Lecture 16:

Architectural Support for Ray Tracing

Visual Computing Systems
Stanford CS348K, Spring 2020
Ray Tracing for architects (in 15 minutes)

Take that Pete Shirley!
Rasterization and ray casting are two algorithms for solving the same problem: determining “visibility along rays”
Visibility problem

Question 1: what samples does the triangle overlap? ("coverage")

Question 2: what triangle is closest to the camera in each sample? ("occlusion")
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: triangles
    t_proj = project_triangle_to_screen(t)
for each 2D sample s in frame buffer: // loop 2: visibility samples
    if (t_proj covers s)
        compute color of triangle at sample       // run fragment shader
        if (depth of t at s is closer than z_closest[s])  // depth buffer check/update
            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)
Recall: depth test logic

depth_test(frag_depth, frag_color, x, y) {
    if (tri_depth < depth_buffer[x][y]) {
        // triangle is closest object seen so far at this
        // sample point. Update depth and color buffers.
        depth_buffer[x][y] = tri_depth;  // update depth_buffer
        color[x][y] = frag_color;        // update color_buffer
    }
}
The visibility problem (described differently)

- In terms of casting rays from a simulated camera:
  - What scene primitive is “hit” by a ray originating from a point on the virtual sensor and traveling through the aperture of the pinhole camera? (coverage)
  - What primitive is the first hit along that ray? (occlusion)
Does a ray (in 3D) hit a triangle (in 3D)?
Ray-plane intersection

- Suppose we have a plane $N^T x = c$
  - $N$ - unit normal
  - $c$ - offset
- How do we find intersection with ray $r(t) = o + td$?

**Key idea:** again, replace the point $x$ with the ray equation $t$:

$$N^T r(t) = c$$

- Now solve for $t$:
  $$N^T (o + td) = c \quad \Rightarrow t = \frac{c - N^T o}{N^T d}$$
- And plug $t$ back into ray equation:
  $$r(t) = o + \frac{c - N^T o}{N^T d} d$$
Ray-triangle intersection

- **Algorithm:**
  - Compute ray-plane intersection
  - Q: What do we do now?
  - A: Compute barycentric coordinates of hit point?
  - If barycentric coordinates are all positive, point is in triangle

---

**Möller–Trumbore intersection algorithm - Wikipedia**
https://en.wikipedia.org/wiki/Möller–Trumbore_intersection_algorithm

The Möller–Trumbore ray-triangle intersection algorithm, named after its inventors Tomas Möller and Ben Trumbore, is a fast method for calculating the ...

**Fast Minimum Storage Ray-Triangle Intersection**

We present a clean algorithm for determining whether a ray intersects a triangle. ...

**Optimizing Ray-Triangle Intersection via Automated Search**
www.cs.utah.edu/~aek/research/triangle.pdf

A method is used to further optimize the code produced via the fitness function. ... For these 3D methods we optimize ray-triangle intersection in two different ways.

**Comparative Study of Ray-Triangle Intersection Algorithms**

V Shumsky - Cited by 1 - Related articles
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: visibility samples (rays)
    r = ray from s on sensor through pinhole aperture
    r.closest_dist = INFINITY // only store closest-so-far for current ray
    r.closest_tri = NULL;
    for each triangle tri in scene: // loop 2: triangles
        if (intersects(r, tri)) {
            if (intersection distance along ray is closer than r.closest_dist)
                update r.closest_dist and r.closest_tri = tri;
        }
        color[s] = compute surface color of triangle r.tri at hit point // hit shader

Compared to rasterization approach: just a reordering of the loops! (+ math in 3D)
“Given a ray, find the closest triangle it hits”

The brute force “for each triangle” loop is typically implemented using a search acceleration structure. (A rasterizer’s “for each sample” inner loop is not just a loop over all screen samples either.)
A simpler problem

- Imagine I have a set of integers $S$
- Given an integer, say $k=18$, find the element of $S$ closest to $k$:

  10 123 2 100 6 25 64 11 200 30 950 111 20 8 1 80

What’s the cost of finding $k$ in terms of the size $N$ of the set?

Can we do better?

Suppose we first sort the integers:

  1 2 6 8 10 11 20 25 30 64 80 100 111 123 200 950

How much does it now cost to find $k$ (including sorting)?

Cost for just ONE query: $O(n \log n)$
Amortized cost over many queries: $O(\log n)$

worse than before! :-(

...much better!
Bounding volume hierarchy (BVH)

- **Leaf nodes:**
  - Contain *small* list of primitives

- **Interior nodes:**
  - Proxy for a *large* subset of primitives
  - Stores bounding box for all primitives in subtree
Ray-scene intersection using a BVH

```c
struct BVHNode {
    bool leaf;  // true if node is a leaf
    BBox bbox;  // min/max coords of enclosed primitives
    BVHNode* child1;  // "left" child (could be NULL)
    BVHNode* child2;  // "right" child (could be NULL)
    Primitive* primList;  // for leaves, stores primitives
};

struct HitInfo {
    Primitive* prim;  // which primitive did the ray hit?
    float dist;  // what is dist along ray to the hit?
};

void find_closest_hit(Ray* ray, BVHNode* node, HitInfo* closest) {
    HitInfo hit = intersect(ray, node->bbox);  // test ray against node’s bounding box
    if (hit.prim == NULL || hit.dist > closest->dist)
        return;  // don’t update the hit record

    if (node->leaf) {
        for (each primitive p in node->primList) {
            hit = intersect(ray, p);
            if (hit.prim != NULL && hit.dist < closest->dist) {
                closest->prim = p;
                closest->dist = hit.dist;
            }
        }
    } else {
        find_closest_hit(ray, node->child1, closest);
        find_closest_hit(ray, node->child2, closest);
    }
}
```

How could this occur?
Workload:
basic rasterization vs. basic ray casting

- **Basic rasterization:**
  - *Stream over triangles* in order (never have to store in entire scene in memory, naturally supports unbounded size scenes)
  - Do not need random access to triangle data
  - Must keep depth buffer in memory
    - Need *random access to a regular, fixed-size data structure (color/depth buffer)*

- **Ray casting:**
  - *Stream over screen samples* (rays)
    - No need to store closest depth so far for the entire screen (just current ray)
  - Must store acceleration structure for scene in memory (*unpredictable access to irregular search tree*)
Generality of ray-scene queries

Ray-scene intersection is a general visibility primitive

What object is visible along this ray?

What object is visible to the camera?

What light sources are visible from a point on a surface (Is a surface in shadow?)

What reflection is visible on a surface?
Direct illumination + reflection + transparency

Image credit: Henrik Wann Jensen
Global illumination solution

Image credit: Henrik Wann Jensen
Direct illumination

\( p \)
One-bounce global illumination
Sixteen-bounce global illumination
Sampling light paths

Image credit: Wann Jensen, Hanrahan
The reflection equation

\[ L_o(p, ω_o) = L_e(p, ω_o) + \int_{H^2} f_r(p, ω_i → ω_o) L_i(p, ω_i) \cos θ_i \, dω_i \]
1 area light sample
(high variance in irradiance estimate)
16 area light samples
(lower variance in irradiance estimate)
Accounting for indirect illumination

Incoming light energy from direction \( \omega_i \) may be due to light reflected off another surface in the scene (not an emitter)

\[
L_o(p, \omega_o) = L_e(p, \omega_o) + \int_{H^2} f_r(p, \omega_i \rightarrow \omega_o) L_i(p, \omega_i) \cos \theta_i \, d\omega_i
\]
Path tracing pseudocode

```
Color pathtrace(Ray ray) {
    Intersection isect = scene.intersect(ray);
    Vector2D wo = -ray.d;
    Vector2D wi;
    float pdf;
    generate_direction_sample(isect.brdf, wo, &wi, &pdf); // random direction to sample indirect
    Color f = isect.brdf.f(wo, wi);
    float terminateProbability = 1.f - f.rho(); // termination probability based on reflectance.
    // Lower reflectance = high chance of terminating
    Lo = estimate_direct_lighting(isect, wo); // reflectance in outgoing direction
    if (RandomFloat() >= terminateProbability)
        Lo += ((f * pathtrace(Ray(isect.P, wi)) * Dot(wi, isect.N) / (pdf * (1-terminateProbability)));
    return Lo;
}
```
One sample per pixel
1024 samples per pixel
Ray tracing performance challenges

3D ray-triangle math is more expensive than point-in-triangle math of rasterization

Ray-scene intersection requires traversal through bounding volume hierarchy acceleration structure
- Unpredictable data access
- Rays are essentially randomly oriented after enough bounces

To simulate advanced effects in a ray tracer, renderer must trace many rays per pixel to reduce variance (noise) that results from numerical integration (via Monte Carlo sampling)
Ray tracing architectures
Keep in mind

- An application developer has always been able to write a ray tracer in CUDA

- So the ability to use a GPU to perform ray tracing is not new

- But by establishing a new architecture for ray tracing, GPU vendors give themselves the opportunity to accelerate/optimize implementations
Recall: traditional graphics pipeline architecture

Vertices
- Vertex Generation
- Vertex Processing

Primitives
- Primitive Generation
- Primitive Processing

Fragments (Fragment Generation)
- Rasterization
- Fragment Processing

Pixels
- Frame-Buffer Ops

Memory
- Uniform data
- Texture buffers

Output image buffer
Ray tracing architecture (RTX)

- TraceRay() is a blocking function that can be called by shaders
Ray tracing architecture maintains BVH acceleration structure for all scene objects and “shader table” (which shader to run for each object in scene)
RT acceleration in NVIDIA Turing GPUs

- Custom processing elements for:
  - ray-triangle intersection
  - ray-box intersection / BVH traversal
Ray traversal "coherence"

- r0 visits nodes: A, B, D, E...
- r1 visits nodes: A, B, D, E...

Bandwidth reduction: BVH nodes (and triangles) loaded into cache for computing scene intersection with r0 are cache hits for r1.
Ray traversal “divergence”

r0 visits nodes: A, B, D, E…
r1 visits nodes: A, B, D, E…
r2 visits nodes: A, B, D, E, C…
r3 visits nodes: A, B, D, E, G…

R2 and R3 require different BVH nodes and triangles
Incoherent rays

Incoherence is a property of both the rays and the scene.

Random rays are “coherent” with respect to the BVH if the scene is one big triangle!
Incoherent rays

Incoherence is a property of both the rays and the scene.

Camera rays become “incoherent” with respect to lower nodes in the BVH if a scene is overly detailed.

(Side note: this suggests the importance of choosing the right geometric level of detail)
Ray incoherence

Nearby rays may hit different surfaces, with different shaders
Consider implications for SIMD processing
Recall difference in ordering

Rasterizer:
For each triangle
  shade SIMD batch of pixels in parallel (with same shader)

Ray tracer:
For each ray in a SIMD-sized batch:
  shade hit point with shader for hit triangle
Global ray reordering

Idea: dynamically batch up rays that must traverse the same part of the scene. Process these rays together to increase locality in BVH access.

1. When ray enters treelet, add rays to per-treelet queue
2. When treelet queue is sufficiently large, intersect enqueued rays with treelet (amortize load of treeless data across all enqueued rays)

Buffering overhead to global ray reordering: must store per-ray “stack” (need not be entire call stack, but must contain traversal history) for many rays.

Per-treelet ray queues sized to fit in caches (or in dedicated ray buffer SRAM)

Partition BVH into treelets
(treelets sized for L1 or L2 cache)

Pharr 1997, Navratil 07, Alia 10
Denoising ray traced images
32 samples per pixel
Deep learning-based denoising

- Idea: Use image-to-image transfer methods based on deep learning to convert cheap to compute (but noisy) ray traced images into higher quality images that look like they were produced by shooting many rays per pixel
Denoising examples

Image credit: Intel Open Image Denoise : https://openimagedenoise.github.io/
Denoising examples

Image credit: Intel Open Image Denoise : https://openimagedenoise.github.io/
Denoising examples

Image credit: Intel Open Image Denoise : https://openimagedenoise.github.io/
Denoising examples
Surprising synergies

- New GPU hardware for ray-tracing operations

- But ray tracing still too expensive for noise-free images in real-time

- Tensor core: specialized hardware for accelerated DNN computations (that can be used to perform sophisticated denoising)
Technologies that are making real-time ray tracing possible

- Better algorithms: fast parallel BVH construction and traversal algorithms (many SIGGRAPH/HPG papers circa 2010-2013)

- GPU hardware evaluation:
  - HW acceleration of ray-triangle insect, BVH traversal
  - Increasingly flexible aspects of traditional GPU pipeline (bindless textures/resources)

- DNN-based image denoising
  - Can make plausible images using small number of rays per pixel
  - Make use of DNN hardware acceleration