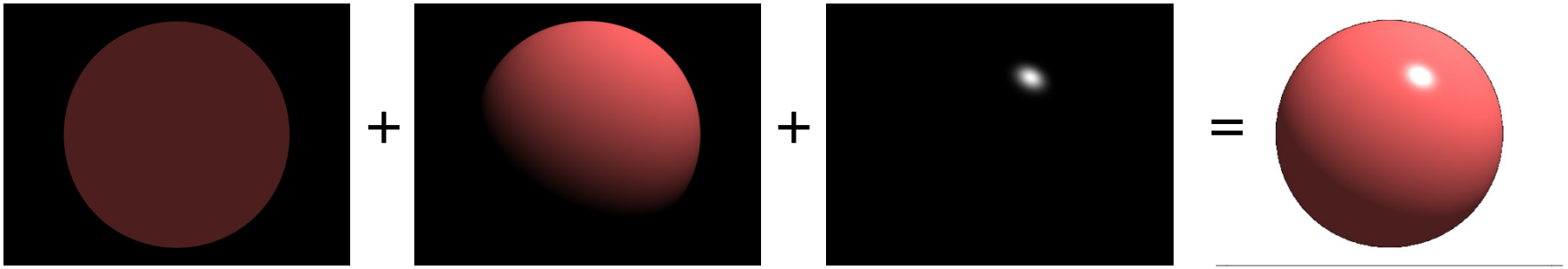
CSC 321 Computer Graphics

Polygon Shading

Review

- Local Illumination Model (1-hop reflection only)
  - Non-physical model: “looks good”
  - Ambient, diffuse and specular components



Ambient Diffuse Specular

$$I = I_{amb} + I_{diff} + I_{spec}$$

$$= I_A k_a + I_L f_{att} (k_d (N \cdot L) + k_s (R \cdot V)^n)$$

Rendering Polygonal Models

- Most scenes are modeled by polygons

Drawing A Polygon

- Compute location of polygon vertices on screen
  - Transformations
- See if the whole or part of the polygon is visible
  - Visibility culling
- Color each pixel in the visible part of the polygon
  - Polygonal shading
  - Texturing

Computing Polygon Vertices

- Object and camera transform
  - Object transformation (obtained from a scene graph)
    - Transformation matrix for o3:

      $M = m2\ m4\ m5$

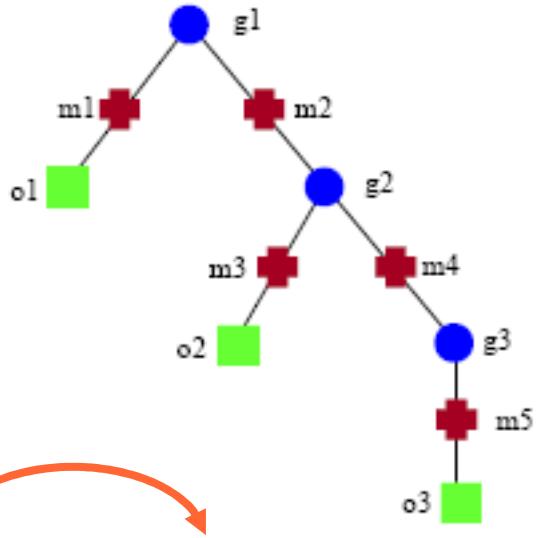
  - Camera transformation
    - World-to-camera, perspective transform

      $C = D\ S_{xyz}\ S_{xy}\ R\ T$

  - Combined:

    $\{x_s, y_s, z_s, w\} = C\ M\ \{x, y, z, 1\}$

    - Normalized: $\left\{\frac{x_s}{w}, \frac{y_s}{w}, \frac{z_s}{w}\right\}$



Screen (X,Y)

Depth into the screen

Visibility

- Is the polygon visible?
  - View-volume clipping: do not draw the polygon if it completely lies outside the view frustum
    - Clip against the 6 walls of the viewing volume
  - Back-face culling: do not draw the polygon if it faces “away” from the viewer.
    - Take the dot product of the polygon normal and the look vector.
      - If positive, the polygon is back facing and not drawn
    - Orientation of the polygon becomes critical
  - What about occlusion?

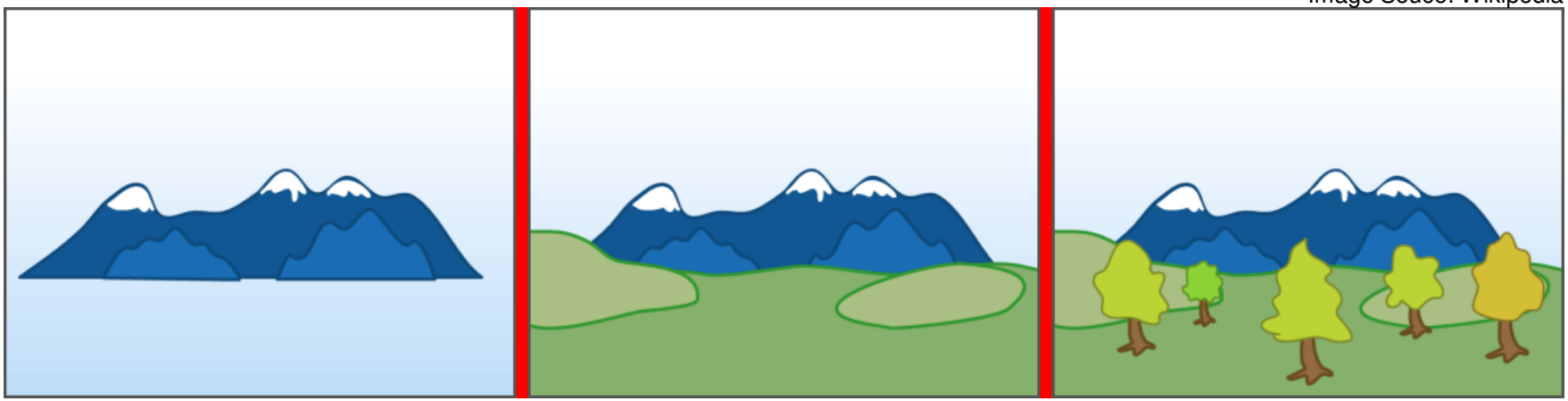
Occlusion culling

$$\left\{ \frac{x_s}{w}, \frac{y_s}{w}, \frac{z_s}{w} \right\}$$

Depth into the screen

- Painter's algorithm
  - Transform all polygons into screen coordinates
  - Sort polygons by the minimum depth of its vertices
  - Draw polygons from back to front

Image Souce: Wikipedia

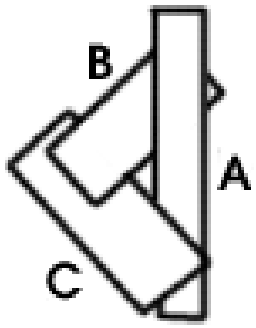


Back → Front

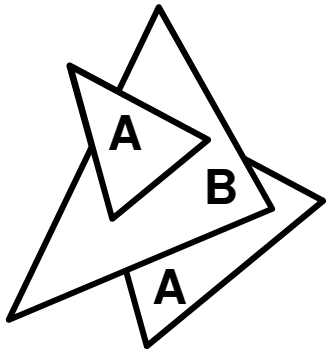
Painter's Algorithm

- Problem
  - Can not properly show intersecting/tangling polygons

Tangling:



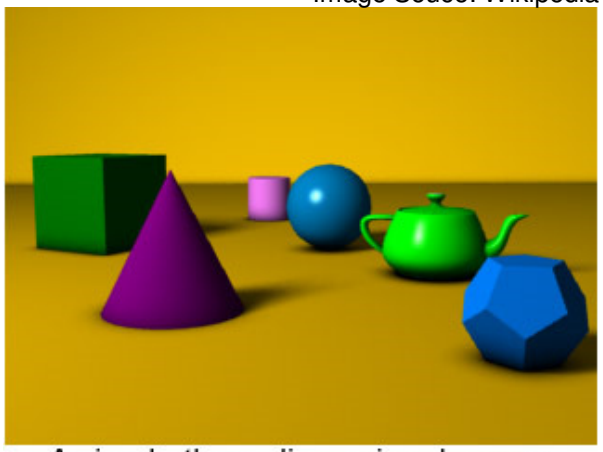
Intersecting:



Z-Buffer

- Z-buffer (depth buffer): an array storing the minimal depth at each pixel
  - After transformation, depth ranges from 0 (near plane) to 1 (far plane)
  - When multiple points projected onto a same pixel, only the point with a smaller depth is drawn.

Image Souce: Wikipedia

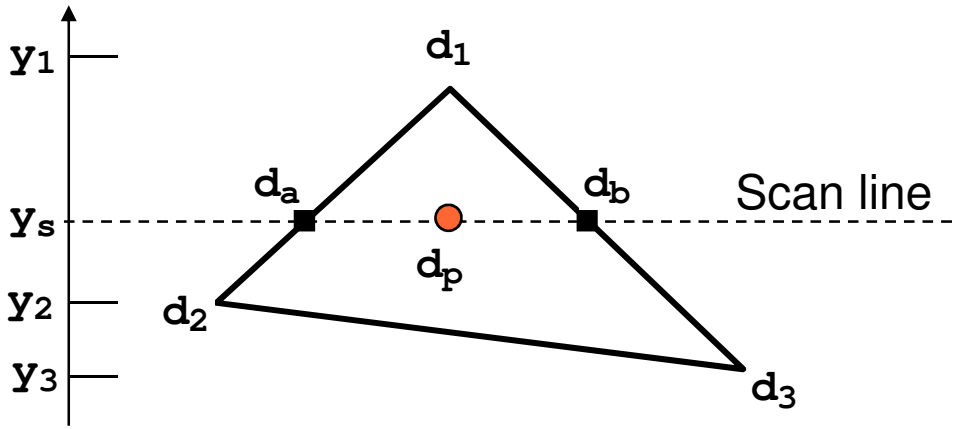


A simple three dimensional scene

Z-buffer representation

Z-Buffer

- Algorithm
  - Z-buffer Initialized to be 1 (far plane) for each pixel
  - Compute projected coordinates of the vertices and their depth
  - Scan-convert interior pixels of the polygon
    - Compute depth at each pixel by linear interpolation
  - Only draw the pixel (and update the z-buffer for that pixel) if the depth of the pixel is smaller than what's currently in the z-buffer



$$d_a = d_1 \frac{y_s - y_2}{y_1 - y_2} + d_2 \frac{y_1 - y_s}{y_1 - y_2}$$

$$d_b = d_1 \frac{y_s - y_3}{y_1 - y_3} + d_3 \frac{y_1 - y_s}{y_1 - y_3}$$

$$d_p = d_a \frac{x_b - x_p}{x_b - x_a} + d_b \frac{x_p - x_a}{x_b - x_a}$$

Better culling

- Discard occluded triangles before projection

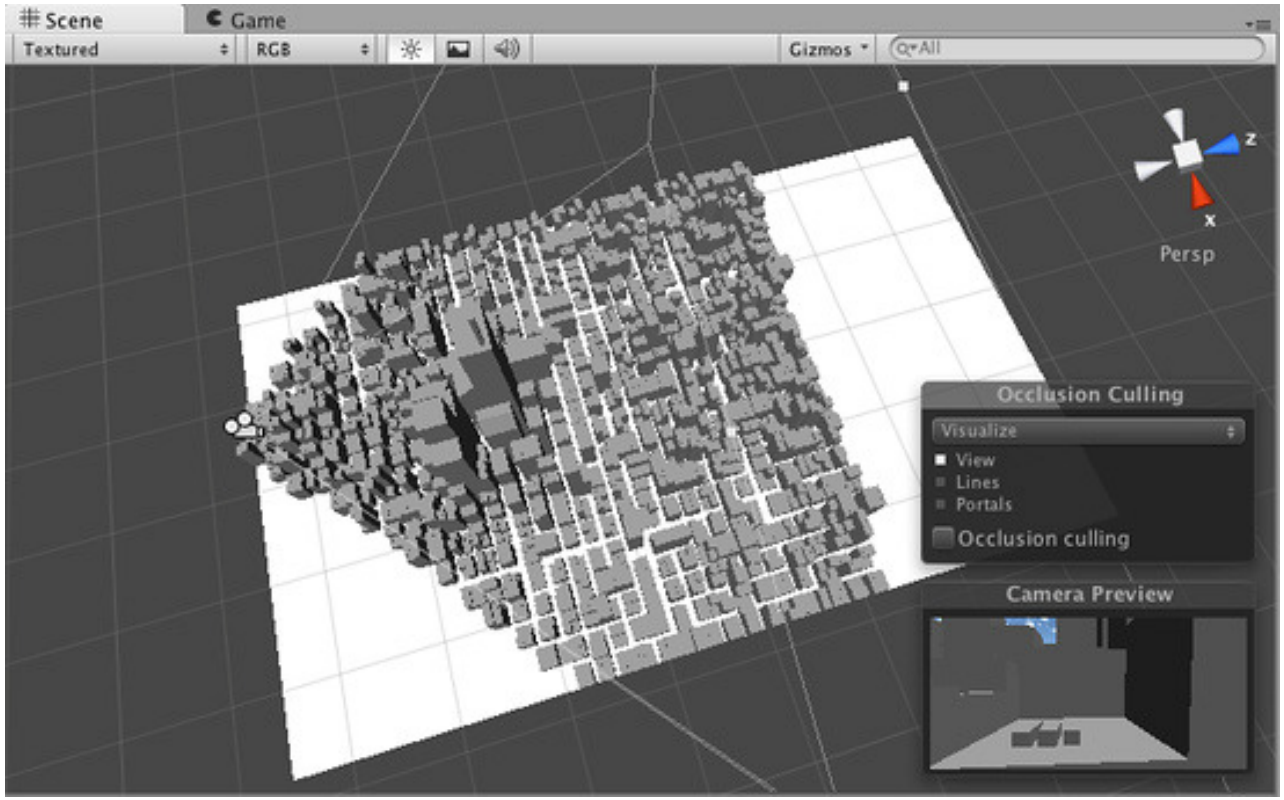


Image source: unity3d.com

Better culling

- Discard occluded triangles before projection

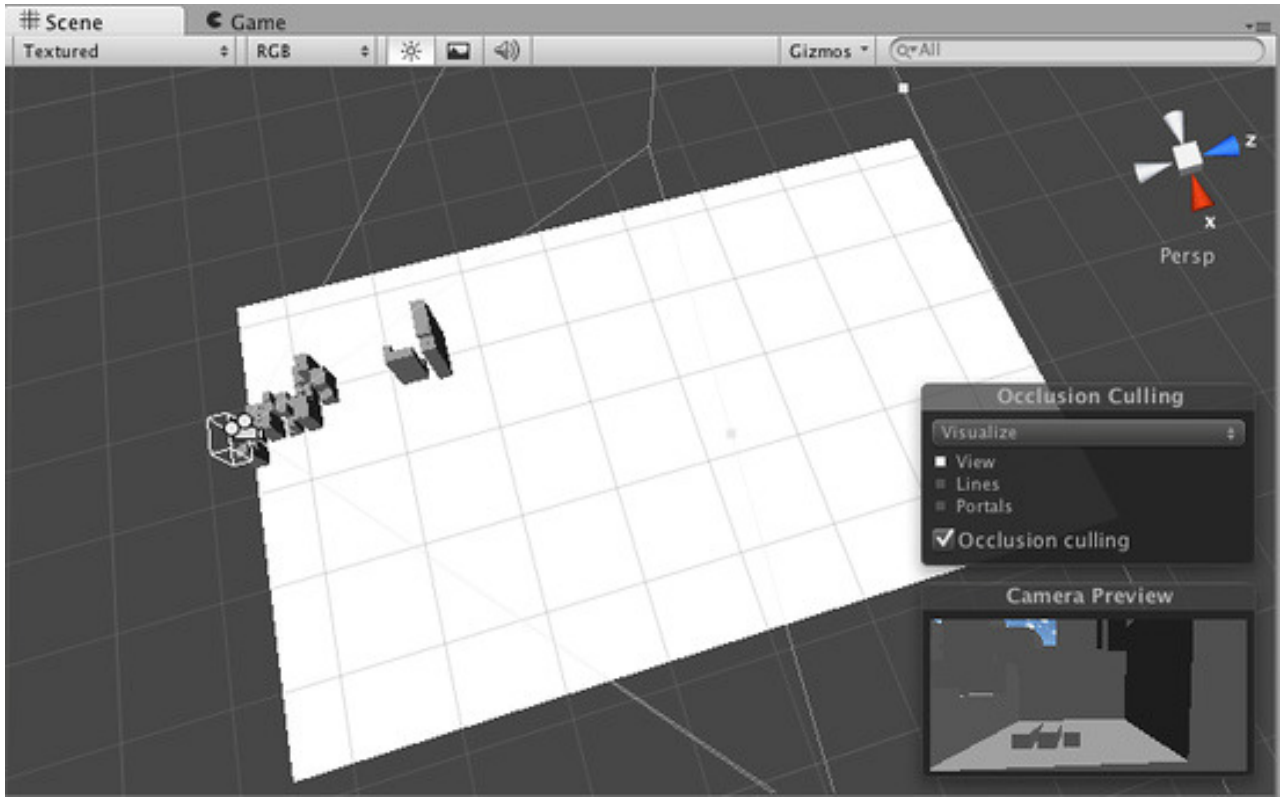


Image source: unity3d.com

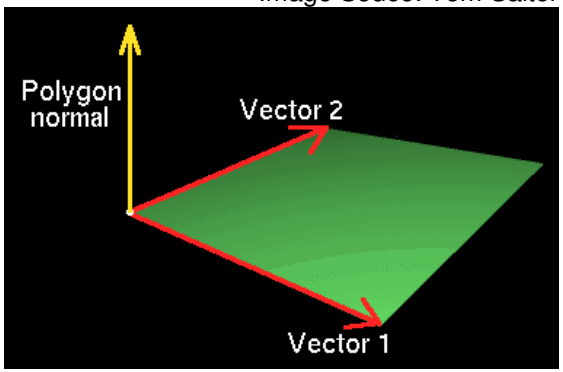
Polygon Shading

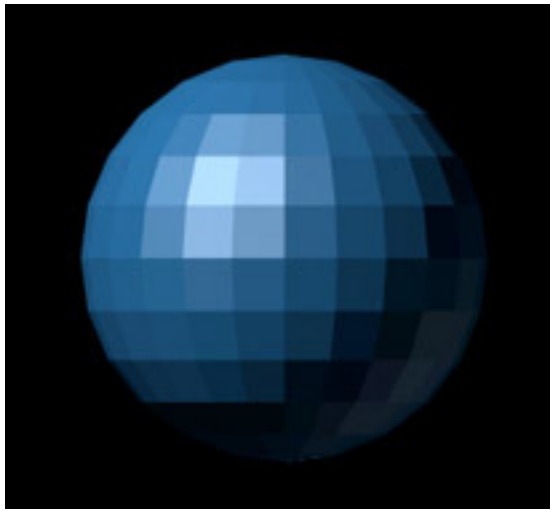
- Using local illumination model (for efficiency)
  - Compute one color per polygon (flat)
  - Compute one color per vertex (Gouraud)
  - Compute one color for each pixel (Phong)

Flat Shading

- Use one color for each polygon
  - Pick any polygon vertex
  - Use the (normalized) cross-product of edge vectors as the normal
  - Compute local illumination
- Pros: Fastest
- Cons: Facet-looking
  - Discontinuity across edges

Image Souce: Tom Salter





Gouraud Shading

- Compute the color at each vertex
  - Use the average normal of the incident polygons
- Blend the color within the polygon
  - Using linear interpolation

$$\mathbf{N_v} = \frac{\mathbf{N_1} + \mathbf{N_2} + \mathbf{N_3} + \mathbf{N_4}}{\mathbf{4}}$$

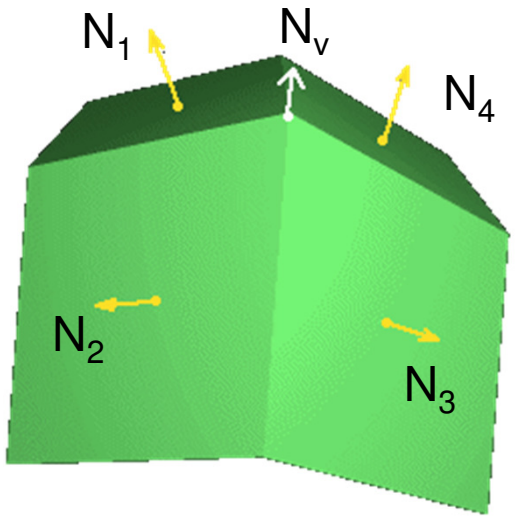
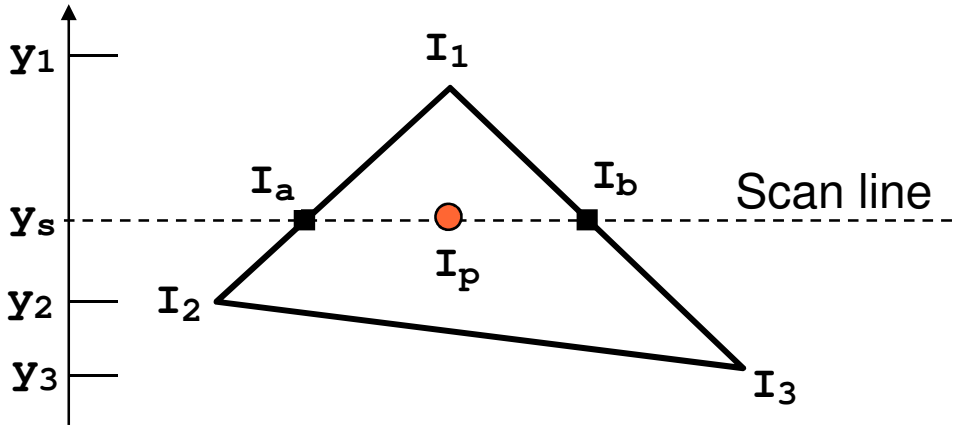


Image Souce: Tom Salter

Gouraud Shading

- Scan-line algorithm (similar to z-buffer):
  - Compute projected coordinates and color at each vertex
  - Scan-convert interior pixels of the polygon
    - Compute color at each pixel by linear interpolation



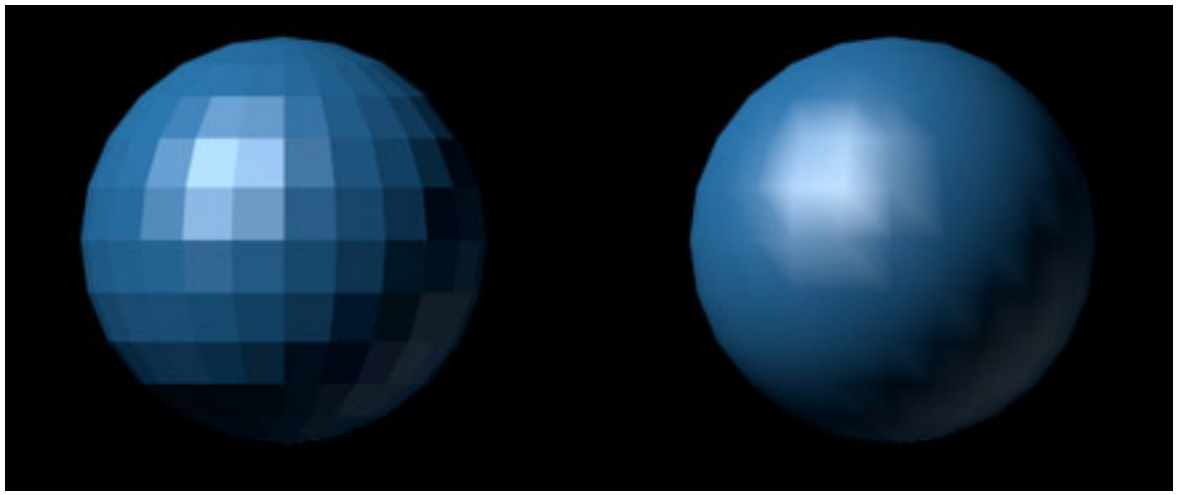
$$I_a = I_1 \frac{y_s - y_2}{y_1 - y_2} + I_2 \frac{y_1 - y_s}{y_1 - y_2}$$

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

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

Gouraud Shading

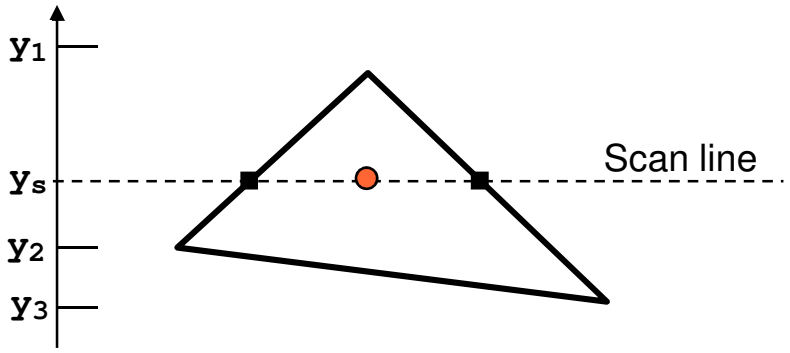
- Pros: Smoother looking than flat shading
  - Color continuous across edges
- Cons
  - Patchy, because color is not smooth (or *derivative* not continuous)
  - Details (e.g., highlight) missing within a polygon

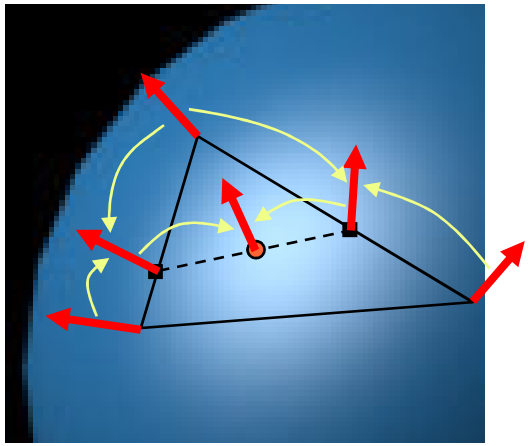


Flat Gouraud

Phong Shading

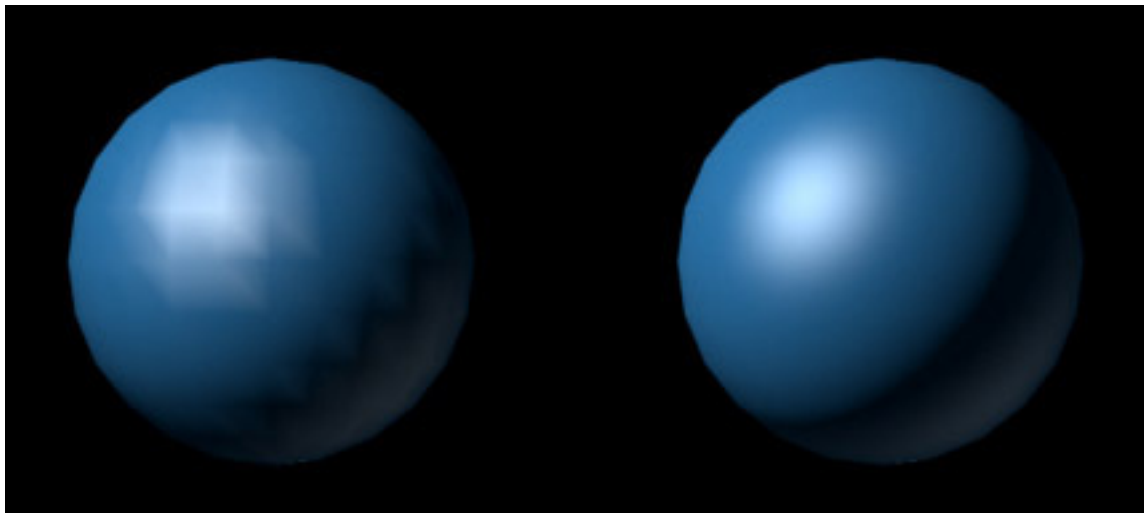
- Using smoothly varying normals
  - Compute a normal for each scan-converted pixel
    - Linear interpolation of normals at the vertices (using scan-line)
  - Compute illumination
    - Using the interpolated normal and the location of the 3D point





Phong Shading

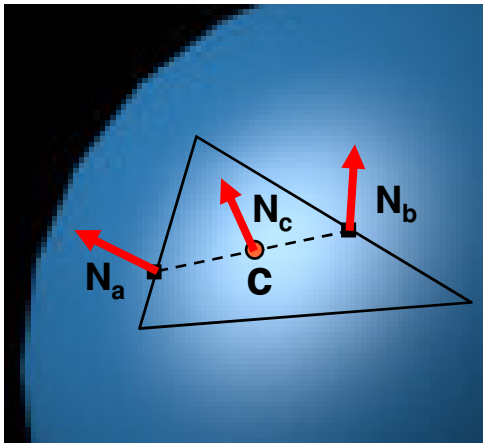
- Pros:
  - Smoother (“Faking” a smooth surface)



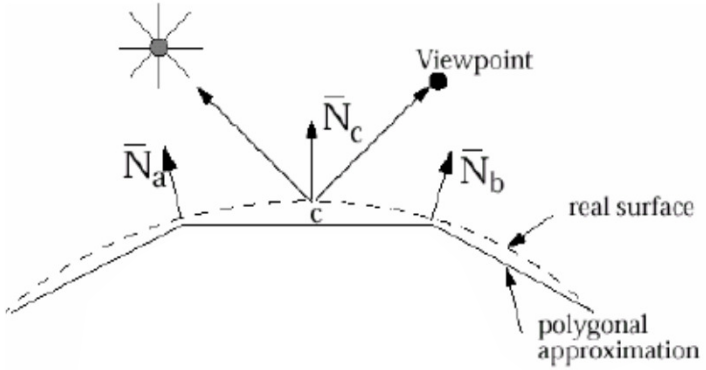
Gouraud Phong

Phong Shading

- Pros:
  - Smoother (“Faking” a smooth surface)
  - Capturing specular lighting inside the polygon

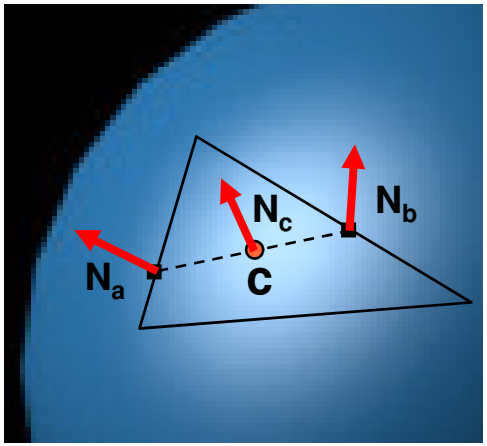


Cross-section view

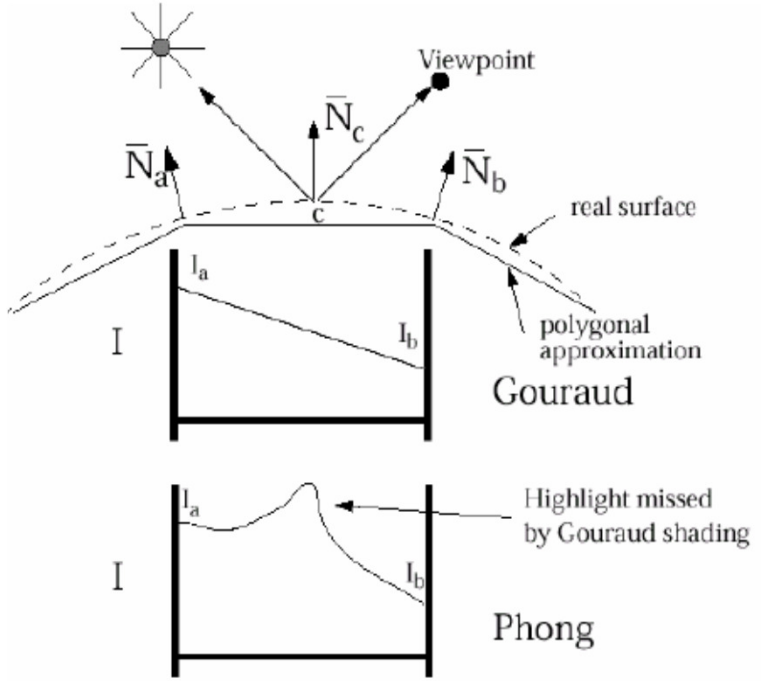


Phong Shading

- Pros:
  - Smoother (“Faking” a smooth surface)
  - Capturing specular lighting inside the polygon



Cross-section view

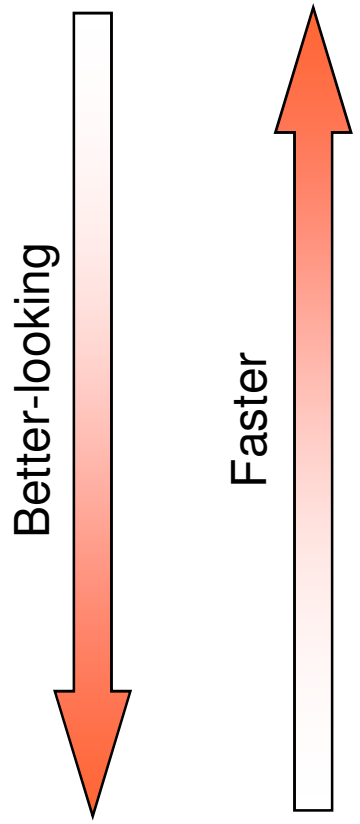


Phong Shading

- Pros:
  - Smoother (“Faking” a smooth surface)
  - Capturing specular lighting inside the polygon
- Cons: much slower
  - OpenGL does not implement Phong

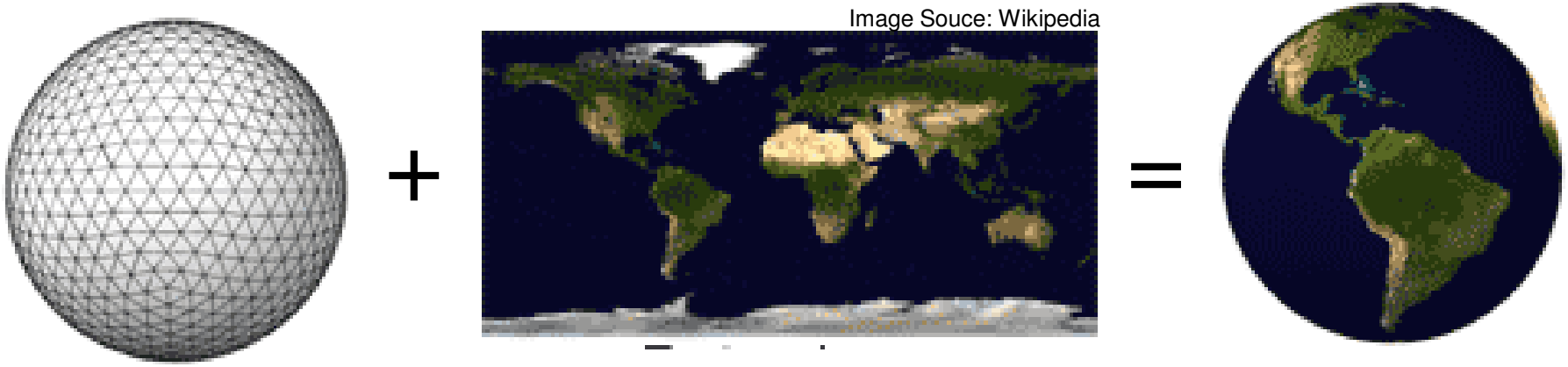
Summary

- Flat shading
  - Compute local illumination once per polygon
- Gouraud shading
  - Compute local illumination once per vertex
  - Interpolate color across the polygon
- Phong shading
  - Interpolate normal across the polygon
  - Compute local illumination per pixel



Texture

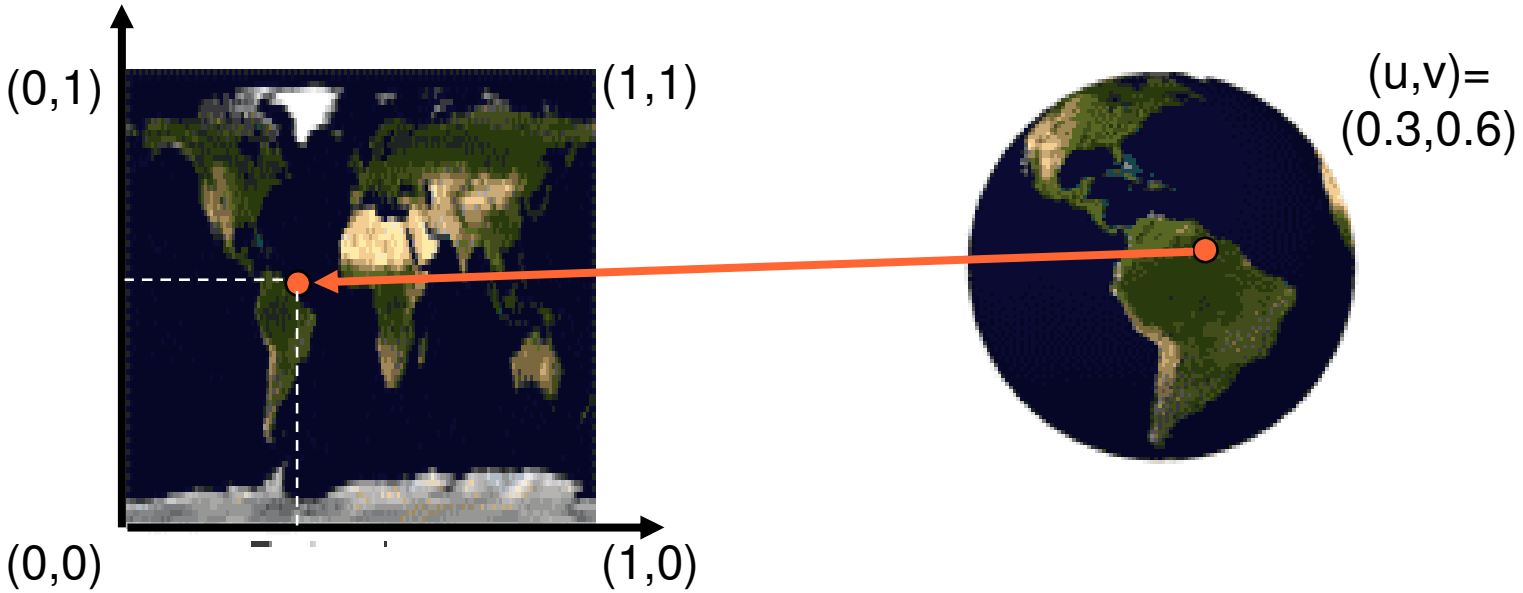
- Simple, effective approach for adding surface detail



Sphere with no texture | Texture image | Sphere with texture

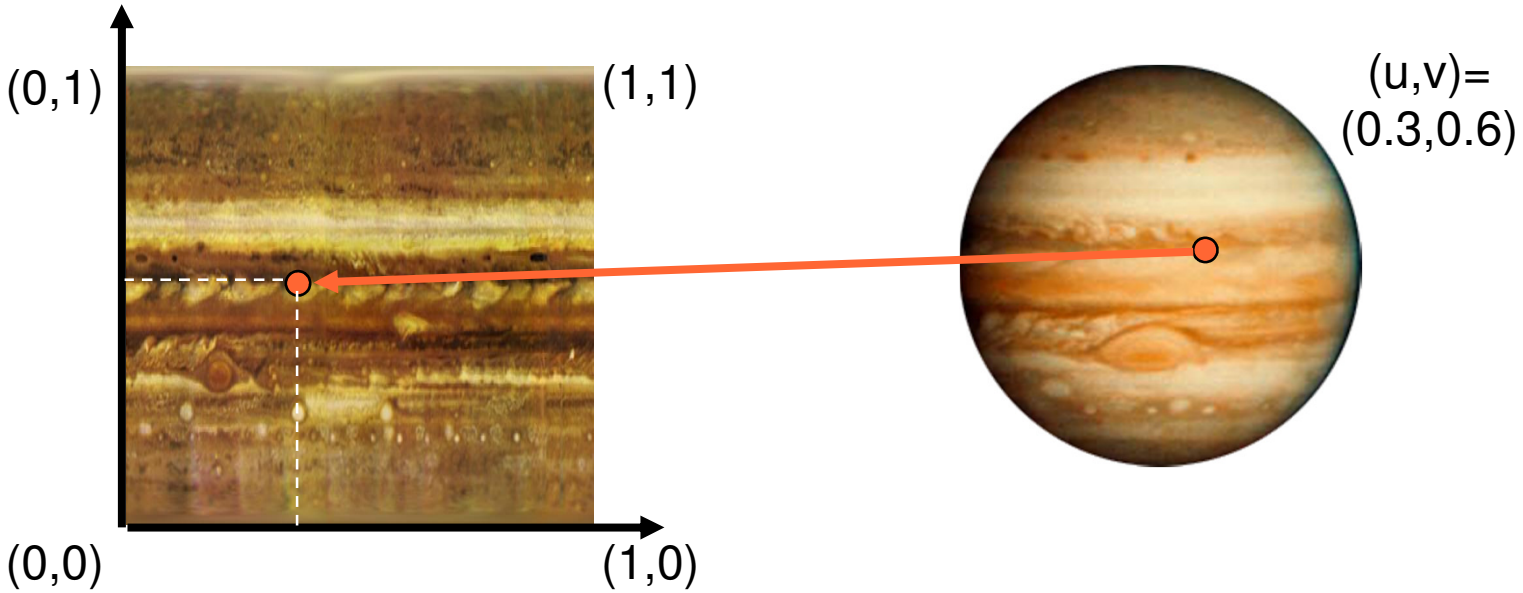
Texture Coordinates

- Uses two parameters (u,v)
  - Each object point is associated with a pair (u,v) between 0 and 1
  - Given a texture image in the unit square, the object point is shaded using the color at (u,v) on the texture image



Texture Coordinates

- Uses two parameters (u,v)
  - Each object point is associated with a pair (u,v) between 0 and 1
  - Given a texture image in the unit square, the object point is shaded using the color at (u,v) on the texture image



Texture Mapping

- Find a texture image (the easy part)
- Get texture coordinates for each point on the model (the hard part)
  - First, find texture coordinates for each polygon vertex
    - Parameterization
  - Interpolate texture coordinates within each polygon
    - Using scan-line algorithm

Mapping a Cylinder

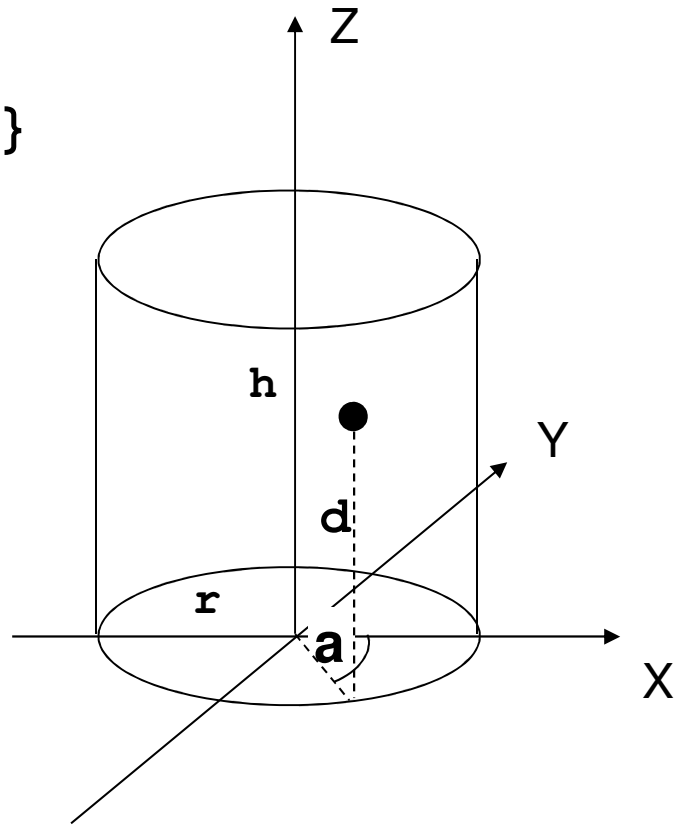
- Cylindrical parameterization

$$p[d, \alpha] = \{r\,\mathrm{Cos}[\alpha],\ r\,\mathrm{Sin}[\alpha],\ d\}$$

$$0 \le d \le h,\ 0 \le \alpha < 2\pi$$

  - Normalize d, a to correct range.

$$u = \frac{\alpha}{2\pi},\ v = \frac{d}{h}$$



Mapping a Sphere

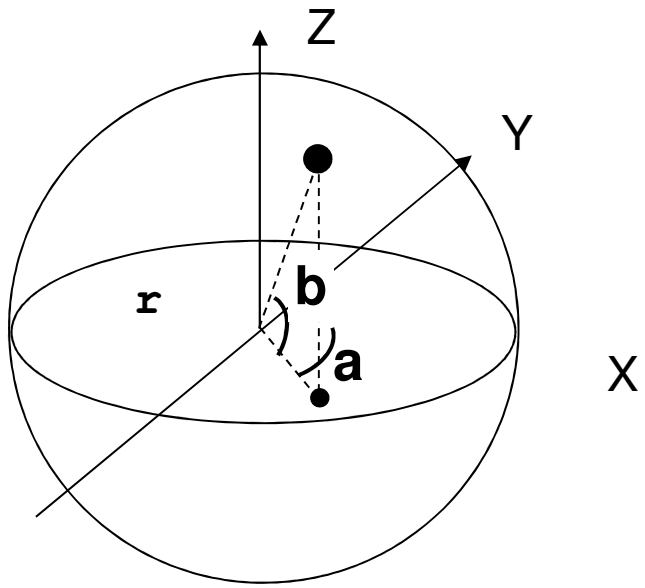
- Longitude-latitude parameterization

$$\mathbf{p}[\alpha, \beta] = \{\mathbf{r}\,\mathbf{Cos}[\beta]\,\mathbf{Cos}[\alpha],\ \mathbf{r}\,\mathbf{Cos}[\beta]\,\mathbf{Sin}[\alpha],\ \mathbf{r}\,\mathbf{Sin}[\beta]\}$$

$$0 \le \alpha < 2\pi,\ \frac{-\pi}{2} \le \beta \le \frac{\pi}{2}$$

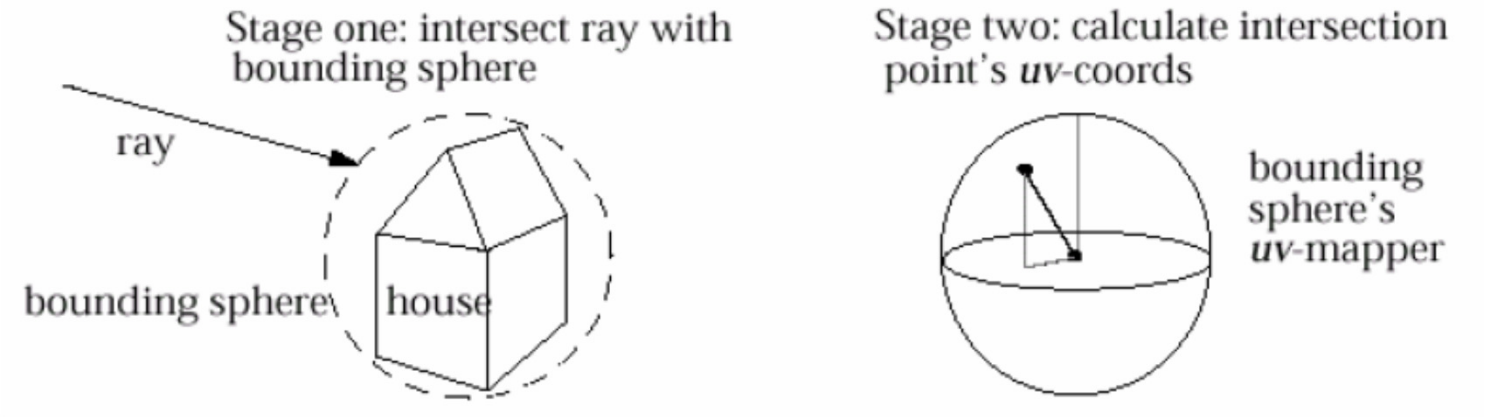
  - Normalize $\mathbf{a}, \mathbf{b}$ to correct range

$$\mathbf{u} = \frac{\alpha}{2\pi},\ \mathbf{v} = \frac{\beta + \pi/2}{\pi}$$



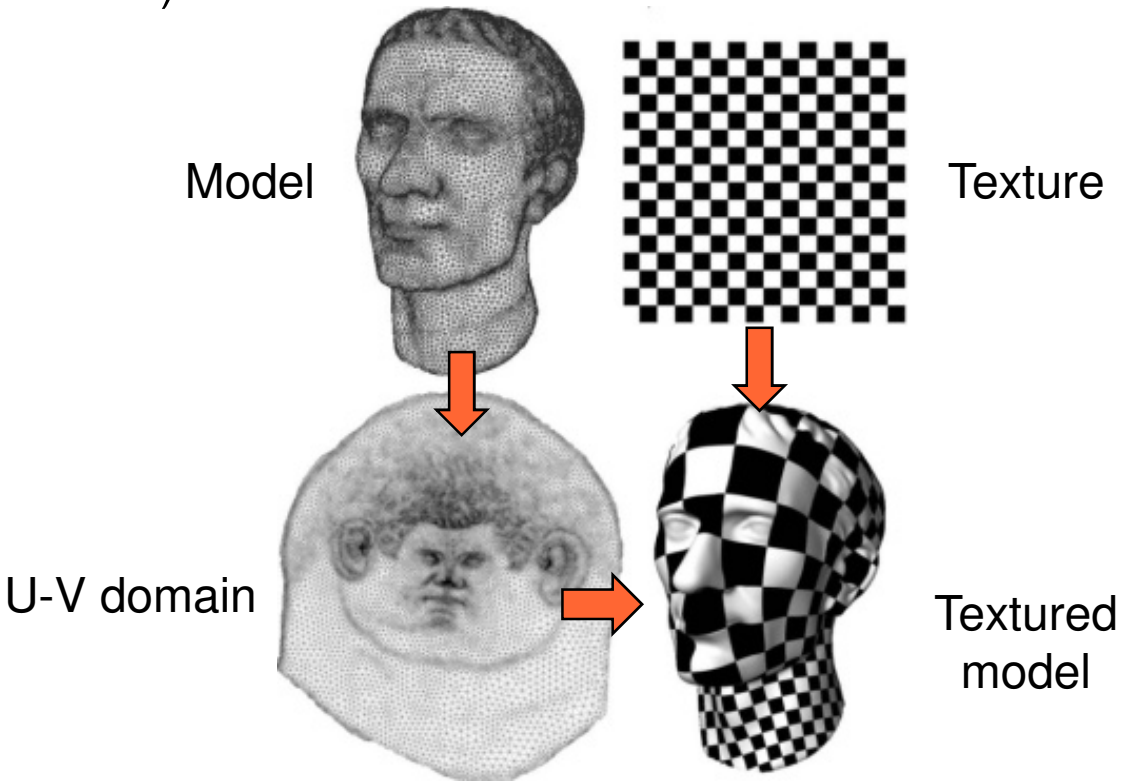
What about arbitrary shapes?

- Mapping to a sphere
  - Wrap the object in a sphere
  - Use the u,v parameter of the projected point on the sphere



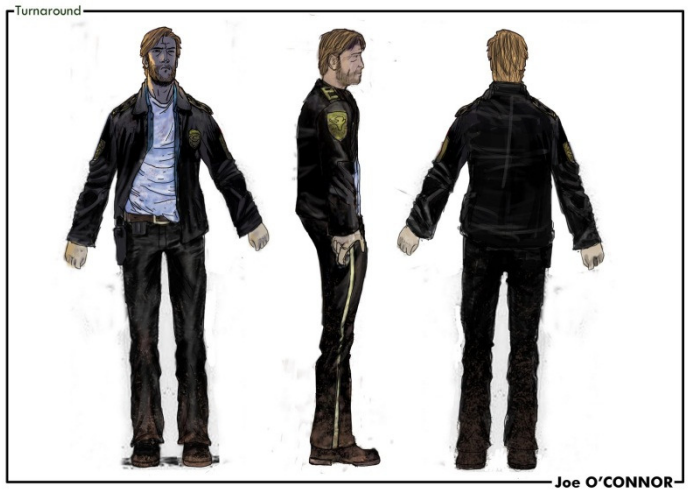
What about arbitrary shapes?

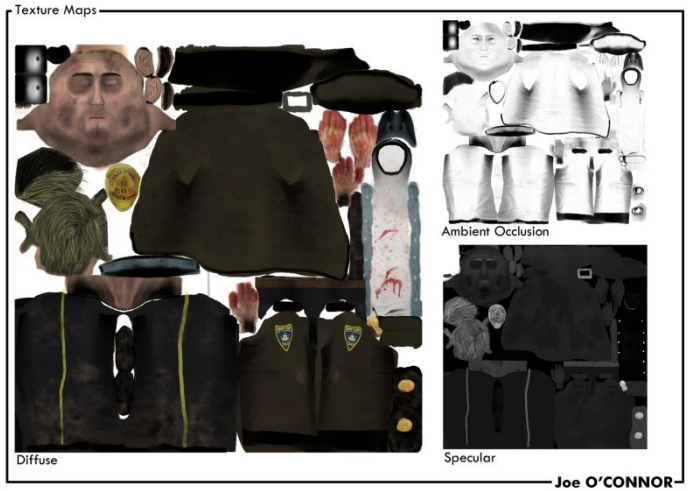
- Global parameterization
  - Minimizing distortions (e.g., preserving triangle angles or areas in the UV domain)



What about arbitrary shapes?

- Using a texture atlas
  - Cut the shape into different patches (manually or automatically) and compute U,V coordinates for each patch
  - Give one texture image for each patch
  - Often used for characters/animals in games





Bump Mapping

- "Fake geometry" by varying surface normal
  - The map encodes height variation (as grayscale), from which the normal can be computed
    - Needs Phong shading to compute per-pixel lighting

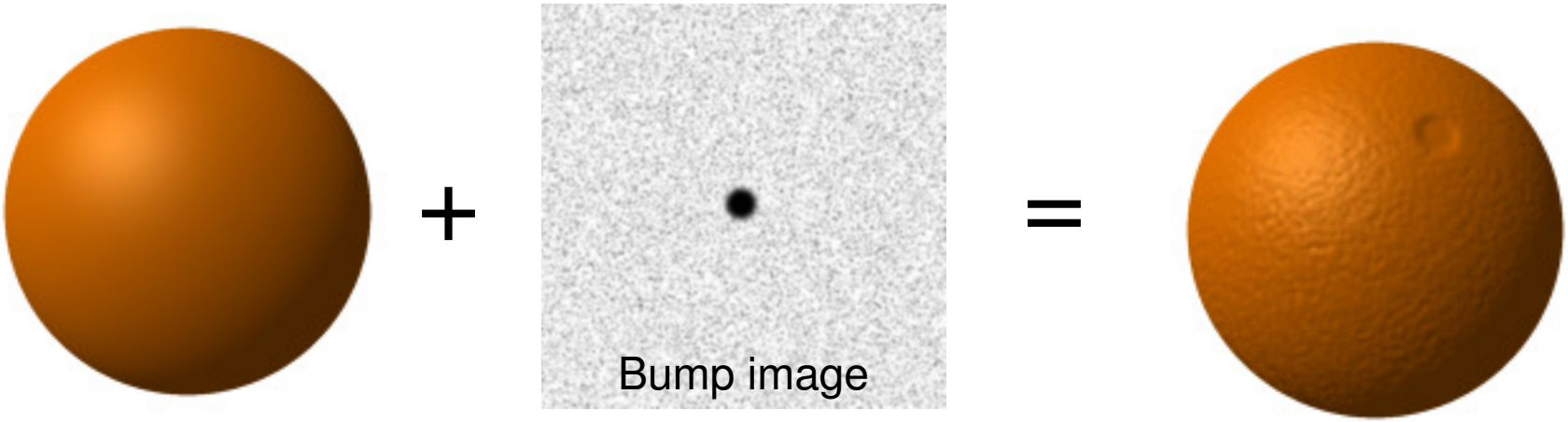
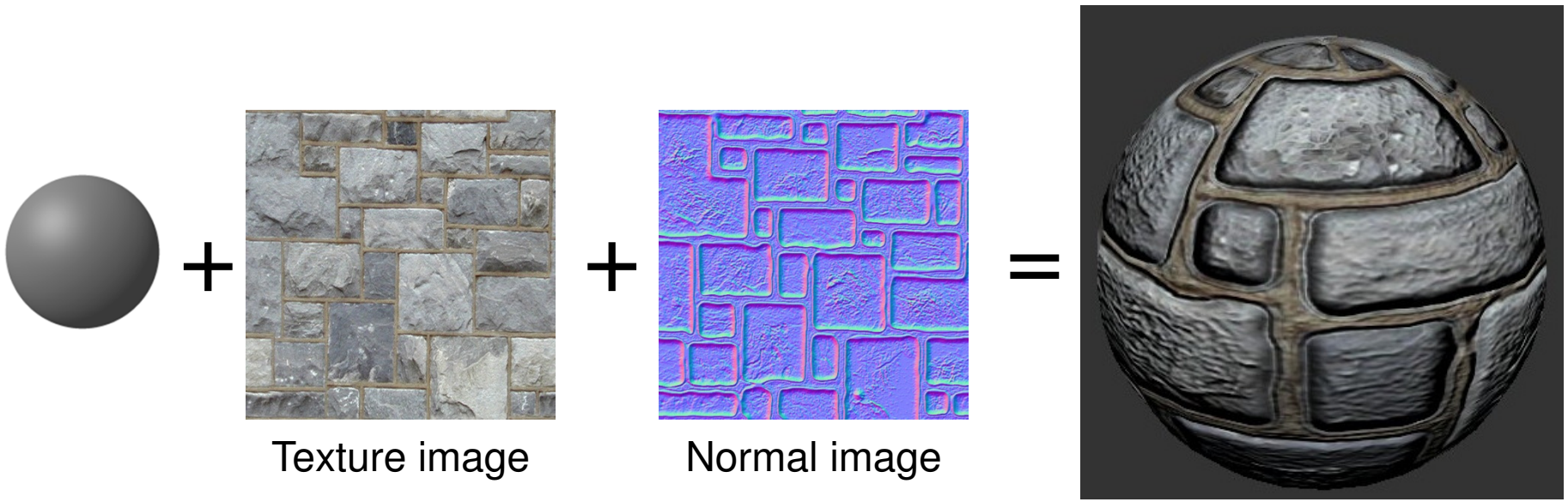


Image Souce: Wikipedia

Normal Mapping

- "Fake geometry" by defining new normal at each point
  - Encodes actual normal (X,Y,Z as R,G,B)



Texture image

Normal image

Bump & Normal Mapping

- Limitation: cannot change the silhouette of the shape