Lecture 16:
High-Performance Ray Tracing

Computer Graphics
CMU 15-462/15-662, Fall 2015
Ray tracing is a mechanism for answering “visibility” queries

\[ v(x_1, x_2) = 1 \text{ if } x_1 \text{ is visible from } x_2, \ 0 \text{ otherwise} \]

\[ v(x, x') = 1 \]
\[ v(x', x'') = 0 \]
Using rasterization to answer visibility queries
Another way to think about rasterization

- Rasterization is an efficient algorithm for servicing large batches of visibility queries... for rays with specific properties
  - Assumption 1: Rays have the same origin
  - Assumption 2: Rays are uniformly distributed over plane of projection (within specified field of view)

- Assumptions → significant optimization opportunities
  - Project triangles: reduce ray-triangle intersection to 2D point-in-polygon test
  - Projection to canonical view volume enables use of efficient fixed-point math, custom GPU hardware for rasterization
Shadow mapping: ray origin need not be the scene's camera position [Williams 78]

- Place ray origin at position of point light source
- Render scene to compute depth to closest object to light along uniformly distributed “shadow rays” (answer stored in depth buffer)
- Store precomputed shadow ray intersection results in a texture

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

Shadow rays shown in red:
Distance to closest object in scene is precomputed and stored in texture map ("shadow map")
Shadow aliasing due to undersampling

Shadows computed using shadow map (shadow map resolution is too low)

Correct hard shadows (result from computing $v(x', x'')$ directly using ray tracing)
Environment mapping

Place ray origin at location of reflective object.

Yields approximation to true reflection results. Why?

Cube map: stores results of approximate mirror reflection rays

Center of projection

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

Scene rendered 6 times, with ray origin at center of reflective box (produces “cube-map”)

Image credit: http://en.wikipedia.org/wiki/Cube_mapping
Ambient occlusion
Ambient occlusion

Idea: precompute “amount of hemisphere” that is occluded from a point, attenuate direct environment lighting by this amount.

\[ E(p) = \int_{H^2} V(\omega) L_i(p, \omega) \cos \theta \, d\omega \]

\[ \approx \int_{H^2} V_d(\omega) \, d\omega \int_{H^2} L_i(p, \omega) \cos \theta \, d\omega \]

\[ \approx O \int_{H^2} L_i(p, \omega) \cos \theta \, d\omega \]

Occlusion term: does ray from \( p \) in direction \( w \) hit any scene object within distance \( d \)
“Screen-space” ambient occlusion in games

1. Render scene to depth buffer
2. For each pixel \( p \) (“ray trace” the depth buffer to estimate occlusion of hemisphere - use a few samples per pixel)
3. Blur the occlusion map to reduce noise
4. Shade pixels, darken direct environment lighting by occlusion amount
 Ambient occlusion

Direct Lighting (no shadowing computations)

Lighting modulated by occlusion
Motivations for real-time ray tracing

Many shadowed lights (big pain to manage many shadow maps)
Accurate reflections from curved surfaces

Reduce content creation and game engine development time: single general solution rather than a specialized technique for each lighting effect.
Less parameter tweaking (e.g., choosing shadow map texture size)

Estimate indirect illumination effects (But unclear if ray tracing is best real-time solution for low frequency effects... more to come)

VR may demand more flexible control over what pixels are drawn. (e.g., row-based display rather than frame-based, higher resolution where eye is looking, correct for distortion of optics)
Efficient ray tracing
Imagine I give you a 16-core CPU
How would you parallelize your ray tracer to render this picture?

Image credit: NVIDIA (this ray traced image can be rendered at interactive rates on modern GPUs)
Partition(node)
For each axis: x, y, z:
   initialize buckets
   For each primitive p in node:
       b = compute_bucket(p.centroid)
       b.bbox.union(p.bbox);
       b.prim_count++;
   For each of the B-1 possible partitioning planes evaluate SAH
Execute lowest cost partitioning found (or make node a leaf)
Modern computer architecture 101
What does a processor do?

Fetch/Decode

ALU (Execute)

Execution Context

input

```
ld r0, addr[r1]
mul r1, r0, r0
mul r1, r1, r0
...  
...  
...  
...  
...  
st addr[r2], r0
```

output
A processor executes an instruction stream

My very simple processor: executes one instruction per clock

Fetch/Decode

ALU (Execute)

Execution Context
Execute program

My very simple processor: executes one instruction per clock

- **Fetch/Decode**
- **ALU (Execute)**
- **Execution Context**

### Input
- `ld r0, addr[r1]`
- `mul r1, r0, r0`
- `mul r1, r1, r0`
- ...
- ...
- ...
- ...
- ...
- `st addr[r2], r0`

### Output
Sixteen cores: process 16 tasks in parallel

Sixteen tasks processed at once

Sixteen cores
An efficient ray tracer implementation must use all the cores on a modern processor (this is quite easy)
Multi-core processors

Intel Core i7 (Haswell): quad-core CPU

Intel Xeon Phi: 60 core CPU
Multi-core processors

NVIDIA
GeForce GTX 980
(Maxwell) GPU
SIMD processing
Single instruction, multiple data

Each core can execute the same instruction simultaneously on multiple pieces of data:

e.g., add vector A to vector B (length 8)
32-bit addition performed in parallel for each vector element.
An efficient ray tracer implementation must also utilize the SIMD execution capabilities of modern processors

- CPUs: up to a factor of 8
- GPUs: up to a factor of 32
### Accessing memory has high cost

#### High latency: 100’s of cycles

- Too little bandwidth: modern processors can perform arithmetic much faster than memory can provide data.

---

<table>
<thead>
<tr>
<th>Product</th>
<th>Peak flops (fp)</th>
<th>DRAM Bandwidth (compute/BW ratio)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA Tesla K40</td>
<td>4.4 TF (fp32)</td>
<td>288 GB/sec (8.75 flops/byte)</td>
</tr>
<tr>
<td>(2014, high-end discrete GPU)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NVIDIA Tegra X1</td>
<td>512 GF (fp32)</td>
<td>25.6 GB/sec (10 flops/byte)</td>
</tr>
<tr>
<td>(2015, high-end mobile GPU)</td>
<td>1 TF (fp16)</td>
<td></td>
</tr>
<tr>
<td>Imagination PowerVR GX6450</td>
<td>115.2 GF (fp32)</td>
<td>12.4 GB/sec (9.2 flops/byte)</td>
</tr>
<tr>
<td>(2014, iPhone 6)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Imagination PowerVR GT7900</td>
<td>384 GF (fp32)</td>
<td>TBD since processor is not in shipping SoCs</td>
</tr>
<tr>
<td>(2015, high-end mobile GPU)</td>
<td>768 GF (fp16)</td>
<td></td>
</tr>
<tr>
<td>Intel Integrated HD 350 GPU</td>
<td>384 GF (fp32)</td>
<td>34.1 GB/sec (11.2 flops/byte)</td>
</tr>
<tr>
<td>(2014, integrated desktop GPU)</td>
<td>768 GF (fp16)</td>
<td></td>
</tr>
<tr>
<td>Xbox 360 (2005)</td>
<td>240 GF (fp32)</td>
<td>32.1 GB/sec (7.5 flops/byte)</td>
</tr>
</tbody>
</table>

---

**CMU 15-418, Spring 2015**
An efficient ray tracer implementation must be careful to reduce memory access costs as much as possible.
Rules of the game

- Many individual processor cores
  - Run tasks in parallel

- SIMD instruction capability
  - Single instruction carried out on multiple elements of an array in parallel (8-wide on modern GPUs, 16-wide on Xeon Phi, 8-to-32-wide on modern GPUs)

- Accessing memory is expensive
  - Processor must wait for data to arrive
  - Role of CPU caches is to reduce wait time (want good locality)
Efficient ray traversal algorithms
High-throughput ray tracing

- Want work-efficient algorithms (do less)
  - High-quality acceleration structures (minimize ray-box, ray-primitive tests)
  - Smart traversal algorithms (early termination, etc.)

- Implementations for existing parallel hardware (CPUs/GPUs):
  - High multi-core, SIMD execution efficiency
  - Help from fixed-function processing?

- Bandwidth-efficient implementations:
  - How to minimize bandwidth requirements?
Parallelizing ray-triangle tests?
Parallelize ray-box, ray-triangle intersection

- Given one ray and one bounding box, there are opportunities for SIMD processing
  - Can use 3 of 4 SSE vector lanes (e.g., xyz work, point-multiple-plane tests, etc.)

- Similar SIMD parallelism in ray-triangle test at BVH leaf

- If leaf nodes contain multiple triangles, can parallelize ray-triangle intersection across these triangles
Parallelize over BVH child nodes

- Idea: use wider-branching BVH (test single ray against multiple child node bboxes in parallel)
  - BVH with branching factor 4 has similar work efficiency to branching factor 2
  - BVH with branching factor 8 or 16 is significantly less work efficient (diminished benefit of leveraging SIMD execution)

[Wald et al. 2008]
Parallelize across rays

- Simultaneously intersect multiple rays with scene

- Today: we’ll discuss one approach: ray packets
  - Code is explicitly written to trace N rays at a time, not 1 ray
Simple ray tracer (using a BVH)

// stores information about closest hit found so far
struct ClosestHitInfo {
    Primitive primitive;
    float distance;
};

trace(Ray ray, BVHNode node, ClosestHitInfo hitInfo)
{
    if (!intersect(ray, node.bbox) || (closest point on box is farther than hitInfo.distance))
        return;

    if (node.leaf) {
        for (each primitive in node) {
            (hit, distance) = intersect(ray, primitive);
            if (hit && distance < hitInfo.distance) {
                hitInfo.primitive = primitive;
                hitInfo.distance = distance;
            }
        }
    } else {
        trace(ray, node.leftChild, hitInfo);
        trace(ray, node.rightChild, hitInfo);
    }
}
Ray packet tracing

Program explicitly intersects a collection of rays against BVH at once

RayPacket
{
    Ray rays[PACKET_SIZE];
    bool active[PACKET_SIZE];
};

trace(RayPacket rays, BVHNode node, ClosestHitInfo packetHitInfo)
{
    if (!ANY_ACTIVE_intersect(rays, node.bbox) ||
        (closest point on box (for all active rays) is farther than hitInfo.distance))
        return;

    update packet active mask

    if (node.leaf) {
        for (each primitive in node) {
            for (each ACTIVE ray r in packet) {
                (hit, distance) = intersect(ray, primitive);
                if (hit && distance < hitInfo.distance) {
                    hitInfo[r].primitive = primitive;
                    hitInfo[r].distance = distance;
                }
            }
        }
    } else {
        trace(rays, node.leftChild, hitInfo);
        trace(rays, node.rightChild, hitInfo);
    }
}
Ray packet tracing

Blue = active rays after node box test

Note: r6 does not pass node F box test due to closest-so-far check, and thus does not visit F
Advantages of packets

- Enable wide SIMD execution
  - One vector lane per ray

- Amortize BVH data fetch: all rays in packet visit node at same time
  - Load BVH node once for all rays in packet (not once per ray)
  - Note: there is value to making packets bigger than SIMD width! (e.g., size = 64)

- Amortize work (packets are hierarchies over rays)
  - Use interval arithmetic to conservatively test entire set of rays against node bbox (e.g., think of a packet as a beam)
  - Further arithmetic optimizations possible when all rays share origin
  - Note: there is value to making packets much bigger than SIMD width!
Disadvantages of packets

- If any ray must visit a node, it drags all rays in the packet along with it.

- Loss of efficiency: node traversal, intersection, etc. amortized over less than a packet’s worth of rays.

- Not all SIMD lanes doing useful work.
Ray packet tracing: incoherent rays

When rays are incoherent, benefit of packets can decrease significantly. This example: packet visits all tree nodes. (So all eight rays visit all tree nodes! No culling benefit!)
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!
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)
Improving packet tracing with ray reordering

Idea: when packet utilization drops below threshold, resort rays and continue with smaller packet

- Increases SIMD utilization
- Amortization benefits of smaller packets, but not large packets

Example: consider 8-wide SIMD processor and 16-ray packets (2 SIMD instructions required to perform each operation on all rays in packet)

16-ray packet: 7 of 16 rays active

Reorder rays
Recompute intervals/bounds for active rays

Continue tracing with 8-ray packet:
7 of 8 rays active

[Boulos et al. 2008]
Giving up on packets

- Even with reordering, ray coherence during BVH traversal will diminish
  - Diffuse bounces result in essentially random ray distribution
  - High-resolution geometry encourages incoherence near leaves of tree

- In these situations there is little benefit to packets (can even decrease performance compared to single ray code)
Packet tracing best practices

- Use large packets for eye/reflection/point light shadow rays or higher levels of BVH
  - Ray coherence always high at the top of the tree

- Switch to single ray (intra-ray SIMD) when packet utilization drops below threshold
  - For wide SIMD machine, a branching-factor-4 BVH works well for both packet traversal and single ray traversal

- Can use packet reordering to postpone time of switch
  - Reordering allows packets to provide benefit deeper into tree
  - Not often used in practice due to high implementation complexity
Let’s stop and think

- One strong argument for high-performance ray tracing is to produce advanced effects that are difficult or inefficient to compute given the single point of projection and uniform sampling constraints of rasterization
  - e.g., soft shadows, diffuse interreflections

- But these phenomenon create situations of high ray divergence! (where packet- and SIMD-optimizations are less effective)
Emerging hardware for ray tracing
Emerging hardware for ray tracing

- **Modern academic/announced industry implementations:**
  - Trace single rays, not ray packets (assume most rays are incoherent rays…)

- **Two areas of focus:**
  - Custom logic for accelerating ray-box and ray-triangle tests
    - MIMD designs: wide SIMD execution not beneficial
  - Support for efficiently reordering ray-tracing computations to maximize memory locality (ray scheduling)

- **See “further reading” on web site for a list of references**
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

- Partition BVH into treelets (treelets sized for L1 or L2 cache)
- When ray (or packet) enters treelet, add rays to treelet queue
- When treelet queue is sufficiently large, intersect enqueued rays with treelet (amortize treelet load over 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)

[Pharr 1997, Navratil 07, Alia 10]
PowerVR GR6500 ray tracing GPU
Constructing High-Quality BVHs Quickly
Building a “poor” BVH quickly

1. Discretize each dimension of scene into $2^B$ cells
2. Compute index of centroid of bounding box of each primitive: $(c_i, c_j, c_k)$
3. Interleave bits of $c_i$, $c_j$, $c_k$ to get $3B$ bit-Morton code
4. Sort primitives by Morton code (primitives now ordered with high locality in 3D space: in a space-filling curve!)
   - $O(N)$ radix sort

Simple, highly parallelizable BVH build:

```python
Partition(int i, primitives):
    node.bbox = bbox(primitives)
    (left, right) = partition primitives by bit i
    if there are more bits:
        Partition(left, i+1);
        Partition(right, i+1);
    else:
        make a leaf node
```

2D Morton Order

- $B=1$
- $B=2$
- $B=3$
- $B=4$
Modern, fast BVH build schemes

- Build “poor” BVH quickly using Morton Codes
- Use initial BVH to accelerate construction of high-quality BVH
- Example: [Kerras 2013]

For all treelets of size < N in original “poor” BVH: (in parallel)
- try all possible trees, keep “optimal” topology that minimizes SAH for treeless