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

CIS565 Fall 2012
University of Pennsylvania
by Yining Karl Li

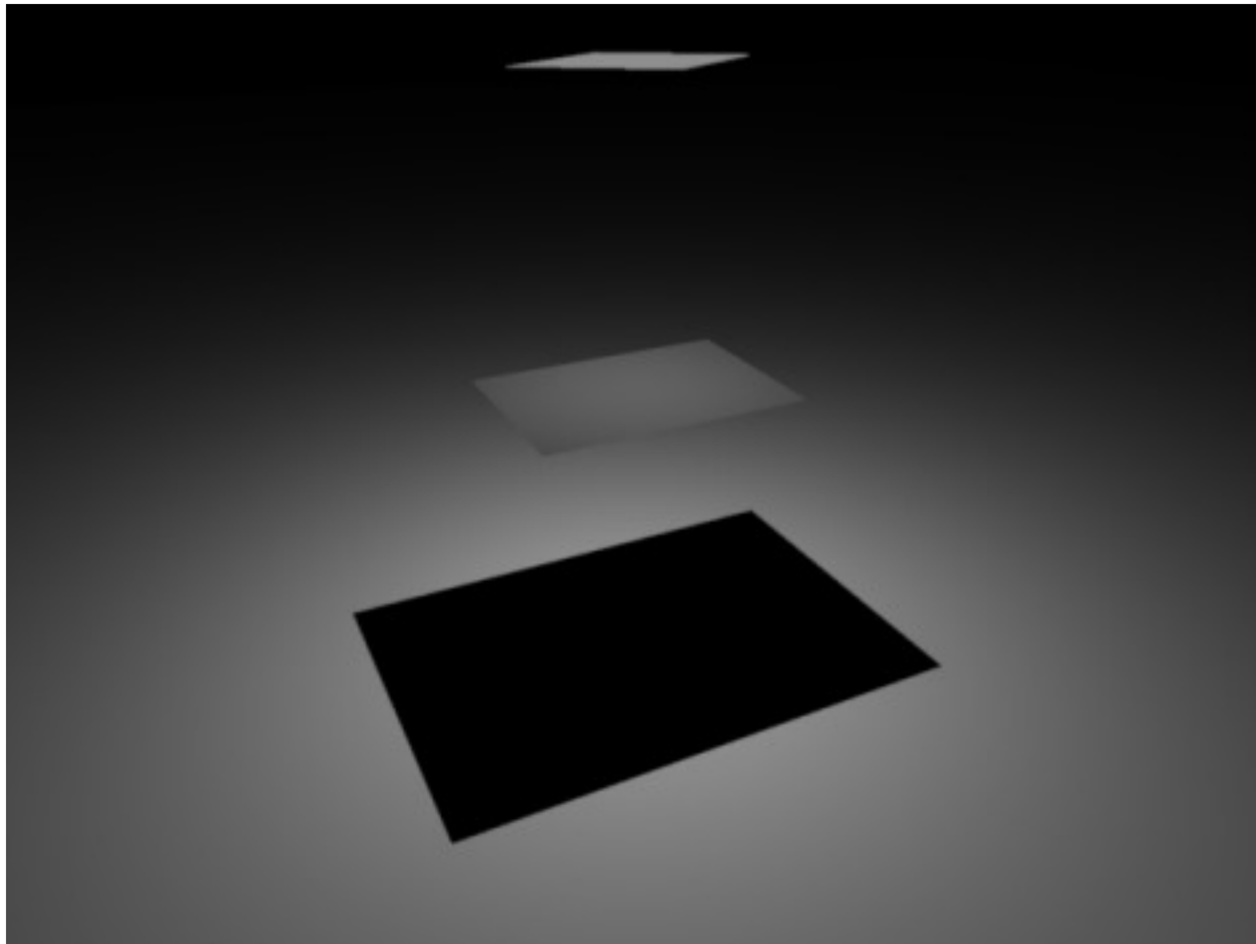
Agenda

- Part 1 (Last Monday):
  - 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 (Today):
  - 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

“1984” Distributed Ray-tracing Sample Render
Lucasfilm/Pixar 1984

Distributed Ray-tracing

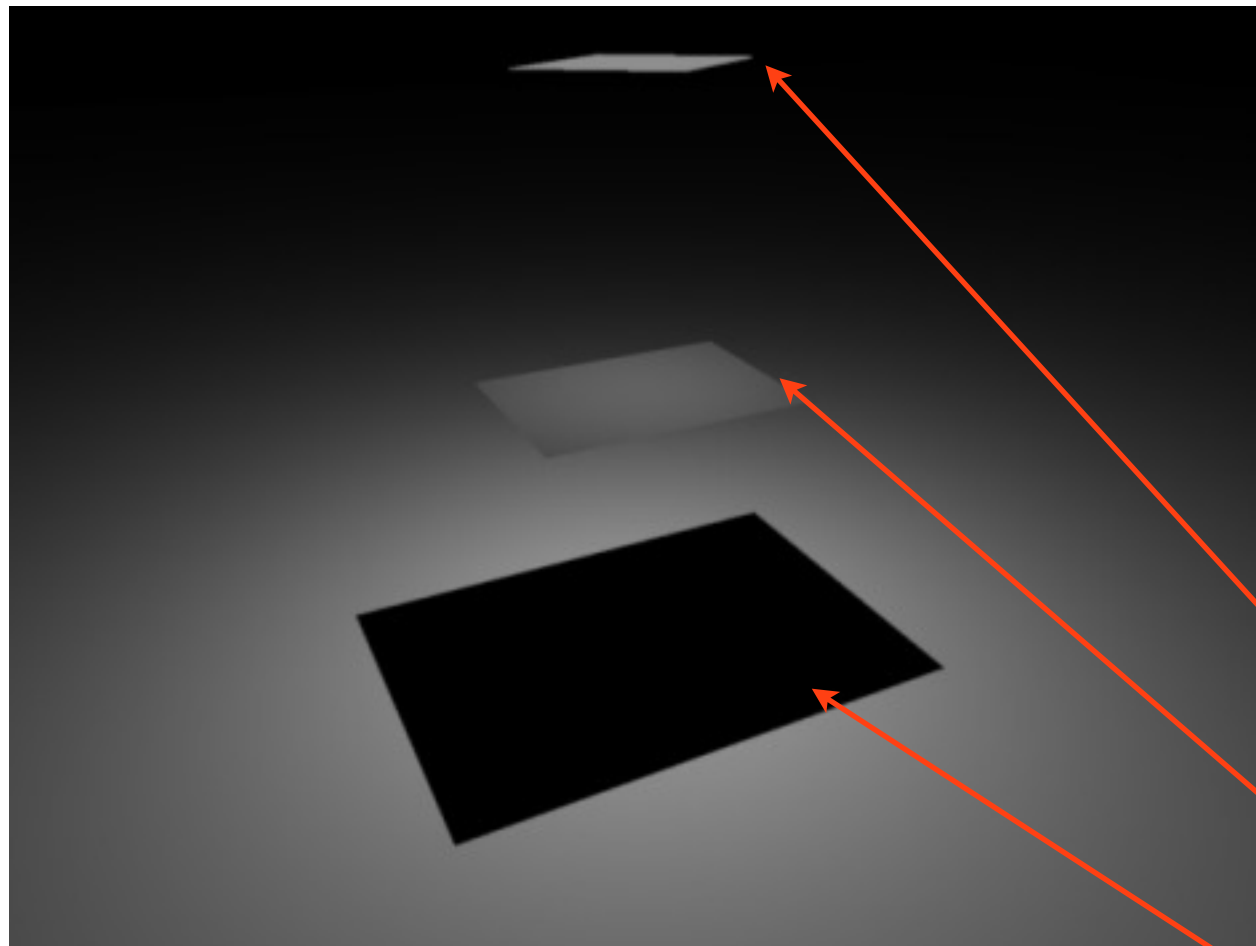
What is distributed ray-tracing?



What is wrong with this picture?

*The sample images from slides 4 to 6 are borrowed from Pat Hanrahan's course CS348: Image Synthesis at Stanford University

What is distributed ray-tracing?



What is wrong with this picture?

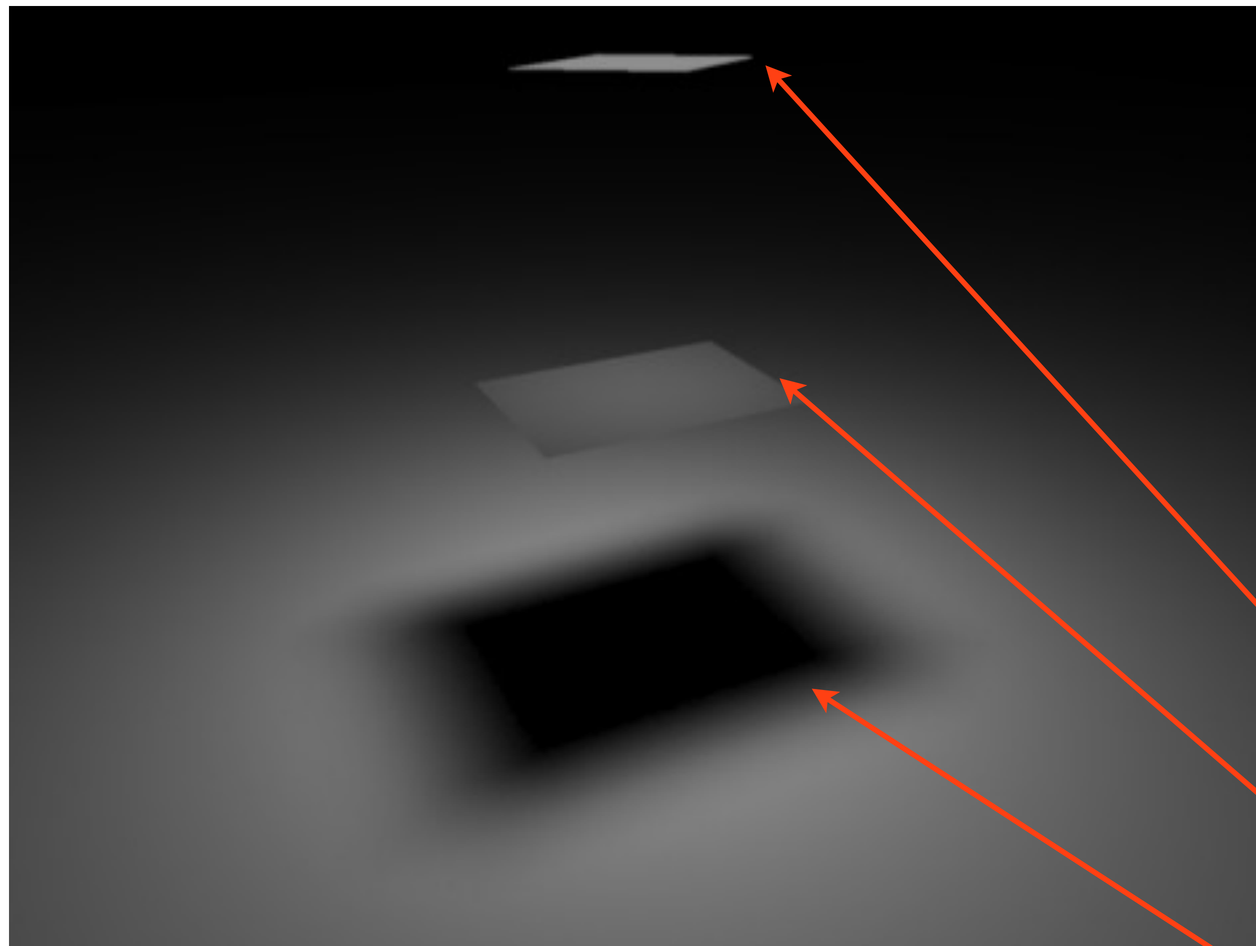
There are no soft shadows, even though we have an area light!

Area light source

Occluder

Hard shadow????

What is distributed ray-tracing?



Instead, we expect the correct solution to look something more like this image.

So how do we accomplish this effect?

Area light source

Occluder

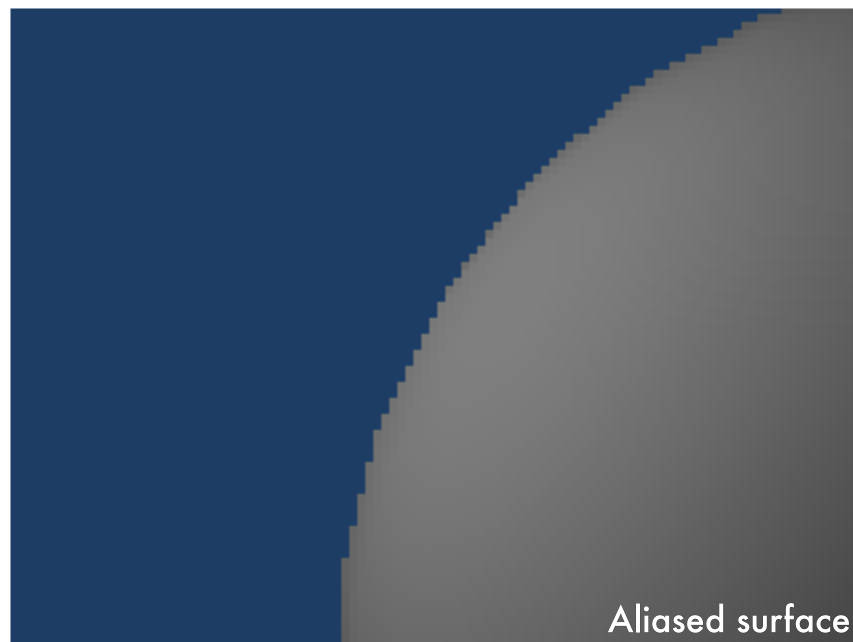
Nice soft shadow

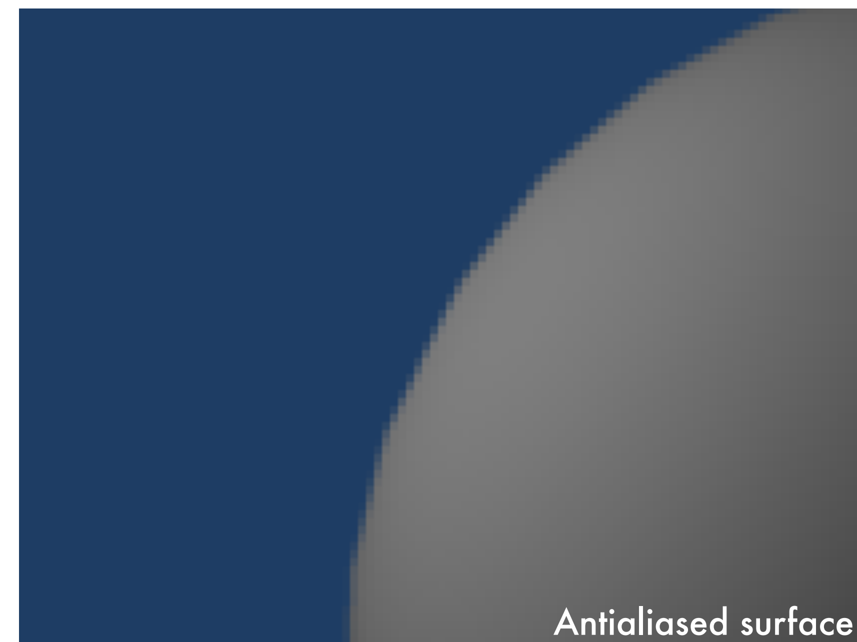
What is distributed ray-tracing?

- Distributed ray-tracing is a ray-tracing method based on the idea of using randomly distributed oversampling to produce effects that require integrating over some property
- Standard ray-tracing sends one ray per pixel and traces that ray through the scene; in distributed ray-tracing, we send *multiple* rays into the scene from a pixel and jitter our multiple rays over some property
- Use-cases:
  - Antialiasing
  - Soft shadows
  - Motion blur
  - Depth of field
  - Glossy non-perfect specular reflections
  - Etc.

Super-simple distributed ray-tracing application: super-sampled antialiasing

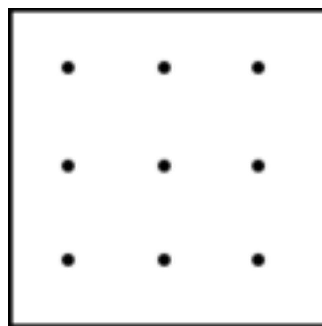
- Aliasing arises from the fact that objects with smooth curves can only be sampled at a discrete number of points
- Solution: for each pixel, sample object surfaces at a number of close, but separate points and average the result
- How do we choose to distribute our samples?
  - Two methods: we can either use a precomputed, fixed sampling pattern, or we can generate random samples on the fly and average the result

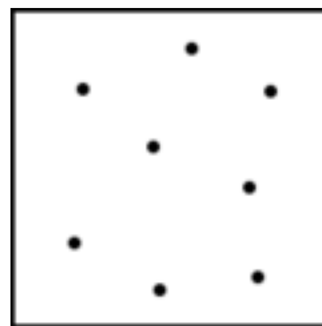

Aliased surface

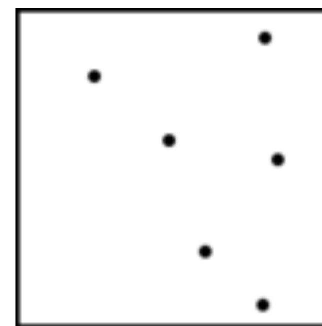

Antialiased surface

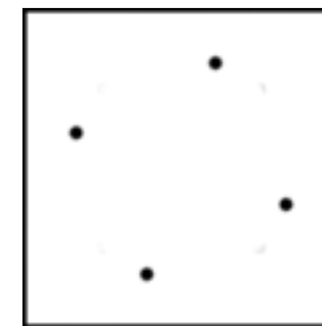
Fixed sampling patterns

- In the antialiasing example, we could simply use a fixed pattern of points within a pixel to shoot rays through and then average the result:


Fixed uniform grid


Poisson Disc

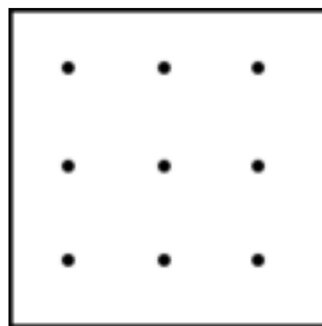

Random

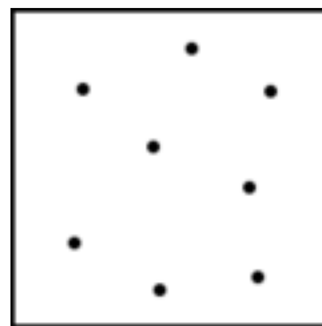

Rotated uniform grid

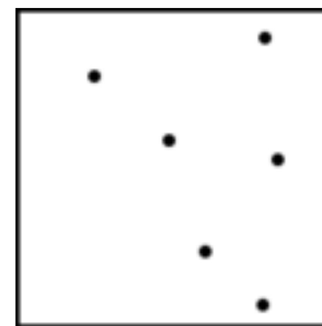
- Advantages: Saves computation time, since we don't need to generate sample points on the fly and can just use pre-cached points
- Disadvantages: Fixed sampling patterns can lead to artifacts and Moire-type banding problems

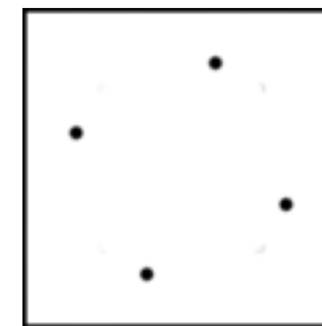
Fixed sampling patterns

- In the antialiasing example, we could simply use a fixed pattern of points within a pixel to shoot rays through and then average the result:


Fixed uniform grid


Poisson Disc


Random


Rotated uniform grid

- Advantages: ~~Saves computation time~~, since we don't need to generate sample points on the fly and can just use pre-cached points
- Disadvantages: Fixed sampling patterns can lead to artifacts and Moire-type banding problems
- Who cares if we need more compute? We're using the GPU, we have plenty of compute! We also don't like caching when we don't have to!

Extending supersampling to other phenomenon...

- If we think about it, supersampled antialiasing is simply *integrating over all possible points on a smooth curved surface* in order to produce a result that closer approximates reality.
- What if we integrate over something else?

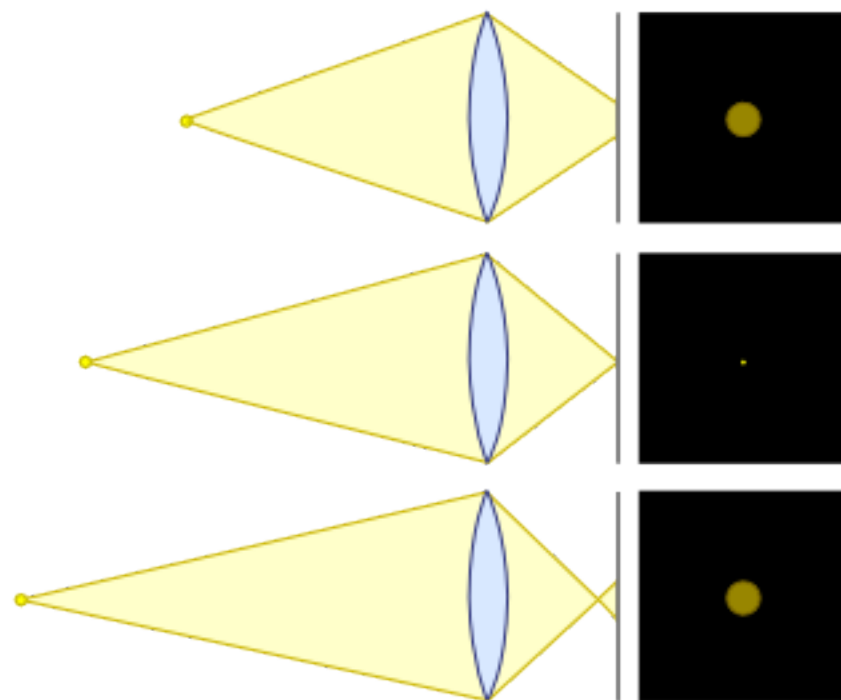
Motion Blur: Integrate over *time*

- Imagine if we jitter our rays in time instead of within a pixel. That is, each ray samples a frame of our scene that is slightly offset in time...

Photorealizer
Peter Kutz 2012

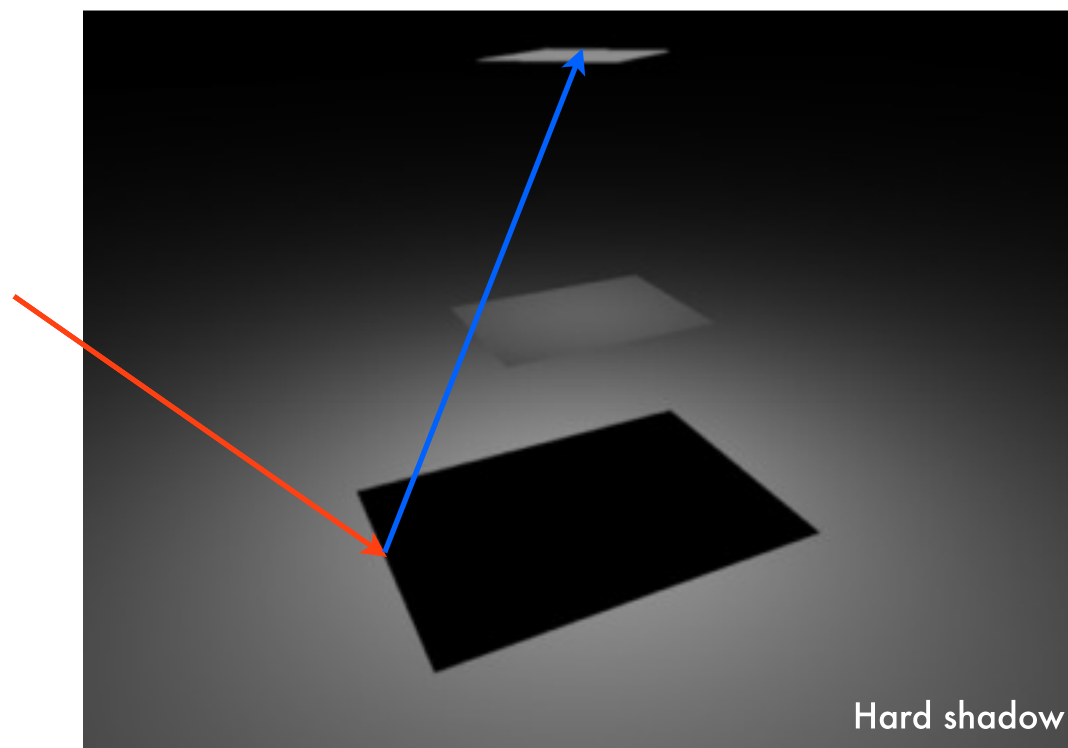
Depth of Field: Integrate over *a lens*

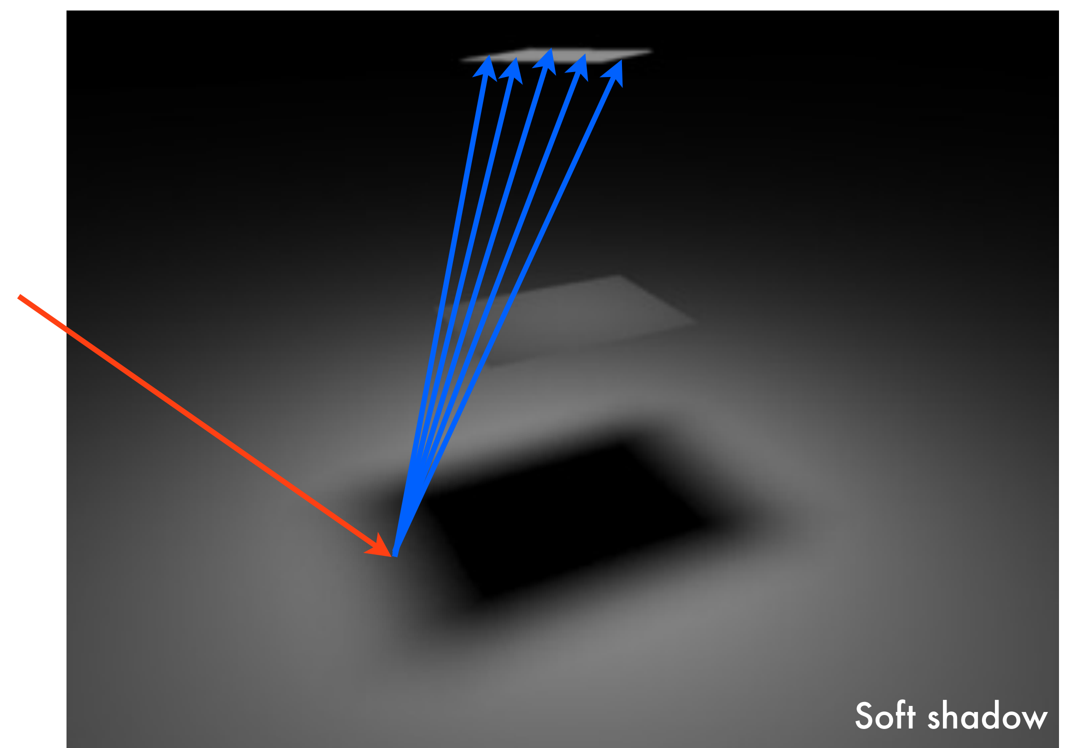
- The standard CIS460/560 style single-ray ray-tracer's camera behaves as a pinhole camera: all rays originate from a single point
- Real cameras have rays that begin at slightly different points and then pass through a lens which refracts rays differently based on where they transmit through the lens
- One possible approach: Jitter the camera's position, but leave the image plane stationary.



Soft Shadows: Integrate over a *light source*

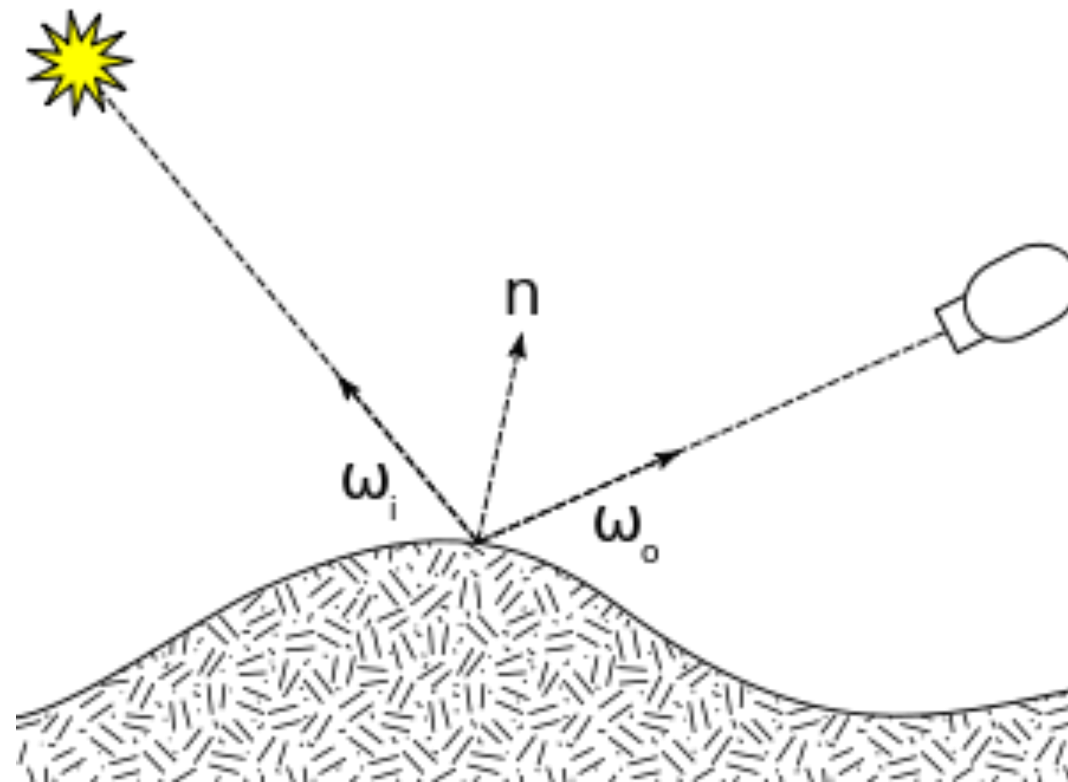
- Think of an area light as a infinite number of point lights spread out of the surface of a piece of geometry, casting an infinite number of slightly varied shadows.
- In a distributed ray-tracer, for each shadow feeler, select a random point on the surface of the area light, and treat that point as a point light. Repeat this process over and over and average the result!



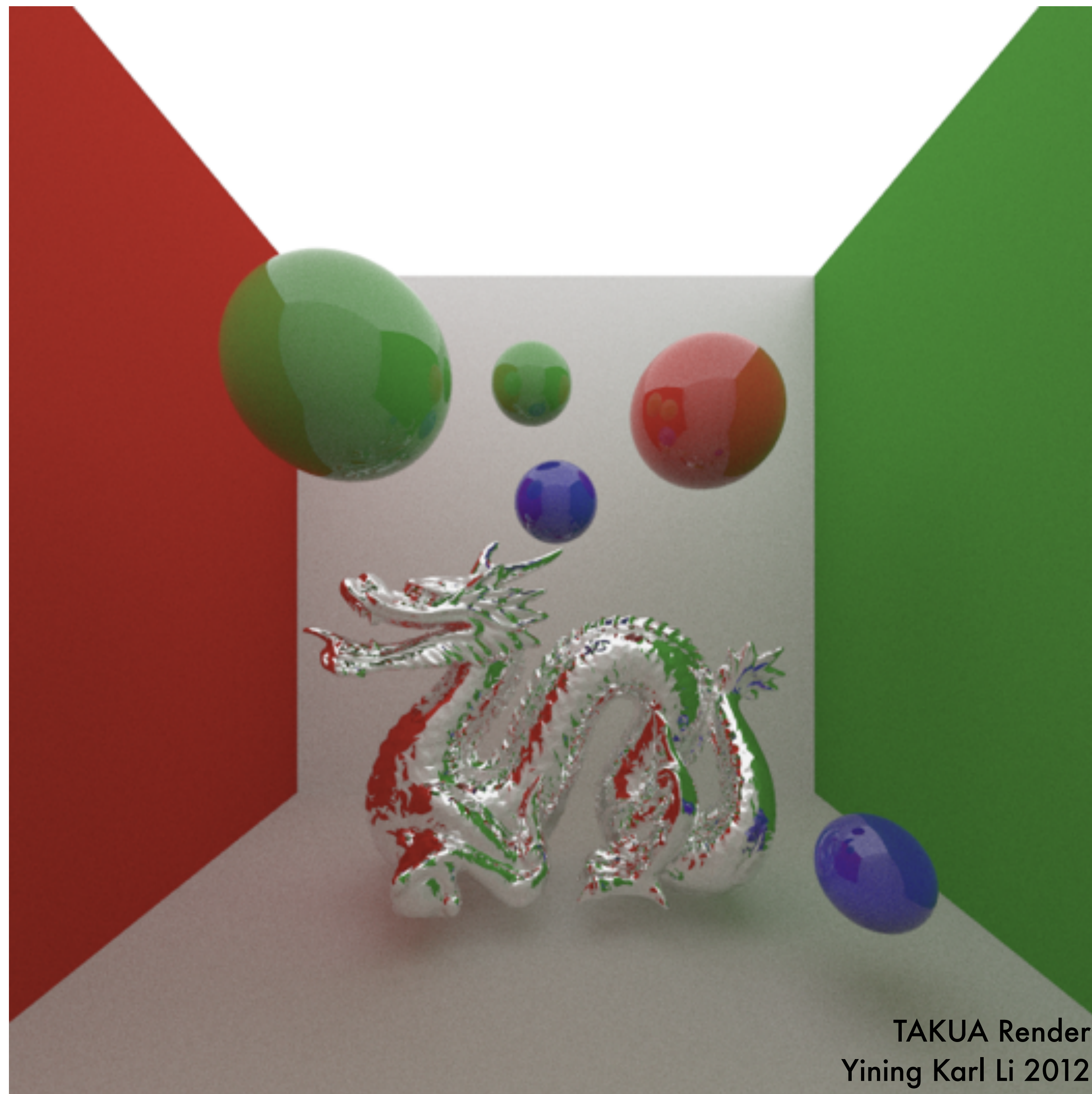


Shading: Integrating over *BRDFs*

- At it's most basic level, a BRDF simply defines how a ray will leave a surface given how it intersected with the surface and the normal of the intersection
- A perfectly reflective BRDF is easy to implement: each incoming ray will always have the same outgoing ray for every sample

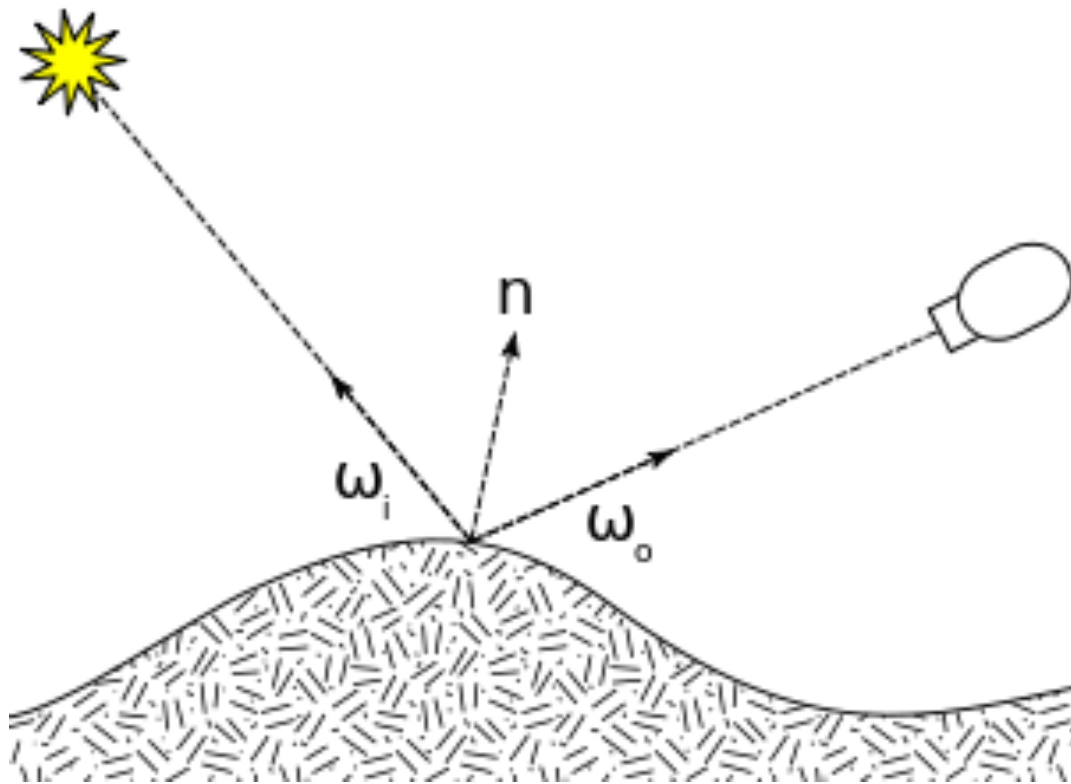


Shading: Integrating over *BRDFs*



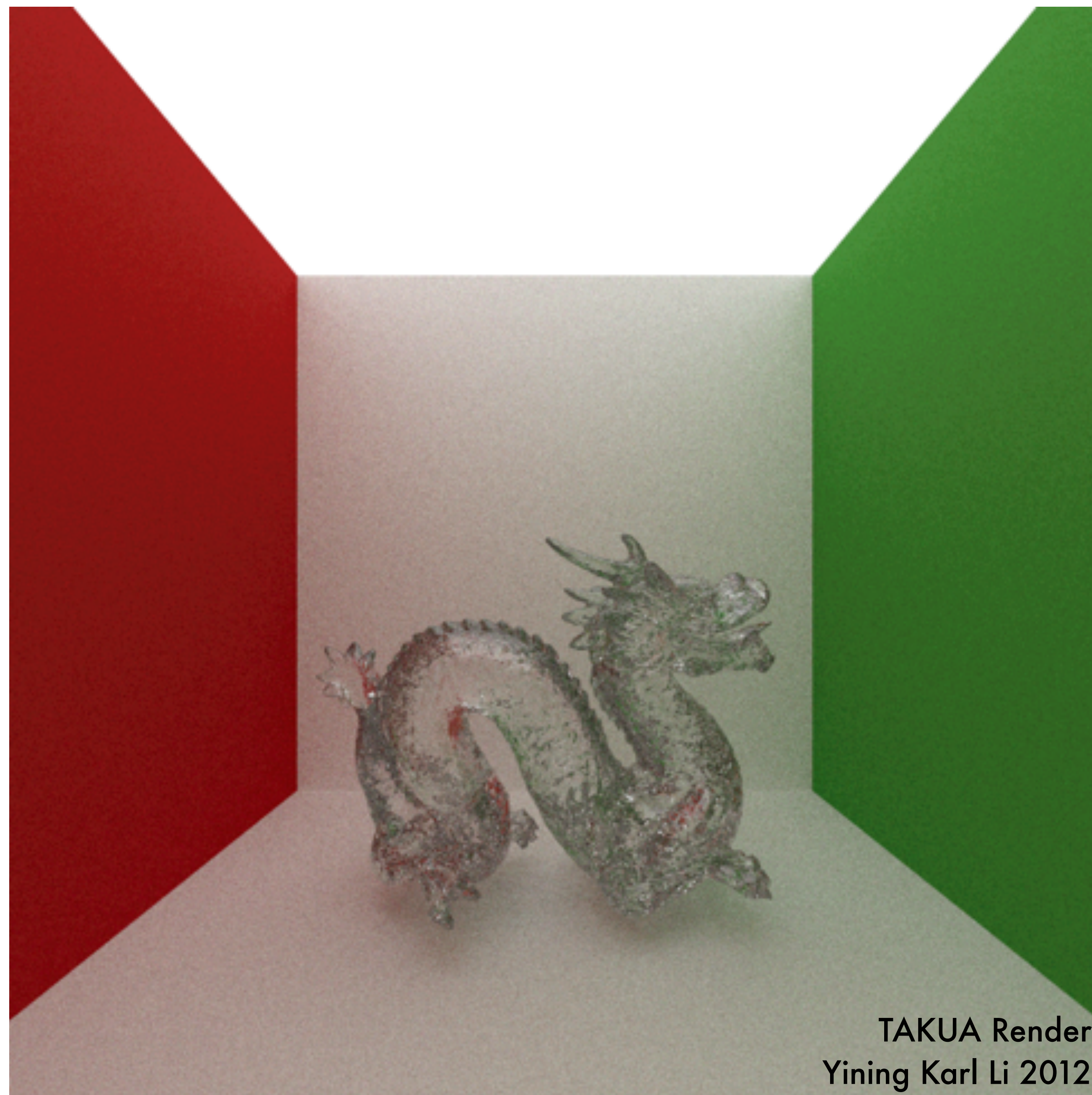
Shading: Integrating over *BRDFs*

- At it's most basic level, a BRDF simply defines how a ray will leave a surface given how it intersected with the surface and the normal of the intersection
- A perfectly reflective BRDF is easy to implement: each incoming ray will always have the same outgoing ray for every sample



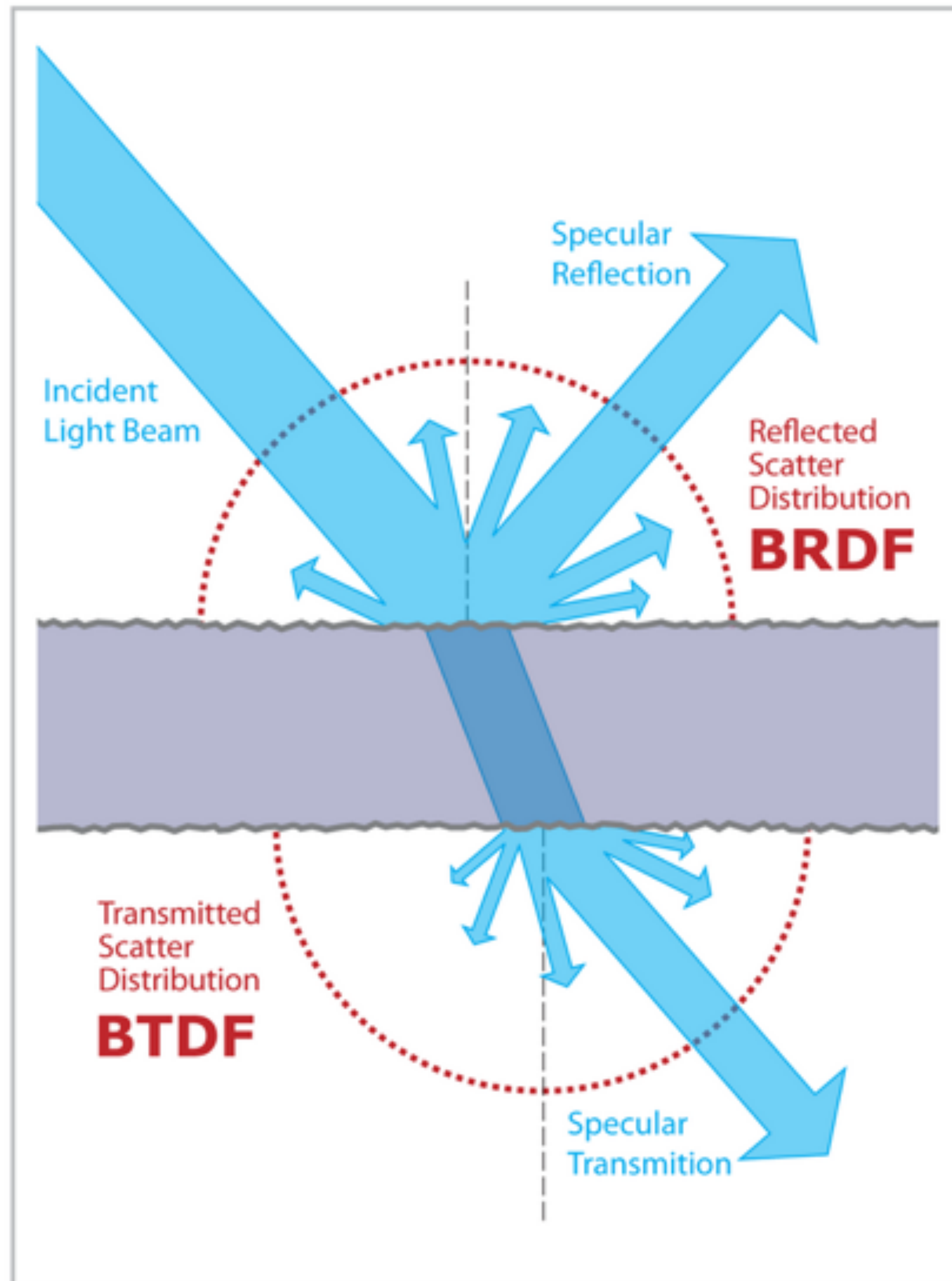
What if we have a BRDF that across samples doesn't return the same result given a fixed input ray?

Shading: Integrating over *BRDFs*



- Example case: glass
- Real glass both *refracts* and *reflects!*
- We need to consider two different possible ray interactions with the surface: the *BRDF* defining reflection, and the *BTDF (Bidirectional Transmission Distribution Function)* defining refraction and transmission
- BRDF + BTDF = BSDF: Bidirectional Scattering Distribution Function

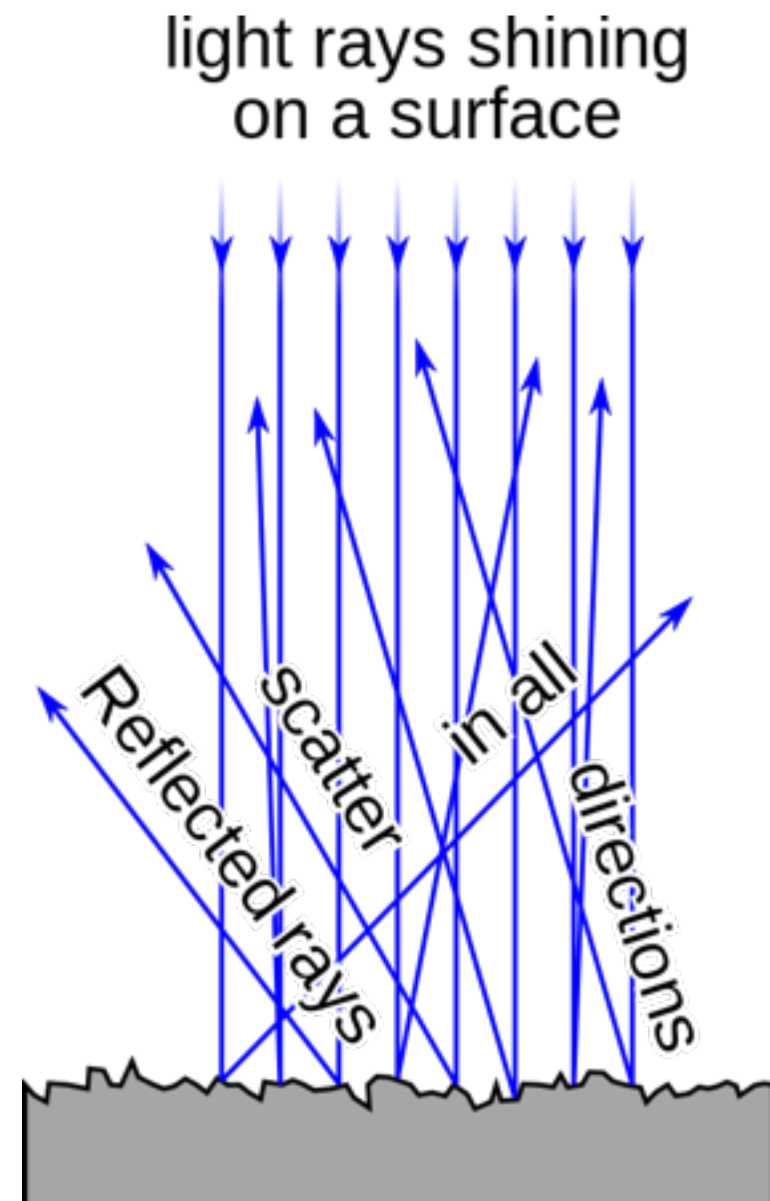
Shading: Integrating over *BRDFs*



- To implement a BSDF, we send *multiple rays* from a pixel. When each ray hits a glass surface, we randomly choose whether that ray reflects or refracts.
- We accumulate the result of multiple random reflection/refraction samples to get the final half reflected, half refracted result
- We can change how reflective or refractive the glass is by simply adjusting the probability distribution for when we choose what to do with each ray

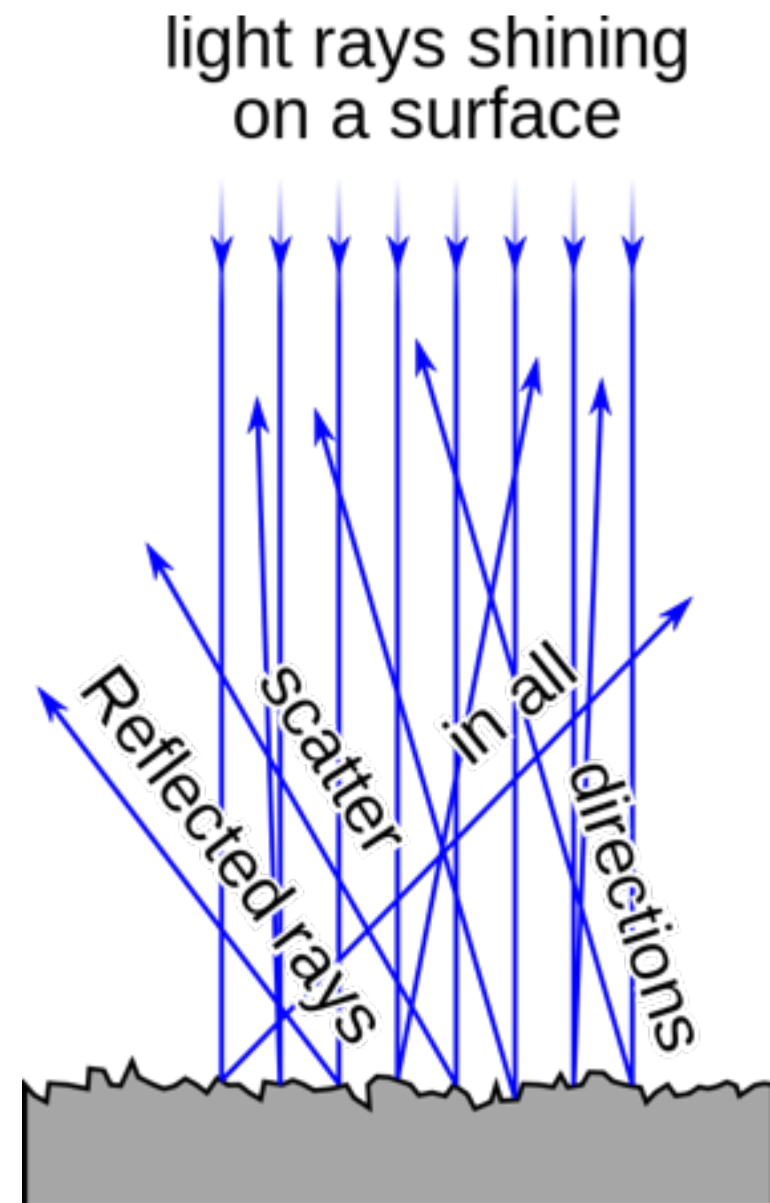
Shading: Integrating over *BRDFs*

- Interesting oddball case: diffuse surfaces!
- In real life, diffuse surfaces are actually reflective- they just reflect in all directions, completely randomly!
- We can simulate this using distributed ray-tracing: send multiple rays, and for each ray, choose a totally random direction to send it off in!



Shading: Integrating over *BRDFs*

- Interesting oddball case: diffuse surfaces!
- In real life, diffuse surfaces are actually reflective- they just reflect in all directions, completely randomly!
- We can simulate this using distributed ray-tracing: send multiple rays, and for each ray, choose a totally random direction to send it off in!
- What if we change the probability distribution from totally random to something that favors a certain direction?



Shading: Integrating over *BRDFs*

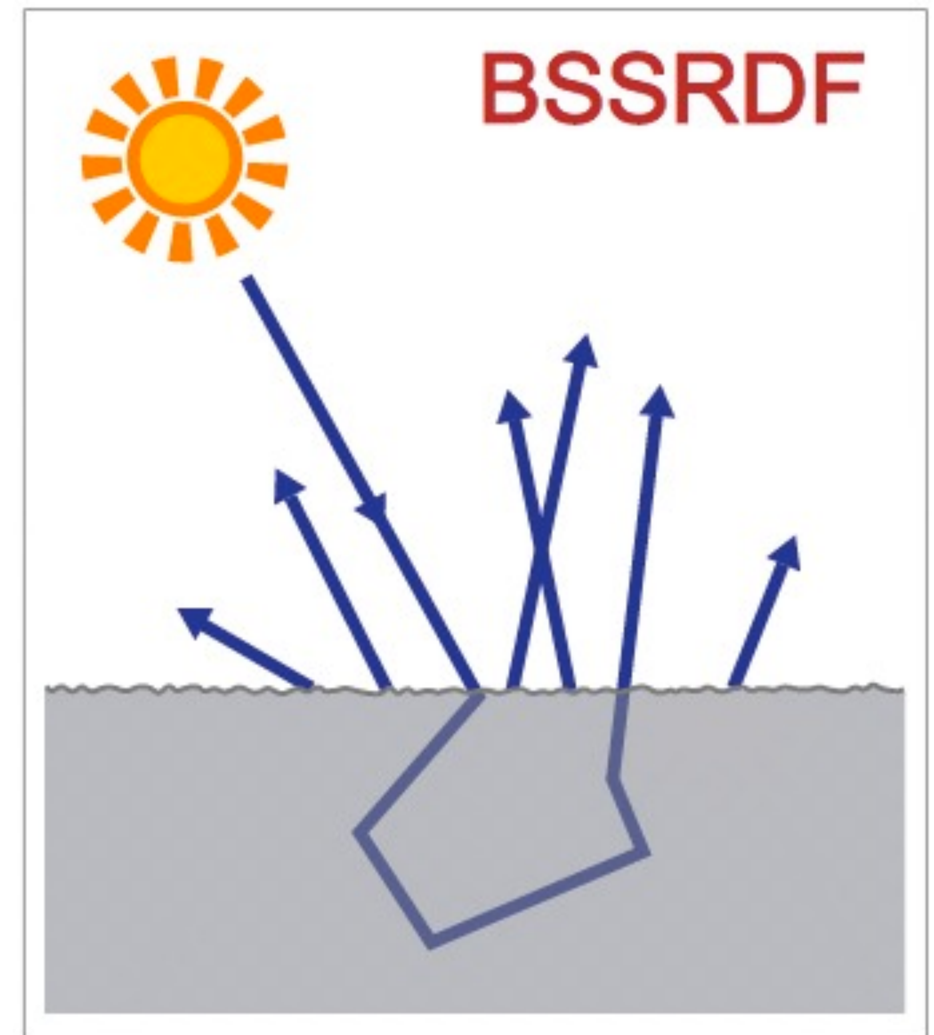
- Combining diffuse and reflective: glossy case
- If we modify the diffuse case to use a probability distribution biased in some direction, we can generate true glossy surfaces.

Shading: Subsurface Scattering

- Subsurface scattering describes how light interacts with translucent objects; light enters a surface, scatters multiple times randomly within the surface, and exits at a different point from which it entered
- Examples of subsurface scattering materials are milk, skin, marble, wax, etc.

Shading: Subsurface Scattering

- Brute-force scattering is implemented by choosing a random scatter direction and distance for each ray that has entered a surface; each time a ray reaches the end of its scatter distance, we pick a new distance and direction until the ray exits the surface
- Brute-force scattering can be extremely computationally expensive, as a LOT of distributed ray-tracing is required



Distributed ray-tracing and Parallelizing

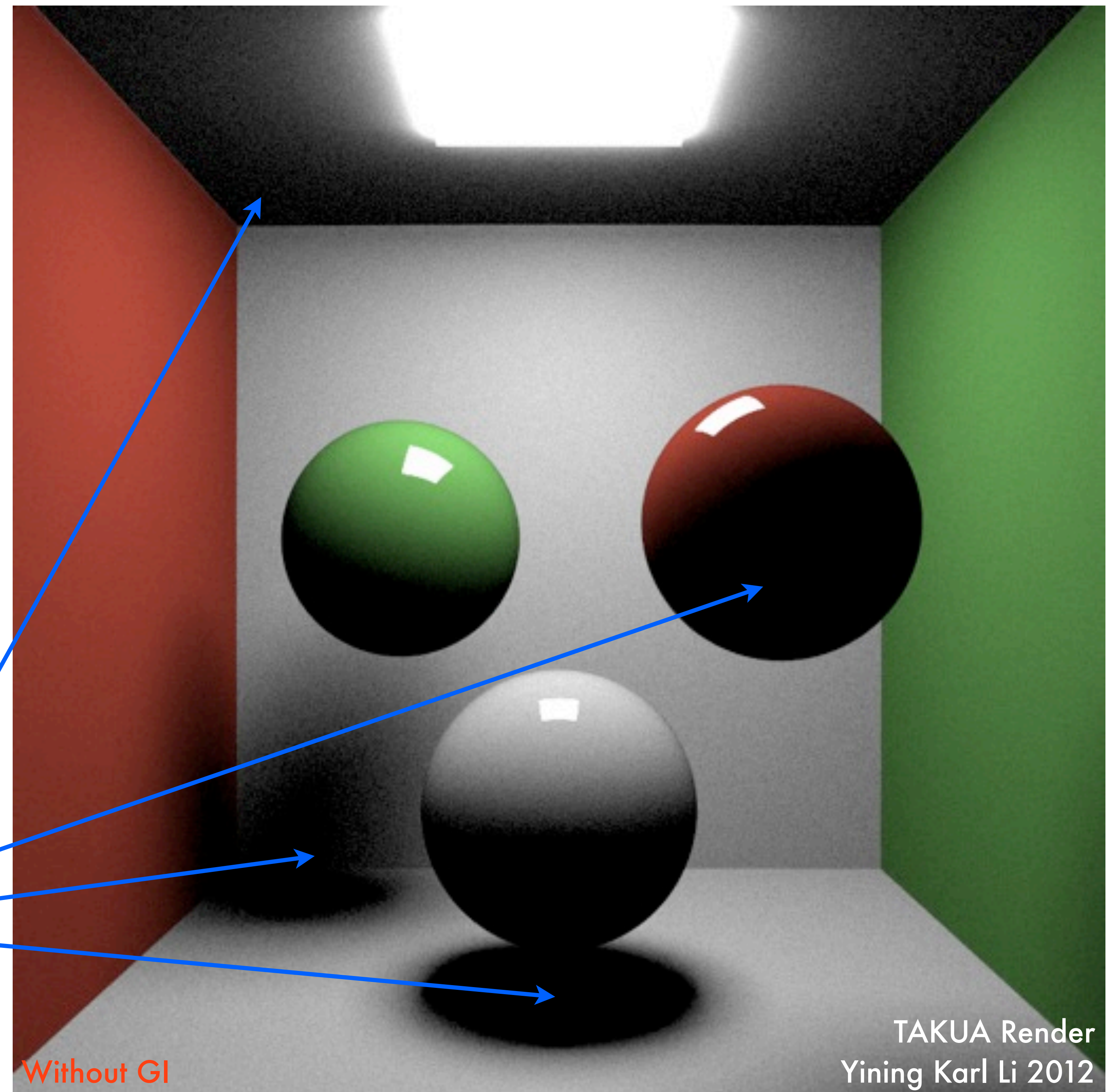
- Fundamentally, distributed ray-tracing simply means casting multiple rays to evaluate properties that cannot be correctly represented with a single ray path
- As we have seen from the previous examples, the runtime for distributed ray-tracing effects can be considerably non-deterministic!
  - BSDF evaluations can be especially difficult to predict runtimes for, even more so when subsurface scattering is involved
- Parallelizing by ray paths/pixels can get extremely inefficient when complex distributed ray-tracing is involved. Parallelizing by ray can save us a lot of wasted compute!

Global Illumination and Path-tracing

What is Global Illumination?

- Global illumination refers to the complete lighting solution in a scene that includes both the *direct illumination* from light sources and the *indirect illumination* from light bouncing off of non-emitting surfaces

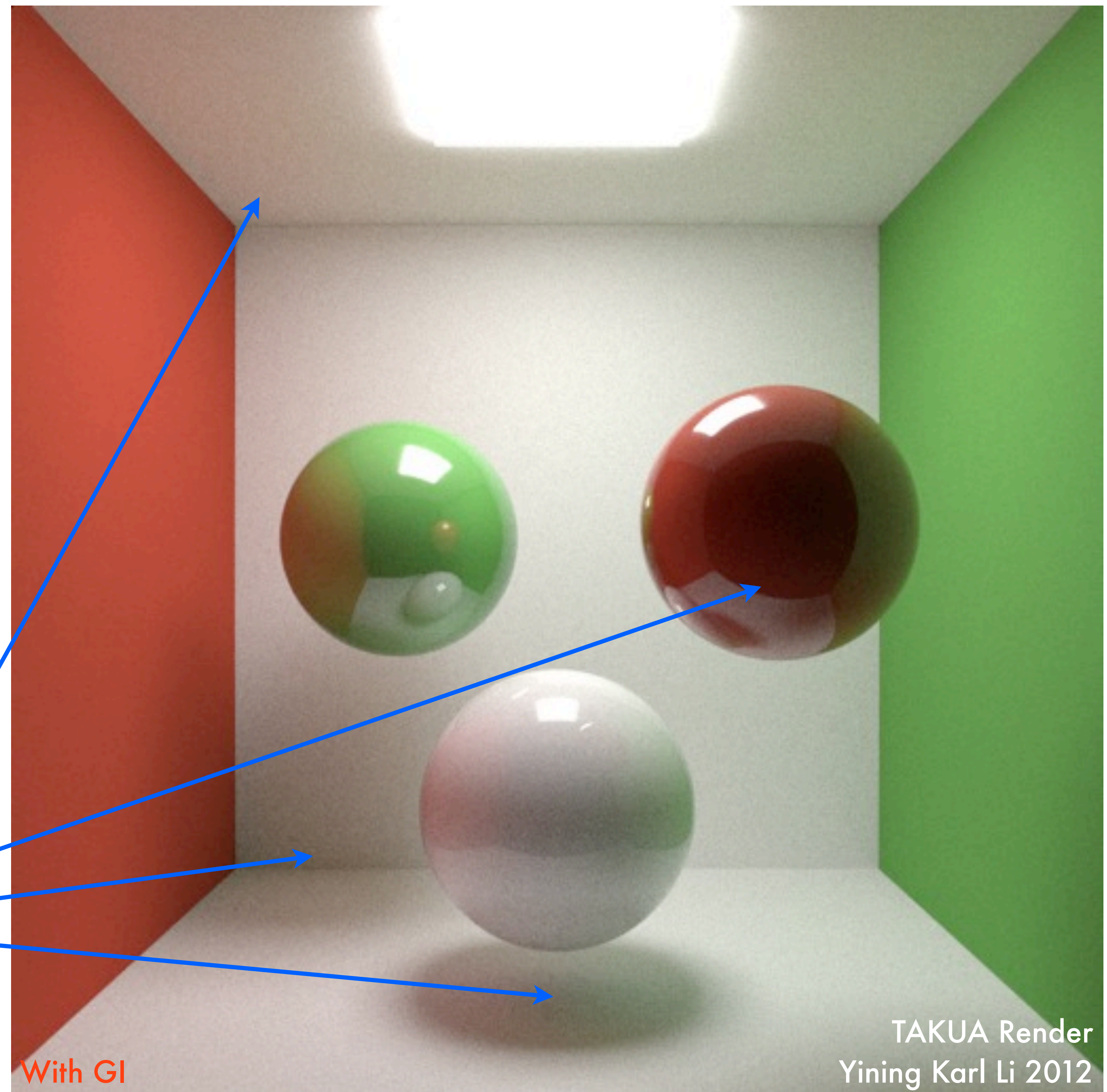
Dark shadows where surfaces are not in direct view of light sources



What is Global Illumination?

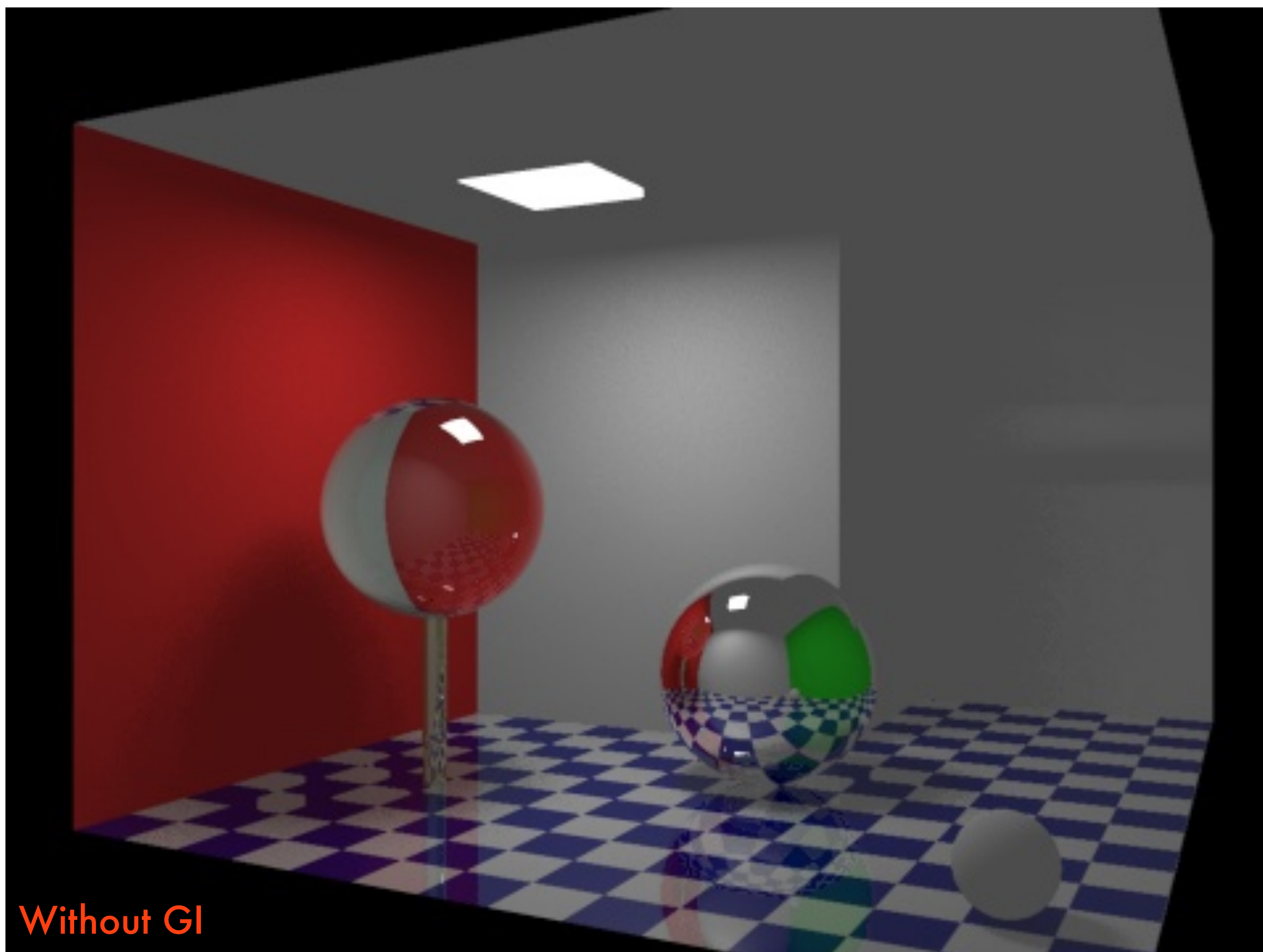
- Global illumination refers to the complete lighting solution in a scene that includes both the *direct illumination* from light sources and the *indirect illumination* from light bouncing off of non-emitting surfaces

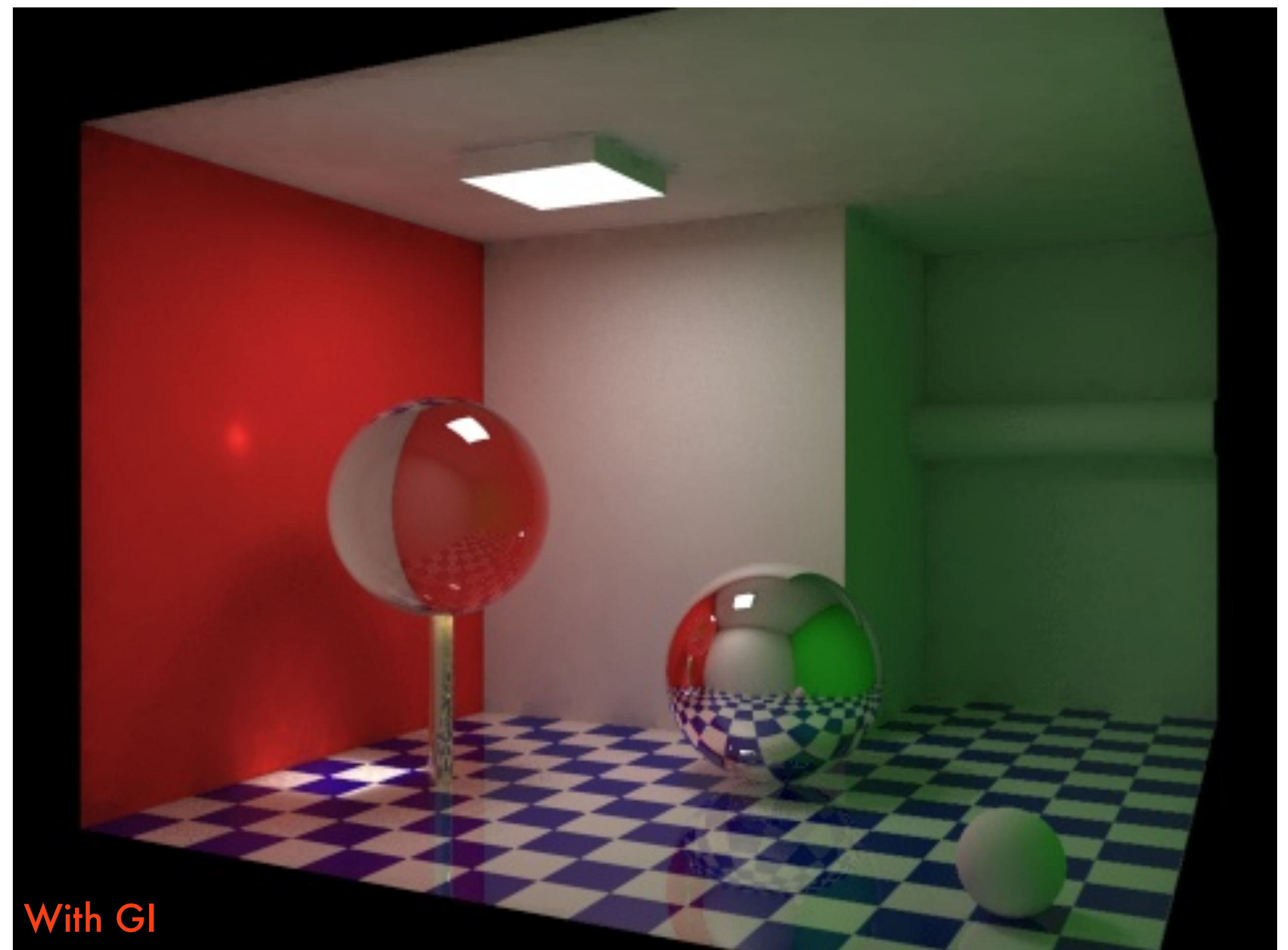
Areas not in direct view of the light source are lit by *indirect light* bouncing off of surfaces



What is Global Illumination?

- Global illumination effects include diffuse color bleeding, reflective and refractive caustics, volumetric scattering, etc.
- Basically, GI encompasses all of the lighting effects that are NOT JUST results of direct lighting





What is Global Illumination?

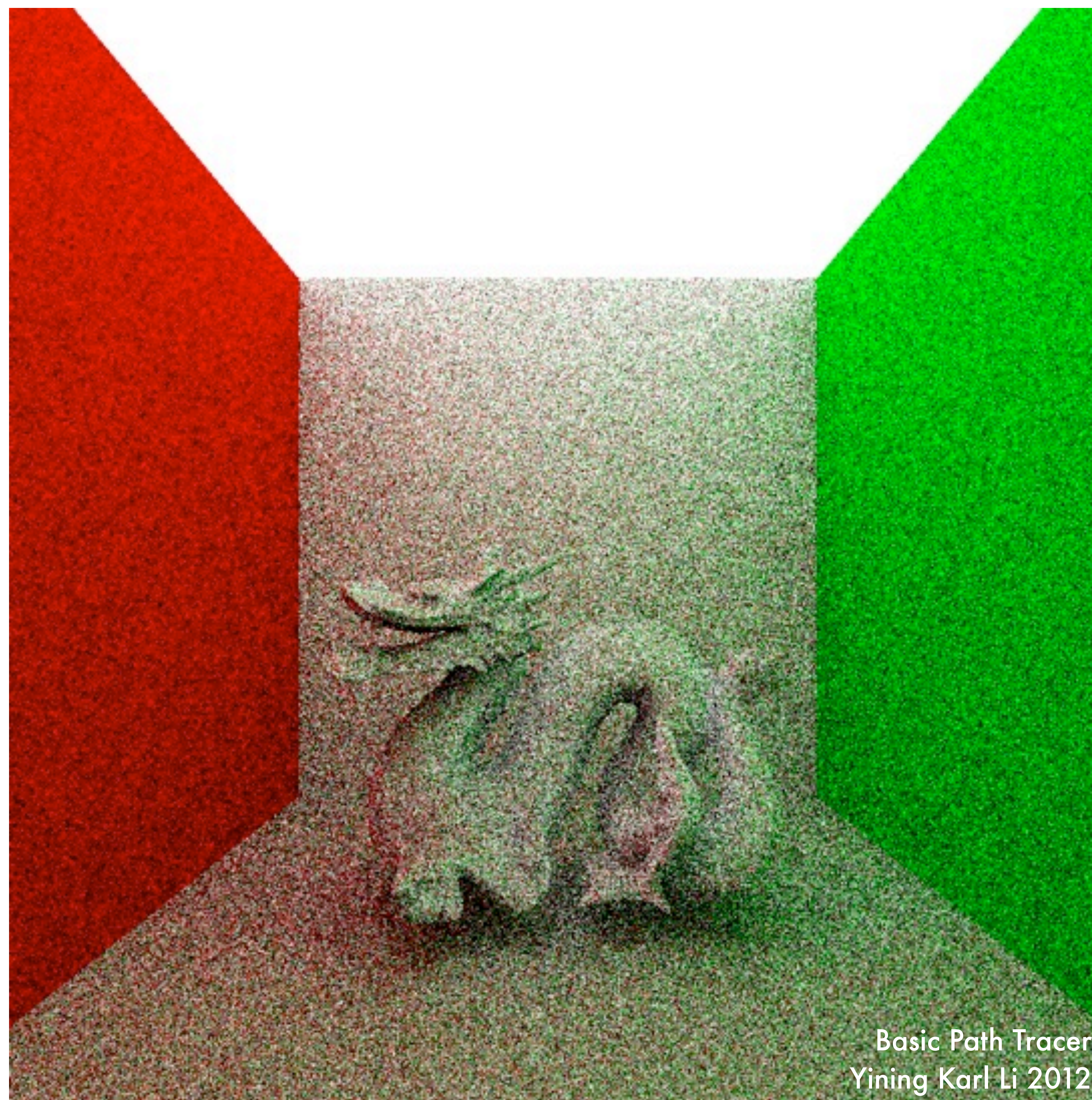
We now have almost all of the pieces we need to implement GI. We just need one more thing...

Monte Carlo Integration

- In order to generate an image with global illumination, we need to solve the rendering equation (from last week). Unfortunately, the rendering equation is somewhere between extremely difficult to impossible to solve analytically
- GI requires integrating across all incoming light from every possible direction. How can we “gather” light from every possible direction?

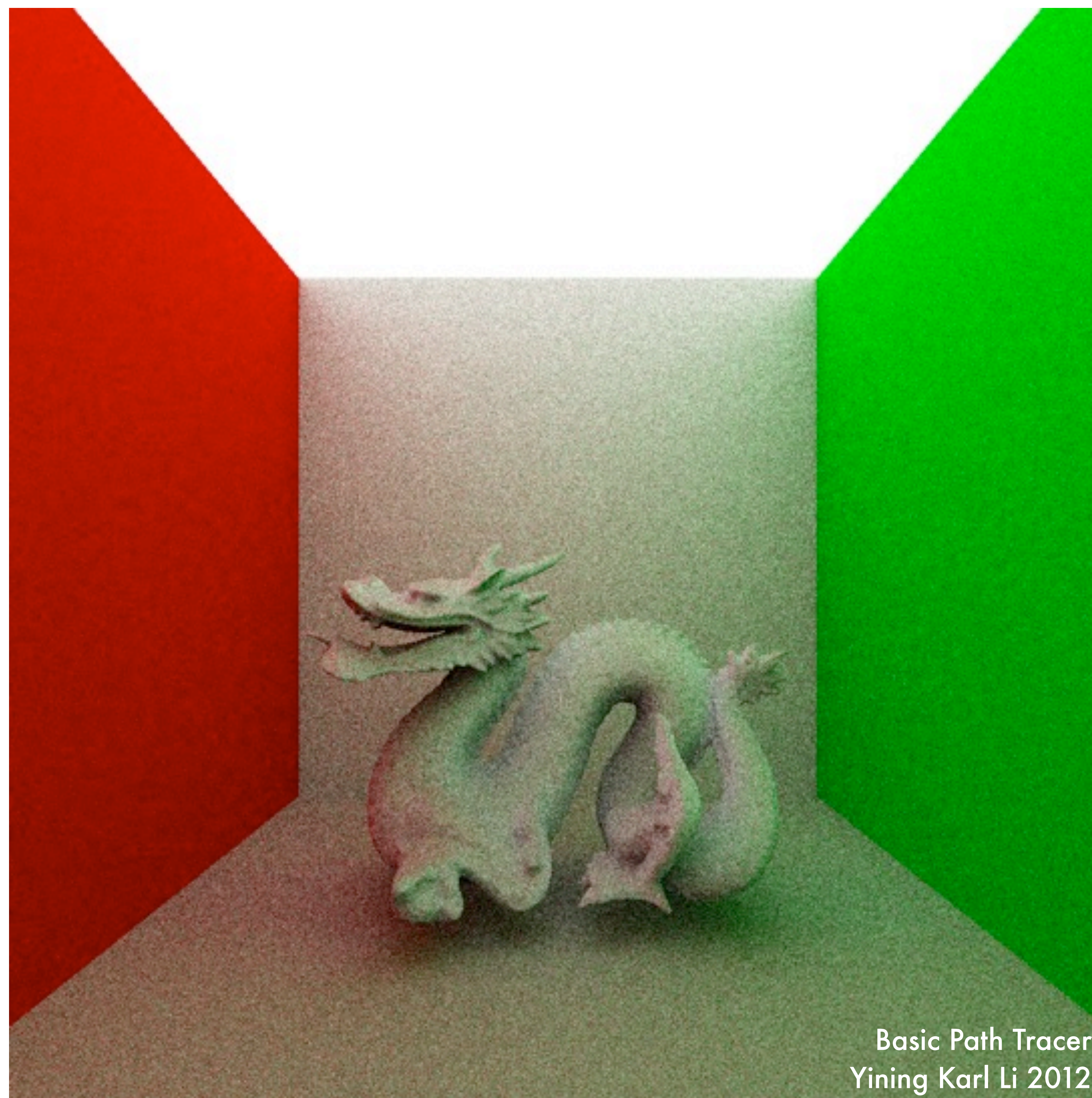
Monte Carlo Integration

- Without going into the math, Monte Carlo integration essentially refers to numerical techniques through which a proper solution to a function is arrived at through repeated random sampling
- In distributed ray-tracing, we converge on a proper representation of various phenomenon by repeatedly randomly sampling a probability distribution function
- We can adapt this idea to arrive at a GI solution!
- We can randomly sample the emittance coming in to a given point from every direction, and over time converge to the total incoming emittance for the given point



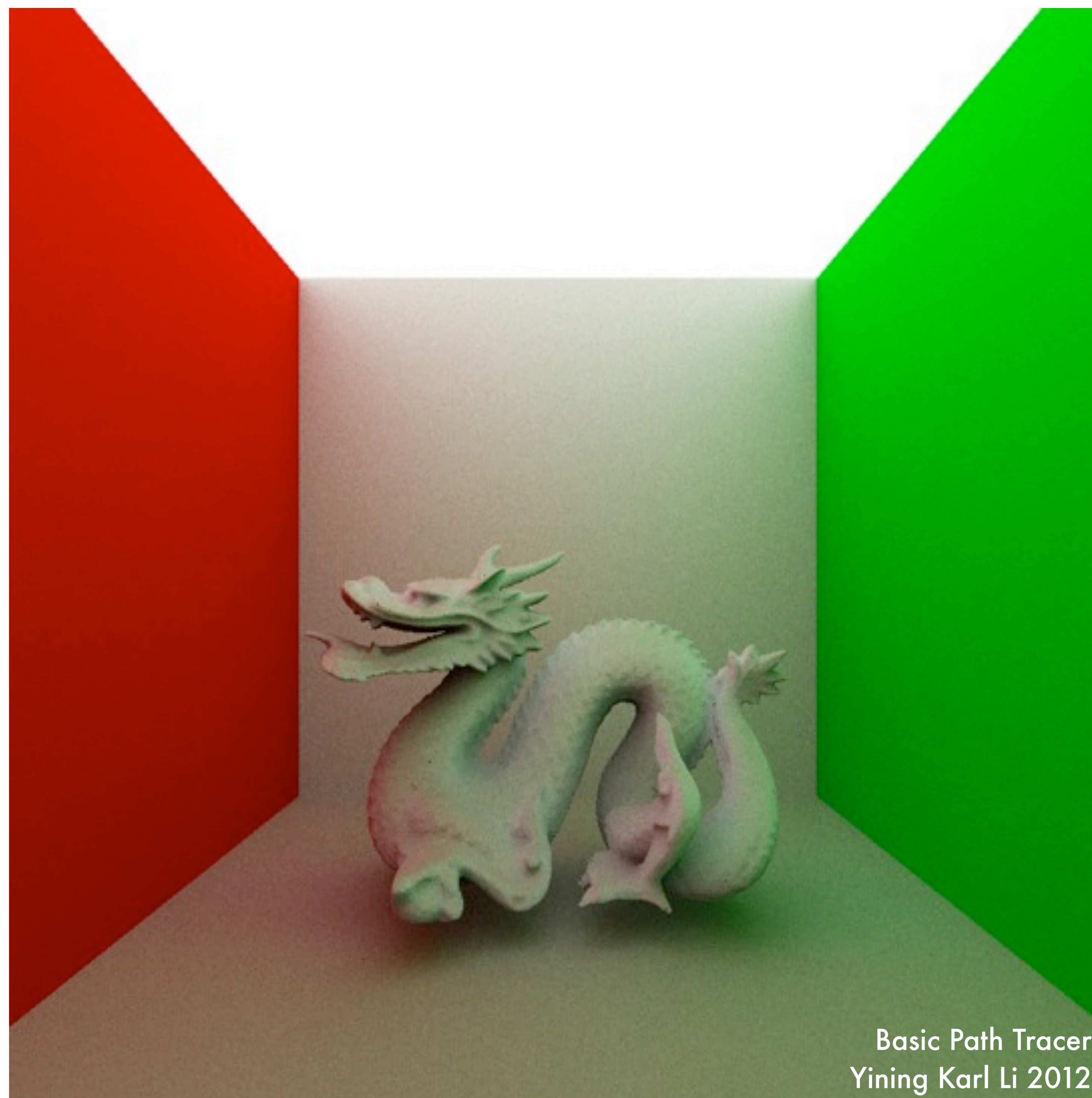
1 Iteration

Since Monte-Carlo integration converges at a solution through repeated random sampling, high variance can occur when there aren't sufficient samples



20 Iterations

Since Monte-Carlo integration converges at a solution through repeated random sampling, high variance can occur when there aren't sufficient samples



250 Iterations

Since Monte-Carlo integration converges at a solution through repeated random sampling, high variance can occur when there aren't sufficient samples

Monte Carlo Integration: GPU Implications

- Monte Carlo style techniques are extremely expensive to run on the CPU, since they require a huge number of samples in order to converge to a usable solution
- However, each sample is computationally independent from all other samples! Thus, Monte Carlo style techniques are often *embarrassingly parallel*.
- What does this imply about Monte Carlo techniques on the GPU?...

So how do we implement a Monte Carlo technique to calculate a global illumination solution?

Path-tracing

- Monte Carlo path-tracing is a full, unbiased method to solve the global illumination problem
- Path-tracing is a brute force solution, first proposed in 1986, but not fully implemented in a practical production environment until the past few years.
- In many ways, path-tracing can be viewed as distributed ray-tracing taken to its absurd logical extreme

The Third and the Seventh
Alex Roman

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

Path-tracing Algorithm

- So in pseudocode...

```
color3 pathTrace(int depth, ray r, vector<geom> objects, vector<lights> light_sources){

   [determine closest intersected object j, intersection normal n, intersection point p]

   if(no object is hit){
      return black;
   }else{

      if(object is light source){
         return light_emittance;
      }

      r = evaluateBSDF(r, n, j.material);

      return material_color * pathTrace(depth-1, r, objects, light_sources);
   }
}
```

- Note that algorithmically, this is actually simpler than normal ray-tracing!

Path-tracing Algorithm

- But remember, we need to accumulate multiple samples and take an average...

```
color3 Accumulate(int iterations){

   color3 accumulatedColor;

   for(int i=0; i<iterations; i++){
      accumulatedColor += pathTrace()
   }

   return accumulatedColor/iterations;
}
```

- Note that Accumulate() is run for each pixel in the image

Path-tracing Algorithm

- Unfortunately, as we saw earlier, path-tracing requires a huge number of samples per pixel in order to converge to a reasonably smooth solution

- Biased techniques such as photon mapping arrive at a smooth solution faster, at the cost of blotchiness and numerical errors. Path-tracing has no blotches or numerical errors, but at the cost of noise that takes a huge number of samples to eliminate

- For these reasons, very few production renderers make use of pure CPU path-tracing

I want my path-tracer to render faster

Parallel Path-tracing

- Fortunately, much like ray-tracing, path-tracing is embarrassingly parallel!
- Remember how we converted recursive ray-tracing to iterative ray-tracing in Part 1? Our path-tracing algorithm here needs to be converted to run iteratively in a similar fashion.
- For parallel ray-tracing, parallelizing by ray path instead of by ray is slower, but overall still fairly acceptable. However, path-tracing, unlike ray-tracing, can be highly non-deterministic.
- What does this imply about how we should parallelize the path-tracing routine?

Parallel Path-tracing

- Remember how we converted recursive ray-tracing to iterative ray-tracing in Part 1? Our path-tracing algorithm here needs to be converted to run iteratively in a similar fashion.

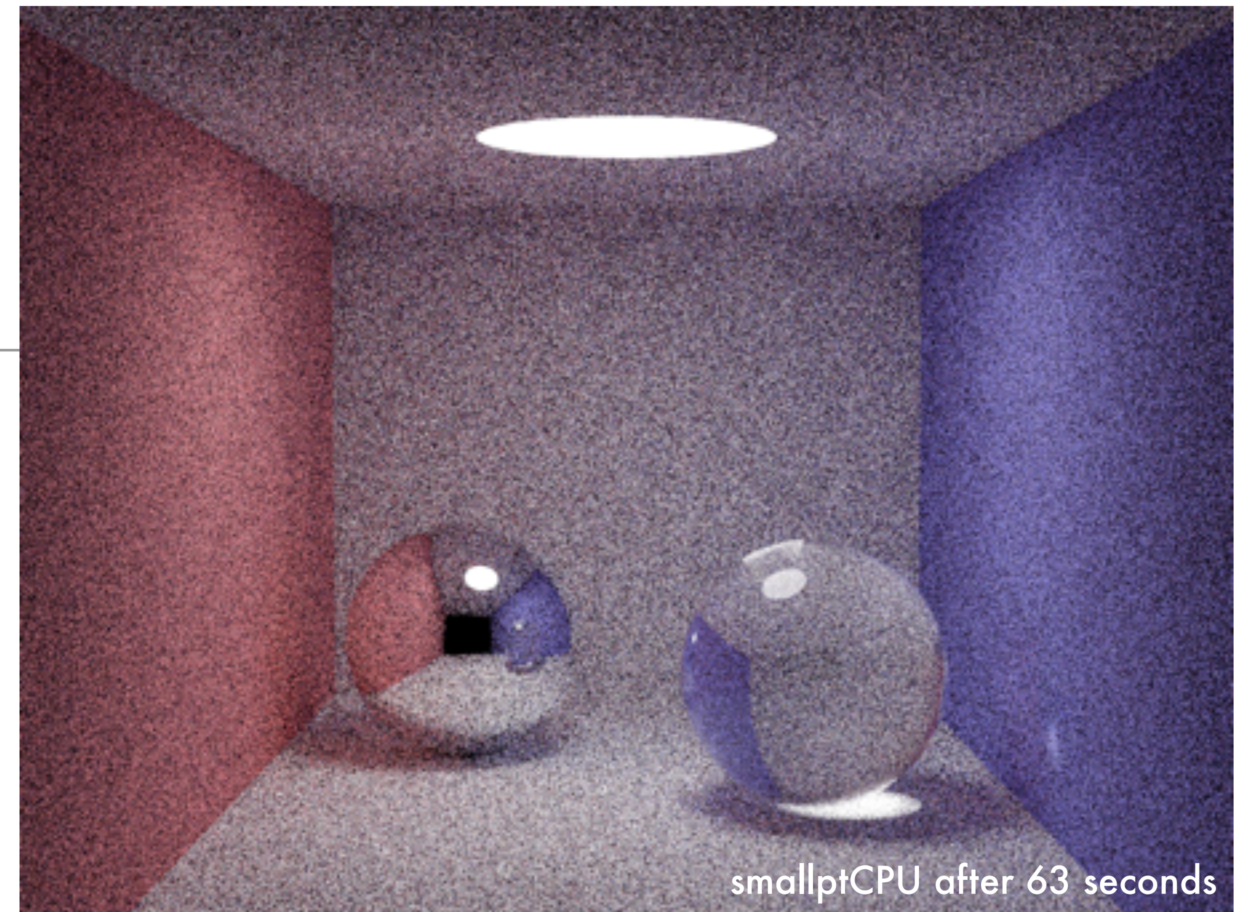
- For parallel ray-tracing, parallelizing by ray path instead of by ray is slower, but overall still fairly acceptable. However, path-tracing, unlike ray-tracing, can be highly non-deterministic.

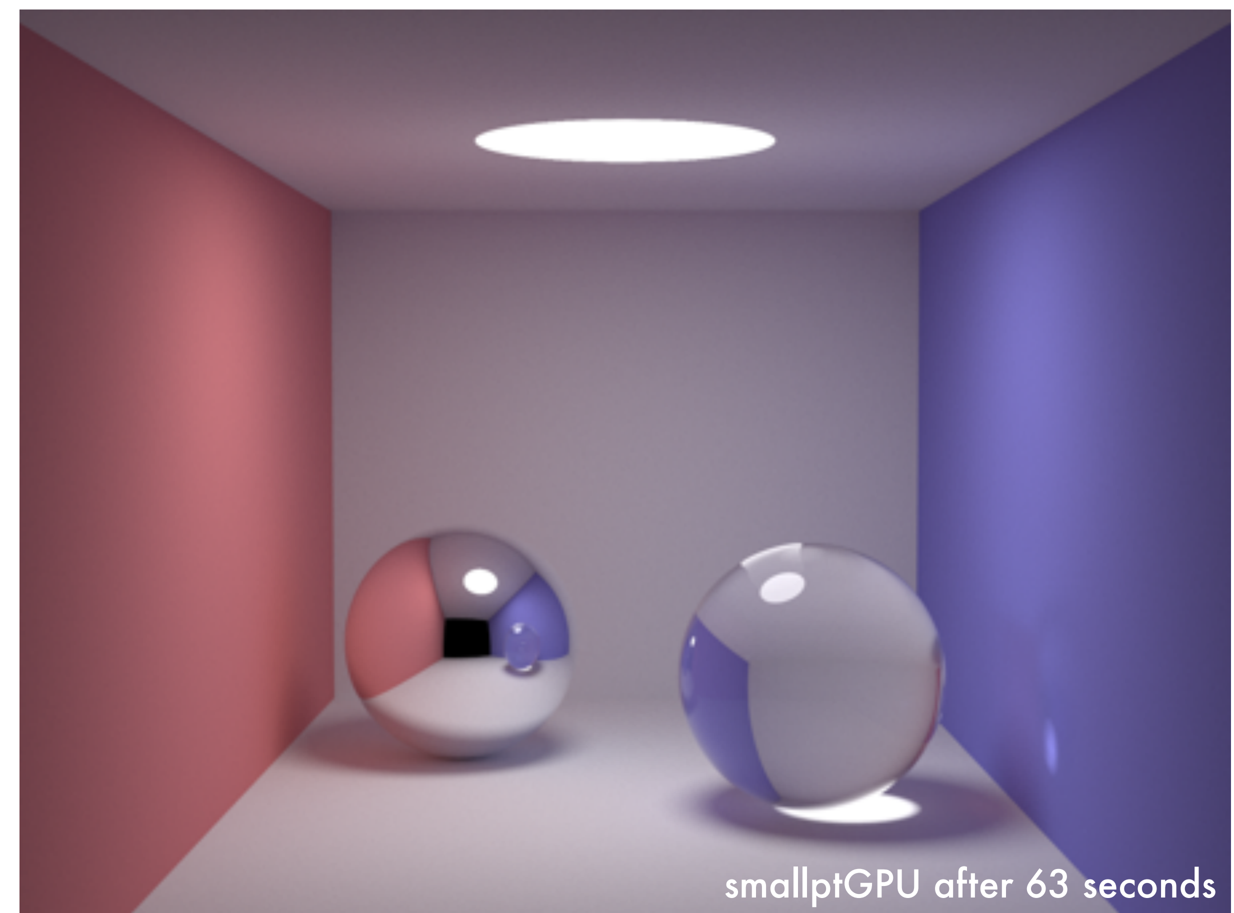
- What does this imply about how we should parallelize the path-tracing routine?

    - We should parallelize by ray!

Parallel Path-tracing

- Fun experiment for you to try at home: try comparing smallptCPU, a small CPU pathtracer, with its sibling GPU version, smallptGPU:
  - smallpt: http://www.kevinbeason.com/smallpt/
  - smallptGPU: http://davibu.interfree.it/opencl/smallptgpu2/smallptGPU2.html
- Expect smallptGPU to be between 10 and 100 times faster than smallptCPU.


smallptCPU after 63 seconds


smallptGPU after 63 seconds

I want my path-tracer to render faster

Path-tracing Algorithm

- What is the most expensive part of this algorithm?

```
color3 pathTrace(int depth, ray r, vector<geom> objects, vector<lights> light_sources){

   [determine closest intersected object j, intersection normal n, intersection point p]

   if(no object is hit){
      return black;
   }else{

      if(object is light source){
         return light_emittance;
      }

      r = evaluateBSDF(r, n, j.material);

      return material_color * pathTrace(depth-1, r, objects, light_sources);
   }
}
```

Path-tracing Algorithm

- What is the most expensive part of this algorithm?

```
color3 pathTrace(int depth, ray r, vector<geom> objects, vector<lights> light_sources){
  [determine closest intersected object j, intersection normal n, intersection point p]
  if(no object is hit){
     return black;
  }else{

     if(object is light source){
        return light_emittance;
     }

     r = evaluateBSDF(r, n, j.material);

     return material_color * pathTrace(depth-1, r, objects, light_sources);
  }
}
```

Tons of intersection testing is where the majority of our compute is going!