Computer Games

CSCI-GA.3033-010
Spring 2012

Hubertus Franke

( frankeh@cims.nyu.edu )
Thanks

Many thanks to:

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

For allowing the reuse of his excellent course material that he created for the Eberle book:

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

Jim Foley  
Professor, College of Computing &  
Stephen Fleming Chair in  
Telecommunications  
Georgia Institute of Technology

Andy van Dam  
T. J. Watson University Professor of  
Technology and Education &  
Professor of Computer Science  
Brown University

Steve Feiner  
Professor of Computer Science &  
Director, Computer Graphics and User  
Interfaces Laboratory  
Columbia University

John F. Hughes  
Associate Professor of Computer  
Science  
Brown University
OpenGL Fixed Function Pipeline

- OpenGL FFP Diagram (for v1.5)

OpenGL 1.5 Fixed Function Pipeline (see OpenGL Reference Manual)
Lecture 4

Lights and Textures
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*.

Adapted from slides © 2010 van Dam et al., Brown University
Review:
Phong Illumination Equation

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

\[ I_\lambda = i_{a\lambda} k_{a\lambda} O_{d\lambda} + \sum_{m \in \text{lights}} f_{\text{att}} i_{d\lambda} \left[ k_{d\lambda} (\mathbf{n} \cdot \mathbf{L}_m) O_{d\lambda} + k_{s\lambda} (\mathbf{R}_m \cdot \mathbf{V})^\alpha O_{s\lambda} \right] \]

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

Adapted from slides © 2010 van Dam et al., Brown University

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.

Open GL implementation? Stay tuned…

Adapted from slides © 2010 van Dam et al., Brown University

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

Adapted from slides © 2003 F. Pfenning, Carnegie Mellon University.
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
    ```c
    glEnable(GL_LIGHTING);
    ```
  - Each individual light must be enabled
    ```c
    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!
    ```c
    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
  
  ```c
  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
  
  ```c
  glLightModelli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);
  ```

- More expensive, but sometimes more accurate
Shading in OpenGL [7]:
Point Sources vs. Directional

- Directional Lights versus Point Lights
  - Directional light given by “position” vector
    \[
    \text{GLfloat light\_position[]} = \{-1.0, 1.0, -1.0, 0.0\};
    \text{glLightfv(GL\_LIGHT0, GL\_POSITION, light\_position)};
    \]
  - Point source given by “position” point
    \[
    \text{GLfloat light\_position[]} = \{-1.0, 1.0, -1.0, 1.0\};
    \text{glLightfv(GL\_LIGHT0, GL\_POSITION, light\_position)};
    \]

- Spotlights: Special Case of Point Lights
  - Create point source as before
  - Specify additional properties to create spotlight
    \[
    \text{GLfloat sd[]} = \{-1.0, -1.0, 0.0\};
    \text{glLightfv(GL\_LIGHT0, GL\_SPOT\_DIRECTION, sd)};
    \text{glLightf(GL\_LIGHT0, GL\_SPOT\_CUTOFF, 45.0)};
    \text{glLightf(GL\_LIGHT0, GL\_SPOT\_EXPO\_NENT, 2.0)};
    \]

Adapted from slides © 2003 F. Pfenning, Carnegie Mellon University.
http://bit.ly/g1J2nj
Shading in OpenGL [8]: Example Material Properties

```c
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
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
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
© 2002 Gröller, E. & Jeschke, S. Vienna Institute of Technology
Other Source Material on Texture Mapping

David W. Jacobs

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

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

CPSC 599.64/601.64 Computer Graphics
Fall 2005
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

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 → inefficient for memory and processing
- new approach necessary: texture mapping

Adapted from slides © 2005 Isenberg, T., University of Calgary (now at U. Groningen)
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

Adapted from slides
© 2005 Isenberg, T., University of Calgary (now at U. Groningen)
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.
Without Textures

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

With Textures

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

Texture Image

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

Adapted from slides © 1997 – 2010 van Dam et al., Brown University
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

Adapted from slides © 2010 van Dam et al., Brown University
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

Adapted from slides © 2010 van Dam et al., Brown University
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)

Adapted from slides © 2010 van Dam et al., Brown University
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
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:
  \[
  (r, g, b) = c_{\text{tex}}(u, v)
  \]
- texture mapping function maps \((u, v)\) values to \((x, y, z)\) positions on objects:
  \[
  (x, y, z) = F_{\text{map}}(u, v)
  \]
- we need to solve the inverse function to find \((u, v)\) values for a \((x, y, z)\) position:
  \[
  (u, v) = F_{\text{map}}^{-1}(x, y, z)
  \]
  \[
  u = s(x, y, z)
  \]
  \[
  v = t(x, y, z)
  \]

Adapted from slides © 2005 Isenberg, T., University of Calgary (now at U. Groningen)
Texture Mapping How-To [4]: General Procedure

- general texture mapping pipeline:
  - determine surface position
  - find texture coordinates
  - find corresponding texel
  - possibly more processing
  - modify illumination

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

Adapted from slides
© 2005 Isenberg, T., University of Calgary (now at U. Groningen)
Texture Mapping How-To [5]:
Projective Textures, Functions

- goal: derive texture coordinates from 3D point
- \( P: \mathbb{R}^3 \rightarrow \mathbb{R}^2 \), 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

Adapted from slides
© 2005 Isenberg, T., University of Calgary (now at U. Groningen)
Texture Mapping How-To [6]: (Manual) Surface Parameterization

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

Adapted from slides
© 2005 Isenberg, T., University of Calgary (now at U. Groningen)
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
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 $\theta_0$

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

using scaling factors $c$, $d$, and the radius $r$

- discontinuity along one line parallel to center axis

Adapted from slides
© 2005 Isenberg, T., University of Calgary (now at U. Groningen)
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

Adapted from slides
© 2005 Isenberg, T., University of Calgary (now at U. Groningen)
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

Adapted from slides
© 2005 Isenberg, T., University of Calgary (now at U. Groningen)
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

Adapted from slides
© 2005 Isenberg, T., University of Calgary (now at U. Groningen)
O Mapping [2]: Illustrations

1. Reflected Ray

2. Object Normal

3. Object Center

4. Normal of Intermediate Surface

Adapted from slides
© 2005 Isenberg, T., University of Calgary (now at U. Groningen)
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

\[
\begin{align*}
u &= s(x, y, z) \\
v &= t(x, y, z)
\end{align*}
\]
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

Adapted from slides
© 2005 Isenberg, T., University of Calgary (now at U. Groningen)
Correspondence Functions [3]: Clamping and Borders Illustrated

Adapted from slides
© 2005 Isenberg, T., University of Calgary (now at U. Groningen)
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 \(\alpha\) transparence value
- is typically used to modify illumination
- methods:
  - replace: surface color value is replaced with texture color
  - decal: \(\alpha\) blending of texture and original color
  - modulate: multiplication of original color value with texture color
The End…?

Done! … well, almost
Surface Detail: Imitating Complexity by Texturing Smooth Objects

Also possible: model very complex objects just by using simple textured geometry
More on Shading
(Surface Detail 2, 4, 5)

- Shading in OpenGL
  - Flat/constant: \texttt{GL\_CONSTANT}
  - Gouraud: \texttt{GL\_SMOOTH}

- Shading Languages
  - Renderman Shading Language (RSL) – \url{http://bit.ly/g229q4}
  - OpenGL Shading Language (OGLSL or GLSL) – \url{http://bit.ly/fX8V0Y}
  - Microsoft High-Level Shading Language (HLSL) – \url{http://bit.ly/eVnjp5}
  - nVidia Cg – \url{http://bit.ly/ewoRic}
Acknowledgements:
Texture Mapping Slides

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

Adapted from slides © 1997 – 2010 van Dam et al., Brown University
Texture Mapping Technique [1]

- Texture mapping is the process of mapping a geometric point in space to a value (color, normal, other...) in a texture
- Our goal is to map any arbitrary geometry to a texture of any dimension
- This is done in two steps:
  - Map a point on the geometry to a point on the unit square
  - Map the unit square point to point on the texture

- Second mapping is much easier, we’ll present it first.
Texture Mapping Technique [2]

- Mapping a point in the unit u, v square to a texture of arbitrary dimension:
  - In general, any point \((u, v)\) on the unit square, the corresponding point on the texture of length \(l\) pixels and height \(h\) pixels is \((u \times l, v \times h)\).

![Diagram](image)

- Above: \((0.0, 0.0) \rightarrow (0, 0); \ (1.0, 1.0) \rightarrow (200, 100); \ (.7, .45) \rightarrow (140, 45)\)
- Once we have coordinates for the texture, we just need to look up the color of the texture at these coordinates
- Coordinates not always a discrete point on texture as they come from continuous space. May need to average neighboring texture pixels (i.e. filter)

Adapted from slides © 2010 van Dam et al., Brown University
Texture Mapping Technique [3]

- Texture mapping polygons
  - \((u, v)\) texture coordinates are pre-calculated and specified per vertex
  - Vertices may have different texture coordinates for different faces

- Texture coordinates are linearly interpolated across polygon

Adapted from slides © 2010 van Dam et al., Brown University
Interpolation Trick: Barycentric Coordinates

- Consider interpolating between two values along a line
  - Given two colors $C_a$ and $C_b$, you can compute any value along the "line" between the two colors by evaluating:

  $C(t) = (1 - t)C_a + tC_b \quad 0 \leq t \leq 1$

- This equation can be written as:

  $C(s, t) = sC_a + tC_b \quad s + t = 1 \quad s, t \geq 0$

- $s$ and $t$ are the Barycentric Coordinates of the line segment between $C_a$ and $C_b$
- The EQ of the line is a convex linear combination of its endpoints. We’ve seen this before (splines, color theory)

- Barycentric coordinates can be generalized to triangles

  $C(s, t, u) = sC_a + tC_b + uC_c \quad s + t + u = 1 \quad s, t, u \geq 0$

Adapted from slides © 2010 van Dam et al., Brown University
Applying Barycentric Coordinates

- When you intersect a ray with a polyhedral object (not needed for our intersect/ray projects):
  - return the vertex data of the triangle intersected
  - return the Barycentric coordinates \((t_1, t_2, t_3)\) of the intersection point
  - These coordinates can be used to interpolate between vertex colors, normals, texture coordinates, or other data
  - What weights do we hang on each vertex such that the triangle would be perfectly balanced on a pin at point \(P\)
  - Alternatively, think of a mobile suspended from \(P\) with 2 arms \(A_1Q\) and \(A_2A_3\).

- Compute \(Q\) as intersection of line through \(A_1\) and \(P\) and line through \(A_2\) and \(A_3\)
- \(t_3' = |Q - A_2|\)
- \(t_2' = |Q - A_3|\)
- \(t_1' = |P - Q|\)
- \((t_1, t_2, t_3) = (t_1', t_2', t_3')/(t_1 + t_2 + t_3)\)

- Another way of thinking about this is by triangle area. The weight at \(A_1\) should be proportional to the area of the triangle \(P, A_2, A_3\), and so on...

Adapted from slides © 2010 van Dam et al., Brown University
Texture Mapping Technique [4]: Map Point to Object on \((u, v)\) Square

- Texture mapping in “Ray”: mapping solids
  - Using ray tracing, we obtain an intersection point \((x, y, z)\) in object space
  - We need to map this point to a point on the \((u, v)\) unit square, so we can map that to a texture value
  - Three easy cases: planes, cylinders, and spheres

- Easiest to compute the mapping from \((x, y, z)\) coordinates in \textit{object space} to \((u, v)\)

- Can cause unwanted texture scaling
- Texture filtering is an option in most graphics libraries

- OpenGL allows you to choose filtering method. (\texttt{GL_NEAREST}, \texttt{GL_LINEAR}, etc...)

Adapted from slides © 2010 van Dam et al., Brown University
Texture Mapping Technique [5]

- Texture mapping large quads:
  - How to map a point on a very large quad to a point on the unit square?
  - Tiling: texture is repeated over and over across infinite plane
  - Given coordinates \((x, y)\) of a point on an arbitrarily large quad that we want to tile with quads of size \((w, h)\), the \((u, v)\) coordinates on the unit square representing a texture with arbitrary dimensions are:

\[
(u, v) = \left( \frac{x \% w}{w}, \frac{y \% h}{h} \right)
\]

Adapted from slides © 2010 van Dam et al., Brown University
Texture Mapping Technique [6]

How to texture map cylinders and cones:

- Given a point P on the surface:
  - If it’s on one of the caps, map as though the cap is a plane
  - If it’s on the curved surface:
    - Use the position of the point around the perimeter to determine $u$
    - Use the height of the point to determine $v$

- Mapping $v$ is trivial, $[-.5, .5]$ gets mapped to $[0.0, 1.0]$ just by adding $.5$

Adapted from slides © 2010 van Dam et al., Brown University
Texture Mapping Technique [7]

Computing the $u$ coordinate for cones and cylinders:
- We need to map all the points on the perimeter of the object to $[0, 1]$.
- The easiest way is to say $u = \frac{\theta}{2\pi}$, but computing $\theta$ can be tricky.

![Diagram]

Standard atan function computes a result for $\theta$ but its always between 0 and $\pi$ and it maps two positions on the perimeter to the same $\theta$ value.
- Example: $\text{atan}(1, 1) = \text{atan}(-1, -1) = \frac{\pi}{2}$
- $\text{atan2}(x, y)$ yields values between $-\pi$ and $\pi$, but isn’t continuous. See above diagram.

Adapted from slides © 2010 van Dam et al., Brown University
Texture Mapping Technique [8]

Texture mapping spheres:
- Find $(u, v)$ coordinates for $P$
- We compute $u$ the same we do for cylinders and cones
- If $v = 0$ or $v = 1$, there is a singularity. Set $u$ to some predefined value. (.5 is good)
- $v$ is a function of the latitude of $P$

$$\phi = \sin^{-1} \frac{P_y}{r} \quad -\frac{\pi}{2} \leq \phi < \frac{\pi}{2} \quad r = \text{radius}$$

$$v = \frac{\phi}{\pi} + .5$$

Adapted from slides © 2010 van Dam et al., Brown University
Texture Mapping Style [1]: Tiling

- We want to create a brick wall with a brick pattern texture.
  - A brick pattern is very repetitive, we can use a small texture and “tile” it across the wall.

- Tiling allows you to scale repetitive textures to make texture elements just the right size.

Adapted from slides © 2010 van Dam et al., Brown University
Texture Mapping Style [2]: Stretching

- With non-repetitive textures, we have less flexibility
  - Have to fill an arbitrarily large object with a texture of finite size
  - Can’t tile, have to stretch

- Example, creating a sky backdrop:

Adapted from slides © 2010 van Dam et al., Brown University
Texture Mapping
Complex Geometry [1]

- Sometimes, reducing objects to primitives for texture mapping doesn’t achieve the right result.

- Consider a simple house shape as an example
- If we texture map it by our old methods, we get discontinuities at some edges.

- Solution: Pretend object is a sphere and texture map using the sphere \((u, v)\) map

Adapted from slides © 2010 van Dam et al., Brown University
Texture Mapping
Complex Geometry [2]

- Intuitive approach: Place a bounding sphere around the complex object
  - Find ray’s object space intersection with bounding sphere
  - Convert to $(u, v)$ coordinates

![Diagram](image)

- Stage one: intersect ray with bounding sphere
- Stage two: calculate intersection point’s $uv$-coords

- We actually don’t need a bounding sphere!
  - Once we have the intersection point with the object, we just treat it as though it were on the sphere. Same results, but be careful with radii.

Adapted from slides © 2010 van Dam et al., Brown University
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

Texture Mapping
Complex Geometry [4]

Results of spherical (u, v) mapping:

You can use cylindrical or planar mappings for complex objects as well

Each has drawbacks

- Spherical: warping at the “poles” of the object
- Cylindrical: discontinuities at the caps
- Planar: one dimension must be ignored

For best overall results, mapping techniques can be swapped

Adapted from slides © 2010 van Dam et al., Brown University
OpenGL Texturing [1]:
Steps

• Create and specify a texture object
  – Create a texture object
  – Specify the texture image
  – Specify how texture has to be applied for each pixel
• Enable texture mapping
• Draw the textured polygons
  – Identify the active texture
  – Specify texture coordinates with vertices

Adapted from slides
© 2007 Jacobs, D. W., University of Maryland
OpenGL Texturing [2]:
Specify 2-D Texture Object

- `glTexImage2D(GLenum target, GLint level, GLint internalformat, GLsizei width, GLsizei height, GLint border, GLenum format, GLenum type, const GLvoid *texels);
  - Eg: `glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 128, 128, 0, GL_RGBA, GL_UNSIGNED_BYTE, image);
  - `format` and `type` used to specify the way the texels are stored
  - `internalFormat` specifies how OpenGL should store the data internally
  - `width` and `height` have to be powers of 2; you can use `gluScaleImage( )` to scale

Adapted from slides
© 2007 Jacobs, D. W., University of Maryland
OpenGL Texturing [3]: Specify How Texture Is Applied

- `glTexParameteri{if}(GLenum target, GLenum pname, TYPE param)`
- `target` can be: `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, ...
  
  `pname` | `param` |
  |-----------------|-----------------|
  `GL_TEXTURE_WRAP_S` | `GL_CLAMP, GL_REPEAT` |
  `GL_TEXTURE_WRAP_T` | `GL_CLAMP, GL_REPEAT` |
  `GL_TEXTURE_MAG_FILTER` | `GL_NEAREST, GL_LINEAR` |
  `GL_TEXTURE_MIN_FILTER` | `GL_NEAREST, GL_LINEAR` |

Adapted from slides
© 2007 Jacobs, D. W., University of Maryland
OpenGL Texturing [4]: Enable Texture and Draw

- `glEnable(GL_TEXTURE_2D)`
  - Enable 2D texturing

- `glTexCoord2f(GL_FLOAT u, GL_FLOAT v)`
  - Specify texture coordinates per vertex (just as normals, color, etc.)
OpenGL Texturing [5]: Create Texture Object

- **glGenTextures** *(GLsizei n, GLuint* textureIDs)*;
  - Returns *n* currently unused texture ID in *textureIDs*
  - Each texture ID is an integer greater than 0

- **glBindTexture** *(GLenum target, GLuint textureID)*;
  - *target* is GL_TEXTURE_1D, GL_TEXTURE_2D, or GL_TEXTURE_3D
  - if *textureID* is being used for the first time a new texture object is created and assigned the ID = *textureID*
  - if *textureID* has been used before, the texture object with ID = *textureID* becomes active

Adapted from slides
© 2007 Jacobs, D. W., University of Maryland
OpenGL Texturing [6]: Putting It All Together

In initialization:

```c
glGenTextures(...);
glBindTexture( ... );
glTexParameteri(...); glTexParameteri(...); ...
glTexImage2D(...);
glEnable(GL_TEXTURE_2D);
```

In display:

```c
glBindTexture( ... ); // Activate the texture defined in initialization
glBegin(GL_TRIANGLES);
  glTexCoord2f(...); glVertex3f(...);
  glTexCoord2f(...); glVertex3f(...);
  glTexCoord2f(...); glVertex3f(...);
  glTexCoord2f(...); glVertex3f(...);
glEnd( );
```
Preview:
Texturing with Blocks

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

Adapted from slides
© 1999 – 2007 van Dam, A., Brown University
Summary

- Last Time: Texture Mapping Explained
  - Definitions and design principles
  - Enclosing volumes: cylinder, sphere, box
  - Mapping methods
    - reflected ray
    - object normal
    - object center
    - intermediate surface normal
- Today: Mappings, OpenGL Texturing
  - Idea: define “texture” to simulate surface detail
  - Shadow, reflection/environment, transparency, bump, displacement
  - Other mappings: gloss, volumetric fog, skins, rainbows, water
  - OpenGL texture mapping how-to
Terminology

- **Texture Mapping - Adding Detail, Raster Image, Color, etc. to CG Model**
  - **Planar projection**: apply flat texture to flat surface(s)
  - **Enclosing volumes**: cylinder, sphere, box
  - **Mapping methods**
    - reflected ray – bounce ray off object $O$
    - object normal – ray from face normal of object (polygon mesh)
    - object center – ray from center of object
    - intermediate surface normal – ray from inside of enclosing $S$

- **Mappings**: Apply Image or Simulated Surface Detail to Object
  - **Shadow**: cast planar projective shadows or calculate volume
  - **Reflection/environment**: take picture of scene from “inside” object
  - **Transparency**: take picture of scene “behind” object; refract
  - **Bump**: perturb color based on height map
  - **Displacement**: perturb face normals, recalculate lighting
Acknowledgements:
Many Mappings

Stefan Jeschke
Research Assistant

Eduard Gröller
Associate Professor
Director, Visualization Working Group

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

Pat Hanrahan
CANON USA Professor
Director, Computer Graphics Laboratory
Computer Science and Electrical Engineering Departments
Stanford University

http://graphics.stanford.edu/~hanrahan/

Mapping material from slides © 1995 – 2009 P. Hanrahan, Stanford University
Overview of Mappings:
Eberly 2e Chapter 20 Sections

- Fine Surface Detail: Bump (§20.5 Eberly 2e)
- 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)
Shadow Mapping [1]:
Basic Concept

- Process for Adding Shadows in 3-D CG
- Compatible with Local Illumination
  - Global method: shadow rays
  - Not needed here as in raytracing
  - Instead, use decaling
- Decals
  - “Paste” surface detail onto model
  - Semi-transparent: alpha blending
  - Can simulate many attributes

Without shadow map

With shadow map


Shadow Mapping [2]:
Techniques

- 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
Shadow Mapping [3]:
Advanced Methods & Research

- Shadow Mattes (Hanrahan)

```java
UberLight()
{
    Clip to near/far planes
    Clip to shape boundary
    foreach superelliptical blocker
        atten *= ...
    foreach cookie texture
        atten *= ...
    foreach slide texture
        color *= ...
    foreach noise texture
        atten, color *= ...
    foreach shadow map
        atten, color *= ...
    Calculate intensity fall-off
    Calculate beam distribution
}
```

- Can Be Layered (See Maya 2011 Tutorial by Maciek Gryka)

Adapted from slides © 1995 – 2009 P. Hanrahan, Stanford University
Reflection/Environment Mapping [1]: Basic Concept

- Reflection Maps (Special Type) ~ Environment Maps (General Case)
  - For a given viewing direction
  - For each normal direction
  - For each incoming direction (hemispherical integral)
    - Evaluate reflection equation
- Idea: Take Picture of Scene Faced by Object, Apply as Map to Object
- Requirements: Need to Take Account of Projective Distortions

Adapted from slides © 1995 – 2009 P. Hanrahan, Stanford University
Reflection/Environment Mapping [2]: Techniques

- Gazing Ball (Mirrorball)

- Reflection Functions
  - Diffuse: irradiance map
  - Glossy: radiance map
  - Anisotropic: for each tangent direction
  - Mirror: reflection map (related to environment map)

- Illumination Functions: Environment Map or Procedural Light Sources

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

Reflection/Environment Mapping [3]:
Advanced Methods & Research

- 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


- Issues: Non-Linear Mapping, Area Distortion, Converting Between Maps

Adapted from slides © 1995 – 2009 P. Hanrahan, Stanford University
Transparency Mapping [1]:
Basic Concept

- Transparency: One Term for Many Techniques
- Goal: “See Through” Objects (Could Be Real Decals)
- Ideas: Render Background Object, Then Foreground Object or Material
  - Blend in color of (semi-)transparent/translucent foreground object
  - Simulate little holes in foreground material (screen door)

Adapted from slides © 1995 – 2009 P. Hanrahan, Stanford University
Transparency Mapping [2]: Techniques

- **Alpha Compositing *aka* Alpha Blending**
  - Combine colors of transparent foreground, opaque background
  - Uses **alpha channel** \(A\) of \((R, G, B, A)\) – think “% transparency”

- **Screen Door Transparency**
  - Simulate little holes in foreground material (screen door)
  - Result: visual effect of being able to see through foreground

Goon Creative, Maya Transparency Tutorial

Technical University of Vienna, IEEE Vis 2004
Transparency Mapping [3]: Advanced Methods & Research

- **OpenGL Transparency How-To at** [OpenGL.org](http://bit.ly/hRaQgk)
- **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
Bump Mapping [1]: Basic Concept

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

**Will This Look Realistic? Why/Why Not?**

Bump Mapping © 2010 Wikipedia
Bump Mapping [2]:

Techniques

\[
P(u,v) = \frac{\partial P(u,v)}{\partial u} \\
S(u,v) = \frac{\partial P(u,v)}{\partial v} \\
T(u,v) = \frac{\partial P(u,v)}{\partial v} \\
N(u,v) = S \times T
\]

- **Displacement**

\[
P'(u,v) = P(u,v) + h(u,v)N(u,v)
\]

- **Perturbed normal**

\[
N'(u,v) = P'_u \times P'_v \\
= N + h_u(T \times N) + h_v(S \times N)
\]

From Blinn 1976

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

Hey, wait a minute! … what’s wrong with the one on the left?

- Right Ball (Displacement Mapped) Casts Rough Shadow
- Left Ball (Bump Mapped) Casts Smooth Shadow – Why?
- Bump Mapping Only Perturbs Normals (Surface Only!)

Displacement Mapping [1]: Basic Concept

- Remember What We Did to Perform Bump Mapping?

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

\[
S(u,v) = \frac{\partial P(u,v)}{\partial u}, \quad T(u,v) = \frac{\partial P(u,v)}{\partial v}
\]

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

- Displacement

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

- Perturbed normal

\[
\mathbf{N}'(u,v) = \mathbf{P}'_u \times \mathbf{P}'_v
\]

\[
= \mathbf{N} + h_u(T \times \mathbf{N}) + h_v(S \times \mathbf{N})
\]

- Q: Can We Make This Permanent? How?

- A: Sure! Let Perturbed Normals Define New Surface; Save Out Vertices

Adapted from slides © 1995 – 2009 P. Hanrahan, Stanford University
Displacement Mapping [2]: Techniques

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

- Displacement Mapping: Uses Open GL Shading Language (GLSL)

Displacement Mapping [3]: Advanced Methods & Research

- When To Consider Using Displacement Mapping
  - Very “deep” texture effect: veins, ridges, etc.
  - Shadows expected

Like Many Mappings and Other Effects, Wanted In Hardware!

The “Imp” © 2008 K. Scott, id Software