CSE 472
More Ray Tracing

Announcement (Today's office hour)

- Yiying's Zoom link

https://msu.zoom.us/j/3952532206
Fun with Ray Tracing

- Shadows and shadow feelers
- Antialiasing
- Stochastic ray tracing
- Reflection
- Transmission
- Penumbra
- Depth of field

The Ray Tracing Process

- Render model into intersection data structure
- For each pixel
  - Compute pixel color
  - Set pixel color
Computing Ray Color

// Compute the color for a ray
color RayColor(ray)
  Test ray for intersection
  If(intersection)
    Determine for intersection:
      intersection point, normal, material, texture color
    color = compute-ambient-color
    foreach light
      color += compute-diffuse-color-from-light
      color += compute-specular-color-from-light
    return color
  else
    return background-color

What you get…
This is nicer…

How do we get shadows?

- What is different about the light computation to get shadows?
Shadow Feelers

- Before we compute the color contribution for a light, shoot a ray in the light direction.
- Does it hit anything?
- Called a:
- “Shadow Feeler”

Computing Ray Color

```c
// Compute the color for a ray
color RayColor(ray)
    Test ray for intersection
    If(intersection)
        Determine for intersection:
        intersection point, normal, material, texture color
        color = compute-ambient-color
        foreach light
            if shadowfeeler does not hit anything
                color += compute-diffuse-color-from-light
                color += compute-specular-color-from-light
        return color
    else
        return background-color
```
But, be careful or you get: Stipple!

Problems w/ numerical precision

How far is this point from the polygon?
Problems w/ numerical precision

A little above or a little below

- Your intersection point will be rounded to the nearest float, double, whatever.
  - Unlikely it will be on the polygon!
  - If it’s above, no problem
  - If it’s below, what happens to our shadow feeler?

Term: Self-shadowing
Self-shadowing Solutions

- Ignore any intersection less than some small amount
  - But, how small? What about corners?
- Ignore the polygon we are intersecting with.
  - Most common solution.

Intersect Function

```cpp
bool Intersect(const CRay &p_ray, // The ray we are testing
double p_maxt, // The maximum allowable t
const Object *p_ignore, // A polygon to ignore
const Object *&p_object, // Object we hit \( (\text{return}) \)
double &p_t, // Distance to intersection \( (\text{return}) \)
CGrPoint &p_intersect); // Intersection point \( (\text{return}) \)
```
Intersect Function

bool Intersect(const CRay &p_ray, // The ray we are testing
double p_maxt, // The maximum allowable t
c const Object *p_ignore, // A polygon to ignore
c const Object *p_object, // Object we hit (return)
double &p_t, // Distance to intersection (return)
CGrPoint &p_intersect); // Intersection point (return)

What is the use for p_maxt?

Summary

- Construct a shadow feeler ray
  - Starts at intersection, direction is towards light
- Shoot ray at light (intersection test)
  - Set ignore to ignore the polygon our intersection point is on.
  - Set maxt to the distance to the light.
- If we get an intersection, we are shadowed
Next amazing thing

Aliasing, aka “Jaggies”

Why does this happen?
Point sampling
But, we don’t see like this…

- Why don’t we see like this?
- Why don’t camera images look like this?

Characteristics of the Human Eye

- Receptors have area
  - They are not points
  - What would this mean?

- Receptors are randomly distributed
  - Not true for cameras, but they do get more artifacts than we see
That “area” issue...

What we want (to simulate nature)

- We want the color of a pixel to be the area-weighted average of what’s under the box for the pixel
  - How can we do this with rays?
General answer to all things ray tracing…

Instead of one ray through the center…
But, where?

- Bad choice
  - Rectangular grid
    - Causes animations problems

Common Alternatives

- Fixed Jitter Tables
  - OpenGL book has one
  - Not really a very great choice

- Random
  - The choice of champions
Antialiasing in Ray Tracing

- For each pixel
  - For i=1 to antialiascnt
    - Generate random location in pixel
    - Shoot ray through random pixel
  - Ray color is average of all rays

uniform random numbers

- First idea is to use uniformly distributed random number (rand())
  - This does not work as well because they tend to have clumps.
The Poisson Disk Distribution

- Simple idea
  - We generate a set of random numbers, no two of which are closer than some minimum distance to each other

- How would you do this
  - Think of generating one random number at a time
  - How do you do this in 2D?
Poisson Disk Sample Generator

For each set of random numbers needed:
    history.clear

For each random number needed:
    do
        valid = true // Till we know otherwise
        point p = (rand(min, max), rand(min, max))
        foreach item in history
            if(Distance(item, p) < minimumD
                valid = false
        while !valid
        history.add(p)
    return p

Poisson Disk Distributions
What’s the minimum distance?

- I usually use: range / (2 * sqrt(N))
  - N points in 2D spaced evenly would be range / sqrt(N) apart
  - We make the minimum distance half that

- When sampling an area that is not square, use: sqrt(area) / (2 * sqrt(N))
Where does the name come from?

Poisson is French for “Fish”
BTW…

- The rods and cones in the eye approximate a Poisson distribution
  - Well, what do you know about that!!

Antialiasing Ray Tracing

```cpp
CPoisson2D poisson
poisson.SetMinDistance( 1 / (2 * sqrt(antialiascnt)) )

for r = 0 to height-1
    for c = 0 to width-1
        poisson.reset()
        color = (0, 0, 0)

        for a=1 to antialiascnt
            p = poisson.generate()
            ray.origin = (0, 0, 0)
            ray.direction.x = xmin + (c + p.x) / width * xwid
            ray.direction.y = ymin + (r + p.y) / height * ywid
            color += RayColor(ray)

        color /= antialiascnt
        pixel(r, c) = RangeBound(color)
```

CSE 472 S2019
Stochastic Ray Tracing

- The term for the use of random rays to increase quality in ray tracing is:
  - Stochastic Ray Tracing
- Sometimes also called:
  - Distributed Ray Tracing
    - A most unfortunate choice of words

Next?
Reflections

- Hall indicated what to do with the reflection, so where does it come from?
  - Shoot more rays
  - Shoot a ray in the reflection direction and see what color you see.
    - The Self-Shadowing issue still applies!

RayColor function

- You should have a RayColor function
- Here’s what mine looks like:

```cpp
void CRaytraceRenderer::RayColor(const CRay &p_ray, CRayColor &p_color, int p_recurse, const CUseIntersection::Object *p_ignore)
```

- This will call itself
  - We call this a recursive ray tracer
  - What is your stopping condition?
Transmission

- I'll let you figure this one out...

Recursion issues

- Don’t recurse if you don’t need to
  - What if the specular color is black?

- Recursion is expensive
  - Decide how much you want to do
What’s different, now?
Penumbra

- Penumbra is a soft-edged shadow
  - You get it when the light has area
  - Often called “soft shadows”

How do we solve this problem?

- All together, now!
How do we solve this problem?

- All together, now!

Implementing Penumbra

Sample point on the surface of the light with a Poisson Distribution. Minimum distance is:

\[ d = \frac{\sqrt{\text{lightArea}}}{\sqrt{\text{numRays}}} \times 0.5 \]
How to find a spot on the surface of a rectangular light

\[ \text{len} = \text{length}(b - a) \]
\[ \text{wid} = \text{length}(d - a) \]
\[ \text{area} = \text{len} \times \text{wid} \]
\[ x = \frac{x_p}{\text{len}} \quad y = \frac{y_p}{\text{wid}} \]
\[ \text{point} = a (1 - x) (1 - y) + b (x) (1 - y) + c (x) (y) + d (1 - x) (y) \]

\((x_p, y_p)\) is from Poisson distribution from \((0, 0)\) to \((\text{len}, \text{wid})\)

RayColor with Penumbra

\begin{verbatim}
C::Poisson2D poisson
Poisson.SetMinDistance( sqrt(lightarea) / (2 * sqrt(antialiascnt) )

color = ambient light

foreach light
  for p=1 to penumbraCnt
    PP = random point on surface of light (Poisson distribution)
    L = PP – intersect
    if shadowfeeler doesn’t hit anything
      color += contribution from this light / penumbraCnt

  color += any reflections
  color += any transmission

return color
\end{verbatim}
Stats for that image

Kd Tree Nodes: 31399
Tree Depth: 39
Intersection Tests: 112,276,007
Polygon Tests: 476,466,446
Average polygon tests per IT: 4.24371
Depth of Field

- What causes depth of field in a camera or your eye?

- How would you implement it?

Putting it all together

- We have three stochastic things:
  - Antialiasing
  - Penumbra
  - Depth of Field

- What’s an efficient way to do all of these together?
Motion blur

- What causes motion blur?

- How would you implement it?

Fog

- Any ideas?
Stars

- Why are they hard?

Stochastic Ray Tracing

- Problem with ray tracing: Point sampling

- We can introduce randomness all over the place
  - Jitter for antialiasing
  - Light area for penumbra
  - Depth of field and motion blur

[Jensen07]
Path Tracing

1) Sample each pixel with N rays
2) At each intersection, trace only one ray
   proportion of rays in each category
   should match the actual (surface) distribution

<table>
<thead>
<tr>
<th>Ray Tracing</th>
<th>Path Tracing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reduces the amount of work deeper in the ray tree</td>
</tr>
</tbody>
</table>

Adaptive sampling

- Determine if number of samples is enough
  - If not, increase the number
- Works for uniform and stochastic sampling
Ray tracing from the light sources

- In regular ray tracing we don’t see
  - Lighting reflected through mirrors
  - Surfaces lit by reflection from other surfaces

- Large set of rays from each light
  - Computes light that hits surfaces
  - General solution for diffuse reflection?
  - Still requires tracing from the eye as pass 2

Photon map

- Photon by light-based raytracing
- Separating GI from rendering
- Stored in kd-trees
- Good for
  - Caustics, subsurface scattering,
  - …
Photon path

- Start from light sources
- Intensity -> photon

Final Gathering

- Radiance Computation
- Distributed ray tracing
Another example

[Jensen00]

RTX (check http://intro-to-dxr.cwyman.org/)