Alternative Rendering Approaches

• This semester’s material has been influenced by a key constraint: “real-time” or “interactive” computer graphics — this implies that a user can actively influence a displayed scene at a sufficiently rapid rate.

• However, we have seen that this constraint results in certain tradeoffs — in particular, lighting and shading is necessarily local, to avoid the computational complexity of multiple objects interacting with each other.

• So…what if it doesn’t have to be real-time?

A Parting Shot

• Before we completely chuck “real time” out the window, let’s address a frequent target of alternative rendering techniques: shadows.

• We know that shadows can’t be done “for free” by OpenGL; we also know that the computational difficulty lies in having to consider interactions across multiple objects — something that the alternative rendering approaches address.

• But, there is a middle ground: what if we only worry about shadows on a specific surface?
Shadows on the Ground

• Despite the difficulty of general purpose shadowing, casting shadows to a specific surface — in this example, a simulated “ground” — remains doable, and doable in real time.

• The key observation is this: a shadow is actually a perspective projection, with the light source serving as the camera or eye point, and the “ground” serving as the near plane.

• Suppose that the ground is at some yg value of a light source is at some location \((xl, yl, zl)\).

• If we translate the light source to the origin, the projection matrix that projects onto \(y = 0.0\) is actually fairly simple — see Angel Section 5.10.

• The algorithm summarizes to:
  1. Translate the origin to the “ground” (yg)
  2. Translate the light source to the origin
  3. Multiply by the shadow projection matrix
  4. Translate the light source back to its location
  5. Draw the object

• This produces an image of the object as if it were projected onto a flat surface (which is precisely what the projection matrix does) — thus, a simple shadow.
Ray Tracing

• Instead of rendering from vertex to pixel, render from pixel to vertex
• For every pixel in the screen display, cast a ray from the eye through the pixel into the scene
• As the ray travels through the scene, modify the light that affects that ray as it hits objects and light source
• Naturally recursive: cast a ray, and when the ray hits “something,” cast one or more rays or terminate — the color upon termination is the color of the pixel

Where the Ray Can Go

• Ray intersects nothing — return background color
• Ray intersects a surface — recursively cast one or more rays from that surface
  • Reflected ray: light bouncing off that surface
  • Transmitted ray: lighting emitted by or passing through that surface
• Ray intersects a light source — return the color of that light source
• Arbitrary shadows, reflections, and translucency “for free” — natural consequence of following rays through each pixel

• Computational complexity: numbers of surfaces and rays cast are not easily bounded, and thus this is not a “real time” approach

• Key function is intersection — which objects does a ray hit? Function varies according to type of object and how they are represented, thus limiting modeling options

• Ray tracing without recursion (e.g. terminate at background or a single surface) == local lighting

### The Rendering Equation

• Theoretical foundation for a number of specific implementations

• Based on physics conversation laws: the amount of energy emitted (light sources) is the same as the amount of energy absorbed and reflected (material properties)

\[
i(p, p') = v(p, p') \left[ \epsilon(p, p') + \int \rho(p, p', p'') i(p', p'') \, dp'' \right]
\]
• $i$ is the light at point $p$ coming from point $p'$

• $v$ is either:
  
  • zero if an opaque surface lies in between $p$ and $p'$
  
  • $1 / \text{distance}^2$ otherwise

• $\epsilon$ is the light, if any, that is emitted at $p'$ (in other words, $p'$ is a light source)

• $p''$ represents the set of points whose light is reflected by $p'$ toward $p$; $\rho$ represents how the material properties of $p'$ affect that light, and $i$ represents this same function for $p'$ and $p''$

Radiosity

• Simplification of the rendering equation through a key assumption: what if all surfaces were perfectly diffuse — e.g. they reflect light equally in all directions

• Then, we can capture diffuse–diffuse interactions — in other words, how does light reflected by a perfectly diffuse surface affect the other perfectly diffuse surfaces around it?

• With radiosity rendering, each surface is called a patch, and each patch has a single color, derived through conventional lighting models
• Given $n$ patches from 1 to $n$, for every patch $i$, let $b_i$ be the light reflected by that patch per unit area. If $a_i$ is the area of patch $i$, then the total light reflected is $b_i a_i$. Given a possible component $e_i$, representing light emitted by patch $i$, a reflective component $\rho_i$ that represents the light from other patches that strike patch $i$, and a form factor $f_{ij}$ that represents how the light from some patch $j$ affects patch $i$, one can model the total light from patch $i$ as:

$$b_i a_i = e_i a_i + \rho_i \sum_{j=0}^{n} f_{ij} b_j a_j$$

• The reciprocity equation expresses that the relationship between two patches $i$ and $j$ has a degree of symmetry:

$$f_{ij} a_i = f_{ji} a_j$$

• Thus, we can swap $a_i$ for $a_j$ (since we are looping through all $i$ then summing through all $j$):

$$b_i a_i = e_i a_i + \rho_i \sum_{j=0}^{n} f_{ij} b_j a_i$$

• Then divide through by $a_i$ for the final radiosity equation:

$$b_i = e_i + \rho_i \sum_{j=0}^{n} f_{ij} b_j$$
• Note how, as the area gets infinitely smaller, the summation becomes integration — and the radiosity equation becomes the rendering equation

• Solving the radiosity equation serves as the basis for radiosity rendering

• Key trick — calculating the form factor — essentially, how the relative distances and angles from one patch to another modify the energy sent by one patch to the other patch