

CIS 636/736 Computer Graphics Lecture 7 of 42

Surface Detail 1: Shading - Pipeline, Phong Illumination

Wednesday, 04 February 2008

Reading: Section 2.5 (Rasterizing), pp. 77 – 92, Eberly 2e

Next: Section 3.1

Adapted with permission from slides by A. VanDam

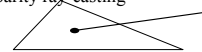
W. H. Hsu

<http://www.kddresearch.org>

Rendering Polygons

Need to render triangle meshes

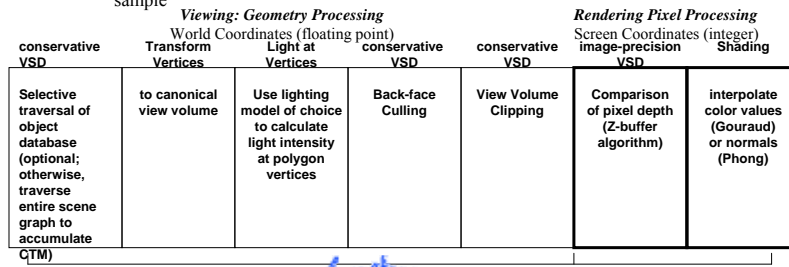
- Raytracing and implicit surface model definition go well together
- But many of our existing models and modeling apps are still based on polygon meshes. How do we render them?
 - can raytrace polygons...
 - better solution: traditional polygons-to-pixels pipeline
- Ray-Triangle intersection is not too hard...
 - do ray-intersect-plane using the plane of the triangle.
 - check if resulting point is inside the triangle, e.g., using odd parity ray-casting
- Ray-Polygon intersection is similar...
 - can decompose into triangles
- The number of polygons is a real problem
 - in a typical mesh representation, there are a lot of triangles, and each is an object that has to be considered in our intersection tests
- Traditional hardware polygon pipeline faster
 - uses the very efficient, one-polygon-at-a-time, Z-buffer Visible Surface Determination algorithm
 - uses an approximate shading rule to calculate most pixels



Traditional Hardware Pipeline

Polygonal rendering w/ Z-buffer

- Polygons (usually triangles) approximate actual geometry
- Non-global illumination approximates lighting
- Shading approximates lighting of each sample point. It's fast, and it looks ok, especially for small triangles
- Based on per-pixel calculation and comparison of z (depth) values
 - much faster than raytracing's solving implicit surface equations for objects in the scene per-sample

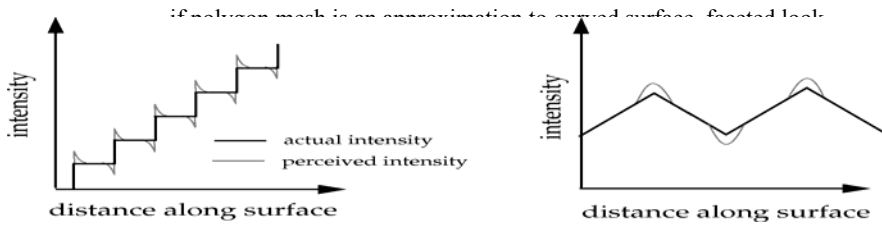


- N.B: Simplified from actual pipelines (no texture maps, or any other kinds of maps, anti-aliasing, transparency)
- Conservative VSD: trivial reject only

Polygon Mesh Shading (1/5)

Interpolating illumination for speed

- Three methods; each treats a single polygon independent of all others (non-global)
 - constant
 - Gouraud (intensity interpolation)
 - Phong (normal-vector interpolation)
- Constant shading
 - single illumination value per polygon

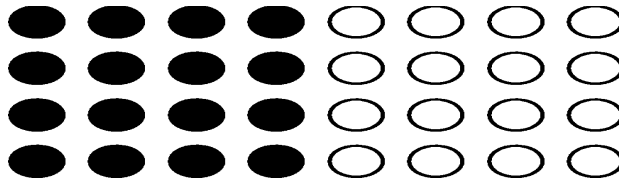


Polygon Mesh Shading (2/5)

Mach banding review

- Mach band effect: discrepancies between actual and perceived intensities due to bilateral inhibition
- A photoreceptor in the eye responds to light according to the intensity of the light falling on it *minus* the activation of its neighbors

$$I(c_j) = e(c_j) - \sum_{k \neq j} \alpha_k e(c_k) \quad 0 \leq \alpha_k \leq 1$$



Mach banding applet :

<http://www.nbb.cornell.edu/neurobio/land/OldStudentProjects/cs490-96to97/anson/MachBandingApplet/>

Polygon Mesh Shading (3/5)

Illumination intensity interpolation

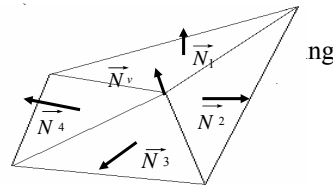
- Gouraud shading
 - use for polygon approximations to curved surfaces
- Linearly interpolate intensity along scan lines
 - eliminates intensity discontinuities at polygon edges; still have gradient discontinuities, so mach banding is only improved, but not eliminated
 - must differentiate desired creases from tessellation artifacts (edges of a cube vs. edges on tessellated spl)

- Step 1: calculate bogus vertex normals

$$\vec{N}_v = \frac{(\vec{N}_1 + \vec{N}_2 + \vec{N}_3 + \vec{N}_4)}{\|\vec{N}_1 + \vec{N}_2 + \vec{N}_3 + \vec{N}_4\|}$$

More generally:

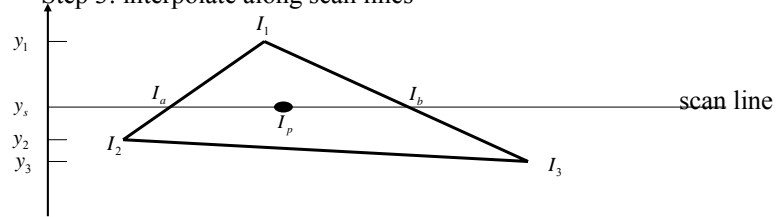
$$\vec{N}_v = \frac{\sum_{i=1}^n \vec{N}_i}{\left\| \sum_{i=1}^n \vec{N}_i \right\|} \quad n = 3 \text{ or } 4 \text{ usually}$$



Polygon Mesh Shading (4/5)

Illumination intensity interpolation (cont.)

- Step 2: interpolate intensity along polygon edges
- Step 3: interpolate along scan lines



$$I_a = I_1 \frac{y_s - y_2}{y_1 - y_2} + I_2 \frac{y_1 - y_s}{y_1 - y_2}$$

$$I_b = I_1 \frac{y_s - y_3}{y_1 - y_3} + I_3 \frac{y_1 - y_s}{y_1 - y_3}$$

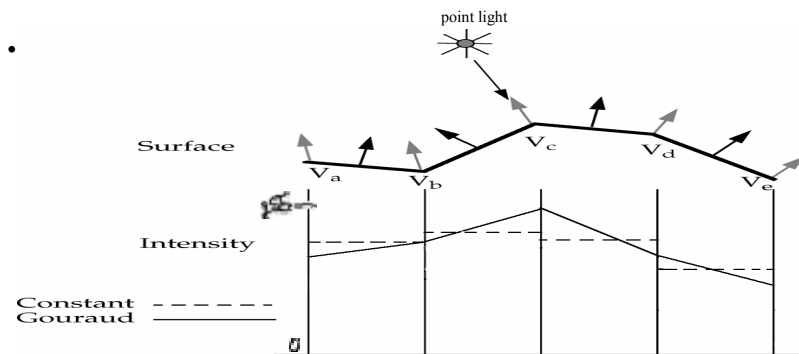
$$I_p = I_a \frac{x_b - x_p}{x_b - x_a} + I_b \frac{x_p - x_a}{x_b - x_a}$$

Polygon Mesh Shading (5/5)

Illumination intensity interpolation (cont.)

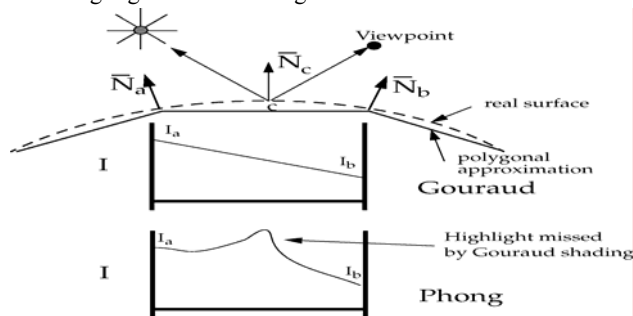
- Gouraud shading
 - Integrates nicely with scan line algorithm:

$$\frac{\Delta I}{\Delta Y} \text{ is constant along polygon edge}$$



What Gouraud Shading Misses

- Gouraud shading can miss specular highlights in specular objects because it interpolates *vertex colors* instead of *vertex normals*
 - here N_a and N_b would cause no appreciable specular component, whereas N_c would. Shading by interpolating between I_a and I_b , therefore misses the highlight that evaluating I at c would catch



Phong Shading

- Also called normal vector interpolation
 - interpolate N rather than I
 - especially important with specular reflection
 - computationally expensive at each pixel to
 - recompute N ; must normalize, requiring expensive square root
 - recompute I_λ
 - Bishop and Weimer developed fast approximation using Taylor series expansion (in SIGGRAPH '86)
- This looks much better and is now done in hardware
- Still, we've lost all the neat global effects that we got with recursive ray tracing

Shading Models Compared

Pixar "Shutterbug" images from:
http://www.siggraph.org/education/materials/HyperGraph/scanline/shade_models/shading.htm

Flat or Faceted Shading:
 Constant intensity over each face



Gouraud Shading:
 Interpolation of intensity



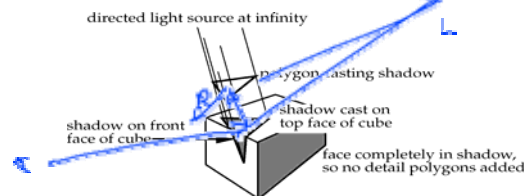
Phong Shading:
 Interpolation of surface normals.
 Note the specular highlights, but
 no shadows – pure local illumination
 model



Shadows

Hacking in Shadows

- Shadow algorithms determine which surfaces can be seen from light sources
 - unseen parts are in shadows
- Visible surface algorithms and shadow algorithms are essentially the same
- One approach for polygon-based systems:
 - add surface-detail polygons that correspond to parts of polygons not visible to light source.
 - surface detail polygons are coplanar with base polygons and are flagged for special treatment (no need to sort in a visible surface algorithm; always on top of surface)

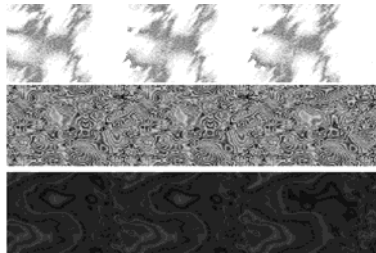


- Much harder than raytracing, which automatically does shadows

Surface Detail

Beautification of Surfaces

- Texture mapping (increasingly in hardware)
 - paste a photograph or painted bitmap on a surface to provide detail (e.g. brick pattern, sky with clouds, etc.)
 - think of a texture map as wrapping paper, but made of rubber
 - map texture/pattern pixel array onto surface to replace (or modify) original color



Texture Mapping (1/5)

Motivation

- Why do we texture map? Need to get detail on surface of models
- Expensive solution: add more detail to model
 - + detail incorporated as a part of object
 - modeling tools aren't very good for adding detail
 - models take longer to render
 - model takes up more space in memory
 - complex detail cannot be reused on other objects
- Efficient solution: map a "texture" onto model
 - + texture maps can be reused for multiple objects
 - + texture maps do take up extra space in memory, but they can be shared, and we can apply compression and caching techniques to reduce overhead
 - + texture mapping can be done quickly (we'll see how)
 - + placing and creation of texture maps can be made intuitive (e.g., tools for adjusting mapping, painting directly onto object)
 - texture maps do not affect the geometry of the object
- What kind of detail goes into these maps?
 - diffuse, ambient and specular colors
 - specular exponents
 - transparency, reflectivity
 - "bumps"
 - data to visualize (e.g., temperature, stress, elevation)
 - projected lighting and shadows

Texture Mapping (2/5)

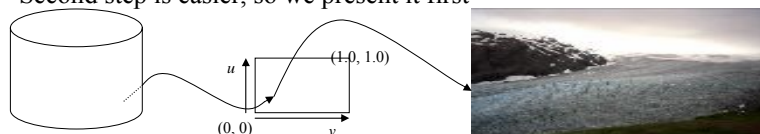
Mappings

- A function is a mapping (CS22)
 - functions map values in their subset of a domain into their subset of a co-domain (this subset is called the range)
 - each value in the domain will be mapped to one value in the co-domain
- Can transform one space into another with a function
 - for intersect, we use linear transformations to move points and vectors into the most convenient space
 - first, we map screen-space points to normalized camera-space points
 - then, we map normalized camera-space rays into unnormalized world-space rays
 - finally, we map unnormalized world-space rays into untransformed object-space rays so that we can compute object-space points of intersection
- We have points on a surface in object-space (the domain)
- We want to get values from a texture map (the co-domain)
- What function(s) should we use?

Texture Mapping (3/5)

Basic Idea

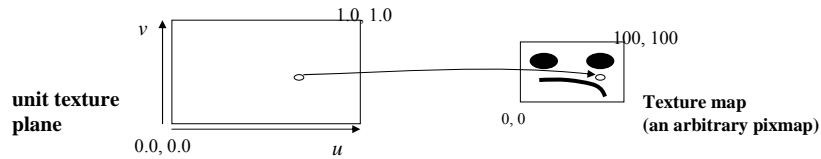
- Definition: texture mapping is the process of mapping a geometric point to a color in the texture map
- Ultimately want ability to map arbitrary geometry to a concrete pixmap of arbitrary dimension
- We do this in two steps:
 1. map a point on the arbitrary geometry to a point on an abstract unit square representing the concrete pixmap
 2. map a point on abstract unit square to a point on the concrete pixmap of arbitrary dimension
- Second step is easier, so we present it first



Texture Mapping (4/5)

From unit square to pixmap

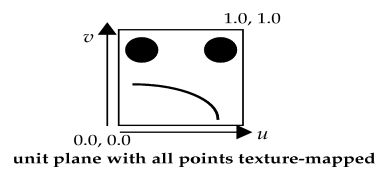
- A 2D example: mapping from a unit u, v square to a texture map



- Step 1: transform a point on abstract continuous texture plane to a point in discrete texture map
- Step 2: get color at transformed point in texture image
- Above Example:
 - $(0.0, 0.0) \Rightarrow (0, 0)$
 - $(1.0, 1.0) \Rightarrow (100, 100)$
 - $(0.5, 0.5) \Rightarrow (50, 50)$

Texture Mapping (5/5)

- In general, for a point (u, v) on a unit plane, the corresponding point in image space is $(u \cdot \text{pix-map width}, v \cdot \text{pix-map height})$



- But, there are infinitely many points on the unit plane... so we get sampling errors; the uv plane is the continuous version of the pixmap, which serves as the texture map
- The unit uv square acts as a stretchable rubber sheet to wrap around the object to be texture mapped
 - the mapping to uv is key, less so is the pixmap used
 - the pixels indexed may be transient (projecting a movie onto a virtual screen, painting onto surfaces)



Texture Mapping a 3D Point

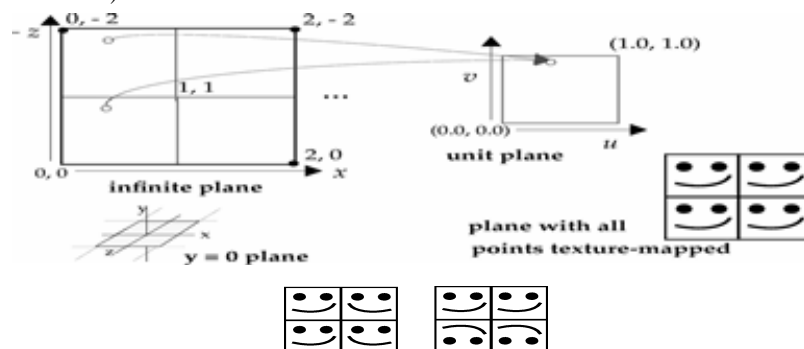
Raytracing Textures

- Using ray tracing we obtain a point in object space, (x, y, z)
- In texture mapping the goal is to go from a point (x, y, z) , to a color
- We know how to map a 2D point (u, v) in the unit square to a color in the pixmap; therefore, only need to map (x, y, z) to (u, v)
- Let's consider 3 easy cases:
 - planes
 - cylinders
 - spheres
- Note: it is easiest to calculate the projection from (x, y, z) to (u, v) from untransformed object space
 - much easier to project from simpler object definition in object space (x, y, z) to texture space (u, v)
 - drawback: texture map will be transformed along with object into world space, e.g., if a texture-mapped sphere is scaled by 2.0 in y , then the texture map will stretch by a factor of 2.0 in y as well

Texture Mapping Planes

Tiling

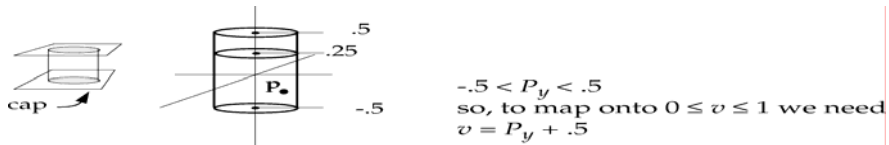
- How to map from a point on an infinite plane to a point on the unit plane?
- Tiling: use decimal portion of x and z coordinates to compute 2D (u, v) coordinates



Mapping Circles (1/2)

Texture mapping cylinders and cones

- Imagine a standard cylinder or cone as a stack of circles
 - use pos. of point on circular perimeter to determine u
 - use height of point in stack to determine v
 - map top and bottom caps separately, as onto a plane
- The easy part: calculating v
 - height of point in object space, which ranges between $[-.5, .5]$, gets mapped to v range of $[0, 1]$

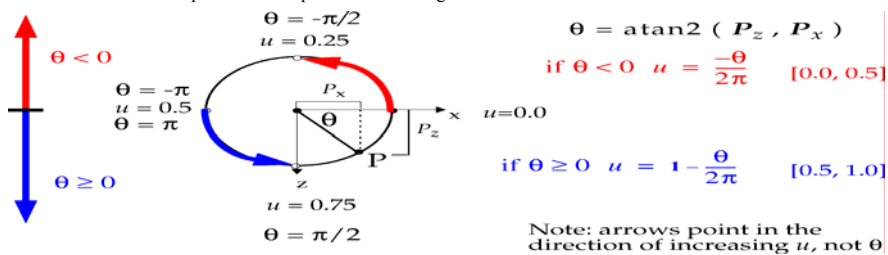


- Calculating u : map points on circular perimeter to u values between 0 and 1; 0 radians is 0, 2π radians is 1
- Then $u = \theta/2\pi$
- These mappings are arbitrary: any function that maps the angle around the cylinder (θ) into the range 0 to 1 will work
 - let's look at a mapping that follows the "law of least astonishment"

Texture Mapping Cylinders (2/2)

Mapping a circle

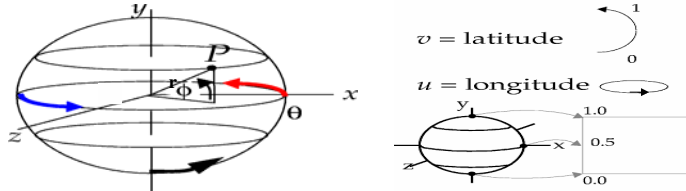
- Convert a point P on the perimeter to an angle:



- Circle radius is not necessarily unit length
 - we need to use arctangent rather than sine or cosine (arctangent only considers the ratio of lengths) of P_z/P_x
- Use standard function $atan2$ because it yields the entire circle (values ranging from $-\pi$ to π , whereas $atan$ only returns a half circle)
 - going around the circle in the direction the image would be mapped, $atan2$ returns angles that range from 0 to $-\pi$, then abruptly changes to $+\pi$ and returns to 0

Texture Mapping Spheres

- Imagine a sphere as a stack of circles of varying radii



- P is a point on the surface of the unit sphere
- Defining v

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

$$v = \frac{\phi}{\pi} + 0.5$$
 - calculate u as for the cylinder (use same mapping for cone as well)
- Defining u
 - if $v=0$ or 1 , there is a singularity and u should equal some specific value (i.e., $u = 0.5$)
 - never really code around these cases because rarely see these values within floating point precision

Texture Mapping Complex Geometries (1/5)

How do I texture-map my Möbius strip?

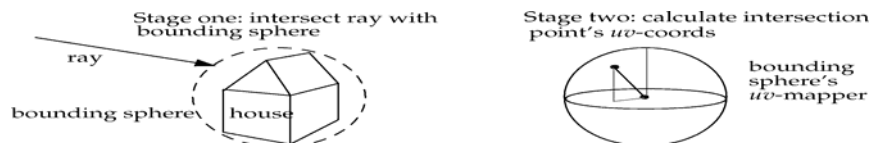
- Texture mapping of simple geometrical primitives was easy. How do I texture map more complicated shapes?
- Let's say we have a relatively "simple" complex shape: a house
- How should we texture map it?
- We could texture map polygon by polygon onto the faces of the house (we know how to do this already, as they are planar).
- However, this causes discontinuities at edges of polygons. We want smooth mapping without edge discontinuities
- Intuitive approach: reduce to a previously solved problem. Let's pretend the house is a sphere for texture-mapping purposes



Complex Geometries (2/5)

Texture mapping a house with a sphere

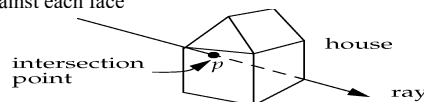
- Intuitive approach: place bounding sphere around complicated shape
- Texture mapping has two stages: finding the ray's object-space intersection point, and converting it into uv -coordinates. Simplify by finding intersection with sphere instead of with house. Then can easily convert into uv -coords



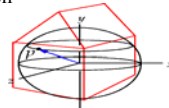
- But sphere intersection calculation is extra effort. If we're at the texture-mapping stage, that means we've already found the intersection point with the complicated shape!
- Non-intuitive approach: we can treat point on the complicated object as a point on a sphere, and project using spherical uv -mapping

Complex Geometries (3/5)

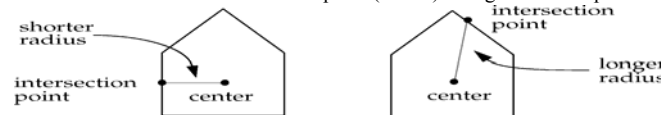
- First, calculate object-space intersection point with geometrical object (house), intersecting against each face



- Then, use object-space intersection point to calculate uv -coords of sphere at that point using spherical projection

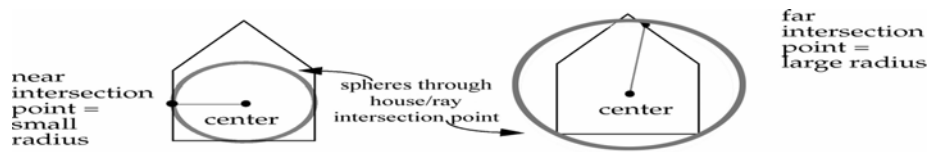


- Note that a sphere has constant radius, whereas our house doesn't. Distance from the center of the house to the intersection point (radius) changes with the point's location



Complex Geometries (4/5)

- How do we decide what radius to use for the sphere in our uv -mapper?
Intersections with our house happen at different radii
- Answer: spherical mapping of (x,y,z) to (u,v) presented above was a function of radius, does not assume that the sphere has unit radius
- Always use a sphere with its center at the house's center, and with a radius equal to the distance from the center to the current intersection point



Complex Geometry (5/5)

Choosing appropriate functions

- We have chosen to use a spherical uv -mapper to simplify texture mapping. However, any mapping technique may be used



- Each type of uv -projection has its own drawbacks
 - spherical: warping at poles of sphere
 - cylindrical: discontinuities at edges of caps

- pl sphere mapped with spherical projection



sphere mapped with planar projection

- Swapping uv -projection techniques allows drawbacks of one technique to be traded for another. Deciding which drawbacks are preferable depends on the user's wishes