Phong Shading and Texture Mapping

15 March 2007
CMPT370
Dr. Sean Ho
Trinity Western University

TRINITY WESTERN UNIVERSITY

Review last time

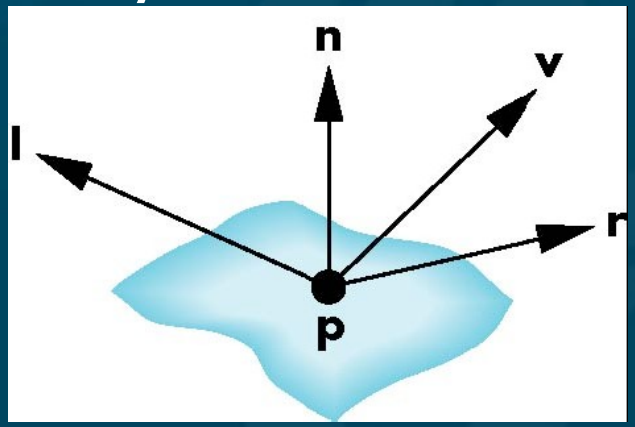
- Lighting and shading
  - The global rendering equation
  - Light-material interaction
  - Kinds of light sources
- The OpenGL local illumination model
  - Ambient term
  - Diffuse term
  - Specular term
  - Specifying in OpenGL
- See RedBook ch5

What's on for today

- Shading polygons
  - Flat shading
  - Gouraud shading
  - Phong shading
- Texture mapping
  - Coordinate transforms
  - Cylinder, sphere, cube maps
  - Bump mapping
  - Environment mapping

Computing the local illumination

- The illumination model relies on four vectors:
  - To light (l): specified by the model/scene
  - To viewer (v): specified by model-view matrix
  - Surface normal (n)
  - Reflection (r): compute from l, n
- Computing normals is not always easy
  - Depends on how we represent the surface
  - OpenGL leaves this up to us (in our application)



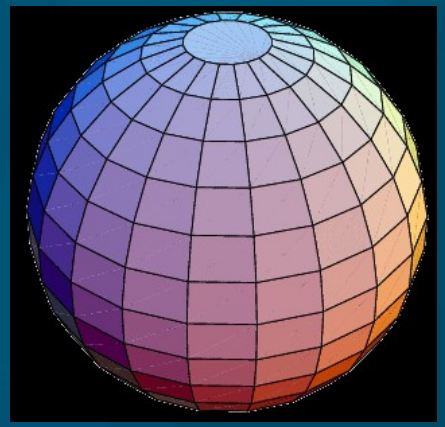
Shading polygons

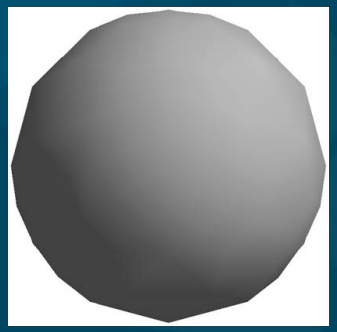
- We specify in our model for each vertex:
  - Vertex coordinates
  - Vertex colours
  - Vertex normal

- Use lighting model to calculate vertex shades
- Smooth shading: vertex shades are interpolated across the polygon
  - glShadeModel(GL_SMOOTH);
- Flat-shading uses the colour of the first vertex:
  - glShadeModel(GL_FLAT);

Calculating normals

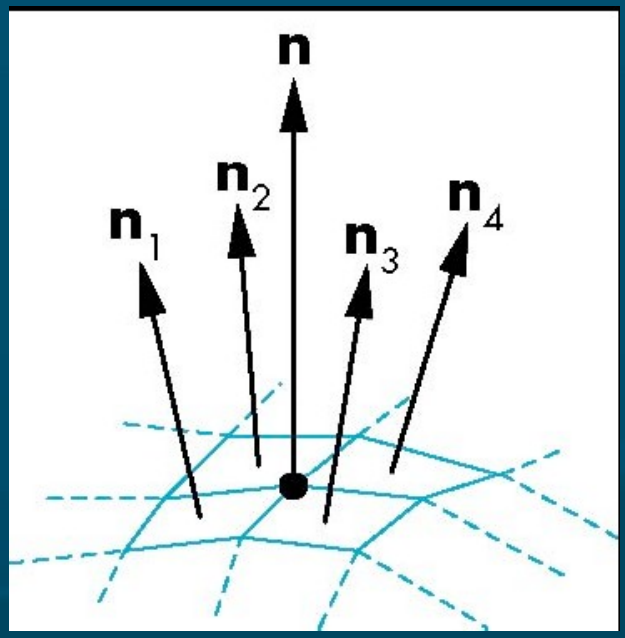
- OpenGL expects us to find the normals
- Some shapes can be done analytically:
  - Sphere: normal points from centre
  - No vertex normals: flat-shaded
  - With vertex normals: smooth-shaded
    - Note silhouette edge
- How to find normals for general surfaces?





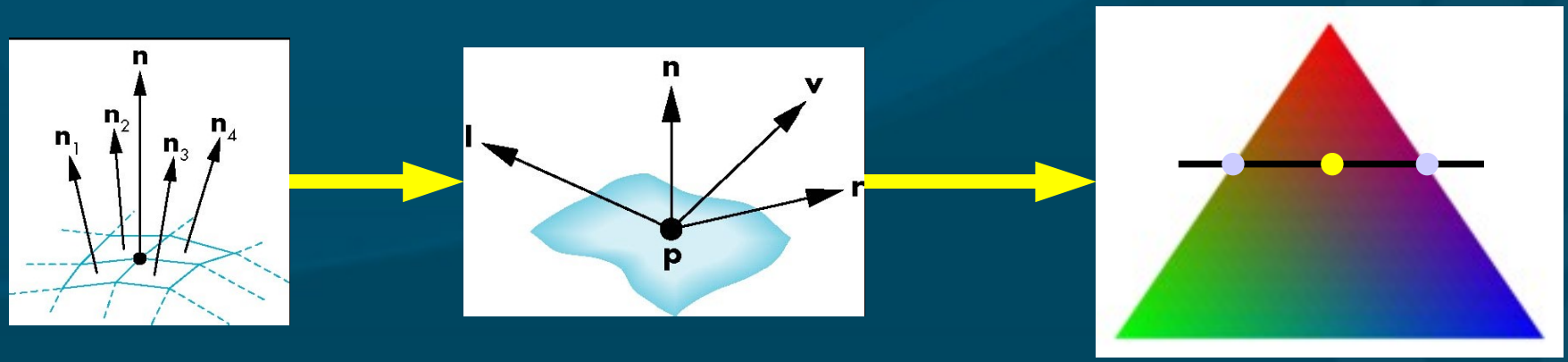
Mesh shading

- Each polygon is flat and has a normal vector we can find
- To calculate normal at a vertex, average the normals of the faces surrounding that vertex:



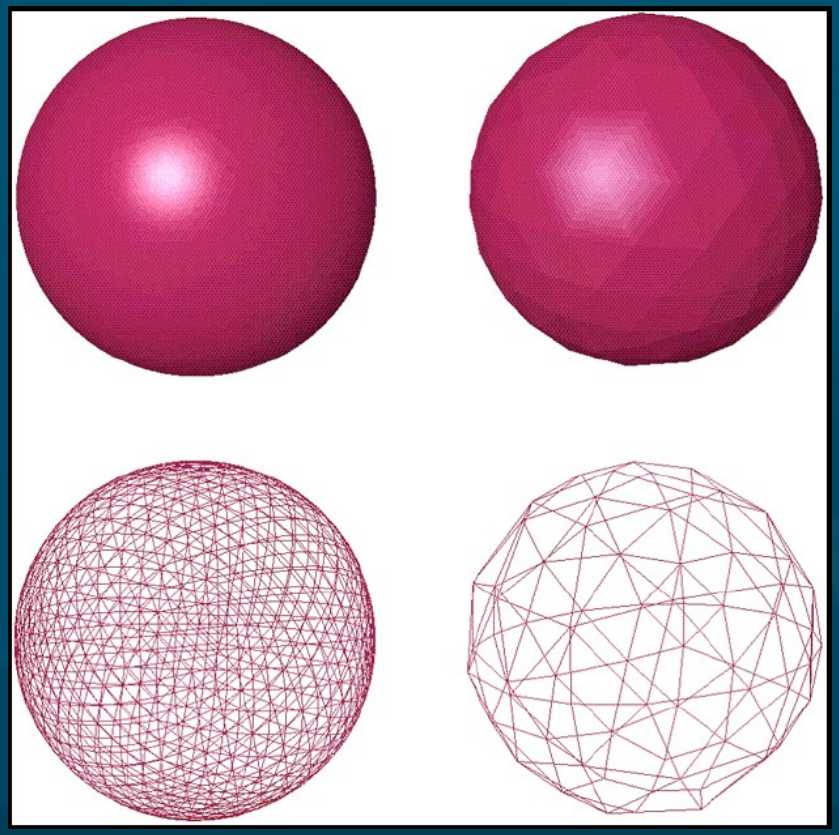
Gouraud shading

- Find vertex normals (average polygon normals)
- Apply lighting model to each vertex to get vertex shades
- Interpolate vertex shades across polygon
  - Interpolate along edges first
  - Then along each scan line (done in hardware)



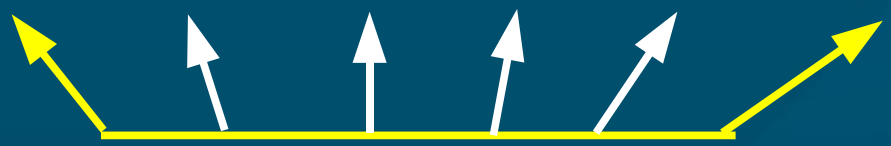
Gouraud shading: quality

- Depends on how big each polygon appears on screen, compared to pixel size
  - Fewer polygons => bigger on screen => worse



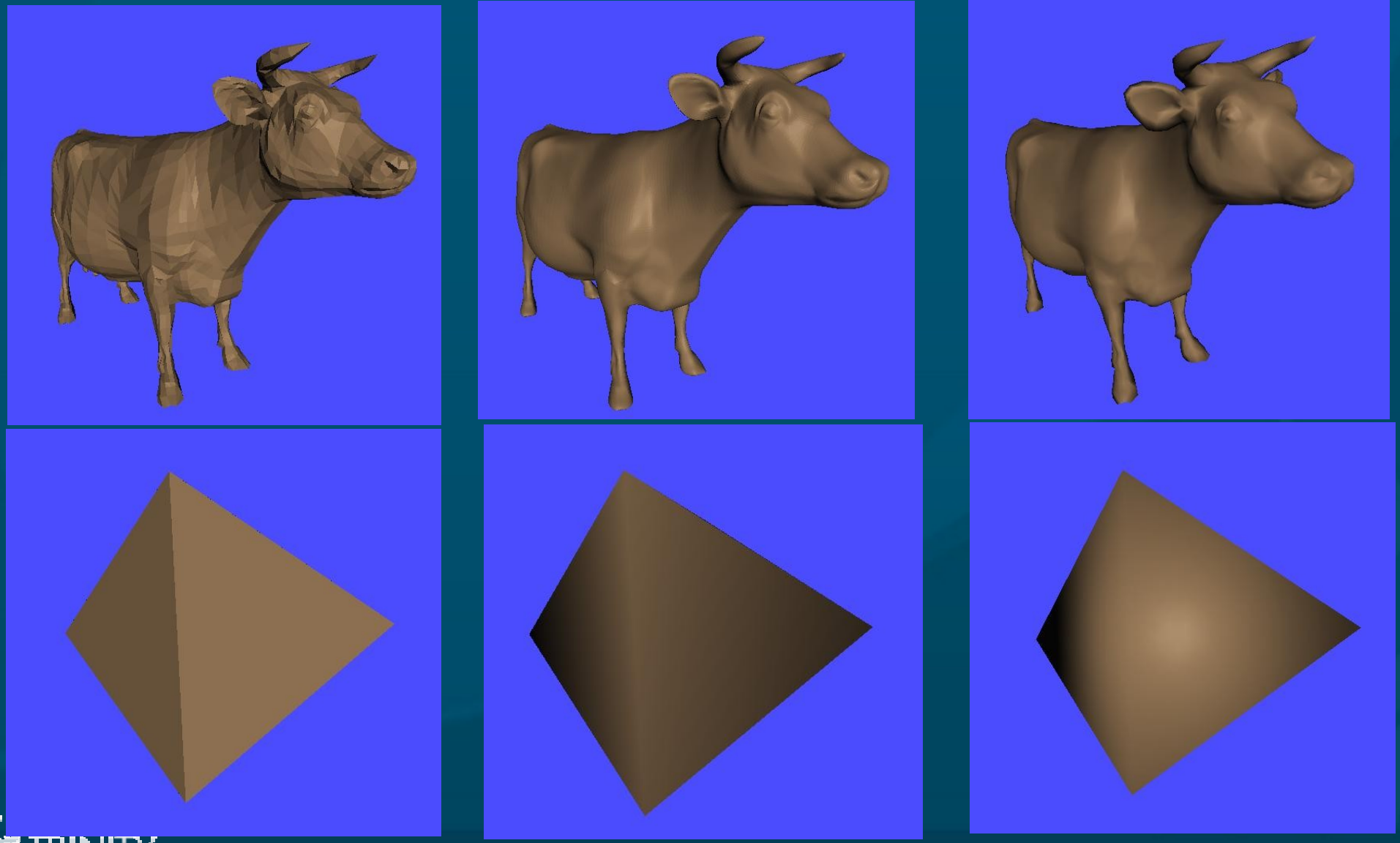
Phong shading

- Find vertex normals (average polygon normals)
- Interpolate vertex normals across polygon
  - Interpolating vectors, not intensities!
- Apply lighting model at each pixel to get shades



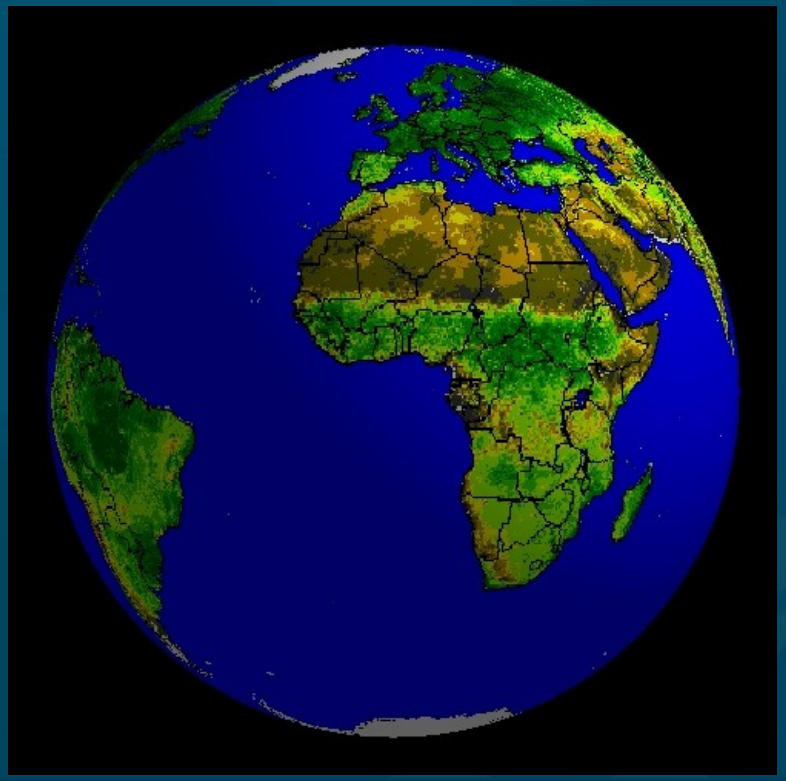
- Gouraud may miss small specular highlights
  - OpenGL implements Gouraud but not Phong
- Calculate lighting model at each pixel: work
  - Can use programmable shaders now

Flat vs. Gouraud vs. Phong



Texture mapping

- Complex objects with many varying shades:
  - Could use a new polygon for every shade
  - Or use an image pasted on top of the surface
- E.g., modeling the earth:
  - Blue sphere is too simple
  - Modeling every continent and mountain range with little polygons is too much
  - Texture-map a picture onto the sphere



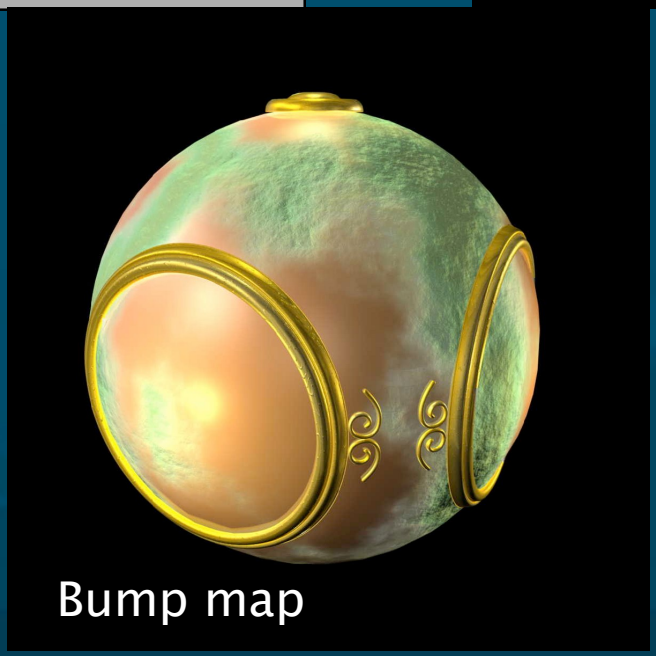
Bump mapping

- e.g., modeling an orange:
  - Geometry is just a simple sphere
  - Texture map colours, striations, etc.
  - But surface is still smooth: what about small dimples?
    - Shading should change as light and view directions change
- Bump mapping tweaks the normal vectors to simulate dimples or bumpiness
  - Silhouette still reflects underlying geometry

Kinds of maps

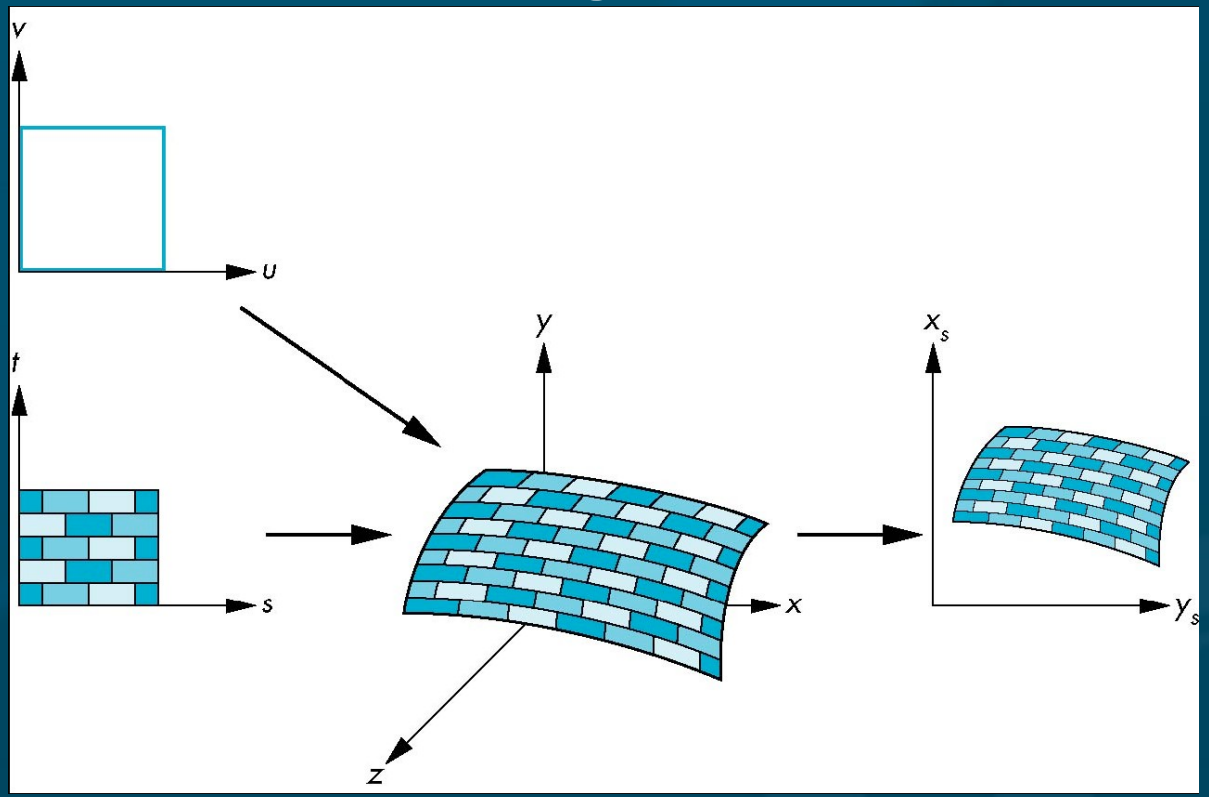
- Texture map:
  - Paste an image onto a surface
- Bump map:
  - Perturbs normal vectors in lighting model to simulate small changes in surface orientation
- Environment (reflection) map:
  - Use a picture of the surrounding room/sky for a texture map
  - Simulates reflections in highly specular surfaces
- Only texture maps are built-in to OpenGL

Texture/bump/environment maps



Mapping: coordinate systems

- Essential question for maps: how to map coordinate systems?
  - Parametric coords (u,v) describing the surface
  - Texture coords (s,t)
  - World coords (x,y,z)
  - Window coords (x_s, y_t)

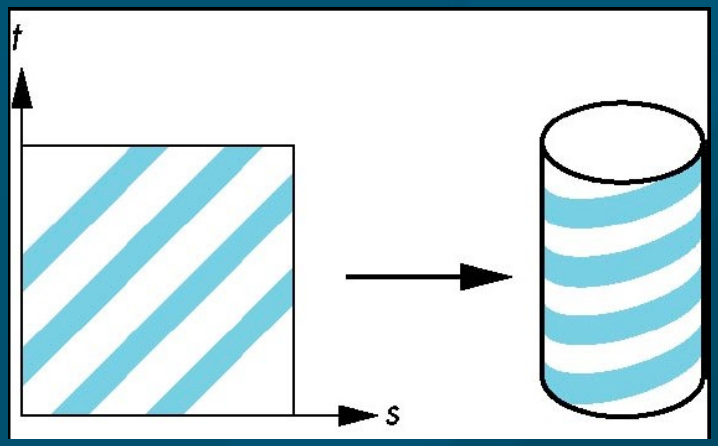


Backward mapping

- For each point (x,y,z) on the surface in world coords, we want to go backwards to find which pixel (s,t) in the texture we should paste:
  - s = s(x,y,z);
  - t = t(x,y,z);
- Two-part mapping:
  - First map texture onto a simple intermediate shape
    - Cylinder
    - Sphere
    - Cube

Cylindrical mapping

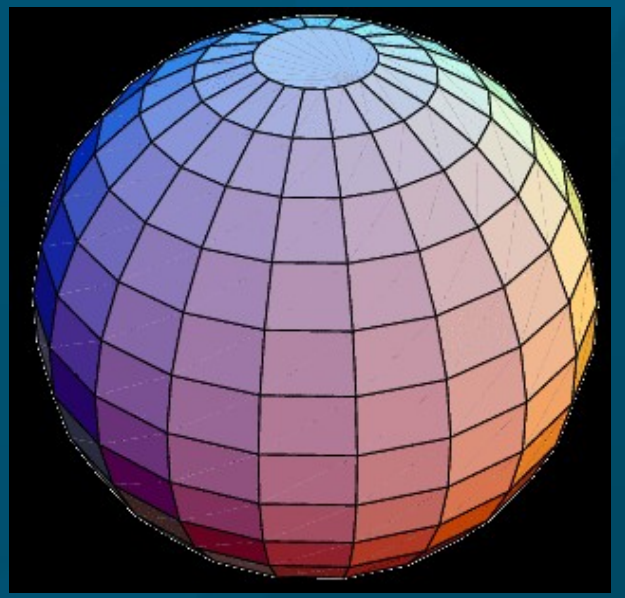
- Parametric cylinder:
  - $x = r \cos(2\pi s)$
  - $y = r \sin(2\pi s)$
  - $z = t/h$

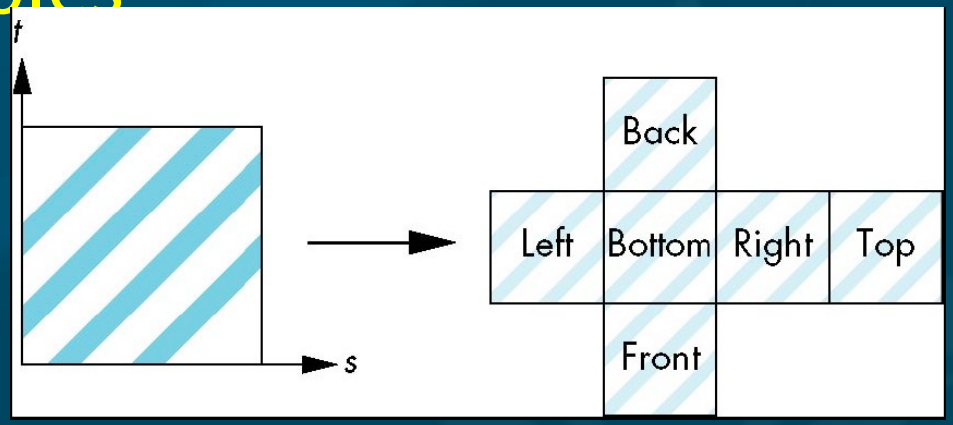


- Map from
  - Square [0,1] x [0,1] in (s,t) texture space to
  - Cylinder of radius r, height h
    in (x,y,z) world coordinates

Spherical maps, cube maps

- Parametric sphere:
  - $x = r \cos(2\pi s)$
  - $y = r \sin(2\pi s) \cos(2\pi t)$
  - $z = r \sin(2\pi s) \sin(2\pi t)$
  - Bad distortions at the poles
- Cube/box mapping:
  - Easy with orthographic projection





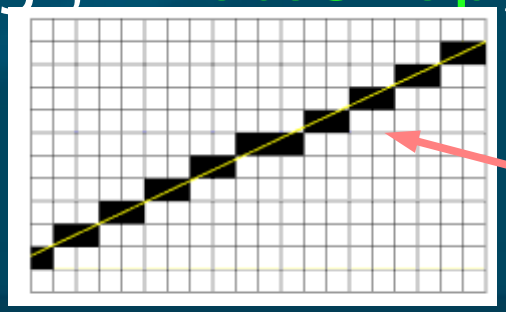
- Both are widely used for environment maps

Implementing bump mapping

- Parameterized surface:
  - $p(u,v) = (x(u,v), y(u,v), z(u,v))$
  - Tangent vectors: $p_u = \partial p/\partial u$, $p_v = \partial p/\partial v$
  - Normal vector: $n = p_u \times p_v$
- Perturbed surface: $p'(u,v) = p(u,v) + d(u,v)\, n(u,v)$
  - $d(u,v)$ is the displacement function/map
- Perturbed normal: $n' = p'_u \times p'_v$
  - $n' \approx (\partial d/\partial u)(n \times p_v) + (\partial d/\partial v)(n \times p_u)$

Rasterization

- Our model is represented with mathematical precision, but
- Ultimately we need to produce a framebuffer: 2D array of RGB values (pixels)
- The rasterization process means that we lose some precision
  - Vector graphics (e.g., Illustrator, EPS) vs.
  - Raster graphics (e.g., PhotoShop, PNG)



TODO

- Lab4: due tonight
  - Add a virtual trackball using quaternions