Lecture 11of 41

Surface Detail 2 of 5: Textures
OpenGL Shading

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

KSOL course pages: http://bit.ly/hGvXlH / 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: Sections 2.6.3, 20.3 – 20.4, Eberly *2e* – see **http://bit.ly/ieUq45**
Next class: Sections 20.5 – 20.13, Eberly *2e*
Brown CS123 slides on Polygons/Texture Mapping – **http://bit.ly/h2VZn8**
Wayback Machine archive of Brown CS123 slides: **http://bit.ly/gAhJbh**
CMU 15-462 slides on OpenGL Shading – **http://bit.ly/g1J2nj**

Lecture Outline

- Reading for Last Class: §2.7, Eberly 2e; Direct 3D Handout
- Reading for Today: §2.6.3, 20.3 – 20.4, Eberly 2e
- References: Gröller & Jeschke (2002), Isenberg (2005), Jacobs (2007)
- Reading for Next Class: §20.5 – 20.13, Eberly 2e
- Last Time: Intro to Illumination and Shading
  - Local *vs.* global models
  - Illumination (vertex shaders) *vs.* shading (fragment/pixel shaders)
  - Bidirectional reflectance distribution function (BRDF) $\rho(p, \omega_i, \omega_o, \lambda)$
  - Phong illumination equation: introduction to shading
- Texture Mapping Explained
  - Definitions
  - Design principles
- Texture Pipeline
- Using Simple Intermediate Surfaces (Cylinder, Sphere, Plane, Box)
- OpenGL Shading: Flat Shading, Smooth Shading (Gouraud)

Where We Are

Lecture	Topic	Primary Source(s)
0	Course Overview	Chapter 1, Eberly 2e
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, 16[1], 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: Illumination & Shading

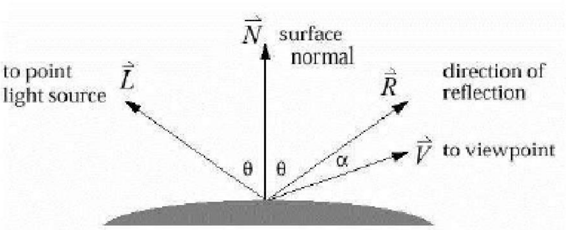
- ***Lighting***, or ***illumination***, is the process of computing the intensity and color of a sample point in a scene as seen by a viewer
  - lighting is a function of the geometry of the scene (including the model, lights and camera and their spatial relationships) and material properties
- ***Shading*** is the process of *interpolation* of color at points in-between those with known lighting or illumination, typically vertices of triangles or quads in a mesh
  - used in many real time graphics applications (e.g., games) since calculating illumination at a point is usually expensive
- On the GPU, lighting is calculated by a ***vertex shader***, while shading is done by a fragment or ***pixel shader***

Review: Phong Illumination Equation

- The full Phong model is a combination of the Lambertian and specular terms (summing over all the lights)

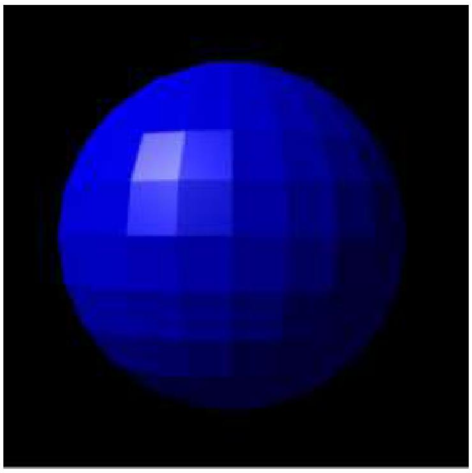
$$I_\lambda = i_{a_\lambda} k_{a_\lambda} O_{d_\lambda} + \sum_{m \in lights} f_{att} i_{d_\lambda} \left[k_{d_\lambda} (\mathbf{n} \cdot \boldsymbol{L}_{\boldsymbol{m}}) O_{d_\lambda} + k_{s_\lambda} (\mathbf{R}_{\boldsymbol{m}} \cdot \mathbf{V})^\alpha O_{s_\lambda} \right]$$

- Subscript s represents specular (so k_s) would be the specular coefficient
- $\boldsymbol{R_m}$ is the reflected direction of the light ray about the surface normal
- f_{att} is the lighting attenuation function
  - function of distance from the light



Review:
Flat/Constant Shading

- We define a normal at each polygon (not at each vertex)
- **Lighting:** Evaluate the lighting equation at the center of each polygon using the associated normal
- **Shading:** Each sample point on the polygon is given the calculated lighting value



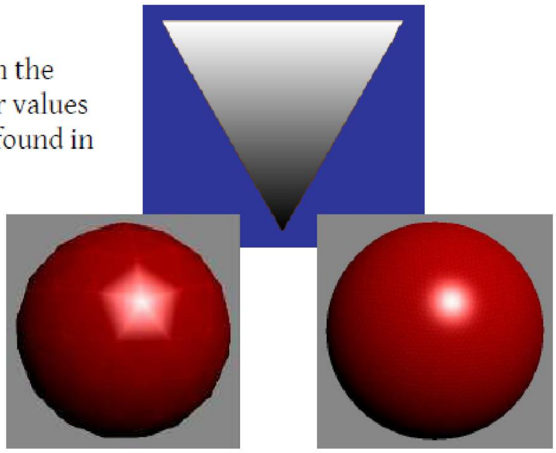
GL_CONSTANT

Adapted from slides © 2010 van Dam *et al.*, Brown University
http://bit.ly/hiSt0f Reused with permission.

Review: Gouraud Shading

- We define a normal vector at each *vertex*
- **Lighting:** Evaluate the lighting equation at each vertex using the associated normal vector
- **Shading:** Each sample point's color on the polygon is interpolated from the color values at the polygon's vertices which were found in the lighting step

GL_SMOOTH



Review: Phong Shading

- Each vertex has an associated normal vector
- **Lighting:** Evaluate the lighting equation at each vertex using the associated normal vector
- **Shading:** For every sample point on the polygon we interpolate the normals at vertices of the polygon and compute the color using the lighting equation with the interpolated normal at each interior pixel

OpenGL implementation?
Stay tuned…

Adapted from slides © 2010 van Dam *et al.*, Brown University
http://bit.ly/hiSt0f Reused with permission.

Source Material on Texturing: Gröller & Jeschke (Vienna Tech)

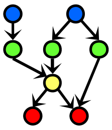
Texturing

Eduard Gröller
(today: Stefan Jeschke)

Institute of Computer Graphics and Algorithms

Vienna University of Technology

Adapted from slides



Other Source Material on Texture Mapping

David W. Jacobs

CMSC 427 Computer Graphics
University of Maryland – College Park (UMD)
Fall 2007
Course: http://bit.ly/fXVA1A
Instructor: http://www.cs.umd.edu/~djacobs

Associate Professor, Computer Science Department and UMIACS, at the University of Maryland.

Tobias Isenberg

Scientific Visualization and Computer Graphics Group
Department of Mathematics and Computing Science
University of Groningen

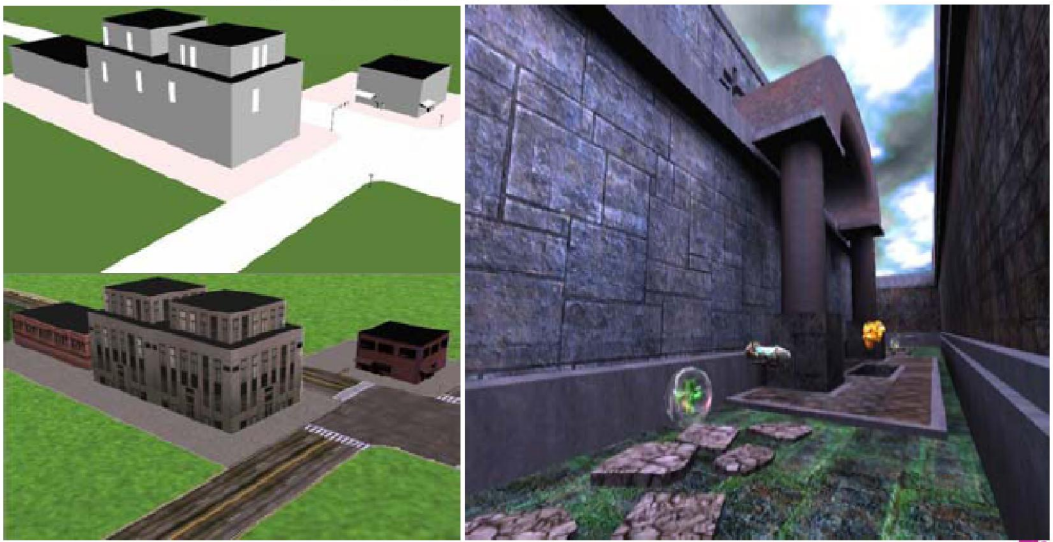
Formerly
Graphics Jungle Lab
Department of Computer Science
University of Calgary

CPSC 599.64/601.64 Computer Graphics
Fall 2005
Course: http://bit.ly/idD2qX
Instructor: http://www.cs.rug.nl/~isenberg

- Computer Graphics II lecture by Stefan Schlechtweg, Department of Simulation and Graphics, Otto-von-Guericke University of Magdeburg, Germany
- CPSC 407 and CPSC 453 lectures by Brian Wyvill, Department of Computer Science, University of Calgary, Canada

Why Texturing?

- Idea: enhance visual appearance of plain surfaces by applying fine structured details



Eduard Gröller, Stefan Jeschke 1

Adapted from slides
© 2002 Gröller, E. & Jeschke, S. Vienna Institute of Technology

Introduction [1]: Motivation

- so far: detail through polygons & materials
- example: brick wall
- problem: many polygons & materials needed for detailed structures $\rightarrow$ inefficient for memory and processing
- new approach necessary: texture mapping
- introduced by Ed Catmull (1974), extended by Jim Blinn (1976)

Adapted from slides

Introduction [2]: Properties and their Mappings

- several properties can be modified
  - color: diffuse component of surface
  - reflection: specular component of surface to simulate reflection (environment mapping)
  - normal vector: simulate 3D surface structure (bump mapping)
  - actual surface: raise/lower points to actually modify surface (displacement mapping)
  - transparency: make parts of a surface entirely or to a certain degree transparent

Concerning Textures

- Pattern of Intensity and color.
  - Can be generalized to 3D texture.
- How do we get them?
  - Take pictures.
  - Write a program (procedural textures).
  - Synthesize from examples
- How do we apply them? (Texture mapping)
  - Specify a mapping from texture to object.
  - Interpolate as needed.
  - This can be a challenging problem, but we'll consider simpler version.

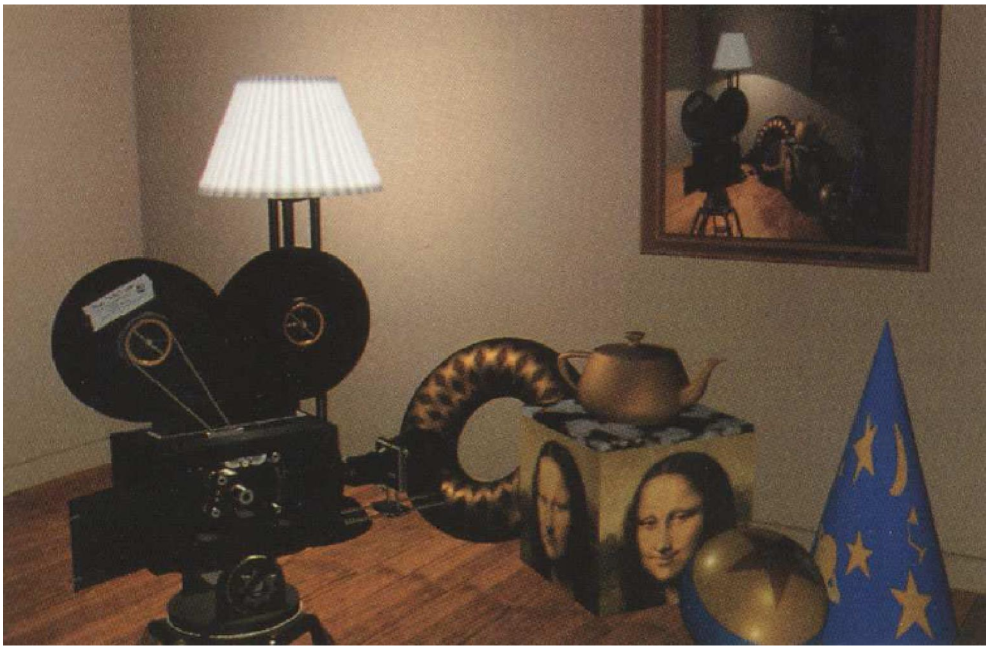
Adapted from slides

Without Textures

Adapted from slides

From *Computer Graphics: Principles and Practice*

With Textures



Adapted from slides

From *Computer Graphics: Principles and Practice*

Texture Image

Adapted from slides

Acknowledgements

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

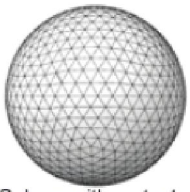
Texture Mapping

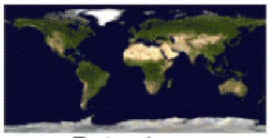
Beautification of Surfaces

Adapted from slides © 1997 – 2010 van Dam *et al.*, Brown University
http://bit.ly/hiSt0f **Reused with permission.**

Texture Mapping Overview [1]: Technique

- Texture mapping:
  - Implemented in hardware on every gpu
  - Simplest surface detail hack, dating back to the '60s GE flight simulator and its terrain generator
- Technique:
  - "Paste" photograph or bitmap (the texture, for example: a brick pattern, a wood grain pattern, a sky with clouds) on a surface to add detail without adding more polygons.
  - Map texture onto the surface get the surface color or alter the object's surface color
  - Think of texture map as stretchable contact paper


Sphere with no texture


Texture image


Sphere with texture

Adapted from slides © 2010 van Dam *et al.*, Brown University
http://bit.ly/hiSt0f Reused with permission.

Texture Mapping Overview [2]: Motivation

- How do we add more detail to a model?
  - Add more detailed geometry; more, smaller triangles:
    - Pros: Responds realistically to lighting, other surface interaction
    - Cons: Difficult to generate, takes longer to render, takes more memory space
  - Map a texture to a model:
    - Pros: Can be stored once and reused, easily compressed to reduce size, rendered very quickly, very intuitive to use, especially useful on far-away objects, terrain, sky,...
    - Cons: Very crude approximation of real life. Texture mapped but otherwise unaltered surfaces still look smooth.
- What can you put in a texture map?
  - Diffuse, ambient, specular, or any kind of color
  - Specular exponents, transparency or reflectivity coefficients
  - Surface normal data (for bump mapping or normal mapping)
  - Projected reflections or shadows

Texture Mapping Overview [3]: Mappings

- A function is a mapping
  - Takes any value in the domain as an input and outputs ("maps it to") one unique value in the co-domain.
- Mappings in "Intersect": linear transformations with matrices
  - Map screen space points (input) to camera space rays (output)
  - Map camera space rays into world space rays
  - Map world space rays into un-transformed object space for intersecting
  - Map intersection point normals to world space for lighting
- Mapping a texture:
  - Take points on the surface of an object (domain)
  - Return an entry in the texture (co-domain)

Texture Mapping How-To [1]: Goals and Texture Elements (Texels)

- **texture**: typically 2D pixel image
- **texel**: pixel in a texture
- determines the appearance of a surface
- procedure to map the texture onto the surface needed
  - easy for single triangle
  - complex for arbitrary 3D surface
- goal: find easy way to do this mapping

Adapted from slides

Texture Mapping How-To [2]: Adapting Polygons-to-Pixels Pipeline

- rendering pipeline slightly modified to use new texture mapping function
- algorithm: for each pixel to be rendered
  - find depicted surface point
  - find point in texture (texel) that corresponds to surface point
  - use texel color to modify the pixel's shading

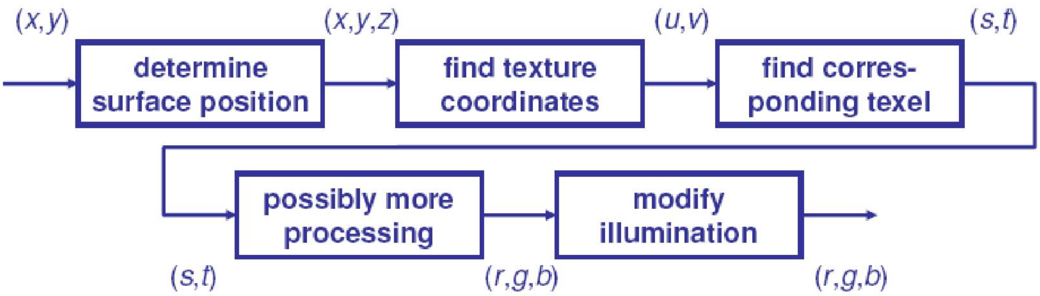
Texture Mapping How-To [3]: Mapping Definition

- 2D texture: function that maps points on the (u, v) plane to (r, g, b) values:
  $(r, g, b) = c_{tex}(u, v)$
- texture mapping function maps (u, v) values to (x, y, z) positions on objects:
  $(x, y, z) = F_{map}(u, v)$
- we need to solve the inverse function to find (u, v) values for a (x, y, z) position:
  $(u, v) = F_{map}^{-1}(x, y, z)$

$u = s(x, y, z)$
$v = t(x, y, z)$

Texture Mapping How-To [4]: General Procedure

- general texture mapping pipeline:



1. compute texture color for surface point
2. use to modify parameters in Phong illumination

Texture Mapping How-To [5]: Projective Textures, Functions

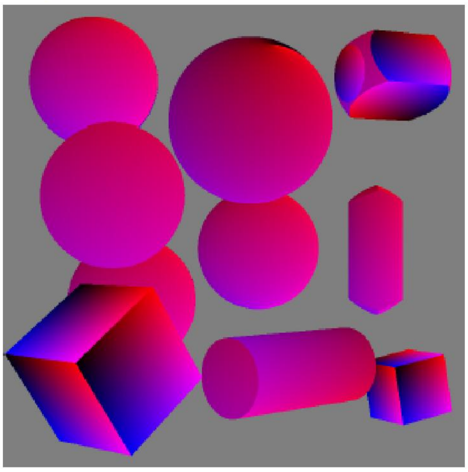
- goal: derive texture coordinates from 3D point
- P: $\Re^3 \rightarrow \Re^2$, so $P(x, y, z) = (u, v)$
- several typical possibilities
  - (manual) parameterization of the surface
  - use of inherent (u, v) coordinates (e.g., freeform surfaces or primitive shapes)
  - two step technique

Texture Mapping How-To [6]: (Manual) Surface Parameterization

- simplest technique
- specification of texture coordinates during modeling
- (u, v) coordinates specified for all vertices of a polygon
- interpolation between these values for points inside the polygon
  (e.g. barycentric interpolation for triangles)

Texture Mapping How-To [7]: Inherent (u, v) Coordinates

- (u, v) coordinates derived from parameter directions of surface patches (e.g., Bézier and spline patches)
- obvious (u, v) coordinates derived for primitive shapes (e.g., boxes, spheres, cones, cylinders, etc.)
- used as defaults



Adapted from slides

Two-Step Approach [1]: Duality (Again)

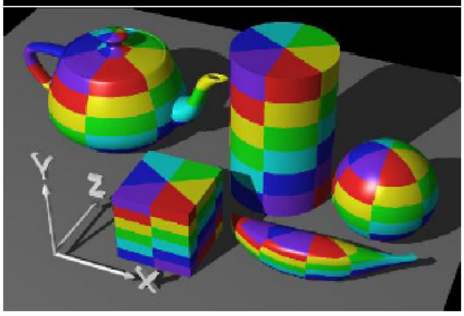
- two steps:
  - mapping of 2D texture coordinates onto simple 3D surface (s-mapping)
  - mapping of the now 3D texture pattern onto complex object (o-mapping)
- in practice – inverse approach:
  - mapping of object point onto simple surface
    O: $f(x_o, y_o, z_o) = (x_i, y_i, z_i)$
  - mapping of surface point onto texture
    S: $f(x_i, y_i, z_i) = (u, v)$

Two-Step Approach [2]: Example – Cylindrical Mapping

- mapping onto cylinder surface given by height h_0 and angle θ_0

$$S:(\theta,h)\to(u,v)=\left(\frac{r}{c}(\theta-\theta_0),\frac{1}{d}(h-h_0)\right)$$

  using scaling factors c, d, and the radius r
- discontinuity along one line parallel to center axis



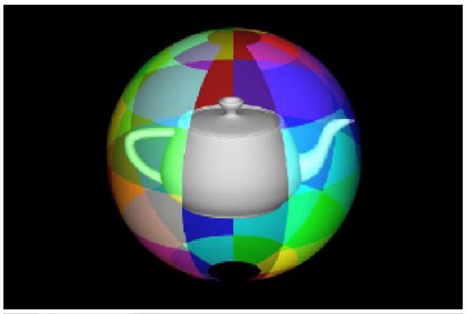
from R. Wolfe: *Teaching Texture Mapping*

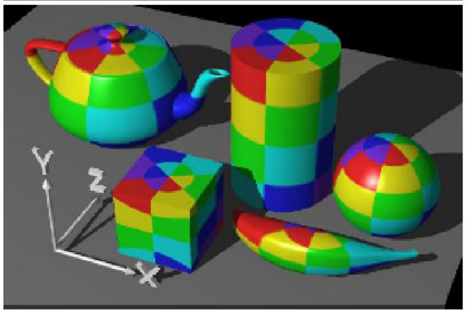
Two-Step Approach [3]: Example – Spherical Mapping

- mapping onto surface of a sphere given by spherical coordinates

$$S:(r,\phi,\theta)\rightarrow(u,v)=\left(\frac{\theta}{2\pi},\frac{(\pi/2)+\phi}{\pi}\right)$$

- no non-distorting mapping possible between plane and sphere surface





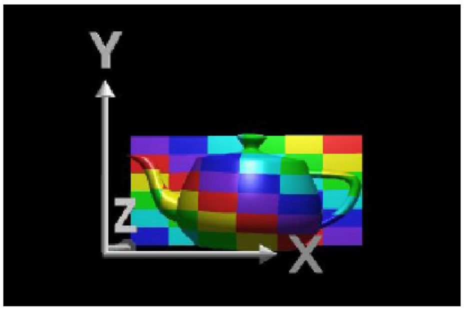
from R. Wolfe: *Teaching Texture Mapping*

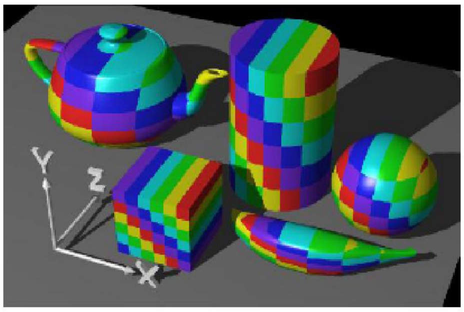
Two-Step Approach [4]: Example – Planar Mapping

- mapping onto planar surface given by position vector $\vec{v}_0$ and two vectors $\vec{s}$ and $\vec{t}$

$$S:(x,y,z)\rightarrow(u,v)=\left(\frac{\vec{v}\cdot\vec{s}}{k},\frac{\vec{v}\cdot\vec{t}}{k}\right)$$

- scaling factor k and $\vec{v} = \vec{P}_i - \vec{v}_0$ (describes point position w.r.t. the origin of the plane)

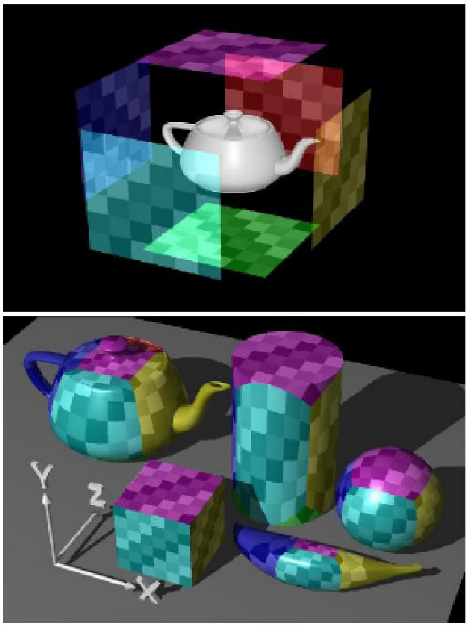




from R. Wolfe: *Teaching Texture Mapping*

Two-Step Approach [5]: Example – Cuboid/Box Mapping

- enclosing box is usually axis-parallel bounding box of object
- six rectangles onto which the texture is mapped
- similar to planar mapping

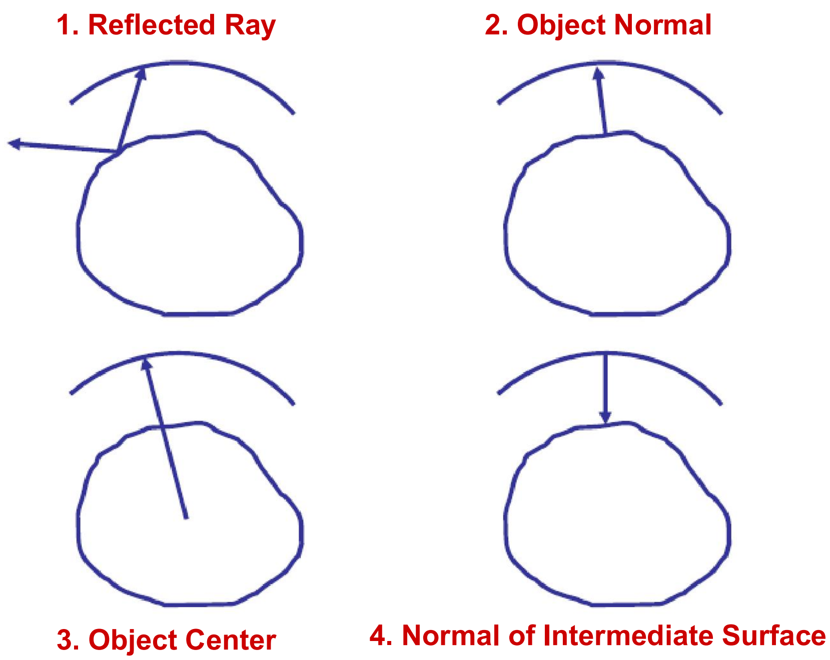


from R. Wolfe: *Teaching Texture Mapping*

O Mapping [1]: Object-to-Surface

- necessary for all named techniques
- four methods
  - *reflected ray*: trace a ray from viewer to object and reflect it onto the intermediate surface
  - *object normal*: intersection of normal vector of object with intermediate surface
  - *object center*: intersection of ray from object center through the object surface with the intermediate surface
  - *normal of intermediate surface*: trace this normal vector towards the object and determine intersection with it

O Mapping [2]: Illustrations



Adapted from slides

Correspondence Functions [1]: Texture Coordinates

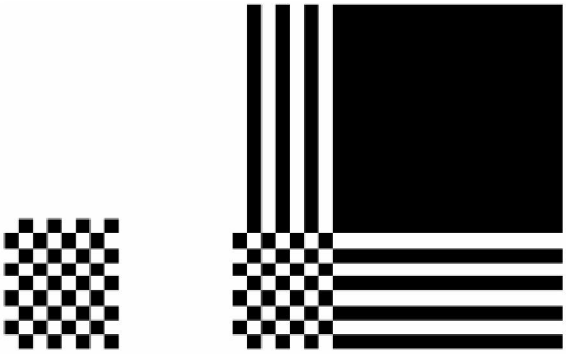
- projector functions yield (*u*, *v*) coordinates in texture parameter space
- typically values of *u* and *v* in [0, 1]
- correspondence functions transform these into texel positions
- rotations, translations, scaling possible
- in most simple cases only scaling necessary

u = s(x, y, z)
v = t(x, y, z)

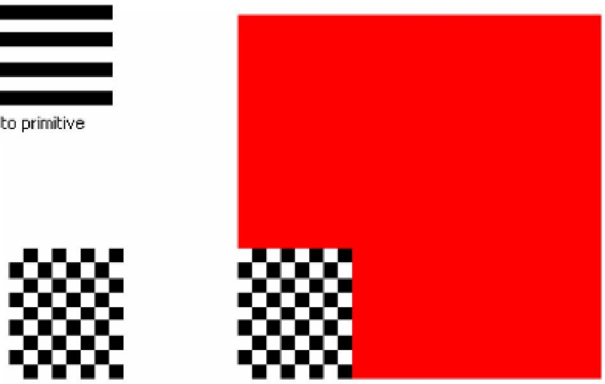
Correspondence Functions [2]: Tiling, Mirroring, Clamping, Borders

- problem: what happens outside of [0, 1]?
- typical approaches
  - texture repetition (tiling) using modulo function
  - texture mirroring – better continuity at texture seams
  - clamping: repeat the last value of the texture edges for values outside of [0, 1]
  - border color: use a specified color for all non-defined values

Correspondence Functions [3]: Clamping and Borders Illustrated


Texture   Clamped texture applied to primitive

from Microsoft Developer Network


Texture   Texture with red border applied to primitive

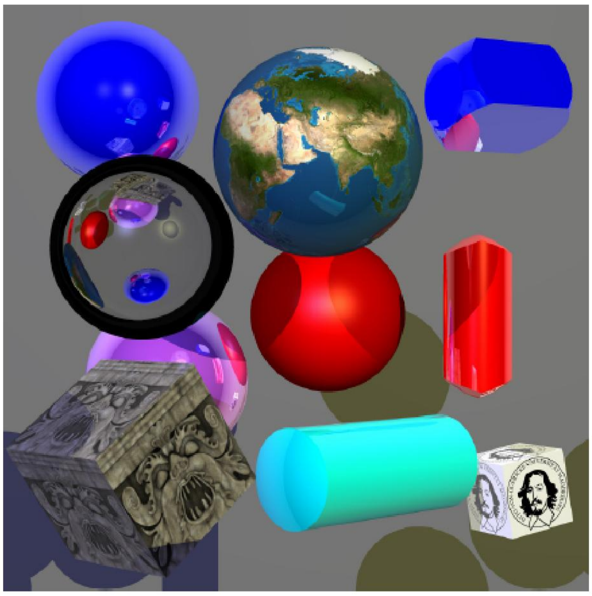
Adapted from slides

Application of Texture Values: Combining Texturing and Lighting

- from an (x, y, z) position we derived an (r, g, b) color value from the texture, potentially with α transparence value
- is typically used to modify illumination
- methods:
  - replace: surface color value is replaced with texture color
  - decal: α blending of texture and original color
  - modulate: multiplication of original color value with texture color

The End…?

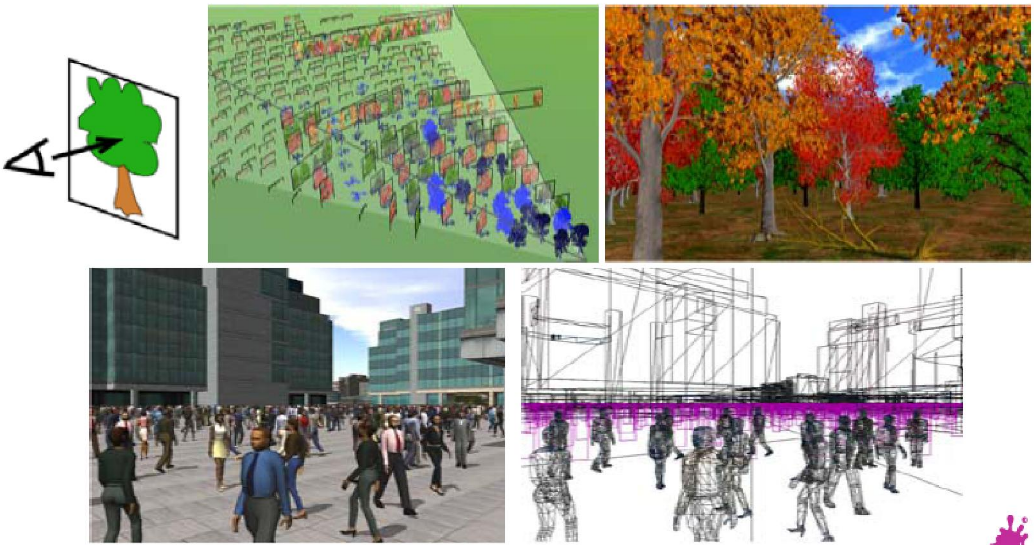
Done!   … well, almost



Adapted from slides

Surface Detail: Imitating Complexity by Texturing Smooth Objects

- Also possible: model very complex objects just by using simple textured geometry



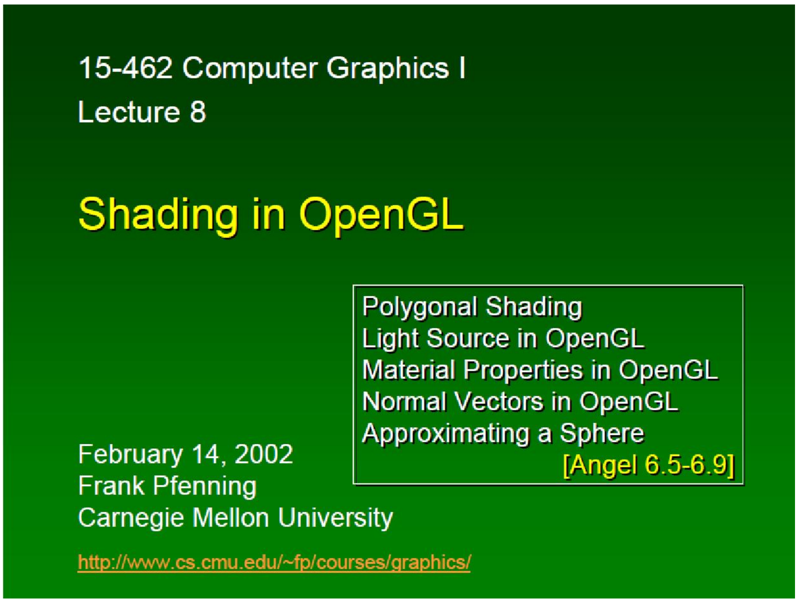
Adapted from slides

More on Shading (Surface Detail 2, 4, 5)

- Shading in OpenGL
  - Flat/constant: GL_CONSTANT
  - Gouraud: GL_SMOOTH
- Shading Languages
  - Renderman Shading Language (RSL) – http://bit.ly/g229q4
  - OpenGL Shading Language (OGLSL or GLSL) – http://bit.ly/fX8V0Y
  - Microsoft High-Level Shading Language (HLSL) – http://bit.ly/eVnjp5
  - nVidia Cg – http://bit.ly/ewoRic
- Vertex *vs.* Pixel Shaders
- How to Write Shaders

Source Material on OpenGL Shading

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

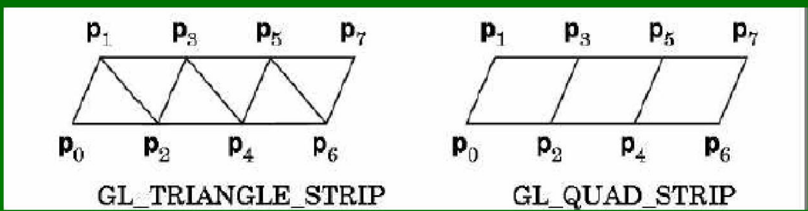


Shading in OpenGL [1]: Flat Shading

- Normal: given explicitly before vertex

```
glNormal3f(nx, ny, nz);
glVertex3f(x, y, z);
```

- Shading constant across polygon
- Single polygon: first vertex
- Triangle strip:Vertex n+2 for triangle n

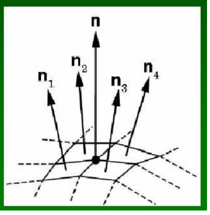


http://bit.ly/g1J2nj

Shading in OpenGL [2]: Interpolative (aka Smooth), Gouraud

- **Interpolative Shading**
  - Enable with glShadeModel(GL_SMOOTH);
  - Calculate color at each vertex
  - Interpolate color in interior
  - Compute during scan conversion (rasterization)
  - Much better image (see Assignment 1)
  - More expensive to calculate
- **Gouraud Shading**
  - Special case of interpolative shading
  - How do we calculate vertex normals?
  - Gouraud: average all adjacent face normals

$$n = \frac{n_1 + n_2 + n_3 + n_4}{|n_1 + n_2 + n_3 + n_4|}$$

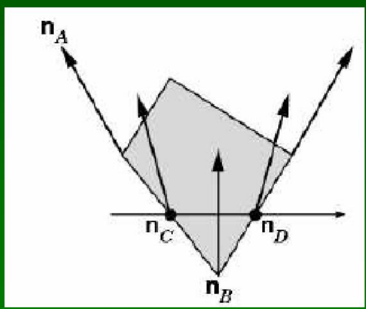


  - Requires knowledge about which faces share a vertex

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

Shading in OpenGL [3]: Phong Shading

- Interpolate normals rather than colors
- Significantly more expensive
- Mostly done off-line (not supported in OpenGL)



... kind of

-WHH

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

Shading in OpenGL [5]: Specifying & Enabling Light Sources

- **Enabling Light Sources**
  - Lighting in general must be enabled
    ```
    glEnable(GL_LIGHTING);
    ```
  - Each individual light must be enabled
    ```
    glEnable(GL_LIGHT0);
    ```
  - OpenGL supports at least 8 light sources
- **Specifying Point Light Source**
  - Use vectors {r, g, b, a} for light properties
  - Beware: light source will be transformed!

```
GLfloat light_ambient[] = {0.2, 0.2, 0.2, 1.0};
GLfloat light_diffuse[] = {1.0, 1.0, 1.0, 1.0};
GLfloat light_specular[] = {1.0, 1.0, 1.0, 1.0};
GLfloat light_position[] = {-1.0, 1.0, -1.0, 0.0};
glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

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

Shading in OpenGL [6]: Global Ambient Light

- Set ambient intensity for entire scene

```
GLfloat al[] = {0.2, 0.2, 0.2, 1.0};
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, al);
```

- The above is default
- Also: local vs infinite viewer

```
glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER,
                    GL_TRUE);
```

- More expensive, but sometimes more accurate

http://bit.ly/g1J2nj

Shading in OpenGL [7]: Point Sources vs. Directional

- **Directional Lights versus Point Lights**
  - 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);
    ```

- **Spotlights: Special Case of Point Lights**
  - Create point source as before
  - Specify additional properties to create spotlight

    ```
    GLfloat sd[] = {-1.0, -1.0, 0.0};
    glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, sd);
    glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 45.0);
    glLightf(GL_LIGHT0, GL_SPOT_EXPONENT, 2.0);
    ```


http://bit.ly/g1J2nj

Shading in OpenGL [8]: Example Material Properties

```
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 */
```

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

Summary

- Last Time: Intro to Illumination and Shading
  - Local *vs.* global models
  - Illumination (vertex shaders) *vs.* shading (fragment/pixel shaders)
  - Phong illumination: derivation of ambient, diffuse, specular terms
  - Introduction to shading
- Texturing: Adding Detail, Raster Image, Color, *etc.* to CG Model
- Texture Pipeline
  - Part of polygons-to-pixels
  - Uses same spaces (coordinate systems) and more
- Using Simple Intermediate Surfaces (Cylinder, Sphere, Plane, Box)
- OpenGL Shading: Flat *aka* Constant, Interpolative (Specifically, Gouraud)
- Next: Patterns, Procedural Textures, Anisotropic Filtering
- References: Gröller & Jeschke (2002), Isenberg (2005), Jacobs (2007)

Terminology

- **Texture Map / Texture Mapping**
  - **Method of adding surface detail to CGI or 3-D model (Wikipedia)**
  - **Kinds of surface detail**
    - **Detail: roughness, grain, bumps/dimples, *etc.***
    - **Surface texture: finish, veneer, *etc.* (represented by raster image)**
    - **Color: monochrome, patterns, polychromatic**
- **Coordinate Systems (Spaces)**
  - **Model / Object: 3-D (x, y, z)**
  - **World / Scene: 3-D (x, y, z)**
  - **Camera / Eye: 3-D (u, v, n)**
  - **Window / Screen: 2-D (u, v)**
  - **Texture: 1-D, 2-D, or 3-D; (s, t) for 2-D**
- **Texture Pipeline – End-to-End System for Calculating, Applying Textures**
- **Gouraud Shading – Interpolative Shading with Color Interpolation**