Outline

- scene graphs
- normal mapping
- shadow mapping
Scene Graphs

- general idea
- scene composed of meshes
- hierarchical composition
- can add subgraphs (objects) to scene
Scene Graphs
Scene Graphs
Scene Graphs

root
Scene Graphs
Scene Graphs

root

S

ocean
mesh
Scene Graphs

- root
- S
- hill
- T
- ocean mesh
Scene Graphs

- root
  - S
    - hill
      - T
        - house1
        - ocean mesh
          - hill mesh
Scene Graphs

- root
- S
- house1
- hill
- T
- ocean mesh
- hill mesh
Scene Graphs

```
root
  ↓
S
  ↓
house1
  ↓
T
  ↓
hill
  ↓
T
  ↓
ocean mesh

hill mesh
```

Mesh nodes:
- Ocean mesh
- Hill mesh
- Roof mesh

Relationships:
- Root node
- House1 node
- Ocean
- Hill
- Roof
Scene Graphs

- ocean mesh
- hill mesh
- roof mesh
- wall mesh
- house1
- hill
- root
Scene Graphs

root
  ↓
  S
  ↓
  house1
  ↓
  T
  ↓
  wall mesh
  ↓
  roof mesh
  ↓
  mesh

  ↓
  hill
  ↓
  T
  ↓
  mesh

  ↓
  ocean mesh

  ↓
  house2
Scene Graphs

- root
  - S
    - hill
      - T
        - hill mesh
    - house1
      - T
    - house2
      - TR
- ocean
  - mesh
- mesh
- roof
  - mesh
- wall
  - mesh
Scene Graphs

- root
  - S
    - T
      - hill
        - mesh
      - T
        - hill
          - mesh
      - T
        - roof
          - mesh
        - wall
          - mesh
      - TR
        - roof
          - mesh
  - house1
  - house2
Scene Graph

aiScene contains:
- list of materials
- list of meshes
- root node of scene graph
Scene Graph

- each aiMesh contains:
  - list of vertex attributes
  - list of triangles with vertex indexes
  - material index
Scene Graph

- each aiMaterial contains:
  - specular, diffuse, etc. properties
  - list of textures
  - ignore AssImp material semantics
  - just use texture name and colors
  - _s for specular texture, etc.
Scene Graph

- each aiNode in scene graph contains:
  - transformation matrix
  - applies to all descendants
  - list of mesh indexes
Scene Graph

- scene graphs are hierarchical structures holding meshes, etc. in the scene
- render a scene by traversing the entire scene graph, depth-first search
- DO NOT render just the mesh list
Normal Mapping

- recall bump-mapping lecture
- perturb normals using texture
- normal maps are easier
Normal Mapping

- normal maps contain the normal vector in tangent space
- \((r,g,b) \rightarrow (\text{tangent}, \text{bitangent}, \text{normal})\)
- tangent goes along positive \(u\)
- bitangent is remaining orthogonal direction
Normal Mapping

- Normal: n
- Bitangent: b
- Tangent: t
Normal Mapping

normal map
value
(0.7, 0.14, 0.7)

normal map: -1.0 to 1.0, color: 0.0 to 1.0
need to “expand” from texture color
Normal Mapping

normal map
value
(0.7, 0.14, 0.7)

normal map: -1.0 to 1.0, color: 0.0 to 1.0
need to “expand” from texture color
Normal Mapping

normal map value
(0.7, 0.14, 0.7)

actual normal retrieved from normal map
\[ n' = 0.7t + 0.14b + 0.7n \]

normal map: -1.0 to 1.0, color: 0.0 to 1.0
need to “expand” from texture color
Normal Mapping

how do we get these n,b,t vectors in eye space for lighting at each fragment?

can compute at vertex and interpolate

BUT, orthogonality not guaranteed!

solution: do all lighting in tangent space

compute eye vec and light vec at vertex shader and interpolate
Normal Mapping

- set matrix $A$, columns are $t, b, n$
- then $Ax$ will transform from tangent space to eye space
- $A^{-1}$ will transform from eye space to tangent space
- $A^{-1} = A^T$
Shadow Mapping

two pass technique

first pass:
- compute image from pov of light
- Assignment3 - directional light
  - ortho? frustum?

second pass:
- use info to determine shadowed regions
Shadow Mapping

- how do we use info from first pass in second pass?
- i.e. what pixel in shadow map does our current fragment correspond to?
Shadow Mapping

first pass

$M_s \rightarrow P_s \rightarrow V \rightarrow$ shadow map coordinate

X
Shadow Mapping

X → M → P → V → framebuffer coordinate

second pass
Shadow Mapping

\[ \text{eye coordinate} \rightarrow \text{framebuffer coordinate} \]

\[ x \rightarrow M \rightarrow P \rightarrow V \]

second pass
Shadow Mapping

First Pass:
- $M_s$ → $P_s$ → $V$ → Shadow map coordinate

Second Pass:
- $M$ → $P$ → $V$ → Framebuffer coordinate
- $X$ → Eye coordinate
Shadow Mapping

- do first pass and save \( VP_s M_s \)
- apply \( VP_s M_s \) to vertex position in vertex shader
- take interpolated result
  - use \( \frac{x}{w}, \frac{y}{w} \) to sample shadow map
- compare with \( \frac{z}{w} + \text{bias} \) ← tweak this