Chapter 8 - Shading and Rendering

- Local Illumination Models: Shading
- Global Illumination: Ray Tracing
- Global Illumination: Radiosity
- Non-Photorealistic Rendering

The 3D rendering pipeline (our version for this class)

3D models in model coordinates → 3D models in world coordinates → 2D Polygons in camera coordinates → Pixels in image coordinates

Scene graph → Camera

Animation, Interaction

Rasterization

Lights
Local Illumination: Shading

- Local illumination:
  - Light calculations are done locally without the global scene
  - No cast shadows
    (since those would be from other objects, hence global)
  - Object shadows are OK, only depend on the surface normal

- Simple idea: Loop over all polygons
- For each polygon:
  - Determine the pixels it occupies on the screen and their color
  - Draw using e.g., Z-buffer algorithm to get occlusion right

- Each polygon only considered once
- Some pixels considered multiple times
- More efficient: Scan-line algorithms
Scan-Line Algorithms in More Detail

• Polygon Table (PT):
  – List of all polygons with plane equation parameters, color information and inside/outside flag (see rasterization)

• Edge Table (ET):
  – List of all non-horizontal edges, sorted by \( y \) value of top end point
  – including a reference back to polygons to which the edge belongs

• Active Edge Table (AET):
  – List of all edges crossing the current scan line, sorted by \( x \) value

\[
\text{for } v = 0..V \text{ (all scan lines):}
\]
\[
\text{Compute AET, reset flags in PT;}
\]
\[
\text{for all crossings in AET:}
\]
\[
\text{update flags;}
\]
\[
\text{determine currently visible polygon } P \text{ (Z-buffer);}
\]
\[
\text{set pixel color according to info for } P \text{ in PT;}
\]
\[
\text{end}
\]
\[
\text{end}
\]

• Each polygon considered only once
• Each pixel considered only once
Reminder: Phong’s Illumination Model

\[ I_o = I_{amb} + I_{diff} + I_{spec} = I_a k_a + I_i k_d (\vec{l} \cdot \vec{n}) + I_i k_s (\vec{r} \cdot \vec{v})^n \]

- **Prerequisites for using the model:**
  - Exact location on surface known
  - Light source(s) known
- **Generalization to many light sources:**
  - Summation of all diffuse and specular components created by all light sources
- **Light colors easily covered by the model**
- **Do we really have to compute the formula for each pixel?**
Flat Shading

• Determine one surface normal for each triangle
• Compute the color for this triangle
  – using e.g., the Phong illumination model
  – usually for the center point of the triangle
  – using the normal, camera and light positions
• Draw the entire triangle in this color

• Neighboring triangles will have different shades
• Visible „crease“ between triangles

• Cheapest and fastest form of shading
• Can be a wanted effect, e.g., with primitives
Mach Band Effect

• Flat Shading suffers from an optical illusion
  – Human visual system accentuates discontinuity at brightness boundary
  – Darker stripes appear to exist at dark side, and vice versa

Source: keithwiley.com

Gouraud Shading

• Determine normals for all mesh vertices
  – i.e., triangle now has 3 normals
• Compute colors at all vertices
  – using e.g., the Phong illumination model
  – using the 3 normals, camera and light positions
• Interpolate between these colors along the edges
• Interpolate also for the inner pixels of the triangle

• Neighboring triangles will have smooth transitions
  – If normals at a vertex are the same for all triangles using it

• Simplest form of smooth shading
  – Specular highlights only if they fall on a vertex by chance
Phong Shading

- Determine normals for all mesh vertices
- Interpolate between these normals along the edges
- Compute colors at all vertices
  - using e.g., the Phong illumination model
  - using the interpolated normal, camera and light positions

- Neighboring triangles will have smooth transitions
  - If normals at a vertex are the same for all triangles using it

- Has widely substituted Gouraud shading
  - Specular highlights in arbitrary positions
  - Have to compute Phong illumination model for every pixel
Chapter 8 - Shading and Rendering

• Local Illumination Models: Shading
• Global Illumination: Ray Tracing
• Global Illumination: Radiosity
• Non-Photorealistic Rendering
Global illumination: Ray Tracing

• Global illumination:
  – Light calculations are done **globally** considering the entire scene
  – i.e. cast shadows are OK if properly calculated
  – Object shadows are OK anyway

• Ray *casting*:
  – From the eye, cast a ray through every screen pixel
  – Find the first polygon it intersects with
  – Determine its color at intersection and use for the pixel
  – Also solves occlusion (makes Z-Buffer unnecessary)

• Ray *tracing*: recursive ray casting
  – From intersection, follow reflected and refracted beams
  – up to a maximum recursion depth
  – Works with arbitrary geometric primitives

http://pclab.arch.ntua.gr/03postgra/mladenstamenico/ (probably not original)
Brainstorming: What Makes Ray Tracing Hard?
Optimizations for Ray Tracing

• Bounding volumes:
  – Instead of calculating intersection with individual objects, first calculate intersection with a volume containing several objects.
  – Can decrease computation time to less than linear complexity (in number of existing objects).

• Adaptive recursion depth control
  – Maximum recursion limit is always necessary.
  – Recursion should be stopped as soon as possible.
  – E.g., stop if intensity change goes below a threshold value.

• Monte Carlo Methods
  – Improve complexity (cascading recursion = exponential).
  – Use one random ray for recursive tracing (instead of refracted/reflected rays).
  – Carry out multiple experiments (e.g. 100) and compute average values.
Recent (2007) development: Real Time Ray Tracing

- Various optimizations presented over the last few years
- Real time ray tracing has become feasible
- Used to be http://openrt.de/ (images from there, now dead)
Chapter 8 - Shading and Rendering

• Local Illumination Models: Shading
• Global Illumination: Ray Tracing
• Global Illumination: Radiosity
• Non-Photorealistic Rendering
Reminder: The rendering equation [Kajiya ‘86]

\[ I_o(x, \vec{\omega}) = I_e(x, \vec{\omega}) + \int_{\Omega} f_r(x, \vec{\omega}', \vec{\omega}) I_i(x, \vec{\omega}') (\vec{\omega}' \cdot \vec{n}) d\vec{\omega}' \]

- \( I_o \) = outgoing light
- \( I_e \) = emitted light
- Reflectance Function
- \( I_i \) = incoming light
- angle of incoming light

- Describes all flow of light in a scene in an abstract way
- doesn’t describe some effects of light:
  - _
  - _
Global Illumination: Radiosity

- Simulation of energy flow in scene
- Can show „color bleeding“
  - blueish and reddish sides of boxes
- Naturally deals with area light sources
- Creates soft shadows
- Only uses diffuse reflection
  - does not produce specular highlights

http://www.webreference.com/3d/lesson46/
Radiosity Algorithm

• Divide all surfaces into small patches
• For each patch determine its initial energy
• Loop until close to energy equilibrium
  – Loop over all patches
    • determine energy exchange with every other patch
• „Radiosity solution“: energy for all patches
• Recompute if __________________________ changes
Combinations

• Ray Tracing is adequate for reflecting and transparent surfaces
• Radiosity is adequate for the interaction between diffuse light sources
• What we want is a combination of the two!
  – This is non-trivial, a simple sequence of algorithms is not sufficient

• Example for a state-of-the-art “combination” (more like another innovative approach): *Photon Maps* (Jensen 96)
  – First step:
    Inverse ray tracing with accumulation of light energy
    Photons are sent from light sources into scene, using Monte Carlo approach
    Surfaces accumulate energy from various sources
  – Second step:
    “Path tracing” (i.e. Monte Carlo based ray tracing) in optimized version
    (e.g. only small recursion depth)
Chapter 8 - Shading and Rendering

- Local Illumination Models: Shading
- Global Illumination: Ray Tracing
- Global Illumination: Radiosity
- Non-Photorealistic Rendering
Non-Photorealistic Rendering (NPR)

• Create graphics that look like drawings or paintings

• One method: stroke-based NPR
  – instead of grey shades, determine a stroke density and pattern
  – imitates pencil drawings or etchings (Kupferstich)

• Other methods: using image manipulation on rendered images
  – can in principle often be done in Photoshop

• Active field of research
  – http://graphics.uni-konstanz.de/forschung/npr/watercolor/
  – many others