, Turn Lights On

```
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[]={0.8, 0.6, 0.4, 1.0};
GLfloat mat_shininess={20.0};
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialf(GL_FRONT, GL_SHININESS, mat_shininess);

glShadeModel(GL_SMOOTH); /*enable smooth shading */
glEnable(GL_LIGHTING); /* enable lighting */
glEnable(GL_LIGHT0); /* enable light 0 */
```

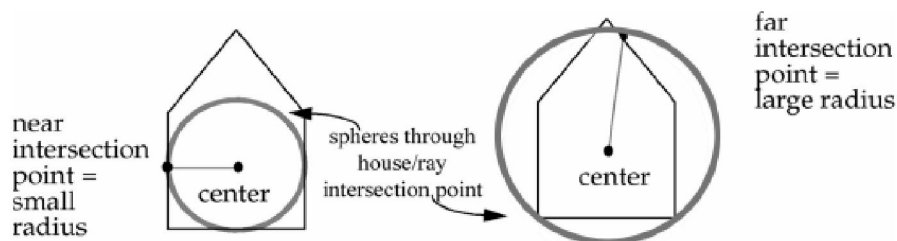
● Start Drawing (glBegin ... glEnd)





Review [7]: Texturing – Object Center Method

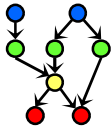
- ▶ When we treat the object intersection point as a point on a sphere, our “sphere” won’t always have the same radius



- ▶ 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
<http://bit.ly/hiSt0f> Reused with permission.





Review [8]: OpenGL Texturing

In initialization:

```
glGenTextures(...);  
glBindTexture( ... );  
glTexParameterf(...); glTexParameterf(...); ...  
glTexImage2D(...);  
glEnable(GL_TEXTURE_2D);
```

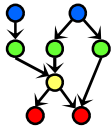
In display:

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

Adapted from slides

© 2007 Jacobs, D. W., University of Maryland





Review [9]: Mappings, Eberly 2^e

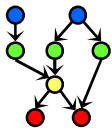
- Fine Surface Detail: Bump (§20.5 Eberly 2^e)
- Material Effects: Gloss (§20.6)
- Enclosing Volumes
 - * Sphere (§20.7)
 - * Cube (§20.8)
- Light
 - * Refraction for Transparency (§20.9)
 - * Reflection aka Environment (§20.10)
- Shadow
 - * Shadow Maps (§20.11, 20.13)
 - * Projective Textures (§20.12)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



Babylon 5

© 1993 – 1998 Warner Brothers Entertainment, Inc.





Acknowledgements



Nathan H. Bean

Instructor
Outreach Coordinator
Department of Computing and Information Sciences
Kansas State University
<http://bit.ly/gC3vwH>

Direct3D material from slides © 2006 – 2010 N. Bean, Kansas State University
<http://bit.ly/gC3vwH>



Randy Fernando

Executive Director
Mindful Schools
<http://www.randima.com>



Mark Kilgard

Principal System Software Engineer
Nvidia
<http://bit.ly/gdjLzR>

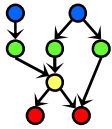
Andy van Dam

T. J. Watson University Professor of
Technology and Education & Professor of
Computer Science
Brown University
<http://www.cs.brown.edu/~avd/>



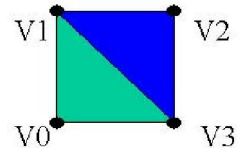
Cg material from figures © 2003 R. Fernando & M. Kilgard, Nvidia, from *The Cg Tutorial*
<http://bit.ly/59ffSR>





Review: Drawing in Direct3D

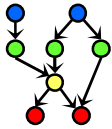
- Specify the material we wish to use for the following triangles
- Specify the texture we wish to use (if we want one or NULL if not)
- Set the stream source to our vertex buffer
- Set the FVF we will be using
- Set the index buffer we will be using
- Call the required `DrawPrimitive` function



```
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 );
}
```



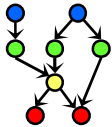


History

- 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 use of lighting effects such as bump mapping and shadowing, higher resolution color values. *Doom III* and *Half-Life 2* usher in a new era of realism

Adapted from slides © 2002 – 2003 van Dam *et al.*, Brown University
<http://bit.ly/fiYmje> Reused with permission.



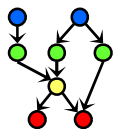


Fixed Function Pipeline

- 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
<http://bit.ly/fiYmje> Reused with permission.





Programmable Hardware



Crysis 2

© 2011 Electronic Arts, Inc. – <http://bit.ly/idNET9>



Starcraft II: Wings of Liberty

© 2010 Blizzard Entertainment, Inc. – <http://bit.ly/9B1qZp>



Rage & id Tech 5

© 2011 id Software, Inc. – <http://bit.ly/eR0m2B>

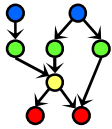


Unreal Tournament 3 & Steamworks

© 2010 Epic Games & Valve – <http://bit.ly/9QKAS7>

Inspired by slides © 2002 – 2003 van Dam *et al.*, Brown University
<http://bit.ly/fiYmje> Reused with permission.

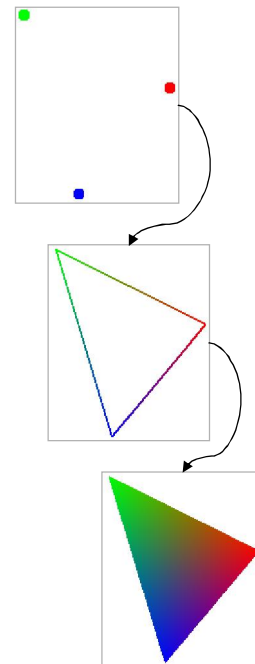




Quick Review: Interpolative Shading in OpenGL

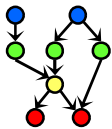
By default, GL will do the following:

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

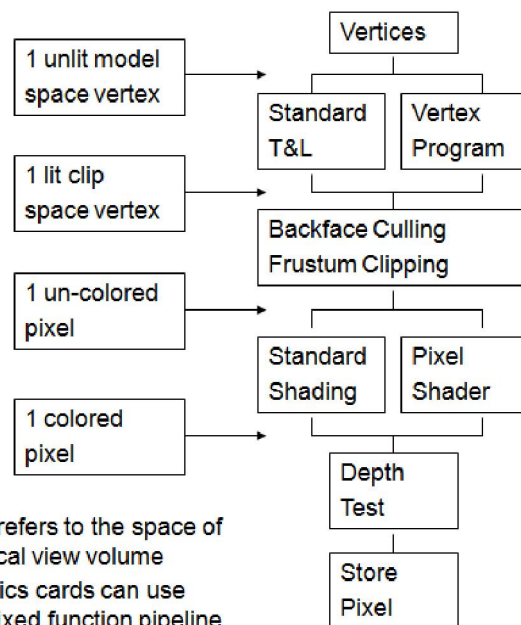


Adapted from slides © 2002 – 2003 van Dam *et al.*, Brown University
<http://bit.ly/fiYmje> Reused with permission.





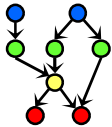
Programmable Hardware Pipeline



- 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

Adapted from slides © 2002 – 2003 van Dam *et al.*, Brown University
<http://bit.ly/fiYmj> Reused with permission.



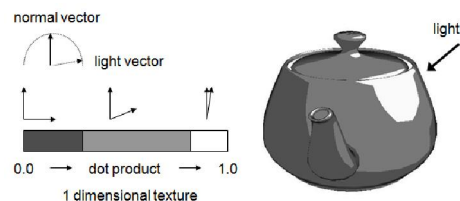


Example: Cartoon Shader & Utah Teapot

- 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

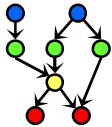


JSRF © 2002 SEGA Games, Inc.
<http://youtu.be/-J3bmKLw5hw>



Adapted from slides © 2002 – 2003 van Dam *et al.*, Brown University
<http://bit.ly/fiYmje> Reused with permission.



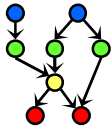


What is Cg?

- Cg is a C-like language that the graphics card compiles in to 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/> has the technical documentation for Cg
- Cg is actually an abstraction of the more primitive assembly language that the programmable hardware originally supported

Adapted from slides © 2002 – 2003 van Dam *et al.*, Brown University
<http://bit.ly/fiYmje> Reused with permission.



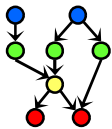


Cg Tips

- 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/> for some helpful documents

Adapted from slides © 2002 – 2003 van Dam *et al.*, Brown University
<http://bit.ly/fiYmje> Reused with permission.



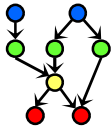


Cg: Big Picture

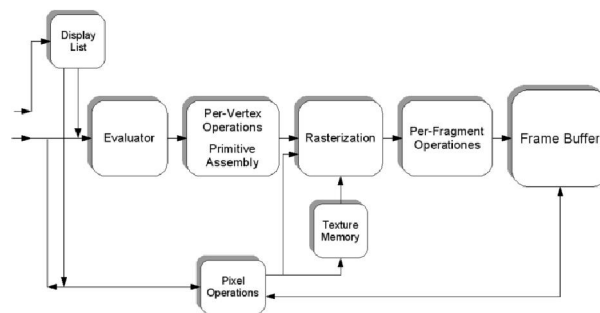
- Write a .cg file. This will invariably take some sort of information as a parameter to its `main()` function
 - Note that this `main()` is not compiled by gcc (or any C/C++ compiler). That would generate a symbol conflict, among other things. It is only processed by NVidia's Cg compiler
- Write a class 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

Adapted from slides © 2002 – 2003 van Dam *et al.*, Brown University
<http://bit.ly/fiYmje> Reused with permission.





Shader Languages Overview

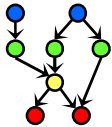


OpenGL State Machine (Simplified) from Wikipedia: *Shader*
<http://bit.ly/fi8tBP>

- **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.

© 2009 Koen Samyn
<http://knol.google.com/k/hisl-shaders>





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)

```

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

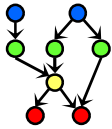
    return Out;
}

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

```

Adapted from slide 2003 Wolfgang Engel, <http://www.wolfgang-engel.info>
 Visiondays 2003, <http://bit.ly/hhkANP>





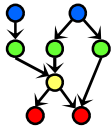
HLSL [2]: Code Example

```
// 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/hsl-shaders>





Cg [1]

- **Q: What Is Cg?**

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

Cg

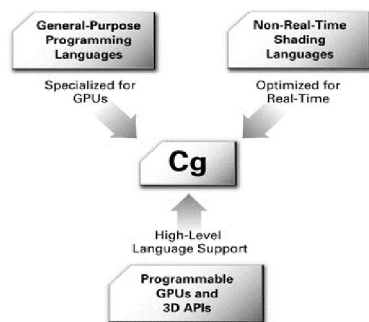


Figure 1-10, Fernando & Kilgard (2003)

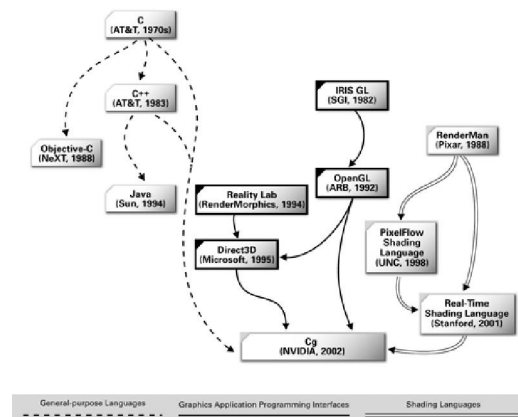
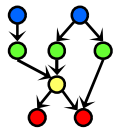


Figure 1-11, Fernando & Kilgard (2003)

- **Intro Slides for Cg Tutorial (Online Book): <http://bit.ly/59ffSR>**

© 2003 R. Fernando & M. Kilgard. *The Cg Tutorial*.
<http://bit.ly/59ffSR>





Cg [2]

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

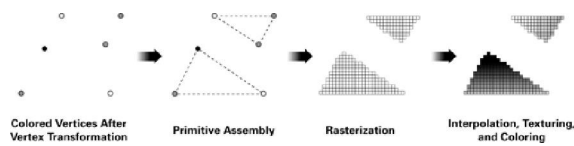


Figure 1-6, Fernando & Kilgard (2003)

- **Programmable Graphics Pipeline**

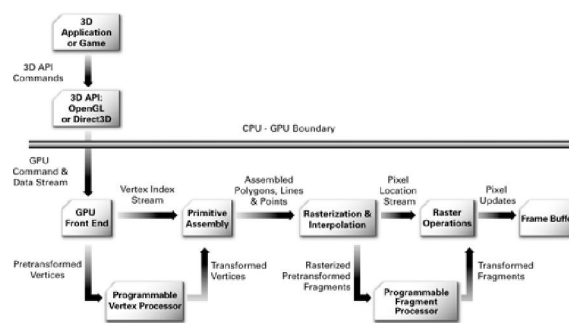
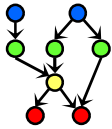


Figure 1-7, Fernando & Kilgard (2003)

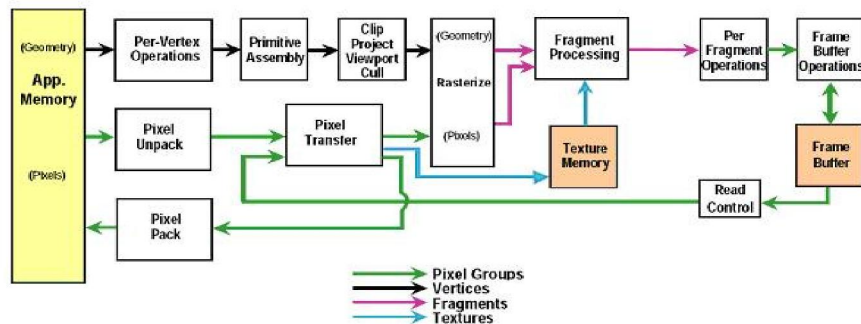
© 2003 R. Fernando & M. Kilgard. *The Cg Tutorial*.
<http://bit.ly/59ffSR>





GLSL [1]: Building on Top of OpenGL

- How It Used to Be (in OpenGL)

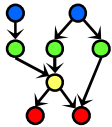


OpenGL 1.5 Fixed Function Pipeline (see [OpenGL Reference Manual](#))
 "GLSL: An Introduction" © 2004 F. Rudolf, NeHe Productions – <http://bit.ly/gi9g47>

- 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

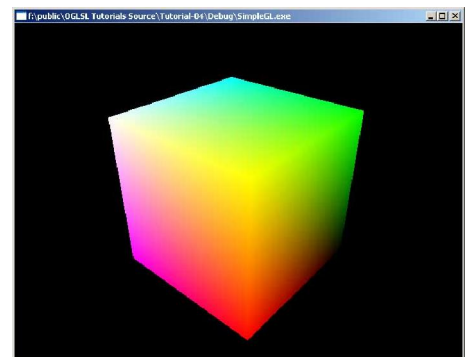
```
varying float xpos;
varying float ypos;
varying float zpos;
void main(void)
{
    xpos = clamp(gl_Vertex.x,0.0,1.0);
    ypos = clamp(gl_Vertex.y,0.0,1.0);
    zpos = clamp(gl_Vertex.z,0.0,1.0);

    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Fragment Shader

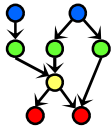
```
varying float xpos;
varying float ypos;
varying float zpos;

void main (void)
{
    gl_FragColor = vec4 (xpos, ypos, zpos, 1.0);
}
```



© 2003 – 2005 M. Christen, ClockworkCoders.com
<http://bit.ly/et5qOp>

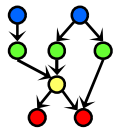




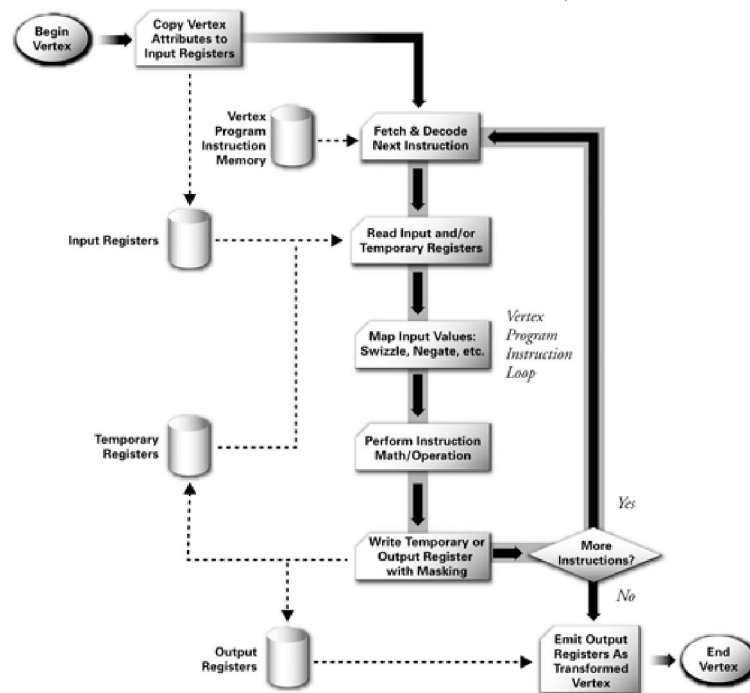
Vertex Shaders [1]

- **“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



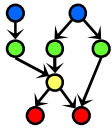


Vertex Shaders [2]: Cg/HLSL-Style Pipeline



© 2003 R. Fernando & M. Kilgard. *The Cg Tutorial*.
<http://bit.ly/59ffSR>





Vertex Shaders [3]: GLSL Example

- Diffuse Shader (NeHe GLSL Example)

Diffuse Shader

The diffuse lighting model is one common used lighting model. It's a little bit harder to implement:

Vertex Shader:

```
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;

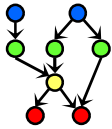
    // Transforming The Vertex Position To ModelView-Space
    vec4 vertex_in_modelview_space = gl_ModelViewMatrix * gl_Vertex;

    // 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>

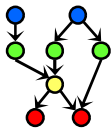




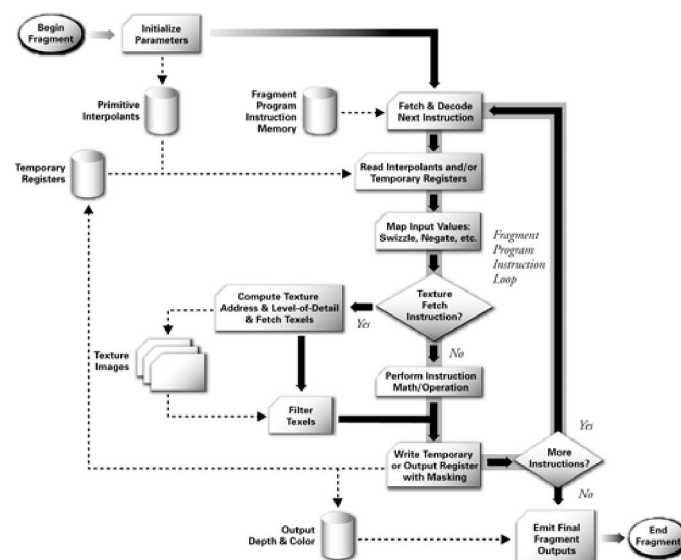
Pixel & Fragment Shaders [1]

- **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



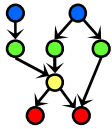


Pixel & Fragment Shaders [2]: Cg/HLSL-Style Pipeline



© 2003 R. Fernando & M. Kilgard. *The Cg Tutorial*.
<http://bit.ly/59ffSR>





Pixel & Fragment Shaders [3]: GLSL Example

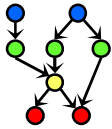
- Consider BRDF (Especially $N \leftarrow L$) From Last Lecture

```
varying vec3 normal;  
varying vec3 vertex_to_light_vector;  
  
void main()  
{  
    // Defining The Material Colors  
    const vec4 AmbientColor = vec4(0.1, 0.0, 0.0, 1.0);  
    const vec4 DiffuseColor = vec4(1.0, 0.0, 0.0, 1.0);  
  
    // Scaling The Input Vector To Length 1  
    vec3 normalized_normal = normalize(normal);  
    vec3 normalized_vertex_to_light_vector = normalize(vertex_to_light_vector);  
  
    // Calculating The Diffuse Term And Clamping It To [0;1]  
    float DiffuseTerm = clamp(dot(normal, vertex_to_light_vector), 0.0, 1.0);  
  
    // Calculating The Final Color  
    gl_FragColor = AmbientColor + DiffuseColor * DiffuseTerm;  
}
```

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

© 2004 F. Rudolf, NeHe Productions
<http://bit.ly/gi9g47>

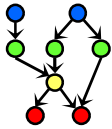




Summary

- **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

