ILLUMINATION MODELS/ALGORITHMS

Local illumination – Fast
Ignore real physics, approximate the look
Interaction of each object with light
  • Compute on surface (light to viewer)

Global illumination – Slow
Physically based
Interactions between objects
BASIC RAY-TRACING ALGORITHM

```python
RayTrace(r, scene)
obj = FirstIntersection(r, scene)

if (no obj) return BackgroundColor;
else {
    if (Reflect(obj))
        reflect_color = RayTrace(ReflectRay(r, obj));
    else
        reflect_color = Black;
    
    if (Transparent(obj))
        refract_color = RayTrace(RefractRay(r, obj));
    else
        refract_color = Black;

    return Shade(reflect_color, refract_color, obj);
}
```
WHEN TO STOP?

• Algorithm above does not terminate

• Termination Criteria
  • No intersection
  • Contribution of secondary ray attenuated below threshold – each reflection/refraction attenuates ray
  • Maximal depth is reached
SIMULATING SHADOWS

• Trace ray from each ray-object intersection point to light sources
  • If the ray intersects an object in between ⇒ point is shadowed from the light source

```plaintext
shadow = RayTrace(LightRay(obj,r,light));

return Shade(shadow,reflect_color,refract_color,obj);
```
RAY TRACING: IDEA

Image Plane

Eye

Light Source

Reflected Ray

Shadow Rays

Refracted Ray

Reflected Ray
RAY-OBJECT INTERSECTIONS

• Core of ray-tracing ⇒ must be extremely efficient
• Usually involves solving a set of equations
  • Using implicit formulas for primitives

Example: Ray-Sphere intersection

**ray:** \( x(t) = p_x + v_x t, \ y(t) = p_y + v_y t, \ z(t) = p_z + v_z t \)

(unit) sphere: \( x^2 + y^2 + z^2 = 1 \)

quadratic equation in \( t \):

\[
0 = (p_x + v_x t)^2 + (p_y + v_y t)^2 + (p_z + v_z t)^2 - 1
= t^2(v_x^2 + v_y^2 + v_z^2) + 2t(p_x v_x + p_y v_y + p_z v_z) + (p_x^2 + p_y^2 + p_z^2) - 1
\]
RAY-TRACING: DIRECT ILLUMINATION

• Local surface information (normal...)
  • For implicit surfaces $F(x,y,z)=0$:
    normal $n(x,y,z)$ is gradient of $F$:

    $$n(x, y, z) = \nabla F(x, y, z) = \left( \frac{\partial F(x, y, z)}{\partial x}, \frac{\partial F(x, y, z)}{\partial y}, \frac{\partial F(x, y, z)}{\partial z} \right)$$

• Example:

    $$F(x, y, z) = x^2 + y^2 + z^2 - r^2$$

    $$n(x, y, z) = \left( \begin{array}{c} 2x \\ 2y \\ 2z \end{array} \right)$$

    Needs to be normalized!
OPTIMIZED RAY-TRACING

• Basic algorithm is simple but VERY expensive
• Optimize...
  • Reduce number of rays traced
  • Reduce number of ray-object intersection calculations
• Parallelize
  • Cluster
  • GPU
• Methods
  • Bounding Boxes
  • Spatial Subdivision
    • Visibility, Intersection/Collision
  • Tree Pruning
HIERARCHICAL BOUNDING VOLUMES

- Save time by testing intersection with a hierarchy of boxes first
BSP TREES: IDEA

• For a plane, objects on the same side of plane as viewer CANNOT be occluded by objects on other side
• Intersect closer side first
• if ray doesn’t intersect plane?
  • can’t intersect other side!
• Idea:
  • Recursively split space by planes
  • Traverse resulting tree to establish rendering/intersection order
    • Test eye location w.r.t. each plane
BSP TREES: CONSTRUCTION
TRAVERSING BSP TREES

• Each plane divides world into near and far
  • For given ray, decide which side is near and which is far
    • Check which side of plane viewpoint is on independently for each tree vertex
    • Tree traversal differs depending on viewpoint!
  • Recursive algorithm
    • Intersect with near side
    • If no intersection, and ray intersects the plane,
      • Intersect with far side
SUMMARY: BSP TREES

- **Pros:**
  - Simple, elegant scheme
  - Faster intersections
  - Correct version of painter’s algorithm back-to-front rendering approach
  - Still very popular for video games

- **Cons:**
  - Slow(ish) to construct tree: $O(n \log n)$ to split, sort
  - Splitting increases polygon count: $O(n^2)$ worst-case
  - => Algorithm restricted to static scenes
SOFT SHADOWS: AREA LIGHT SOURCES

- **So far:**
  - All lights were either point-shaped or directional
    - Both for ray-tracing and the rendering pipeline
  - Thus, at every point, we only need to compute lighting formula and shadowing for **ONE** direction per light

- **In reality:**
  - All lights have a finite area
  - Instead of just dealing with one direction, we now have to **integrate** over all directions that go to the light source
AREA LIGHT SOURCES

• Area Lights:
  • Infinitely many light rays
  • Need to integrate over all of them:

\[
I_{\text{reflected}} = \int \rho(\omega) \cdot V(\omega) \cdot I_{\text{light}}(\omega) \cdot d\omega
\]

• Lighting model visibility and light intensity can now be different for every ray!
NUMERICAL INTEGRATION

• Regular grid of point lights
  • Problem: Too regular
    see 4 hard shadows

• Need LOTS of points
  to avoid this problem
1. Shoot a ray through pixel (i,j). Set attenuation to 1.0.
2. Find the closest intersection of the ray with an object
3. Randomly choose between “emission” and “reflection”
   a. If “emission”, return emissionColor;
   b. If “reflection”,
      Reflect a ray in a random direction
      rayWeight *= reflectance;
      Go to 2.
SIMPLEST PATH TRACER

PathTrace(Ray r) {
    P = closestIntersection(r);
    if (random(emit, reflect) == emit)
        return EmissionColor;
    else {
        Ray v = {intersectionPt, randomDirectionInHemisphere(r.normalWhereObjWasHit)};
        double cos_theta = dot(v.direction, r.normalWhereObjWasHit);
        return PathTrace(v)*cos_theta*reflectance;
    }
}
WHAT YOU GET

- Area light
- Soft shadows
- Reflections from diffuse objects ‘color bleeding’
- Noise
- Soft shadows
GENERATING RANDOM POINTS

• How to generate random points in a unit square?

```plaintext
for (i=0; i< N) {
    x = rand();
    y = rand();
}
```
GENERATING RANDOM POINTS

• How to generate random points in a rectangle?

```
for (i=0...N)
{
    x = w*rand();
    y = h*rand();
}
```
GENERATING RANDOM POINTS

• In a right triangle?

\[
\begin{align*}
    x &= a_1 \cdot w \\
    y &= a_2 \cdot h \\
\end{align*}
\]

for \( i = 0 \ldots N \)

\{
    a_1 = \text{rand}() \\
    a_2 = \text{rand}() \\
    \text{if} \ (a_1 + a_2 < 1) \\
    \{ \\
        x = a_1 \cdot w; \\
        y = a_2 \cdot h; \\
    \} \\
\}
REJECTION SAMPLING

• Say you have a complicated shape, and you want to generate points uniformly in it.

• One elegant way is rejection sampling:
  • Generate a point evenly in the bounding box
  • If it’s not inside our shape, discard

• How many points will be rejected (ratio)?
MONTE CARLO METHODS

• Example: approximating $\pi$:

"Pi 30K" by CaitlinJo - Own work. This mathematical image was created with Mathematica. Licensed under CC BY 3.0 via Commons - https://commons.wikimedia.org/wiki/File:Pi_30K.gif#/media/File:Pi_30K.gif
PathTrace(Ray r) {
    P = closestIntersection(r);
    if (random(emit, reflect) == emit)
        return EmissionColor;
    else {
        Ray v = {intersectionPt,
            randomDirectionInHemisphere(r.normalWhereObjWasHit)};
        double cos_theta = dot(v.direction, r.normalWhereObjWasHit);
        return PathTrace(v)*cos_theta*reflectance;
    }
}
HOW TO CHOOSE BETWEEN 3 ACTIONS?

1. “Go to 314 lecture”, $P = 0.4$
2. “Go have lunch instead”, $P = 0.5$
3. “Sleep in”, $P = 0.1$

```plaintext
x = random();
if (x < 0.4)
    return 1;
else if (x < 0.9)
    return 2;
else
    return 3;
```
BACK TO PATH TRACING

• Now we know how to reflect a ray randomly when surface is diffuse
And we know how to choose between “emit” and “scatter” events.
We can similarly add “absorb”, “reflect”, “refract”.
RAY TRACING VS PATH TRACING

• Global illumination algorithms
• Rays emitted FROM camera

• Ray Tracing
  • Single ray per pixel
  • Supports indirect lighting only from specular surfaces
    • No color bleeding
  • Shoots shadow rays to compute direct illumination
    • Soft shadows are harder to get

• Path Tracing (*may produce renders indistinguishable from photos*)
  • Many rays per pixel, their color averaged
  • At each interaction, ray direction changes randomly with some distribution
  • No difference between light sources and objects
    • Soft shadows, complex materials, etc.
    • Supports all sorts of indirect lighting
THE RENDERING PIPELINE

Vertices and attributes

† vertices and attributes

† model view transform

† per-vertex attributes

† clipping

† viewport transform

† scan conversion

† interpolation

† texturing/...

† lighting/shading

† depth test

† blending

† framebuffer

† fragment shader
OPAQUE VS. TRANSPARENT

• For transparent objects, every time we’re writing into a fragment buffer, we need to consider what there is already

• Per fragment:
  • Fragment’s color: source color
  • What’s in framebuffer: destination color

• Same idea as layers in Photoshop

How to combine those 2 colors into some new color?
BLENDING EQUATIONS

- \( D = (r, g, b, \alpha)_D \) - destination color (what’s already in framebuffer)
- \( S = (r, g, b, \alpha)_S \) - source color (current fragment)
- \( Out = (r, g, b, \alpha)_{out} \) - output color (result of blending)

Blending equations:
\[
\begin{align*}
Out.rgb &= f_1(D.rgb, S.rgb) \\
Out.\alpha &= f_2(D.\alpha, S.\alpha)
\end{align*}
\]

\( \alpha = 1.0 \) (opaque) \hspace{5cm} \alpha = 0.7 \) (semi-transparent)
WHAT CAN WE DO WITH THOSE?

• Simple transparency (“over operator”):
  • $f_1 = ADD, f_2 = ADD$
  • $d_1 = 1 - S.\alpha$
  • $s_1 = S.\alpha$
  • $d_2 = 0$
  • $s_2 = 1$

\[
\text{rgb: } (1 - 0.7) \cdot (0, 0, 1) + 0.7 \cdot (1, 1, 0) \\
= (0, 0, 0.3) + (0.7, 0.7, 0) = (0.7, 0.7, 0.3)
\]

\[
\text{Out.\,rgb} = (1 - S.\alpha) \cdot D.\,rgb + S.\alpha \cdot S.\,rgb \\
\text{Out.\,\alpha} = 0 \cdot D.\alpha + 1 \cdot S.\alpha
\]

$\alpha = 1.0$ (opaque)  
$\alpha = 0.7$ (semi-transparent)
**OVER OPERATOR**

\[ \text{Out.rgb} = (1 - S.\alpha) \cdot D.rgb + S.\alpha \cdot S.rgb \]

• Examples: \( A.\alpha = 1, B.\alpha = 0.4 \)

A over B:
\[ \text{Out.rgb} = (1) \cdot A.rgb + (1 - 1) \cdot B.rgb \]

B over A:
\[ \text{Out.rgb} = (0.4) \cdot A.rgb + (1 - 0.4) \cdot B.rgb \]
WHAT CAN WE DO WITH THOSE?

• “Darken”
  • $f_1 = MIN$, $f_2 = ADD$
  • $d_1 = 1$
  • $s_1 = 1$
  • $d_2 = 0$
  • $s_2 = 1$

\[
\text{Out. rgb} = \min(S. rgb, D. rgb)
\]
\[
\text{Out. } \alpha = S. \alpha
\]

$\alpha = 1.0$ (opaque)
ALIASING & ANTI-ALIASING
CONTINUOUS VS. DISCRETE

- Continuous -> Discrete: **Sampling**
- Discrete -> Continuous: **Reconstruction/Interpolation**
TWO VIEWS OF IMAGES

• A continuous image, $I(x_w,y_w)$, is a bivariate function.
  • range is a linear color space.

• A discrete image $I[i][j]$ is a two dimensional array of color values.

• We associate each pair of integers $i, j$, with the continuous image coordinates $x_w = i$ and $y_w = j$
ALIASING

• Aliasing happens when hi-res image is drawn on low-res media

• The heart of the problem: too much information in one pixel
ANTIALIASING

• We can also model this as an optimization problem.
• These approaches lead to:

\[
I[i][j] \leftarrow \iint_{\Omega} I(x,y)F_{i,j}(x,y)dx\,dy
\]

where \( F_{i,j}(x,y) \) is some function measuring how strongly the continuous image value at \([x,y]_i\) should influence the pixel value \( i, j \)
• We often choose the filters $F_{i,j}(x,y)$ to be something non-optimal, but that can more easily computed with.

• The simplest such choice is a box filter, where $F_{i,j}(x,y)$ is zero everywhere except over the 1-by-1 square center at $x = i, y = j$.

• Calling this square $\Omega_{i,j}$, we arrive at

$$ I[i][j] \leftarrow \iint_{\Omega_{i,j}} I(x,y)dx dy $$
OVER-SAMPLING

• Even that integral is not really easy to compute
• Instead, it is approximated by some sum of the form:

\[ I[i][j] \leftarrow \frac{1}{n} \sum_{k=1}^{n} I(x_k, y_k) \]

• where \( k \) indexes some set of locations \((x_k, y_k)\) called the sample locations.
• The renderer first produces a “high resolution” color and z-buffer “image”,
  • where we will use the term *sample* to refer to each of these high resolution pixels.
SUPER-SAMPLING

• If the sample locations for the high resolution image form a regular, high resolution grid, then this is called super sampling.

• We can also choose other sampling patterns for the high resolution “image”,
  • Such less regular patterns can help us avoid systematic errors that can arise when using the sum to replace the integral.
MULTI-SAMPLING

• Render to a “high resolution” color and z-buffer
• During the rasterization of each triangle, “coverage” and z-values are computed at this sample level.
• But for efficiency, the fragment shader is only called only once per final resolution pixel.
  • This color data is shared between all of the samples hit by the triangle in a single (final resolution) pixel.
• Once rasterization is complete, groups of these high resolution samples are averaged together.
MIPMAPPING

use “image pyramid” to precompute averaged versions of the texture

Without MIP-mapping

With MIP-mapping
WHAT'S INTERPOLATION?

• Intuitively: fill in all the values between the given ones
CONSTANT INTERPOLATION

• Also called “nearest neighbor”
• Very simple
• But discontinuous
LINEAR INTERPOLATION

• Draw a line between each consecutive pair of points

• Continuous
• But not smooth!
• Your animation/color/etc. will be non-smooth as well.
POLYNOMIAL INTERPOLATION

\[ f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \]
RUNGE’S PHENOMENON

If degree of the polynomial is too high, the function becomes very wiggly
SPLINES

- We limit degree of polynomial to something low, e.g. 3
- Resort to piecewise functions
- We can make the overall function smooth!

\[ f_X(x) = \begin{cases} 
\frac{1}{4}(x + 2)^3 & -2 \leq x \leq -1 \\
\frac{1}{4}(3|x|^3 - 6x^2 + 4) & -1 \leq x \leq 1 \\
\frac{1}{4}(2 - x)^3 & 1 \leq x \leq 2 
\end{cases} \]
SPLINES

• We limit degree of polynomial to something low, e.g. 3
• Resort to piecewise functions
• We can make the overall function smooth!

\[
f_X(x) = \begin{cases} 
 \frac{1}{4}(x + 2)^3 & -2 \leq x \leq -1 \\
 \frac{1}{4}(3|x|^3 - 6x^2 + 4) & -1 \leq x \leq 1 \\
 \frac{1}{4}(2 - x)^3 & 1 \leq x \leq 2 
\end{cases}
\]
HERMITE CUBIC BASIS

Four polynomials that satisfy the conditions

\[
\begin{align*}
  h_{00}(t) &= t^2(2t - 3) + 1 \\
  h_{01}(t) &= -t^2(2t - 3) \\
  h_{10}(t) &= t(t - 1)^2 \\
  h_{11}(t) &= t^2(t - 1)
\end{align*}
\]
FROM CURVES TO SURFACES: TENSOR SPLINES

• Curve: linear combination of basis functions $B_i(u)$ with coefficients $P_i$

$$C(u) = \sum_{i=0}^{n} P_i B_i(u)$$

• Surfaces: treat surface as a curve of curves

• Assume $P_i$ is not constant, but a function of second parameter $v$: $P_i(v) = \sum_{j=0}^{m} Q_{ij} B_j(v)$

$$C(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} Q_{ij} B_j(v) B_i(u)$$
INTERPOLATION VS EXTRAPOLATION

Interpolation – approximate values in between data points
Extrapolation – predict function behavior outside data range
BILINEAR PATCHES

Given $P_{00}, P_{01}, P_{10}, P_{11}$ associated parametric bilinear surface is:

$$u, v \in [0, 1]$$

$$P(u, v) = (1-u)(1-v)P_{00} + (1-u)vP_{01} + u(1-v)P_{10} + uvP_{11}$$
RECONSTRUCTION FROM VOLUME DATA

• Volume data: view as voxel grid with values at vertices
  • Defines implicit function in 3D
    • interpolate grid values

• Shape defined by isosurface
  • isosurface = set of points with constant iso-value $\alpha$
  • separates values above $\alpha$ from values below

• Reconstruction – Extract triangulation approximating isosurface
For each voxel produce set of triangles
  - Based on above/below corner configuration
  - Empty for non-intersecting voxels
  - Approximate surface inside voxel
AMBIGUOUS FACES

• Face containing two diagonally opposite marked grid points and two unmarked ones

• Two locally valid interpretations

• Source of MC consistency problem
PHYSIOLOGY OF VISION

• the retina
  • rods
    • b/w, edges
  • cones
    • 3 types
    • color sensors
• uneven distribution
  • dense fovea
TRICROMACY

• three types of cones
  • L or R, most sensitive to red light (610 nm)
  • M or G, most sensitive to green light (560 nm)
  • S or B, most sensitive to blue light (430 nm)

• color blindness results from missing cone type(s)
RGB VS XYZ REVISITED

- another view of why the R curve goes negative
CIE COLOR SPACE

• CIE defined 3 “imaginary” lights X, Y, Z
  • any wavelength \( \lambda \) can be matched perceptually by positive combinations
  • basis functions!

Note that:

\[ X \sim R \]
\[ Y \sim G \]
\[ Z \sim B \]
COLOR INTERPOLATION, DOMINANT & OPPONENT WAVELENGTH

Complementary wavelength

Dominant wavelength

White point

Complementary wavelength
THE CMY COLOR MODEL

• Used in color printing
  • light is absorbed by dyes
• Cyan, Magenta and Yellow primaries are complements of Red, Blue and Green
• Primaries (dyes) subtracted from white paper
  • Red = White-Cyan = White-Green-Blue  (0,1,1)
  • Green = White-Magenta = White-Red-Blue (1,0,1)
  • Blue = White-Yellow = White-Red-Green  (1,1,0)
  • (r,g,b) = (1-c,1-m,1-y)
HSV COLOR SPACE

more intuitive color space for people

• H = Hue
  • dominant wavelength, “color”

• S = Saturation
  • how far from grey/white

• V = Value
  • how far from black
  • also: brightness B, intensity I, lightness L
HSI/HSV AND RGB

- HSV/HSI conversion from RGB is non-linear
  - H=hue same in both
  - V=value is max, I=intensity is average

\[
H = \cos^{-1} \left[ \frac{1}{2} \left( \frac{(R - G) + (R - B)}{\sqrt{(R - G)^2 + (R - B)(G - B)}} \right) \right]
\]

if \( B > G \),

\[
H = 360 - H
\]

HSI:

\[
S = 1 - \frac{\text{min}(R,G,B)}{I}
\]

\[
I = \frac{R + G + B}{3}
\]

HSV:

\[
S = 1 - \frac{\text{min}(R,G,B)}{V}
\]

\[
V = \text{max}(R,G,B)
\]
YIQ COLOR SPACE

color model used for color TV
• Y is luminance (same as CIE)
• I & Q are color (not same I as HSI!)
• using Y backwards compatible for B/W TVs
• conversion from RGB is linear
  • expressible with matrix multiply

\[
\begin{bmatrix}
Y \\
I \\
Q
\end{bmatrix} = \begin{bmatrix}
0.30 & 0.59 & 0.11 \\
0.60 & -0.28 & -0.32 \\
0.21 & -0.52 & 0.31
\end{bmatrix} \begin{bmatrix}
R \\
G \\
B
\end{bmatrix}
\]

• green is much lighter than red, and red lighter than blue
LUV COLOR SPACE

- Our precision in distinguishing close colors is different for different areas in CIE XYZ
  - “Tolerance”
THANK YOU AND GOOD BYE!

FIN