Representing smooth and detailed surfaces using textures
Goal: representing smooth & complex surfaces
Can we do it with a small number of polygons?

**quiz:** how would you do it?
Idea: shading normal

for each point on the surface, assign a “fake” normal to use when rendering
Generating shading normals for smooth surfaces
Naive rendering leads to faceted appearance
Trick: assign a normal per vertex, then interpolate

\[ n = \text{normalize} \left( (1 - b_1 - b_2)n_0 + b_1n_1 + b_2n_2 \right) \]

a.k.a. "Phong interpolation"
without vertex normal

with vertex normal
How do we obtain vertex normal?

- input: normals on triangle faces
- output: normals on triangle vertices

**quiz:** how would you do it?
Option 1: average over facet normals

for each vertex:
    n = 0
    for each neighbor face:
        n += face.normal
    vertex.normal = normalize(n)

quiz: what is the downside?
Plain average does not account for triangle sizes
Option 2: area-based averaging

```
for each vertex:
    n = 0
    for each neighbor face:
        n += face.area * face.normal
    vertex.normal = normalize(n)
```

**quiz:** what is the downside?
Plain average vs area-based average

from Martijn Butjs
https://www.bytehazard.com/articles/vertnorm.html
An issue of area-based averaging

**quiz:** why does this happen?
Option 3: angle+area-based averaging

for each vertex:
    \( n = 0 \)
    for each neighbor face:
        \( n += \text{face.area} \times \theta \times \text{face.normal} \)
    vertex.normal = normalize(n)
How do we deal with sharp discontinuities?

without vertex normal

with vertex normal
Trick: duplicate an edge if it is sharp

for code, check out https://github.com/mitsuba-renderer/mitsuba/blob/10af06f365886c1b6dd8818e0a3841078a62f283/src/librender/trimesh.cpp#L468
Discrepancies between shading normal and real geometry can lead to artifacts.

(*quiz*) Why does this happen?

From “Hacking the Shadow Terminator”, Johannes Hanika
Issue: shadow ray being blocked by the low-poly geometry

From “Hacking the Shadow Terminator”, Johannes Hanika
Issue: shadow ray being blocked by the low-poly geometry

Hanika’s trick: offset the origin of the shadow ray to the smooth surface

From “Hacking the Shadow Terminator”, Johannes Hanika
Discrepancies between shading normal and real geometry can lead to artifacts.
How do we use shading normals in rendering equation?
How do we use shading normals in rendering equation?

\[
\int f(\omega_v, \omega_l, n) \mid n \cdot \omega_l \mid L(\omega_l) d\omega_l
\]

replace these with shading normals
How do we use shading normals in rendering equation?

\[ \int f(\omega_v, \omega_l, n) \left| n \cdot \omega_l \right| L(\omega_l) d\omega_l \]

replace these with shading normals

\[ \int f(\omega_v, \omega_l, n) \left| n \cdot \omega_l \right| \frac{\left| n' \cdot \omega_l \right|}{\left\| x' - x \right\|^2} L(\omega_l) dx' \]

do we replace this?
How do we use shading normals in rendering equation?

\[
\int f(\omega_v, \omega_l, n) \left| n \cdot \omega_l \right| L(\omega_l) d\omega_l
\]

replace these with shading normals

\[
\int f(\omega_v, \omega_l, n) \left| n \cdot \omega_l \right| \frac{\left| n' \cdot \omega_l \right|}{\| x' - x \|^2} L(\omega_l) dx'
\]

do we replace this? (Nope)
Generating shading normals for detailed surfaces

*quiz*: how would you do this?
Idea: use textures to represent details
Normal mapping idea:
perturb the normal using a texture
Normal maps are usually stored in "local" coordinates 

$(0, 0, 1)$
Local coordinates = tangent space of surface

aka “shading frame”

n: normal
t: tangent
b: bitangent

https://www.cs.cornell.edu/courses/cs5625/2016sp/slides/03detail-map.pdf
Constructing tangent space for meshes

\[ n = \text{shading\_normal} \]
\[ t = \text{normalize}(\frac{dp}{du} - n \cdot \text{dot}(n, \frac{dp}{du})) \]
\[ b = \text{cross}(n, t) \]

see https://github.com/BachiLi/lajolla_public/blob/main/src/shapes/triangle_mesh.inl#L65 for code
Applying normal maps

// given n, t, b as our shading frame
n_local = 2 * normal_map(u, v) - 1
n_world = n * n_local.z +
         t * n_local.y +
         b * n_local.z
new_normal = normalize(n_world)
Applying normal maps

// given n, t, b as our shading frame
n_local = 2 * normal_map(u, v) - 1
n_world = n * n_local.z +
          t * n_local.y +
          b * n_local.z
new_normal = normalize(n_world)

new_tangent = normalize(
    dpdu - new_normal * 
    dot(new_normal, dpdu))
new_bitangent =
    cross(new_normal, new_tangent)
Bump mapping: normal mapping using height maps
Getting normal maps from height maps

https://www.cs.cornell.edu/courses/cs5625/2016sp/slides/03detail-map.pdf
Getting normal maps from height maps

Given $f(x, y, z) = 0$, $n \propto \left( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right)$
Getting normal maps from height maps

given $f(x, y, z) = 0$, $n \propto \left( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right)$

for a height field, $f(x, y, z) = h(x, y) - z$

https://www.cs.cornell.edu/courses/cs5625/2016sp/slides/03detail-map.pdf
Getting normal maps from height maps

given \( f(x, y, z) = 0 \), \( n \propto \left( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right) \)

for a height field, \( f(x, y, z) = h(x, y) - z \)

\( n = \text{normalize} \left( \frac{\partial h}{\partial x}, \frac{\partial h}{\partial y}, -1 \right) \)
Getting normal maps from height maps

given \( f(x, y, z) = 0 \), \( n \propto \left( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right) \)

for a height field, \( f(x, y, z) = h(x, y) - z \)

\[ n = \text{normalize} \left( \frac{\partial h}{\partial x}, \frac{\partial h}{\partial y} - 1 \right) \]

can be obtained using finite differences

https://www.cs.cornell.edu/courses/cs5625/2016sp/slides/03detail-map.pdf
Generating normal maps

- option 1: use Perlin noise
Generating normal maps

• option 2: simplify from complex geometry

https://www.cs.cornell.edu/courses/cs5625/2016sp/slides/03detail-map.pdf
Generating normal maps

- option 3: computer vision technology (e.g., photometric stereo)

https://en.wikipedia.org/wiki/Photometric_stereo
Parallax mapping: enhancing normal mapping

https://learnopengl.com/Advanced-Lighting/Parallax-Mapping
Idea: perturb uv coordinates based on height

https://learnopengl.com/Advanced-Lighting/Parallax-Mapping
Idea: perturb uv coordinates based on height

https://learnopengl.com/Advanced-Lighting/Parallax-Mapping
Parallax occlusion mapping: enhancement of parallax mapping

(aka relief mapping, steep parallax mapping)

take multiple steps instead of one step

https://learnopengl.com/Advanced-Lighting/Parallax-Mapping
Parallax occlusion mapping is widely used in video games.
Displacement mapping: directly modify the surfaces

- requires **tessellation** of meshes during rendering
- hardware support exists
- more about this next time

Hierarchy of scales

modeling mesoscale appearance is the hardest!

https://www.cs.cornell.edu/courses/cs5625/2016sp/slides/03detail-map.pdf
Next: tessellation & subdivision surfaces