Parallel Physically Based Path-tracing and Shading
Part 1 of 2

CIS565 Fall 2012
University of Pennsylvania
by Yining Karl Li
Agenda

• Part 1 (Today):
  • Quick introduction and theory review:
    • The Rendering Equation
    • Bidirectional reflection distribution functions
    • Pathtracing algorithm overview
  • Implementing parallel ray-tracing
    • Recursion versus iteration
    • Iterative ray-tracing

• Part 2 (Wednesday):
  • Distributed ray-tracing/Monte-carlo integration, more on BRDFs
  • Implementing parallel path-tracing
    • Path versus ray parallelization, ray compaction
    • Parallel computation, BRDF evaluation, and you!
  • Parallel approaches to spatial acceleration structures
    • Stack-less KD-tree construction and traversal
    • Bounding volume hierarchies
Path-tracing: Quick Introduction and Theory Review
The Rendering Equation

\[ L_o(p, \omega_o) = L_e(p, \omega_o) + \int_{\Omega} \rho(p, \omega_i, \omega_o) L_i(p, \omega_i) \cos \theta \, d\omega_i \]

• Super high level meaning: [outgoing light] = [incoming light] + [emitted light] + [absorbed light]

\[ L_o(p, \omega_o) = \text{Outgoing light} \]

\[ L_e(p, \omega_o) = \text{Emitted light} \]

\[ \int_{\Omega} d\omega_i = \text{Integrate over a hemisphere in the direction } \omega \text{ over the given point } p \]

\[ \rho(p, \omega_i, \omega_o) = \text{BRDF (Bidirectional Reflectance Distribution Function)} \]

\[ L_i(p, \omega_i) = \text{Incoming Light} \]

\[ \cos \theta = \text{Attenuate incoming light based on the cosine of the angle between the normal } n \text{ and the incoming light direction } \omega_i \]
Bidirectional Reflectance Distribution Functions

• Defines how light is reflected at a given opaque surface
• Can be extended with transmittance to produce the BSDF: Bidirectional Scattering Distribution Function
• Reflectance models:
  • Ideal Specular (think mirrors)
  • Ideal Diffuse
  • Specular/Glossy (won’t cover today)
    • Phong Model
    • Microfacet Models
    • Torrance-Sparrow Model

\[
f_r(\omega_i \rightarrow \omega_r) = \frac{dL_r(\omega_i \rightarrow \omega_r)}{dE_i} \begin{bmatrix} 1 \\ sr \end{bmatrix}
\]
Reflectance Models: Ideal Specular

• Ideal specular reflection: incoming light and outgoing light make the same angle across the surface normal, so angle of incidence = angle of reflection

• Fresnel’s law: defines the behavior of light when moving between mediums with different indices of refraction.
  • Can be approximated with Shlick’s approximation.
Reflectance Models: Ideal Diffuse

- Ideal diffuse reflection: light is equally likely to be reflected in any output direction within a hemisphere oriented along the surface normal over a given point
- Think: wall paint.
- Theoretical models:
  - Micro-facet distribution
  - Subsurface reflection
Path-tracing Algorithm

- Solves the rendering equation, which was first proposed by James Kajiya in 1986.

- Generalizes ray tracing to produce accurate, unbiased images with full global illumination. Path tracing allows for effects like soft shadows, DOF, antialiasing for free.

- Potentially extremely slow on the CPU and has only become a feasible technique in recent years due to faster and faster hardware.
Path-tracing Algorithm

1. For each pixel, shoot a ray into the scene

2. For each ray, trace until the ray hits a surface. Upon hitting a surface, sample the emittance and BRDF for the surface and then send the ray in a new random direction

3. Continue bouncing each ray around until a recursion depth is reached

4. Repeat steps 1-3 over and over and continuously accumulate the result until a final image begins to converge
The random “Monte Carlo” method that path tracers use means that they can take some time to converge to a final image.
The random “Monte Carlo” method that path tracers use means that they can take some time to converge to a final image.
The random “Monte Carlo” method that path tracers use means that they can take some time to converge to a final image.
Path-tracing: GPU Motivation

• Even with a naive implementation, GPU path tracing can converge fast enough to be interactive! Contrast with CPU implementations, which can take dozens of minutes to hours to converge.

• Even more performance can be extracted through the use of spatial acceleration structures such as stack-less KD-trees or BVH.

• Single biggest constraint is memory: path tracing requires keeping everything in a scene in memory at once, which is not an issue on the CPU with 16 Gb RAM available, but can become a problem on the GPU with typically <1.5 Gb RAM available
Current Commercial GPU Path-tracers

- Brigade Render by OTOY
- Arion Render by RandomControl
- Octane Render by Refractive Software
CUDA Path-tracing Demos

• Peter and Karl’s GPU Path Tracer: https://vimeo.com/41109177

• BRIGADE Renderer: http://www.youtube.com/watch?feature=player_embedded&v=FJLy-ci-RyY
Parallel Ray-tracing: A stepping stone to path-tracing
Basic Ray-tracing Algorithm

• 1. For each pixel, shoot a ray into the scene

• 2. For each ray, trace until the ray hits a surface.

• 3. For each intersection, cast a shadow feeler ray to each light source to see if each light source is visible and shade the current pixel accordingly

• 4. If the surface is diffuse, stop. If the surface is reflective, shoot a new ray reflected across the normal from the incident ray

• 5. Repeat steps 1-4 over and over until a maximum tracing depth has been reached or until the ray hits a light or a diffuse surface
Recursive Ray-tracing

• The most obvious way to implement basic raytracing is through a purely recursive approach:

```cpp
color3 rayTrace(int depth, ray r, vector<geom> objects, vector<lights> light_sources){
    [determine closest intersected object j, intersection normal n, intersection point p]
    color = black

    if(object j is reflective){
        reflected_r = reflect_ray(r, normal, p);
        reflected_color = rayTrace(depth+1, reflected_r, objects, light_sources);
        color = reflected_color;
    }

    for each light l in light_sources{
        if shadow_ray(p, l)==true{
            light_contribution = calculate_light_contribution(p,l,n,j);
            color += light_contribution;
        }
    }

    return color;
}
```
Parallelizing Ray-tracing

- Ray-tracing is an embarrassingly parallel problem!
  - Tracing each pixel in the image is computationally independent from all other pixels
  - Tracing a single pixel is not a terribly computationally intense task, there’s simply a lot of tracing that needs to happen
- Solution: parallelize along pixels!
  - Launch one thread per pixel, trace hundreds to thousands of pixels in mass parallel!
Parallelizing Ray-tracing

Wait, we have a problem...
CUDA does not support recursion!*  
*Except on Fermi and newer
Iterative Ray-tracing

- Iterative ray-tracing: a slightly less intuitive ray-tracing algorithm that does not need recursion!
- Analogy: think breadth first search versus depth first search
- Recursive model:

  ![Diagram](image)

  Trace all ray bounces in first ray path
Iterative Ray-tracing

• Iterative ray-tracing: a slightly less intuitive ray-tracing algorithm that does not need recursion!

• Analogy: think breadth first search versus depth first search

• Recursive model:

  ![Diagram](image)

  Trace all ray bounces in second ray path
Iterative Ray-tracing

- Iterative ray-tracing: a slightly less intuitive ray-tracing algorithm that does not need recursion!
- Analogy: think breadth first search versus depth first search
- Recursive model:

```
   ├── ➔ ├── ➔ ├── ➔ └── ➔
   │    │    │    │    │
   │    │    │    │    │
   │    │    │    │    │
   │    │    │    │    └──➢
TRACE ALL RAY BOUNCES IN THIRD RAY PATH
```

etc.
Iterative Ray-tracing

• Iterative ray-tracing: a slightly less intuitive ray-tracing algorithm that does not need recursion!
• Analogy: think breadth first search versus depth first search
• Iterative model:

1. Trace first ray
2. Bounce in all ray paths
Iterative Ray-tracing

- Iterative ray-tracing: a slightly less intuitive ray-tracing algorithm that does not need recursion!
- Analogy: think breadth first search versus depth first search
- Iterative model:

  > > Trace second ray bounce in all ray paths
Iterative Ray-tracing

• Iterative ray-tracing: a slightly less intuitive ray-tracing algorithm that does not need recursion!
• Analogy: think breadth first search versus depth first search
• Iterative model:

Trace third ray bounce in all ray paths, and so on and so forth
Iterative Ray-tracing

- Implement ray-tracing as a while or for loop and cache the current ray for use in the next iteration of the loop:

```cpp
color3 rayTrace(int depth, ray r, vector<geom> objects, vector<lights> light_sources){
    ray currentRay = r;
    color = black;
    for(int i=0; i<depth; i++){
        // determine closest intersected object j, intersection normal n, intersection point p
        if(object j is reflective){
            reflected_r = reflect_ray(r, normal, p);
        }
        for each light l in light_sources{
            // if shadow ray(p, l)==true{
            if shadow_ray(p, l)==true{
                color += color * calculate_light_contribution(p,l,n,j);
            }
        }
    return color;
}
```
Parallelizing Ray-tracing

So we can just implement that entire interative ray-tracing algorithm in a CUDA kernal and we’re done, right?
Parallelizing Ray-tracing

So we can just implement that entire interative ray-tracing algorithm in a CUDA kernal and we’re done, right?

Well yes, BUT...
Parallel Ray-Tracing Quirks: Wasted Cycles

• How many bounces does each ray path make before terminating?
Parallel Ray-Tracing Quirks: Wasted Cycles

- How many bounces does each ray path make before terminating?

4 bounces?
Parallel Ray-Tracing Quirks: Wasted Cycles

• How many bounces does each ray path make before terminating?

3 bounces?
Parallel Ray-Tracing Quirks: Wasted Cycles

• How many bounces does each ray path make before terminating?

2 bounces?
Parallel Ray-Tracing Quirks: Wasted Cycles

• How many bounces does each ray path make before terminating?

1 bounce?
Parallel Ray-Tracing Quirks: Wasted Cycles

• How many bounces does each ray path make before terminating?

No bounces?
Parallel Ray-Tracing Quirks: Wasted Cycles

• How many bounces does each ray path make before terminating?
• We have no idea how many bounces each ray path may take!
• What does this uncertainty imply about parallelizing by pixels?
Parallel Ray-Tracing Quirks: Wasted Cycles

• Remember, in CUDA, we can only launch a finite number of blocks at a time, and must wait for blocks to complete before launching more.
• If some threads need to trace more bounces than others, then potentially a large number of threads will spend the majority of the time idling.
• Conclusion: parallelizing by pixels is one possible approach, but ultimately a naive one.

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
<th>Thread 4</th>
<th>Thread 5</th>
<th>Thread 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bounce 1</td>
<td>Bounce 1</td>
<td>DONE</td>
<td>Bounce 1</td>
<td>Bounce 1</td>
<td>Bounce 1</td>
</tr>
<tr>
<td>Bounce 2</td>
<td>DONE</td>
<td>DONE</td>
<td>Bounce 2</td>
<td>DONE</td>
<td>DONE</td>
</tr>
<tr>
<td>Bounce 3</td>
<td></td>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bounce 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>WASTED CYCLES</td>
</tr>
<tr>
<td>DONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Ray Parallelization

- Solution: parallelize by rays, not pixels!
- Instead of a single kernel launch that traces an entire ray path, do multiple kernel launches that trace individual bounces!
  1. Construct pool of rays that need to be intersection tested
  2. Construct grid of colors and unaccumulated colors
  3. Launch a kernel that traces ONE bounce and records the next ray into the ray pool
  4. Remove terminated rays from the ray pool through string compaction type process
  5. Repeat
Ray Parallelization

• With each iteration of the ray-trace, we need less threads as rays terminate! As a result, each iteration requires fewer blocks, meaning each iteration executes faster than the previous iteration.

Iteration 1: 10 blocks executing in groups of 4 = 3 batches

Iteration 2: 4 blocks executing in groups of 4 = 1 batch

• Well, there are a few rare edge cases where this approach does not provide a performance boost. Can you think of any?
Ray Parallelization: Super Simple Example

First Kernel Launch

Ray Pool:
Ray 1, Ray 2, Ray 3

Threads Needed:
3

Result:

Terminated Rays:
Ray 1
Ray Parallelization: Super Simple Example

Second Kernel Launch

Ray Pool:
Ray 2, Ray 3

Threads Needed:
2

Result:
Terminated Rays:
Ray 1
Ray Parallelization: Super Simple Example

Third Kernel Launch

Ray Pool: 
Ray 2, Ray 3

Threads Needed: 
2

Result:
Terminated Rays: 
Ray 1, Ray 3
Ray Parallelization: Super Simple Example

Fourth Kernel Launch

Ray Pool:
Ray 2

Threads Needed:
1

Result:

Terminated Rays:
Ray 1, Ray 3, Ray 2
Parallel Ray-Tracing Quirks: Memory Management

- Assume we cudaMemcpy() all of our geometry and materials and other scene assets from host memory to device global memory.
- What happens in this scene on the first bounce?
Parallel Ray-Tracing Quirks: Memory Management

• Assume we cudaMemcpy() all of our geometry and materials and other scene assets from host memory to device global memory.

• What happens in this scene on the first bounce?

• A lot of rays are hitting the same objects, meaning a lot of threads are concurrently trying to access the same places in global memory!
Parallel Ray-Tracing Quirks: Memory Management

- Possible Solutions:
  - In distributed raytracing scenarios: since the first bounce will always involve the same raycasts from the camera, cache the result of the first bounce and recycle the result.
    - “First bounce cache, second bounce thrash”
  - If the scene is sufficiently small, cache geometry data in shared memory.
    - Why might this be a bad idea in some cases?
References


• Sam Lapere’s “Ray Tracey’s Blog”: http://raytracey.blogspot.de/


• Stanford University's CS348B: Image Synthesis course materials: https://graphics.stanford.edu/wikis/cs348b-12