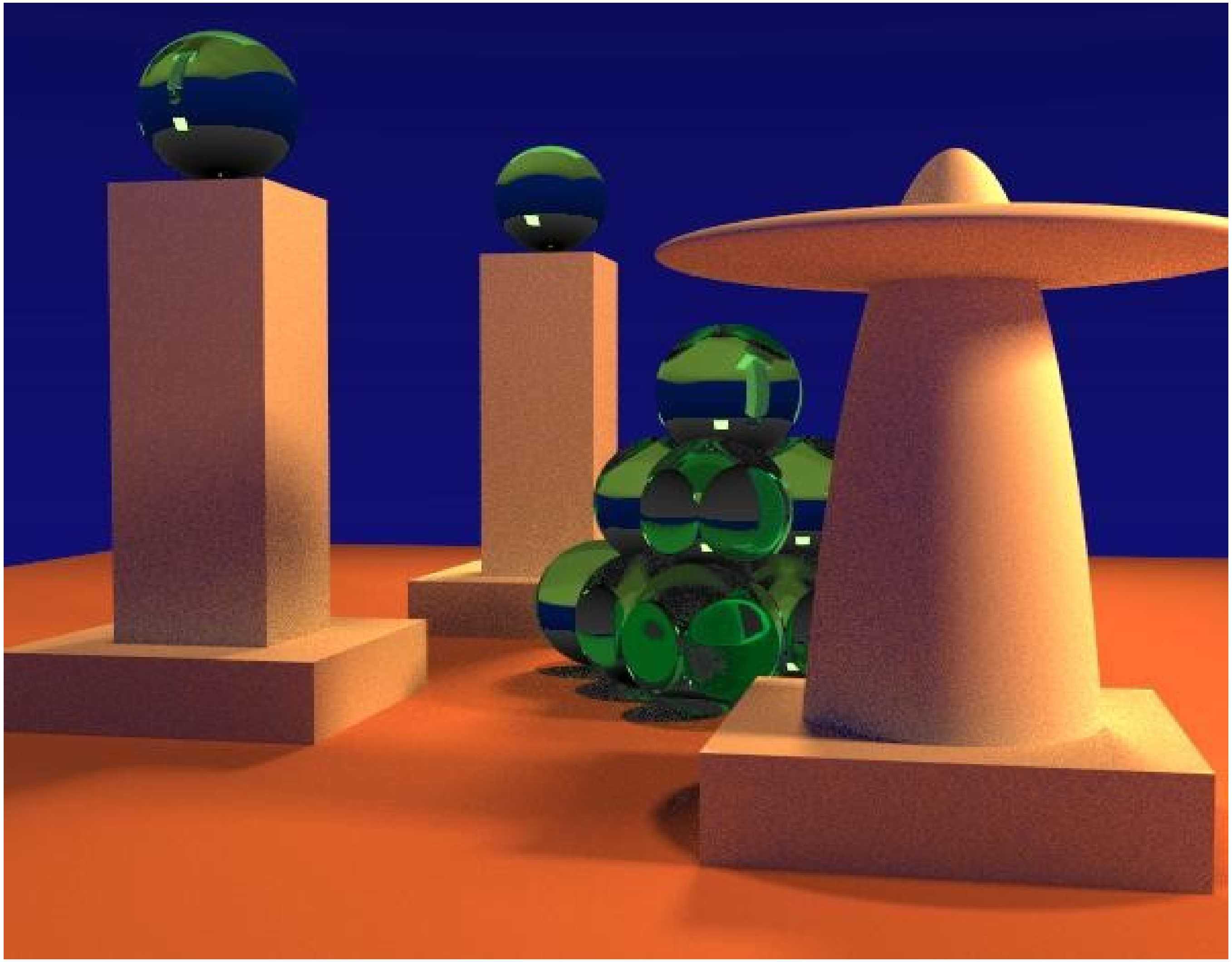
Monte Carlo Ray Tracing

EECS 487

March 21, 2007



outline

- rendering algorithms:
  - scan conversion
  - ray casting
  - ray tracing
  - monte carlo ray tracing

scan conversion

```
for each triangle T
  for each pixel in T
     color the pixel (if depth test ok)
```

scan conversion: analysis

- rendering window has p pixels (e.g. ~1 million)
- scene has n triangles (e.g. ~200,000)
- average depth complexity is d (e.g. $d < 4$)
  - i.e., d is layers of surface at a given pixel
- what is an upper bound on # steps to render?

scan conversion: analysis

number of steps = n + p*d

e.g.: 4.2 million

ray casting

```
for each pixel
   construct corresponding ray r
   intersect r with scene
   compute color via lighting, textures
```

ray casting: analysis

- rendering window has p pixels (e.g. ~1 million)
- scene has n triangles (e.g. ~200,000)
- (depth complexity not relevant)
- what is an upper bound on # steps to render? (assuming ray intersections are brute force)

ray casting: analysis

number of steps = p*n

e.g. 200 billion

(50,000 times slower than scan conversion)

ray tracing

```
for each pixel
   construct corresponding ray r
   intersect r with scene
   compute color via lighting, textures
   spawn 2 more rays and recurse
```

ray tracing: analysis

- rendering window has p pixels (e.g. ~1 million)
- scene has n triangles (e.g. ~200,000)
- depth of recursion is r (e.g. 5)
- what is an upper bound on # steps to render? (assuming ray intersections are brute force)

ray tracing: analysis

number of ray tests depends on level of recursion:

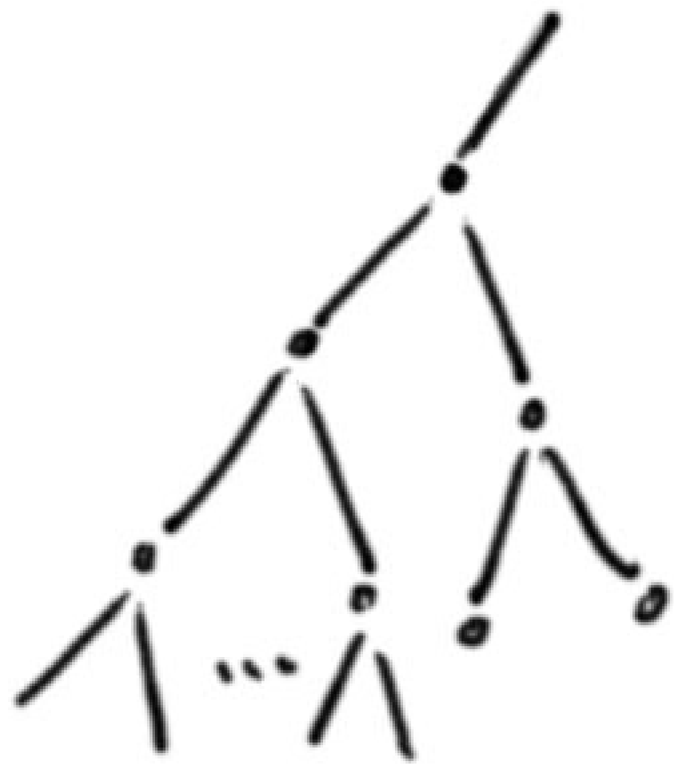
1 level: 1 test (per pixel)

2 levels: 3 tests

3 levels: 7 tests

r levels: $2^r - 1$ tests

round up: $\sim 2^r$ tests



ray tracing: analysis

ray tests (per pixel): 2^r tests

steps per ray test: n

total number of steps: p * n * 2^r

e.g.: 32 times slower than ray casting
in our example (ignoring shadow rays)

note: this is pessimistic, since not every surface is
both specular *and* transparent

ray tracing analysis

- not so bad!

- can we make it slower?

ray tracing analysis

- not so bad!
- can we make it slower?
- yes!

monte carlo ray tracing

```
for each pixel
   construct corresponding ray r
   intersect r with scene
   compute color via lighting, textures
   spawn multiple additional rays
     and recurse
```

Q: how is this different from ray tracing?

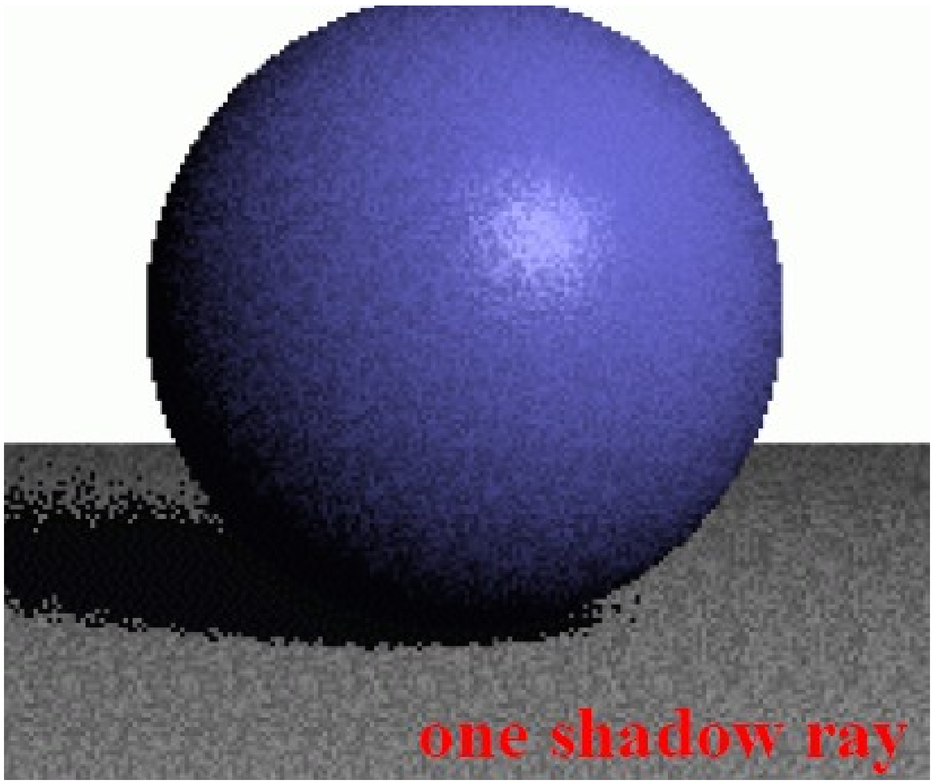
monte carlo ray tracing

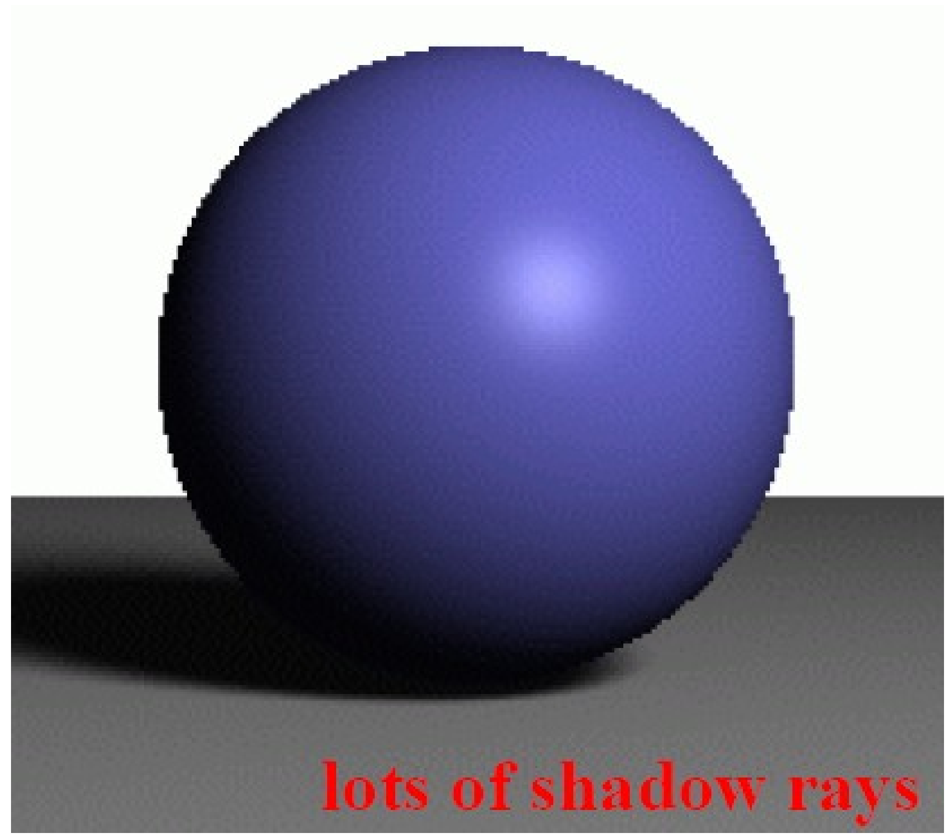
for each pixel
  construct corresponding ray r
  intersect r with scene
  compute color via lighting, textures
  spawn <u>multiple</u> additional rays and recurse

A: “multiple” instead of “2 ”

Q: why cast all these additional rays?

- A: get better simulation of global illumination
- e.g. soft shadows:
  - instead of 1 shadow ray to each *point* light,
  - cast multiple (random) rays to each *area* light
  - or: cast 1 (random) ray to each area light
  - fewer samples yields more “noise”





groups.csail.mit.edu/graphics/classes/6.837/F03/lectures/19_MonteCarlo.pdf

other effects

- soft shadows
- ?

other effects

- soft shadows
- glossy reflection
- color bleeding
- motion blur
- depth of field
- caustics?

monte carlo ray tracing: analysis

same as ray tracing, except the “branching factor” of the ray tree is not 2

call it b (e.g. $b = 100$)

recursion level: r (e.g. $r = 5$)

ray tests (per pixel): b^r tests

monte carlo ray tracing: analysis

total number of steps: $p * n * b^r$

$(b/2)^r$ times more work

(e.g. 50^5, or 300 million times more work than plain ray-tracing in our example)

observations

- actually, maybe b = 100 was a tad high...
  (but low values produce noise)
- brute force ray tests are a bad idea here
  (smarter method could be much faster)
- need to limit the depth of recursion
  (recurse when it will matter)
- and the number of rays cast
- should avoid work that makes no contribution

modification: monte carlo path tracing

- trace only 1 secondary ray per recursion
- but trace many primary rays per pixel
- (performs antialiasing as well)

monte carlo path tracing

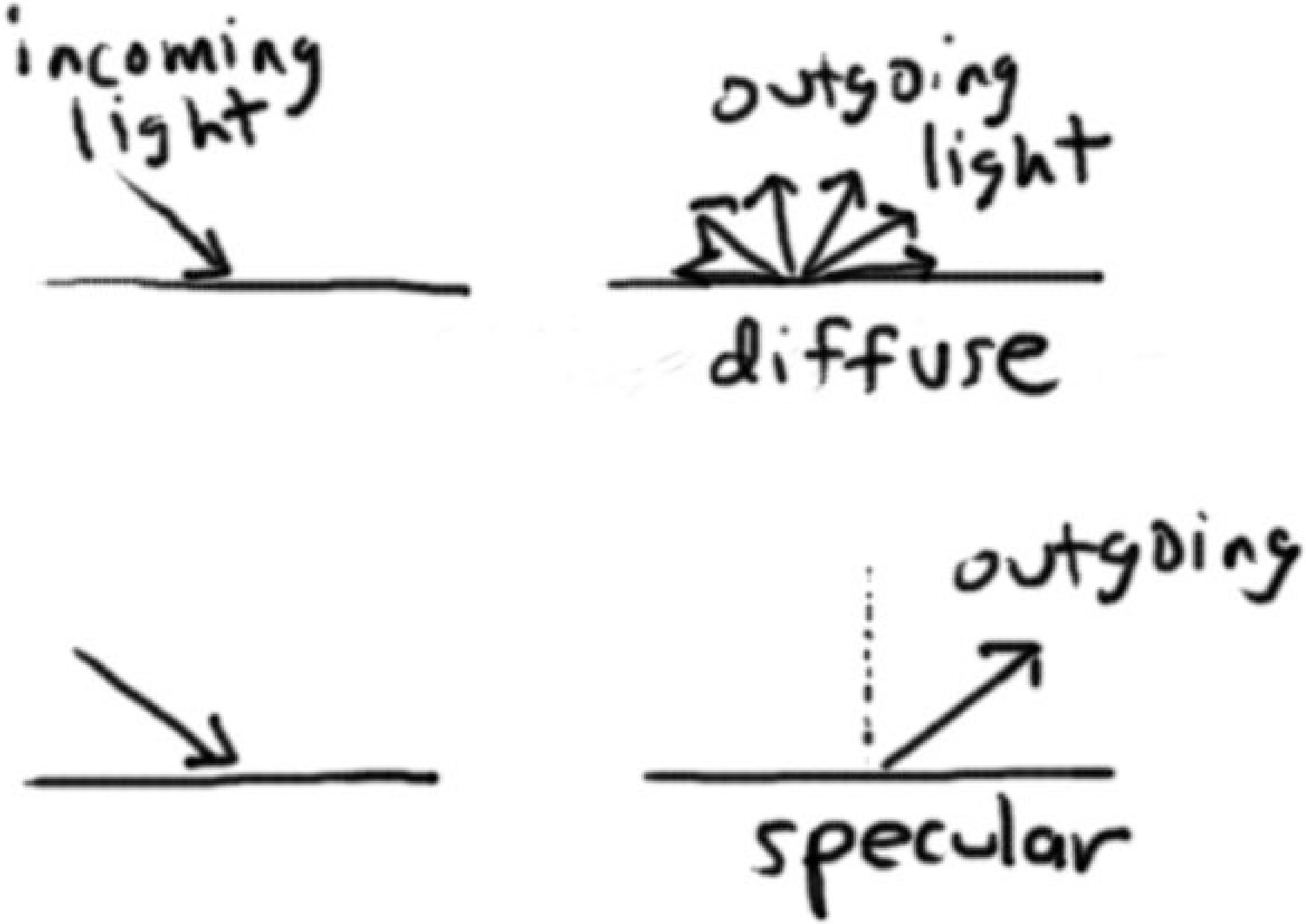
```
trace ray:
  find ray intersection with nearest object
  shade object

shade object:
  sample incoming light(via 1 random ray)
  shade using BRDF
```

Digression: what is a "BRDF"?

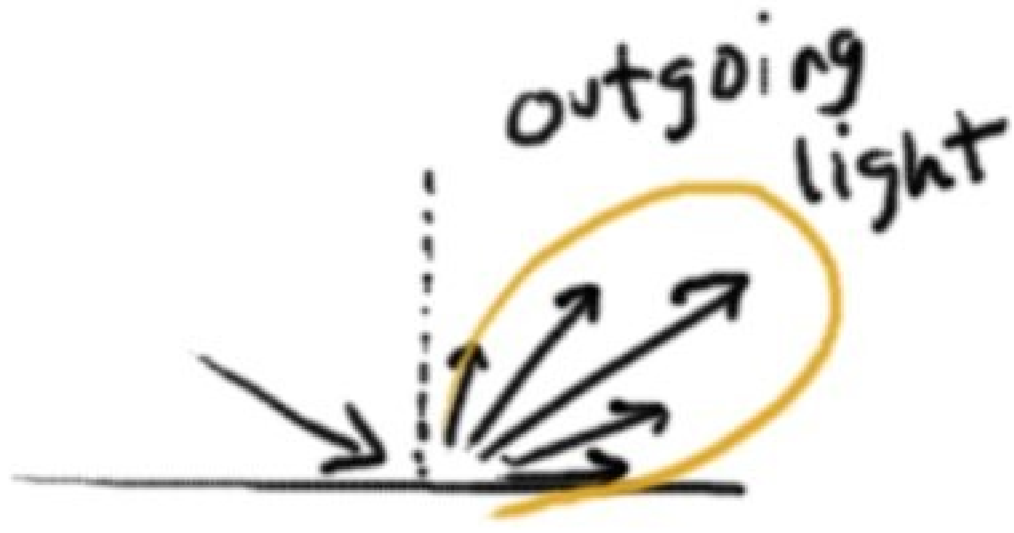
- Bi-directional reflectance distribution function
- Describes how a material reflects light
- We have seen simple cases:
  - pure diffuse
  - pure specular
  - combination of pure diffuse and pure specular
- Real materials are not so simple

Simple BRDFs: diffuse or specular reflection



General BRDFs

- Most real materials do not correspond to either of those extremes (diffuse or specular)
- E.g., a glossy surface:



General BRDFs

- For each incoming direction, tells how much light will be reflected in each outgoing direction
- A BRDF is a function, describing the distribution of outgoing light, given an incoming direction
  - F: function
  - D: distribution

How to encode a BRDF?

- suppose we use a look-up table
- what are the dimensions?

How to encode a BRDF?

First, note that a direction is a 2D entity

- think of a hemisphere representing the sky over your head
- it takes two angles to designate a point on the hemisphere
- each point corresponds to a direction

How to encode a BRDF?

- A BRDF answers this question:
  for this incoming direction, what strength of light results along that outgoing direction?
- I.e.: given this pair of directions,
  what is the light strength?

How to encode a BRDF?

- A BRDF answers this question:
  for this incoming direction, what strength of light results along that outgoing direction?
- I.e.: given this pair of directions, what is the light strength?
- So a BRDF is a 4D entity
  - i.e., the lookup table is 4 dimensional
  - for each quadruple, it returns a single value

More on dimension

- image: 2D data set
- volume or movie: 3D data set
- BRDF: 4D data set
  - not practical to have varying BRDFs over a surface
  - may not need same resolution as in images / movies
  - still expensive

OK but, why “monte carlo”??

next few slides sampled from:

groups.csail.mit.edu/graphics/classes/6.837/F03/lectures/19_MonteCarlo.pdf

and also:

http://www.cs.utah.edu/classes/cs6620/lecture-2006-03-24-6up.pdf

digression: monte carlo integration

- want to evaluate: $$\int f(x)\,dx$$

- Use random variable x_i with uniform probability, convert integral to a sum:

$$\frac{1}{n}\sum_{i=1}^{n} f(x_i)$$

improved version

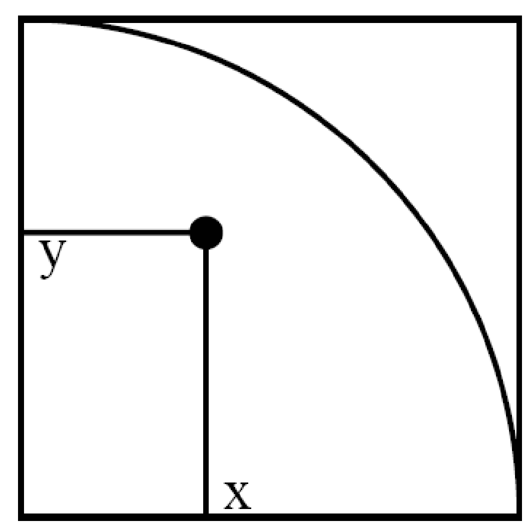
- Use random variable x_i *with probability* p_i

$$\frac{1}{n}\sum_{i=1}^{n}\frac{f(x_i)}{p_i}$$

- the whole trick is to choose the x_i and p_i to sample the interesting places

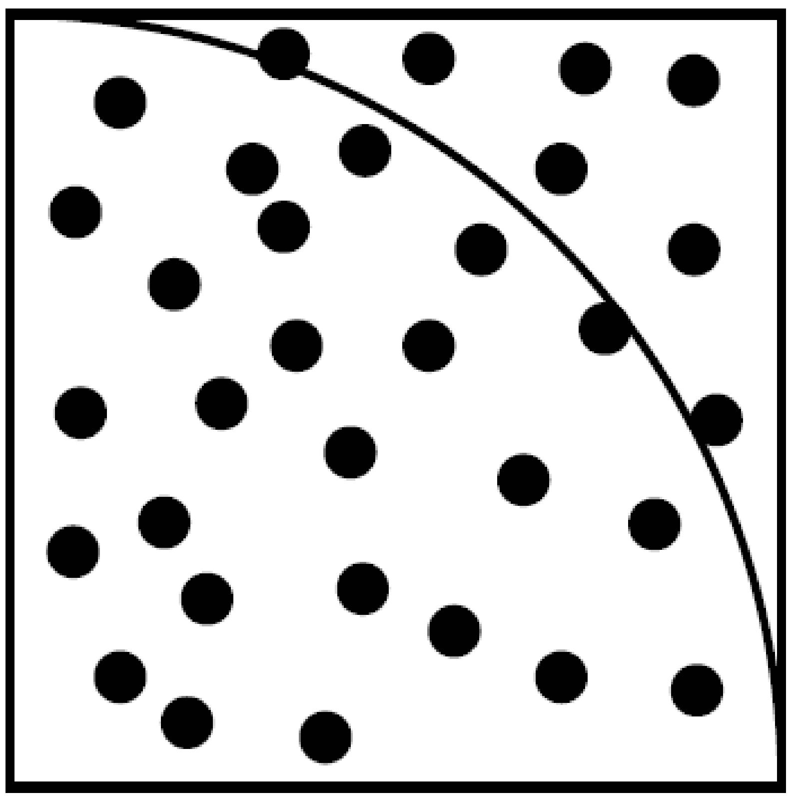
Example: monte carlo integration to compute π

- take a square
- take a random point (x,y) in the square
- test if it is in the ¼ circle ($x^2 + y^2 < 1$)
- run a lot of trials to estimate the probability
- the probability is $\pi/4$
- i.e.: your estimate times 4 is approximately π



Example: monte carlo integration of π

- to reduce the error, use more trials



link to ray tracing

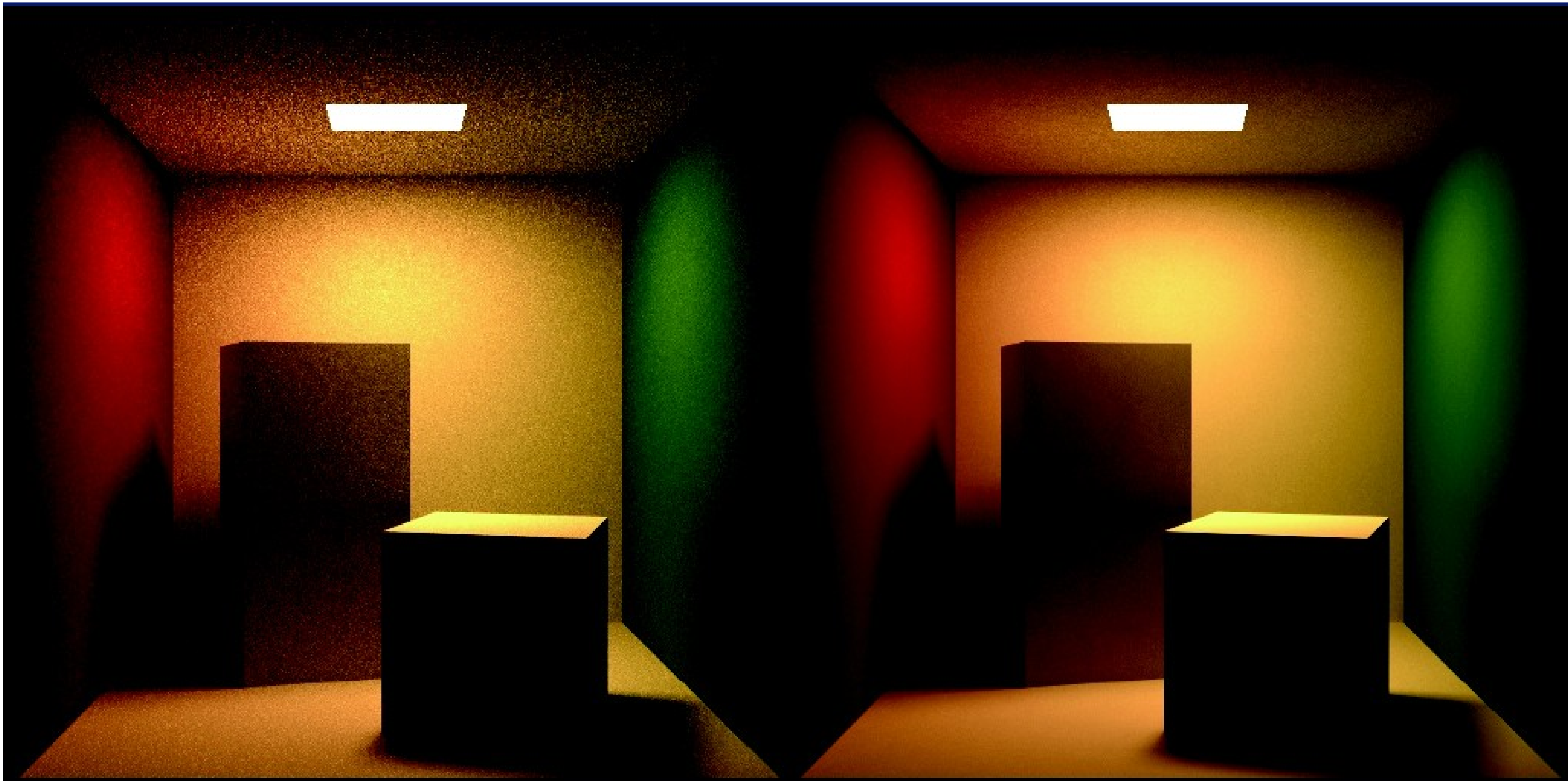
- Integration over light source area:
  – Soft shadows
- Integration over reflection angle:
  – Blurry reflections (gloss)
- Integration over refracted angle:
  – Translucency (fuzzy transparency)

link to ray tracing

- Integration over camera lens:
  - Depth of field
- Integration over time:
  - Motion blur

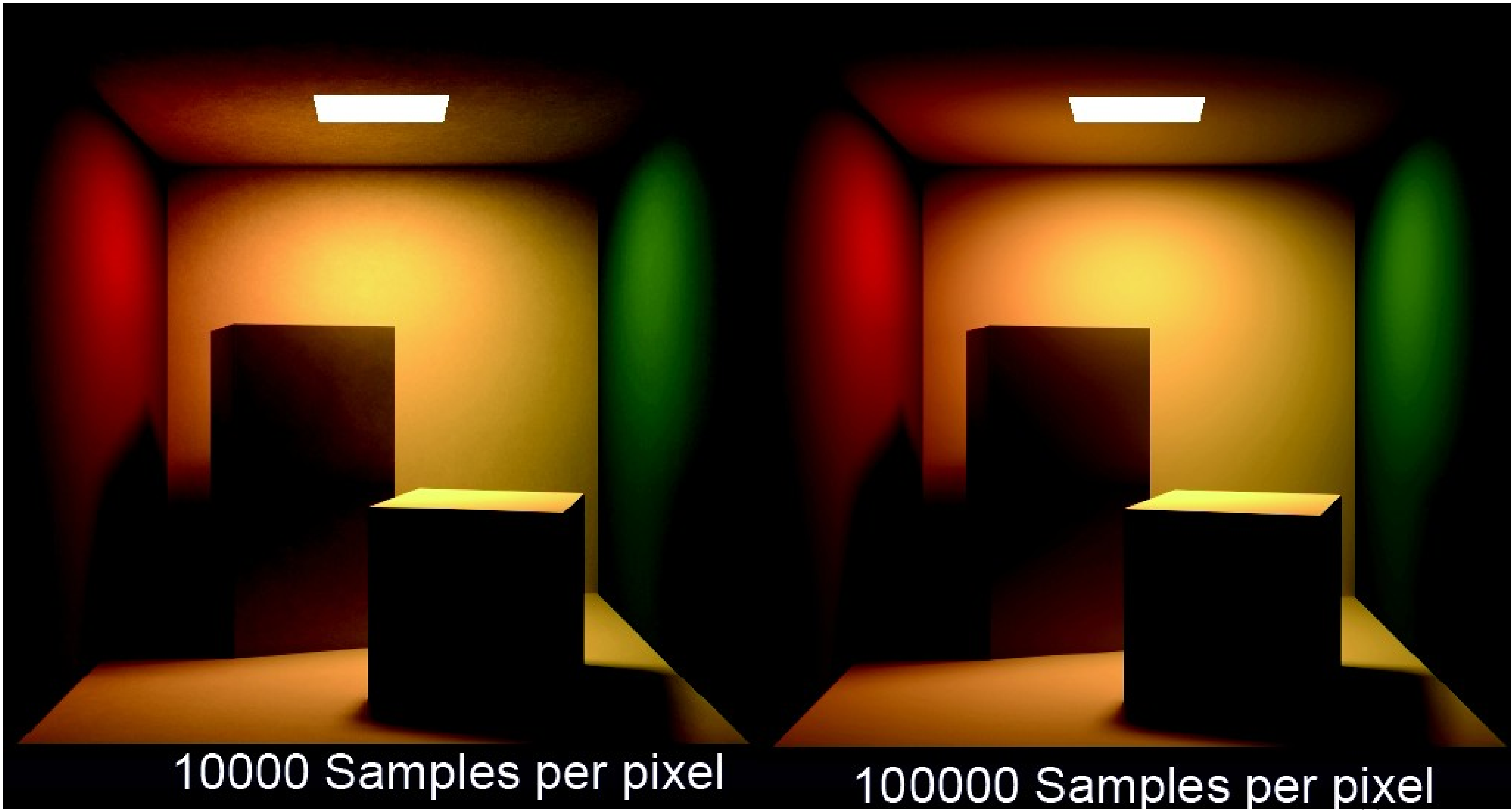
sampling strategies

- Pure Monte Carlo approach says to pick a random direction at each point
- Most rays will not hit a light source
- Kajiya style path tracing: pick a random light source and sample it randomly
    - Good convergence for scenes dominated by direct light



49 Samples per pixel

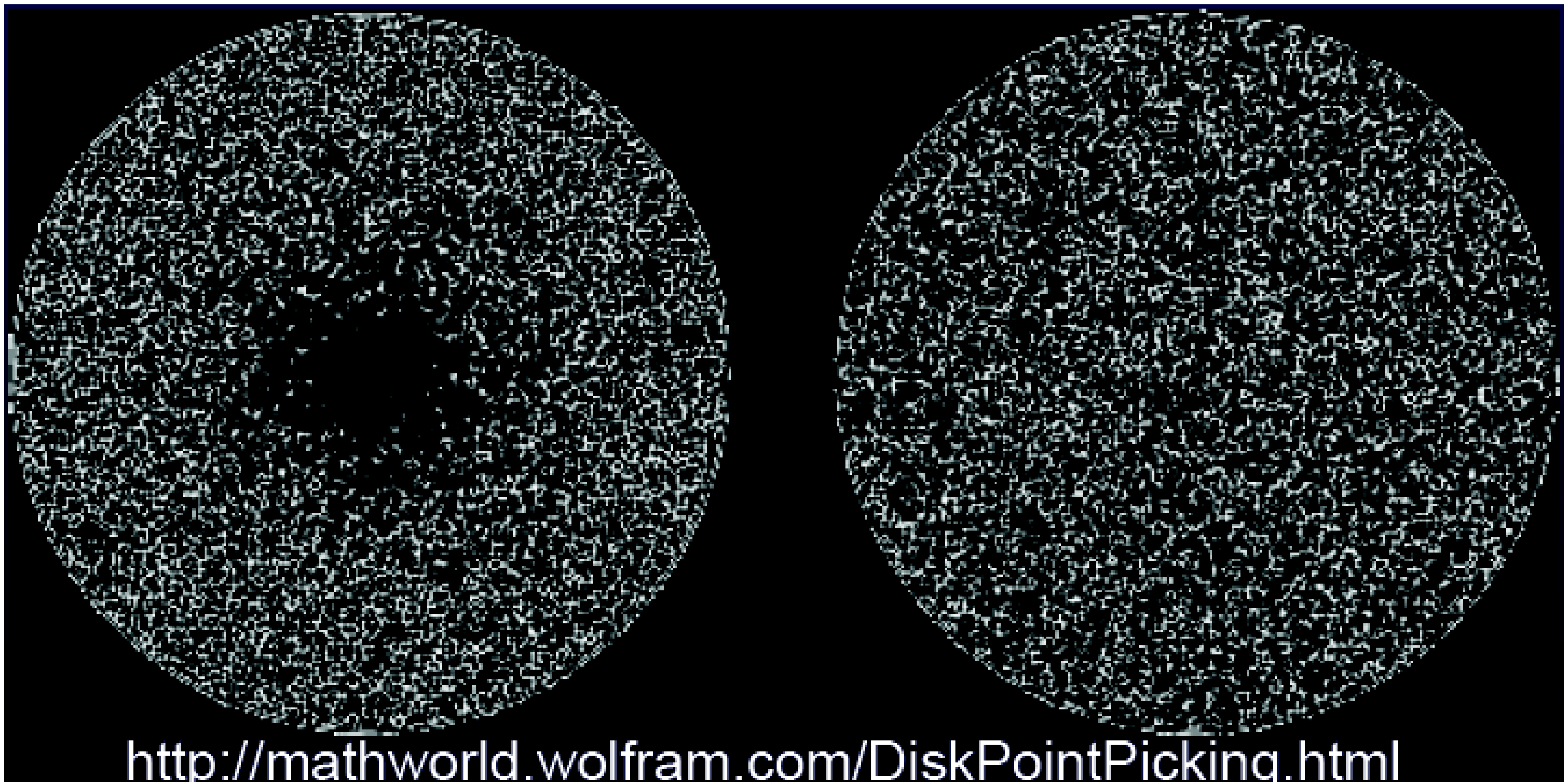
625 Samples per pixel


10000 Samples per pixel
100000 Samples per pixel

random sampling can be tricky

How to sample points on a disk uniformly?

wrong: right:



sampling a disk uniformly

- wrong:
  choose angle and radius uniformly:
  $\theta \in [0, 2\pi]$
  $r \in [0,1]$
  $x = r\,cos(\theta)$, $y = r\,sin(\theta)$

- Q: what's wrong with this?

sampling a disk uniformly

- wrong:
  choose angle and radius uniformly:
  $\theta \epsilon [0, 2\pi]$
  $r \epsilon [0,1]$
  $x = r cos(\theta)$, $y = r sin(\theta)$

- Q: what's wrong with this?
  A: samples are more crowded near center

sampling a disk uniformly

Right:

choose angle and r^2 uniformly:

$\theta \in [0, 2\pi]$

$r^2 \in [0,1]$ **note:** r^2, not r

$x = r\,cos(\theta)$, $y = r\,sin(\theta)$

Creates more samples at larger radiuses

alternate strategy (sampling a disk uniformly)

- pick a random location in the square that contains the disk
  - choose a random x and y coordinate in the disk
- if the point is outside the disk, discard it
- easy to see that this works
- downside: some wasted samples

monte carlo recap

- Turn integral into finite sum
- Use random samples
  - more samples = more accuracy (less noise)
- Very flexible
- Tweak sampling/probabilities for optimal result
- A lot of integration and probability theory
  to get things right

wrap up...

- project 4 due in 1 week + 1 day
- project 5 out then (3/29)
- I'll be out of town next week
  - no office hours