Today

- Shadows
- Procedural modeling

Hard and soft shadows

Hard shadow, point light source  Soft shadow, area light source

Shadows for interactive rendering

- Focus on hard shadows
  - Soft shadows often too hard to compute in interactive graphics
- Two main techniques
  - Shadow mapping
  - Shadow volumes
- Many variations, subtleties
- Active research area

Shadow mapping

Main idea

- Scene point is lit by light source if it is visible from light source
- Determine visibility from light source by placing camera at light source position and rendering scene

Scene points are lit if visible from light source

Determine visibility from light source by placing camera at light source position

Two pass algorithm

First pass

- Render scene by placing camera at light source position
- Store depth image (shadow map)

Depth image seen from light source
### Two pass algorithm

**Second pass**
- Render scene from camera position
- At each pixel, compare distance to light source with value in shadow map
  - If distance is larger, we are in shadow
  - If distance is smaller or equal, pixel is lit

### Issues
- Limited field of view of shadow map
- Z-fighting
- Sampling problems

### Limited field of view

- What if a scene point is outside the field of view of the shadow map?

### Limited field of view

- What if a scene point is outside the field of view of the shadow map?
- Use six shadow maps, arranged in a cube
- Requires rendering pass for each shadow map!

### z-fighting

- Depth values for points visible from light source are equal in both rendering passes
- Because of limited resolution, depth of pixel visible from light could be larger than shadow map value
- Need to add bias in first pass to make sure pixels are lit

### Solution

- Add bias when rendering shadow map
  - Move geometry away from light by small amount
- Finding correct amount of bias is tricky
Bias

- **Not enough**
- **Too much**

Correct

Sampling problems

- Shadow map pixel may project to many image pixels
- Ugly stair-stepping artifacts

Solutions

- Increase resolution of shadow map
  - Not always sufficient
- Split shadow map into several slices
- Tweak projection for shadow map rendering
  - Light space perspective shadow maps (LiSPSM)
    - [Link](http://www.cg.tuwien.ac.at/research/vr/lispsm)
    - With GLSL source code!
- Combination of splitting and LiSPSM
  - Basis for most serious implementations

LiSPSM

Basic shadow map

Light space perspective shadow map

Percentage closer filtering

- Instead of looking up one shadow map pixel, look up several pixels
- Perform depth test for each shadow map pixel
- Compute percentage of pixels that are lit

Percentage closer filtering

- Supported in hardware for small filters (2x2 shadow map pixels)
- Can use larger filters with additional rendering passes
- Fake soft shadows
Shadow mapping with GLSL

First pass
- Render scene by placing camera at light source position
- Compute light view (look at) matrix
  - Similar to computing camera matrix from look-at, up vector
  - Compute its inverse to get world-to-light transform
- Determine view frustum such that scene is completely enclosed
  - Use several view frusta/shadow maps if necessary

First pass
- Each vertex point is transformed by
  \[ P_{light} V_{light} M \]
  - Object-to-world (modeling) matrix \( M \)
  - World-to-light space matrix \( V_{light} \)
  - Light frustum (projection) matrix \( P_{light} \)
- Remember: points within frustum are transformed to unit cube \([-1,1]^3\)

Looking up shadow map
- Need to transform each point from object space to shadow map
- Shadow map texture coordinates are in \([0,1]^2\)
- Transformation from object to shadow map coordinates
  \[ T' = \begin{bmatrix} f \end{bmatrix} V_{shadow} \]
  - Store \( T \) as texture matrix
- After perspective projection we have shadow map coordinates

Looking up shadow map
- Transform each vertex to normalized frustum of light
  \[ \begin{bmatrix} r \end{bmatrix} \begin{bmatrix} s \end{bmatrix} \]
  - Pass \( s,t,r,q \) as texture coordinates to rasterizer
  - Rasterizer interpolates \( s,t,r,q \) to each pixel
  - Use projective texturing to look up shadow map
    - This means, the texturing unit automatically computes \( s/q,t/q,r/q,1 \)
    - \( s/q,t/q \) are shadow map coordinates in \([0,1]^2\)
    - \( r/q \) is depth in light space
  - Shadow depth test: compare shadow map at \((s/q,t/q)\) to \( r/q \)

First pass
- Use \texttt{glPolygonOffset} to apply depth bias
- Store depth image in a texture
  - Use \texttt{glCopyTexImage} with internal format \texttt{GL_DEPTH_COMPONENT}

Second pass
- Render scene from camera
- At each pixel, look up corresponding location in shadow map
- Compare depths with respect to light source
**GLSL specifics**

**In application**
- Store matrix $T$ in OpenGL texture matrix
- Set using `glMatrixMode(GL_TEXTURE)`

**In vertex shader**
- Access texture matrix through predefined uniform `gl_TextureMatrix`

**In fragment shader**
- Declare shadow map as `sampler2DShadow`
- Look up shadow map using projective texturing with `vec4 texture2DProj(sampler2D, vec4)`

**Implementation specifics**

- When you do a projective texture look up on a `sampler2DShadow`, the depth test is performed automatically
  - Return value is $(1,1,1,1)$ if lit
  - Return value is $(0,0,1)$ if shadowed
- Simply multiply result of shading with current light source with this value

**Demo**

- Cg tutorial examples shadowMapping

**Resources**

- Overview, lots of links
- Basic shadow maps
- Avoiding sampling problems in shadow maps
  - [http://www.cg.tuwien.ac.at/research/vr/lipsm/](http://www.cg.tuwien.ac.at/research/vr/lipsm/)
- Faking soft shadows with shadow maps
- Alternative: shadow volumes
  - [http://www.gamedev.net/reference/articles/article1873.asp](http://www.gamedev.net/reference/articles/article1873.asp)

**Today**

- Shadows
  - **Procedural modeling**
    - Recursion, randomness
    - Noise textures

**Modeling**

- Creating 3D objects/scenes and defining their appearance (texture, etc.)
- So far we saw
  - Triangle meshes
  - NURBS surfaces
- Interactive modeling
  - Place vertices, control points manually
- For realistic scenes, need extremely complex models containing millions or billions of primitives
- Modeling everything manually extremely tedious
Alternatives

- **Data-driven modeling**
  - Scan model geometry from real world examples
  - Use laser scanners or similar devices
  - Photographs as textures

- **Procedural modeling**
  - Construct 3D models and textures using algorithms

Examples

- [http://www-graphics.stanford.edu/data/3Dscanrep/](http://www-graphics.stanford.edu/data/3Dscanrep/)
- [http://www.tsi.enst.fr/3dmodels/](http://www.tsi.enst.fr/3dmodels/)
- .ply file format reader [http://www.tecgraf.puc-rio.br/~diego/professional/rply/](http://www.tecgraf.puc-rio.br/~diego/professional/rply/)

Randomness

- Use some sort of randomness to make models more interesting, natural, less uniform, clean
- **Pseudorandom** number generation algorithms
  - Produce a sequence of (apparently) random numbers based on some initial seed value
  - Pseudorandom sequences are repeatable, as one can always reset the sequence
    - E.g., if tree is built using several random numbers, then the entire tree can be rebuilt by just resetting the seed to its initial value
    - If the seed is set to a different value, a different sequence of numbers will be generated, resulting in a slightly different tree

Recursion

- Repeatedly apply the same operation (set of operations) to an object
- Generate objects that are self-similar, **fractals**
  - Object that look the same when viewed at different scales
  - For example, the shape of a coastline may appear as a jagged line when we view a map of California
    - As we zoom in closer and closer, we see that there is more and more detail at finer scales
    - We always see a jagged line no matter how close we look at the coastline

Height fields

- Landscapes are often constructed as **height fields**
- Regular grid in the ground plane (assume xz plane here)
- Store a height (y) value at each point
- Can store large terrain in memory
  - No need to store xz coordinates, connectivity
  - Shaped terrain by operations that modify the y coordinates
  - Can interpret height values as gray scale values
    - Apply image processing tools

Fractal landscapes

- Random midpoint displacement algorithm
  - Recursively subdivide triangles
  - Randomly displace edge midpoints
  - Reduce size of displacement as triangles get smaller
  - Similar for quadrilaterals

[Source: Wikipedia]
Fractal landscapes

- Add textures, nice rendering...
- Terragen, free software
  
  [http://www.planetside.co.uk/terragen/]

L-systems

- Developed by a biologist (Lindenmayer) in 1968 to study growth patterns of algae
- Defined by a grammar
  
  \[ G = \{ V, S, \omega, P \} \]

  - \( V \) alphabet, set of symbols that can be replaced (variables)
  - \( S \) set of symbols that remain fixed (constants)
  - \( \omega \) string of symbols defining initial state
  - \( P \) production rules

- Difference between L-systems and formal languages
  - In each step, apply all possible production rules
  - L-systems are subsets of formal languages

Sierpinski triangle

- Variables: \( A, B \)
  - Draw forward
- Constants: +, −
  - Turn left, right by 60 degrees
- Start: \( A \)
- Rules: \((A\rightarrow B-A-B), (B\rightarrow A+B+A)\)

  2 iterations
  
  ![2 iterations triangle](image)

  4 iterations
  
  ![4 iterations triangle](image)

  6 iterations
  
  ![6 iterations triangle](image)

  9 iterations
  
  ![9 iterations triangle](image)

Fractal fern

- Variables: \( X, F \)
  - \( X \): no drawing operation
  - \( F \): move forward
- Constants: +, −
  - Turn left, right
- Start: \( X \)
- Rules:
  
  \( (X \rightarrow F-[X]+X]+F[+FX]-X), (F \rightarrow FF) \)

- Stochastic L-system
  - If there is more than one production rule for a symbol, randomly choose one

Fractal trees

- Recursive generation of trees in 3D
  
  [http://web.comhem.se/solgrop/3dtree.htm]

- Model trunk, branches as cylinders
- Change color from brown to green at certain level of recursion

  Fractal tree

  Sierpinski tree

Algorithmic beauty of plants

- Online book on algorithmic beauty of plants by Prusinkiewicz
  
  [http://algorithmicbotany.org/papers/#abop]

Buildings, cities

Pascal Mueller
[http://www.vision.ee.ethz.ch/~pmueller/publications.html]

Noise textures

- Form of randomness to model natural, organic looking 2D and 3D structures
- Noise represents a distribution of randomness over some space (2D or 3D)
- Not entirely random, as two points nearby will have a similar value
- Noise has a frequency associated with it
- By combining noise patterns of different frequencies, one can make more complex turbulence patterns

2D noise
Turbulence

Noise

- Often used as a 3D solid texture
  - Texture is defined in 3D
  - Surface visualizes a 2D slice through 3D volume
- Don’t need uv texture coordinates!

Solid noise on a sphere [Ken Perlin]

Perlin noise

- Efficient way to generate noise textures procedurally
- Band limited
  - Features have a certain uniform size
- Translation invariant
  - “Looks the same everywhere”
- Rotation invariant
  - No preferred orientation

2D Perlin noise

Perlin noise

- GLSL provides built in noise functions
  - Unfortunately, hardware may not support it
- Alternatively, generate and store 3D noise texture in pre-process
  - Details see, e.g., GLSL Shading Language book

3D perlin noise

Noise [Ken Perlin] Noise bump mapped [Ken Perlin]
**Fractal noise and turbulence**

- Fractal noise
  \[ F(x) = \sum \alpha_i \cdot N(x \cdot f_i) \]
- Turbulence
  \[ T(x) = \sum \| \alpha_i \cdot N(x \cdot f_i) \| \]
- Frequencies usually in octaves \[ f_i = 2f_{i-1} \]
- Amplitude \[ \alpha_i = p \cdot \alpha_{i-1} \]
- Persistency \[ p < 1, \text{often } p = 0.5 \]

**Variations on noise**

- Noise
- \[ \sin(x + \text{sum } 1/f(\text{noise})) \]
- \[ \text{sum } 1/f(\text{noise}) \]
- \[ \text{sum } 1/f(\text{noise}) \]

**3D Perlin noise**

- Noise
- Turbulence

**Other effects**

- Use colormap and look-up with noise offset
- Wood
  - color = colormap(radius+noise(x,y,z))
- Marble
  - Parallel plane + noise
  - color = colormap( x+turbulence(x,y,z) )

**More on Perlin noise**

- Easy to read intros
  - [http://www.noisemachine.com/talk1/index.html](http://www.noisemachine.com/talk1/index.html)
  - [http://freespace.virgin.net/hugo.elias/models/m_perlin.htm](http://freespace.virgin.net/hugo.elias/models/m_perlin.htm)
- Technical paper on 3D implementation
  - [http://mrl.nyu.edu/~perlin/noise/](http://mrl.nyu.edu/~perlin/noise/)