Lecture 14:

Modern Real-Time Rendering Techniques

Computer Graphics: Rendering, Geometry, and Image Manipulation
Stanford CS248A, Winter 2023
Course projects

- Project deadlines:
  - Proposal: no later than Friday March 3rd... but ungraded (get it in early if you can)
  - Final video: Monday, March 20
  - Final writeup+code: Tuesday, March 21

- On Tuesday, March 21st at 3:30pm we’ll have a showcase where we watch all the videos
  - Highly encouraged to come in person, but you can watch online
Last couple of lectures: ray-scene queries

What object is visible to the camera?
What light sources are visible from a point on a surface (is a surface in shadow?)
How much radiance is incident from a given direction?
Rasterization: algorithm for "camera ray"-scene queries

- Rasterization is an efficient implementation of ray casting where:
  - Ray-scene intersection is computed for a batch of rays
  - All rays in the batch originate from the same origin
  - Rays are distributed uniformly in the plane of projection

Note: rasterization does not yield uniform distribution in angle... angle between rays is smaller away from the view direction than it is in the center of the view because equal steps in Y are not equal steps in angle.
**Review: basic rasterization algorithm**

Sample = 2D point  
Coverage: 2D triangle/sample tests (does projected triangle cover 2D sample point)  
Occlusion: depth buffer

```
initialize z_closest[] to INFINITY   // store closest-surface-so-far for all samples
initialize color[]                  // store scene color for all samples
for each triangle t in scene:       // loop 1: over triangles
    t_proj = project_triangle(t)
    for each 2D sample s in frame buffer:  // loop 2: over visibility samples
        if (t_proj covers s)
            compute color of triangle at sample
            if (depth of t at s is closer than z_closest[s])
                update z_closest[s] and color[s]
```

“*Given a triangle, find the samples it covers*”

(finding the samples is relatively easy since they are distributed uniformly on screen)

More efficient hierarchical rasterization:

For each TILE of image

- If triangle overlaps tile, check all samples in tile
Review: OpenGL/Direct3D graphics pipeline

* Several stages of the modern OpenGL pipeline are omitted

1. Input: vertices in 3D space
2. Vertices positioned in normalized coordinate space
3. Triangles positioned on screen
4. Fragments (one fragment per covered sample)
5. Shaded fragments
6. Output: image (pixels)
Review: basic ray casting algorithm

Sample = a ray in 3D

Coverage: 3D ray-triangle intersection tests (does ray “hit” triangle)

Occlusion: closest intersection along ray

initialize color[] // store scene color for all samples
for each sample s in frame buffer: // loop 1: over visibility samples (rays)
    r = ray from s on sensor through pinhole aperture
    r.min_t = INFINITY // only store closest-so-far for current ray
    r.tri = NULL;
    for each triangle tri in scene: // loop 2: over triangles
        if (intersects(r, tri)) { // 3D ray-triangle intersection test
            if (intersection distance along ray is closer than r.min_t)
                update r.min_t and r.tri = tri;
        }
    color[s] = compute rejected radiance from triangle r.tri at hit point

And as you know now, a performant raytracer will use an acceleration structure like a BVH.

Compared to rasterization approach: just a reordering of the loops!

“Given a ray, find the closest triangle it hits.”
Theme of this part of the lecture

A surprising number of advanced lighting effects can be efficiently approximated using the basic primitives of the rasterization pipeline, without the need to actually ray trace the scene geometry. Instead we are going to approximate the use of ray tracing with:

- Rasterization
- Texture mapping
- Depth buffer for occlusion

These techniques have been the basis of high quality real-time rendering for decades. Although in recent years they are being to be replaced by ray tracing as ray tracing performance is not fast enough to be used in real-time applications.
Shadows
How much light is REFLECTED from p toward p₀

\[ L(p, \omega_o) = \sum_i f(p, \omega_i, \omega_o) V(p, p_i) L_i \cos \theta_i \]

Visibility term:

\[ V(p, p_i) = \begin{cases} 1, & \text{if } P \text{ is visible from } P_i \\ 0, & \text{otherwise} \end{cases} \]

(Point light 1 is at P₁ and emits L₁)

(Point light 2 is at P₂ and emits L₂)
Review: How to compute $V(p, p_i)$ using ray tracing

- Trace ray from point $P$ to location $P_i$ of light source
- If ray hits scene object before reaching light source… then $P$ is in shadow
Convince yourself this algorithm produces “hard shadows” like these (what you’d see on a sunny day).
Or this...
Point lights generate “hard shadows”
(Either a point is in shadow or it’s not)

\[ V(p, p_i) = \begin{cases} 
1, & \text{if } p \text{ is visible from } L_i \\
0, & \text{otherwise} 
\end{cases} \]
What if you didn’t have a ray tracer, just a rasterizer?
We want to shade these points (aka fragments)

What “shadow rays” do you need to compute shading for this scene?
Shadow mapping

[Williams 78]

1. **Place camera at position of the scene’s point light source**

2. Render scene to compute depth to closest object to light along a uniformly spaced set of “shadow rays” (note: answer is stored in depth buffer after rendering)

3. Store precomputed shadow ray intersection results in a texture map

“Shadow map” = depth map from perspective of a point light.
(Store closest intersection along each shadow ray in a texture)
Result of shadow texture lookup approximates visibility result when shading fragment at $P$.

Precomputed shadow rays shown in red:
Distance to closest object in scene has been precomputed and stored in “shadow map”.

Camera position

$P_i$

$P$

Surface
Interpolation error

Bilinear interpolation of shadow map values (red line) only approximates distance to closest surface point in all directions from the camera.
Shadow aliasing due to shadow map undersampling

Shadows computed using shadow map

Correct hard shadows
(result from computing visibility along ray between surface point and light directly using ray tracing)
Area light

Soft shadow boundary

Credit: Jaime Velasco (https://all3dp.com/2/blender-lighting-simply-explained/)
Area light

Penumbra (Region of partial shadow)

Umbra (Region of complete shadow)

Credit: Jaime Velasco (https://all3dp.com/2/blender-lighting-simply-explained/)
Shadow cast by an area light (via ray tracing)

Notice that a fraction of the light from an area light toward a point P may reach that point (partial occlusion)

(Partially illuminated)
Percentage closer filtering (PCF) — hack!

- Instead of sampling shadow map once, perform multiple lookups around desired texture coordinate.
- Tabulate fraction of lookups that are in shadow, modulate light intensity accordingly.

**Hard shadows**
(one lookup per fragment)

**PCF shadows**
(16 lookups per fragment)

<table>
<thead>
<tr>
<th>Shadow Map Values</th>
<th>Consider Case Where Distance From Light to Surface is 0.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0 0</td>
<td>1</td>
</tr>
<tr>
<td>0 0 0 0 0 0 1 1</td>
<td>1</td>
</tr>
<tr>
<td>0 0 0 0 0 1 1 1</td>
<td>1</td>
</tr>
<tr>
<td>0 0 0 0 1 1 1 1</td>
<td>1</td>
</tr>
<tr>
<td>0 0 0 1 1 1 1 1</td>
<td>1</td>
</tr>
<tr>
<td>0 0 1 1 1 1 1 1</td>
<td>1</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1</td>
<td>1</td>
</tr>
</tbody>
</table>
What PCF computes

The fraction of these rays that are shorter than $|P - P_L|$
Shadow cast by an area light

Actual illumination at P is given by fraction of these rays that are occluded.
Q. Why isn’t the surface in shadow completely black?

Answer: Assumption that some amount of “ambient light” (light scattered from off surfaces) hits every surface. Here... ambient light is just a constant.
Ambient occlusion

This scene contains an environment light source that has equal illumination from all directions. (e.g., an overcast day)

All surfaces are diffuse reflectors. Without accounting for shadows, all surfaces should be the same color.
Hack: ambient obscurance

Idea:

Precompute “fraction of hemisphere” that is occluded within distance $d$ from a point (via a ray tracer)  
Store this fraction in a texture map  
When shading, attenuate environment lighting by this fraction
“Screen-space” ambient occlusion in games

1. Render scene to depth buffer
2. For each pixel $p$, “ray trace” the depth buffer to estimate local occlusion of hemisphere - use a few samples per pixel
3. Blur the per-pixel occlusion results to reduce noise
4. When shading pixels, darken direct environment lighting by occlusion amount computed for the current pixel
Ambient occlusion

Direct Lighting (no self-shadowing computations)

Lighting modulated by ambient occlusion

Indirect illumination is a significant factor in realistic lighting. Ev-
i-3.7 [Three-Dimensional Graphics and Realism]: Color, shad-
ing, and texture

This paper presents the screen space AO algorithm we developed
to provide a sense of scale, and as a spatial cue through contact
 shadows. It is based on a new derivation of screen-space
algorithm from the Alchemy engine used at Vicarious Visions in com-
mmercial games. It is based on a new derivation of screen-space

demonstrates its visual impact. The left image shows a scene with
may be obscured from any direction. This is why screen space ap-
methods: I.3.7 [Three-Dimensional Graphics and Realism]: Color,
the limitations are sample variance

Ambient occlusion

Ambient occlusion

Any point may be obscured from any direction. This is why screen space ap-
may be obscured from any direction. This is why screen space ap-
methods: I.3.7 [Three-Dimensional Graphics and Realism]: Color,
The algorithm estimates obscurance at a pixel from sample points
which are independent of the number of polygons,
the latter. We attribute the visual fidelity and robustness of
dependencies of intensity, and maximize temporal coherence.

The Alchemy Screen-Space Ambient Obscurance Algorithm

Morgan McGuire

Ambient obscurance

The Alchemy Screen-Space Ambient Obscurance Algorithm

Vicarious Visions

Ambient occlusion

Ambient occlusion

Ambient occlusion

Ambient occlusion

Ambient occlusion
Reflections
What is wrong with this picture?
Reflections

Image credit: NVIDIA
Reflections
Recall: perfect mirror material
Recall: perfect mirror reflection

Light reflected from $P_1$ in direction of $P_0$ is incident on $P_1$ from reflection about surface at $P_1$. 
Rasterization: “camera” position can be reflection point

Environment mapping:
place ray origin at reflective object

Yields approximation to true reflection results. Why?

Cube map:
stores results of approximate mirror reflection rays

(Question: how can a glossy surface be rendered using the cube-map)

Center of projection
Environment map vs. ray traced reflections


Image credit: Control
Environment map vs. ray traced reflections


Image credit: Control
Indirect lighting
Indirect lighting

Why is this gray wall tinted red?

Why is this point not black?
Precomputed lighting

- Precompute accurate lighting for a scene offline using a ray tracer (possible for static lights)

- “Bake” results of lighting into texture map
Precomputed lighting in Unity Engine

Visualization of light map texture coordinates

Image credit: Unity / Alex Lovett
Growing interest in real-time ray tracing

- I’ve just shown you an array of different techniques for approximating different advanced lighting phenomenon using a rasterizer
- Challenges:
  - Different algorithm for each effect (code complexity)
  - Algorithms may not compose
  - They are only approximations to the physically correct solution (“hacks!”)
- Traditionally, tracing rays to solve these problems was too costly for real-time use
  - That is rapidly changing…

This image was ray traced in real-time on a GPU
This image was rendered in real-time on a single high-end GPU
Real-time ray tracing challenge:

Need to shoot many rays per pixel to accurately simulate advanced lighting effects

Want high-performance interactive rendering 😞
Innovation 1: Hardware innovation: custom GPU hardware for RT

NVIDIA GeForce RTX 3080 GPU
Innovation 2: better importance sampling algorithms

Path traced: 1 path/pixel (8 ms/frame)

Path traced: 1 path/pixel using ReSTIR GI (8.9 ms/frame)

Key idea: cache good paths, reuse good paths found from prior frames or for prior pixels in same frame

[Ouyang et al. 2021]
Innovation 3: Neural network based denoising

Idea: Use neural image-to-image transfer methods to convert cheaper to compute (but noisy) ray traced images into higher quality images that look like they were produced by tracing many rays per pixel.
Surface normals
Recall: numerical integration of light (via Monte Carlo sampling) suffers from high variance, resulting in images with "noise"
Denoised results
16 paths/pixel (denoised)
256 paths/pixel (denoised)
Summary

- Until very recently, it was too expensive to perform ray tracing in real-time graphics systems
- Many rasterization-based methods for approximating ray traced effects (shadows, reflections, etc).
- In the last five years, there’s been a major shift toward using more ray tracing in real-time graphics systems
  - Brute force: new ray tracing hardware supported by graphics APIs (D3D12/Vulkan)
  - Algorithmic innovation: smarter ways to importance sample paths
  - Introduction of ML: use ML to convert noisy low sample count images to images that “look like” images that were ray traced at high sample counts
- Gradual introduction of ray tracing into shipping games
Morphological anti-aliasing (MLAA)

Detect carefully designed patterns in rendered image
For detected patterns, blend neighboring pixels according to a few simple rules
(“hallucinate” a smooth edge.. it’s a hack!)

Note: modern interest in replacing MLAA patterns with DNN-based anti-aliasing.
Morphological anti-aliasing (MLAA)

Aliased image
(one shading sample per pixel)

Zoomed views
(top: aliased, bottom: after MLAA)

After filtering using MLAA
Modern trend: learn anti-aliasing functions

Use modern image processing deep networks to reduce aliasing artifacts from rendered images.

https://wccftech.com/nvidia-dlss-explained-nvidia-ngx/
Learn anti-aliasing functions

Use modern image processing deep networks to reduce aliasing artifacts from rendered images.

Traditional Heuristic (TXAA)  Learned AA (DLSS)
Summary: deferred shading

- Very popular technique in modern games
- Creative use of graphics pipeline
  - Create a G-buffer, not a final image
- Two major motivations
  - Convenience and simplicity of separating geometry processing logic/costs from shading costs
  - Potential for high performance under complex lighting and shading conditions
    - Shade only once per sample despite triangle overlap
    - Often more amenable to "screen-space shading techniques"
      - e.g., screen space ambient occlusion