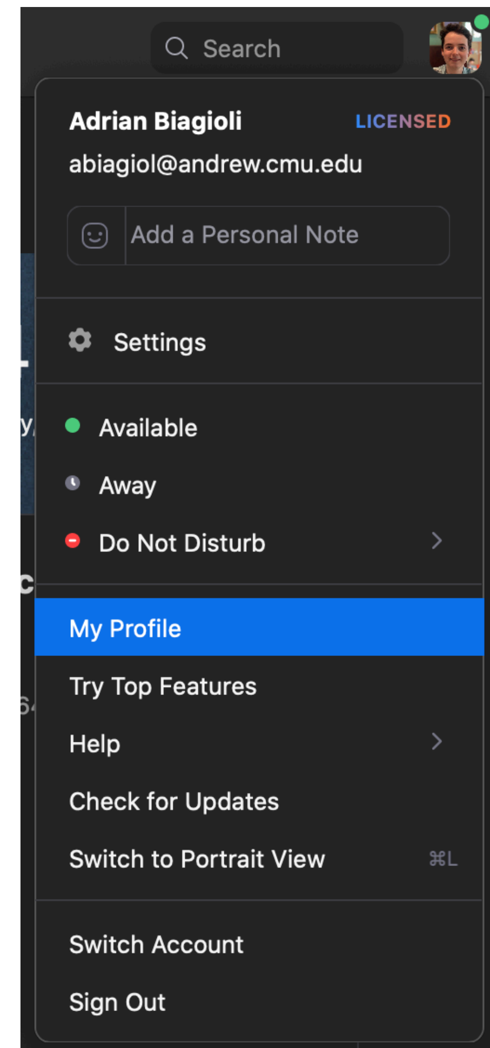
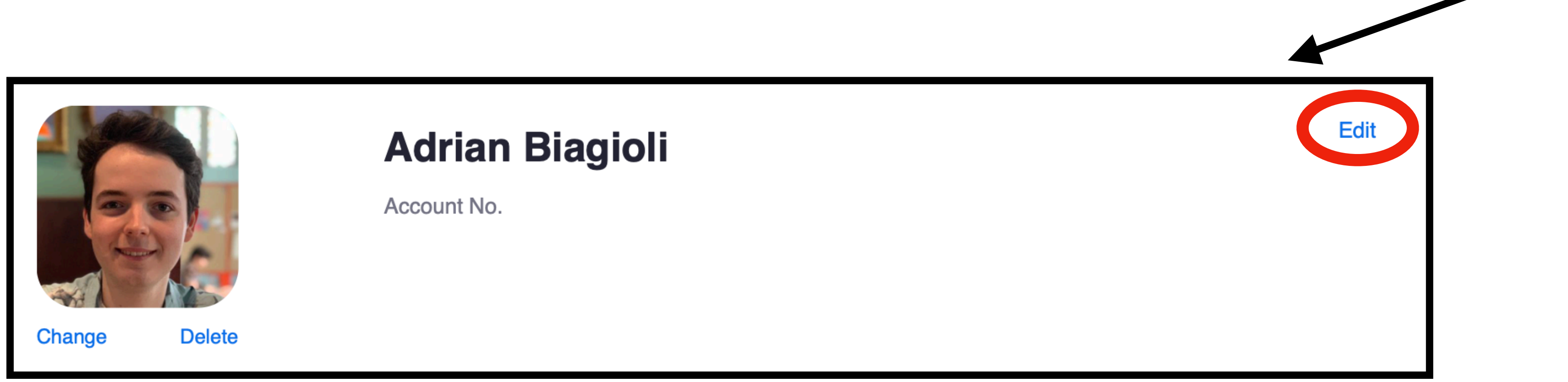
Recitation will begin Shortly

- Make sure your microphone is on mute
- Get used to the raise hand feature on zoom, I will be paying attention to this so you can ask questions
  - Please make sure your name is set on zoom so I can call on you
  - You can also ask questions in the text chat





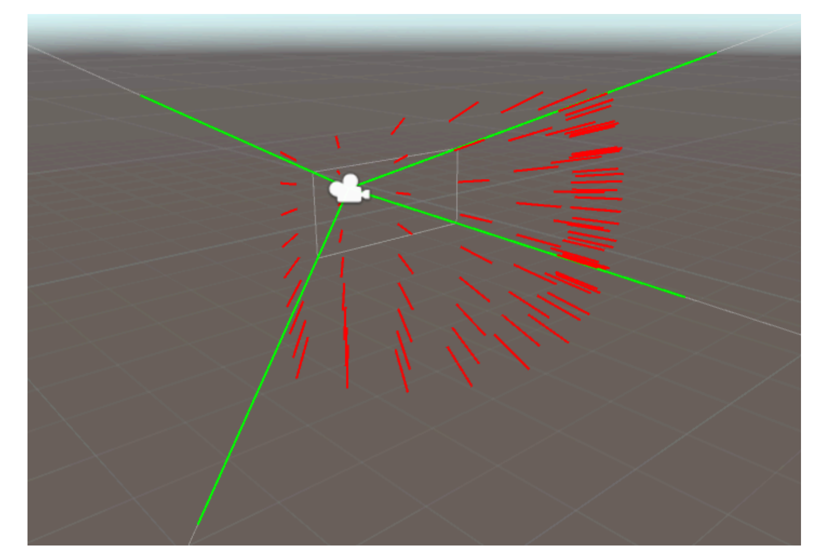
Assignment 3 Part 2 Overview

Computer Graphics
CMU 15-462/662

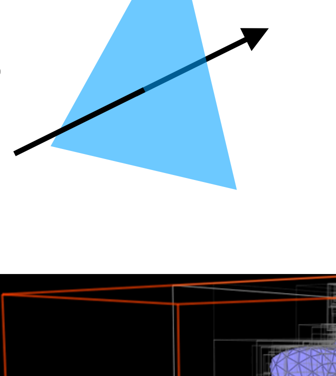
Introduction

- Assignment 3, Part 2 is *extremely theory heavy!*
  - Majority of your debugging time will be debugging math errors, not debugging crashes.
  - The majority of points to be lost are math-related.
    - Important: Include known errors in your writeup
    - We will look at your code, "it looks right" isn't enough
- Today we'll review much of the theory we went over in class, and give some implementation tips.

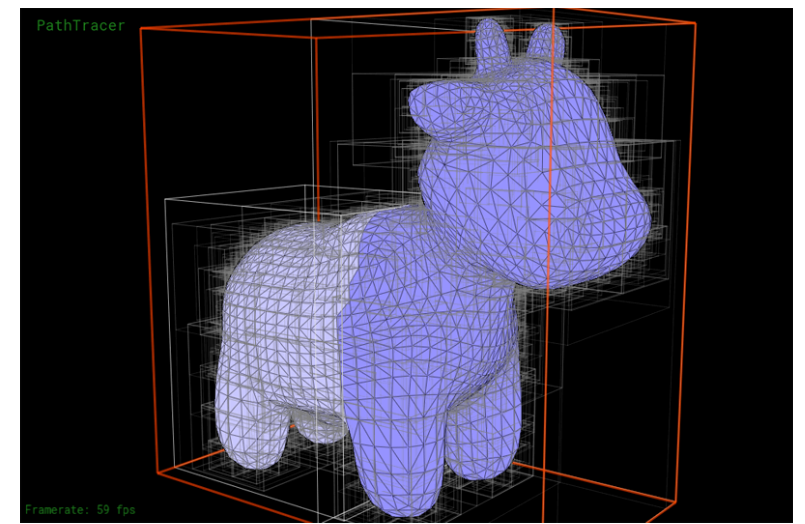
Overview of Tasks



- Task 1: Generate the initial rays to send from the camera
- Task 2: Compute ray-primitive intersection
  - You need to support triangles and spheres



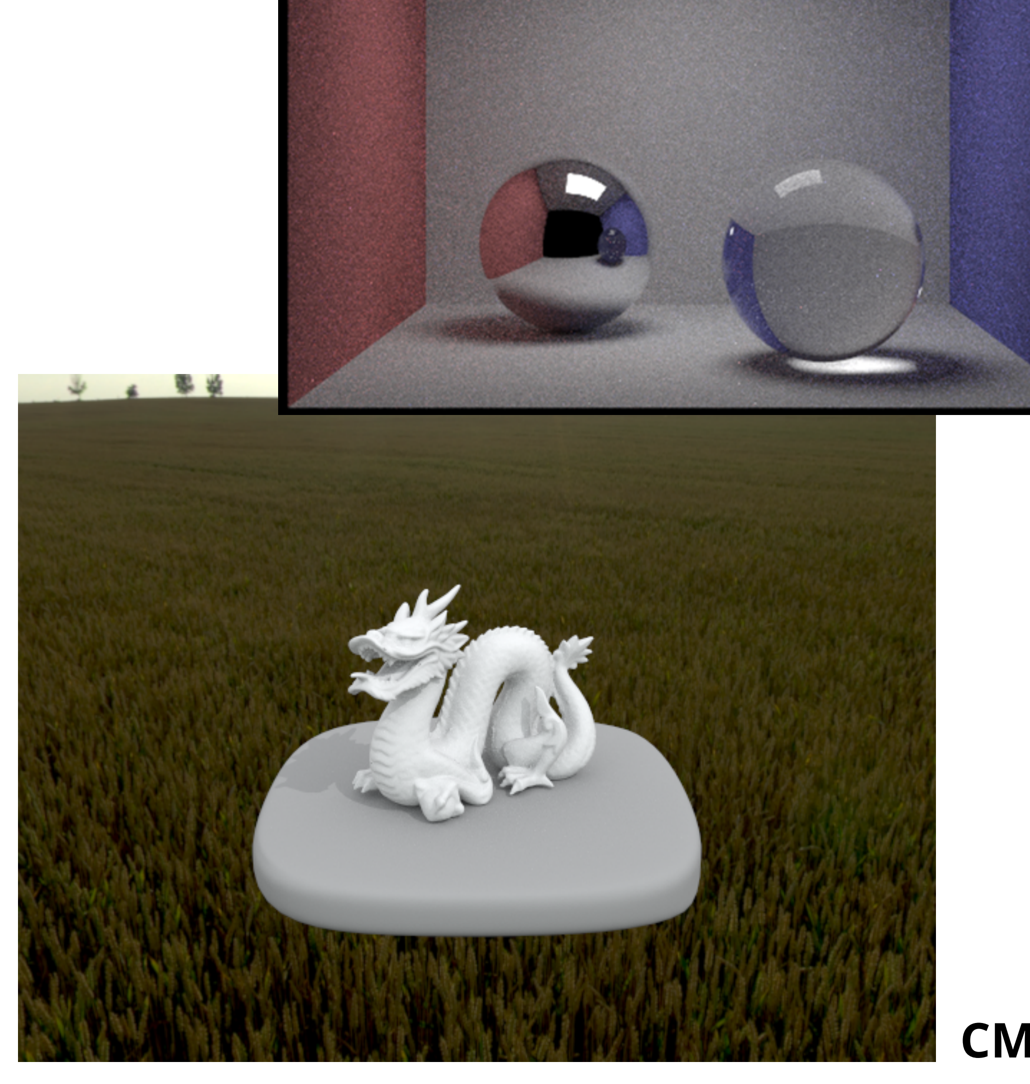
- Task 3: Accelerate ray-scene intersection queries using a Bounding Volume Hierarchy (BVH)



- Task 4: Implement direct lighting with shadows

Overview of Tasks

- Task 5: Support indirect illumination via path tracing
- Task 6: Support non-diffuse materials (mirror, glass)
- Task 7: Support environment lighting via a texture

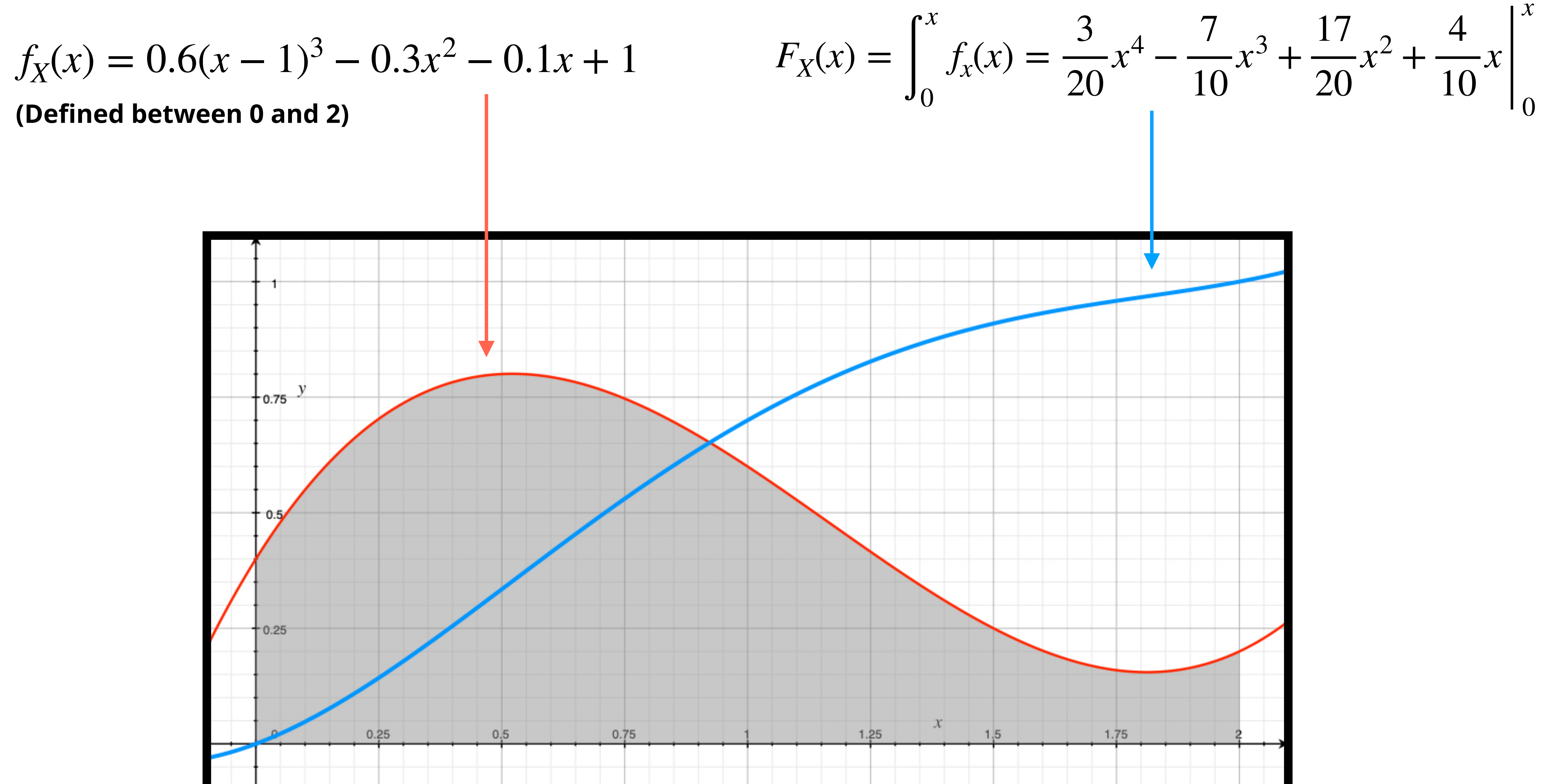


Sampling

PDF and CDF

- Probability Mass function (PMF): Probability that a discrete random variable X has value x ($p_X(x) \in [0,1]$)
- Probability Density Function (PDF): Continuous version of the PMF ($f_X(x) \in [0,1]$ for some continuous random variable X)
- Cumulative Distribution Function (CDF): Probability that a Random variable is less than some value
  - $F_X(x) = P(X \leq x)$
  - $F_X(\infty) = 1$ by definition of a PDF / PMF
  - Continuous (given PDF f_x): $F_X(x) = \int_{-\infty}^{x} f_x(x')dx'$
  - Discrete (given PMF p_x): $F_X(x) = \sum_{x' \leq x} p_x(x')$

PDF and CDF Example



Note, $F_X(\infty) = 1$ for any probability (discrete or continuous) distribution (sum of probabilities is 1). Also CDF is monotonically increasing.

Inversion Sampling

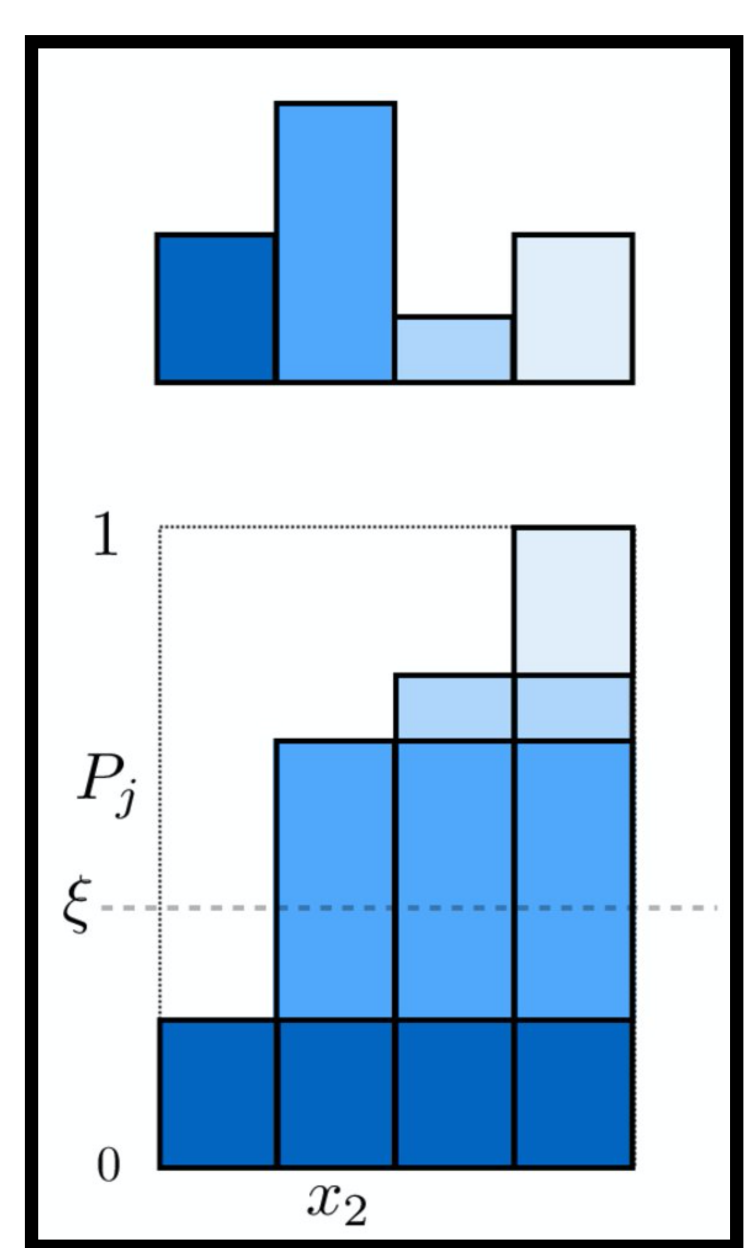
Goal: We are given a PDF $f_X(x)$ for some random variable X. We want to generate X so that it follows $f_X(x)$

Basic Steps:

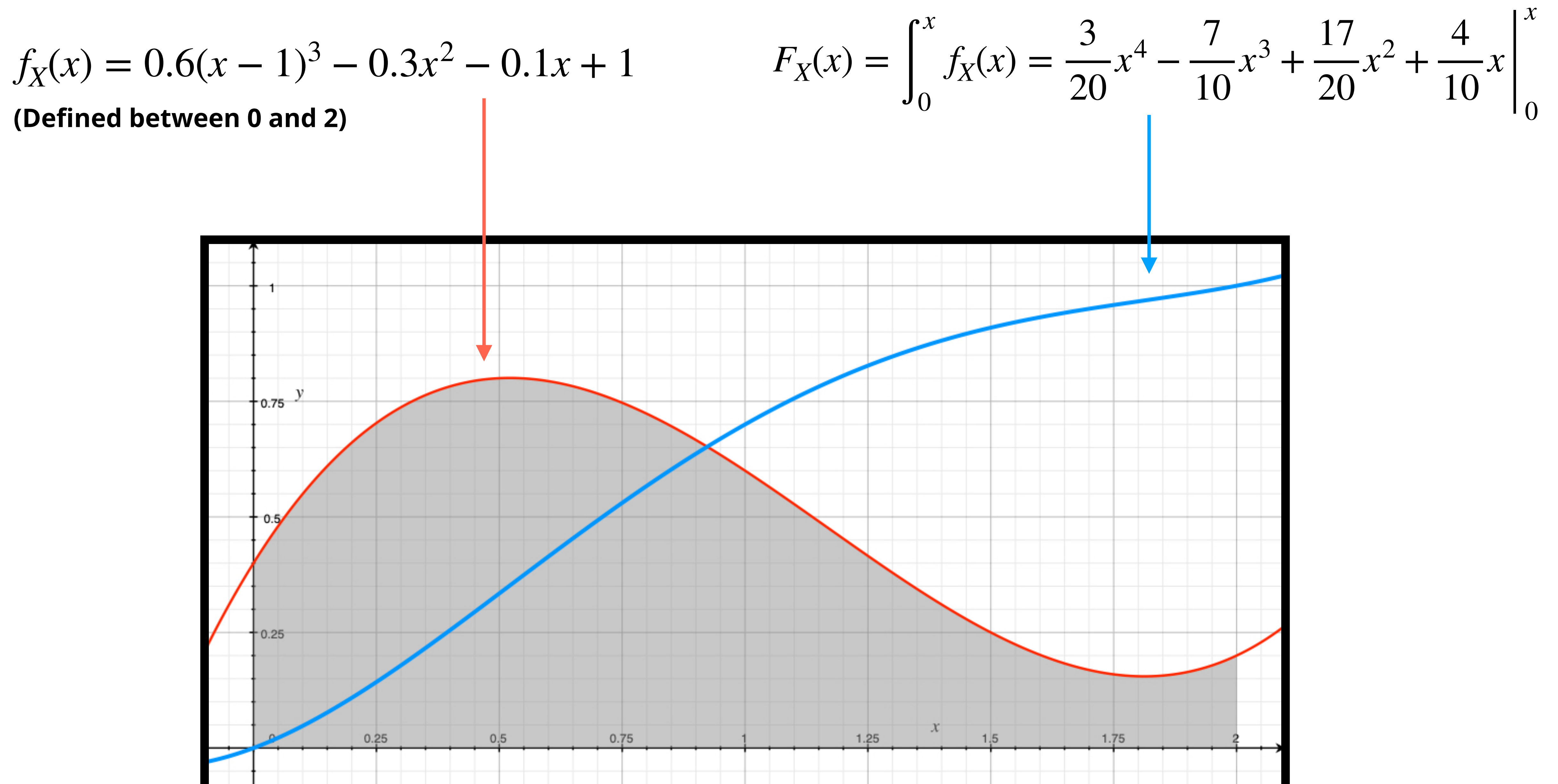
1. Evaluate the CDF $F_X(x)$ on the distribution
2. Invert the CDF to find the *quantile function* $Q_X(q)$. This should satisfy $Q_X(F_X(x)) = x$
3. Generate a random number uniformly between 0 and 1
4. Apply Q_X to the uniform sample

Inversion Sampling (Discrete)

- For discrete RV (PMF), you can imagine the CDF as stacking "blocks"
  - One block for each possible event, each block has height equal to PMF
  - Then we uniformly choose a random "height". Higher probability events will get picked more often (that's our goal!)



Inversion Sampling (Continuous)

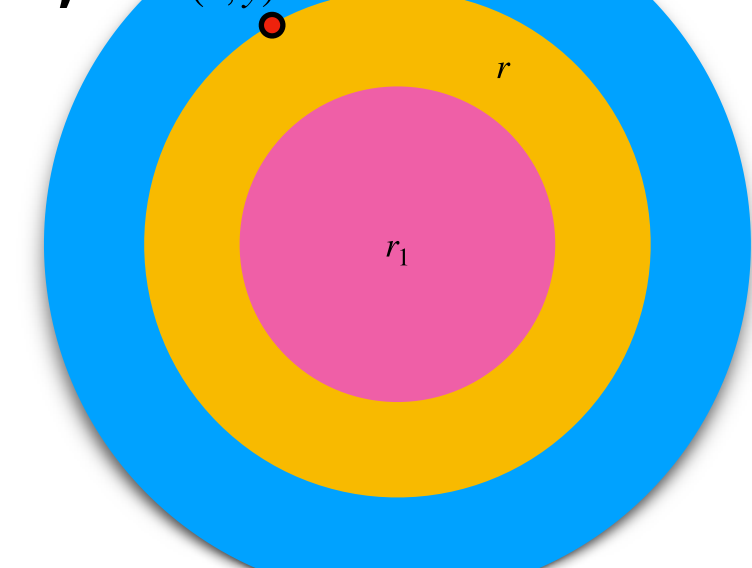


For continuous RV (PDF), we have the same idea, but we integrate instead. The CDF rises more quickly in a region that has a high PDF, just like before.

Example: Sampling between two concentric circles

- Suppose we want to find a function $f(a,b) : \mathbb{R}^2 \to \mathbb{R}^2$ such that if a, b are random numbers in $[0,1]$ then $(x,y) = f(a,b)$ is a random point between two concentric circles, both centered around the origin, with radii r_1, r_2 and $r_1 < r_2$

- We can parameterize (x,y) in polar coordinates, as (Θ, R). So let $R \in [r_1, r_2]$ be the distance of (x,y) from the origin. Let $\Theta \in [0,2\pi]$ be the angle from the horizontal.



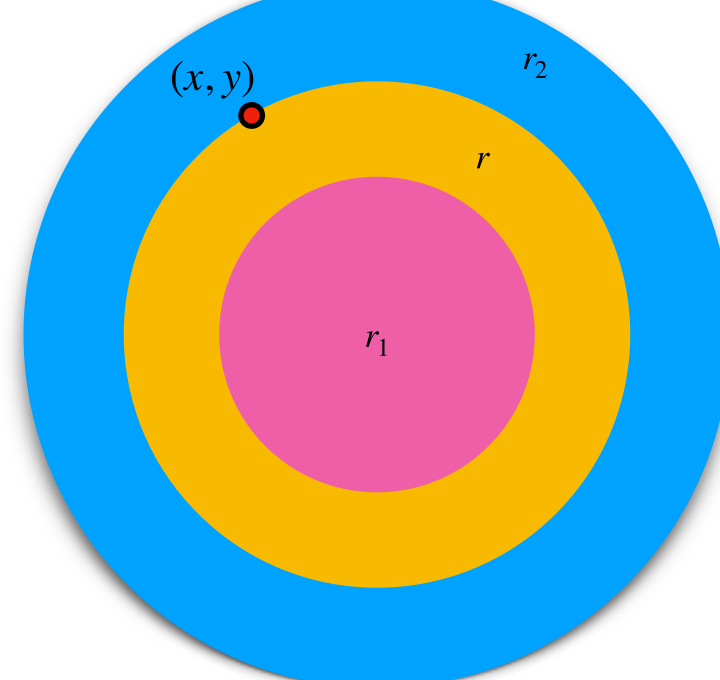
- Want to find the CDF $F_R(r) = P(||R|| \leq r)$
  - Do we need to worry about $F_\Theta(\theta)$? ***No, it's constant wrt θ.***

- The CDF is the ratio of areas: $F_R(r) = \dfrac{r^2 - r_1^2}{r_2^2 - r_1^2}$.

Concentric Circles Continued

Basic Steps:

1. Evaluate the CDF $F_X(x)$ on the distribution
2. Invert the CDF to find $Q_X(q)$. This should satisfy $Q_X(F_X(x)) = x$
3. Generate a random number uniformly between 0 and 1
4. Apply Q_X to the uniform sample

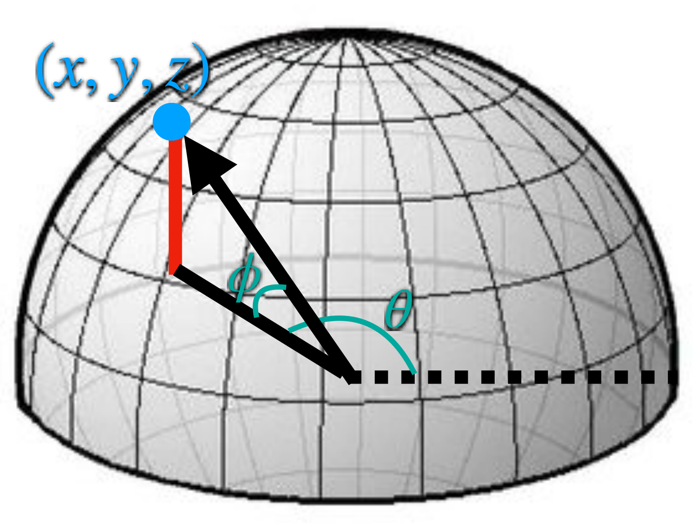


- To find the inverse CDF, solve $q = F_R(r)$ for r:
  - $q = \dfrac{r^2 - r_1^2}{r_2^2 - r_1^2} \Rightarrow \ldots \Rightarrow r = \sqrt{qr_2^2 + (1-q)r_1^2}$
- So, we have the inverse CDF $Q_R(q) = \sqrt{qr_2^2 + (1-y)r_1^2}$
- Say we have a uniformly random variables $X, Y \in [0,1]$. How do we generate the final point (x, y)?
  - First compute polar coords: $(\Theta, R) = (2\pi \cdot X, Q_R(Y))$
  - Finally: $(x, y) = (R\cos\Theta, R\sin\Theta)$

Example: Uniformly Sampling the Hemisphere

- Goal: Given 2 uniform RVs (X, Y) in $[0,1]$, generate a point uniformly randomly on the surface of a unit hemisphere.
- First, just like before, parameterize the hemisphere with two variables (ϕ, θ) which represent latitude and longitude

$$x = \sin\phi\cos\theta$$
$$y = \sin\phi\sin\theta$$
$$z = \cos\phi$$

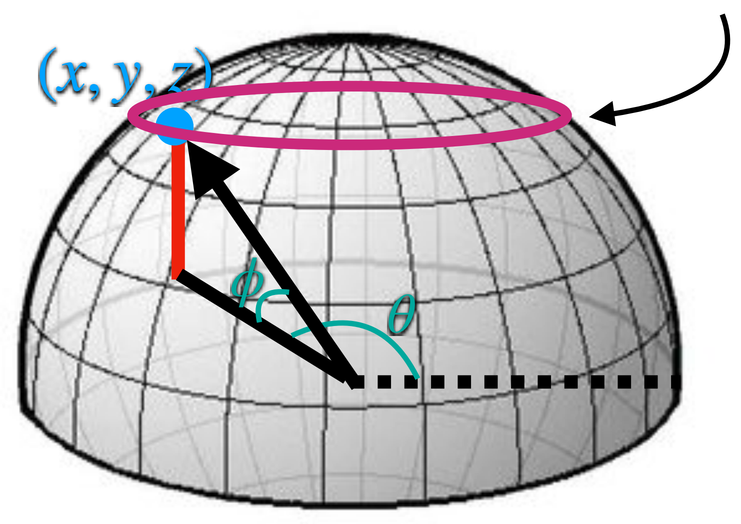


- Sanity check: $x^2 + y^2 + z^2 = 1$
- To derive: First find z from ϕ, then project down onto plane and work in polar coords
(https://math.stackexchange.com/questions/35500/parameterizing-the-upper-hemisphere-of-a-sphere-with-an-upward-pointing-normal)
- As before, PDF is only dependent on ϕ
- Then, what is that PDF $f_\Phi(\phi) = P[\phi \leq \Phi]$?

$$f_\Phi(\phi) = \frac{\text{piece of hemisphere at latitude } \phi}{\text{surface area of hemisphere}}$$

Uniform Hemisphere Sampling Continued

$$x = \sin\phi\cos\theta$$
$$y = \sin\phi\sin\theta$$
$$z = \cos\phi$$



$$f_\Phi(\phi) = \frac{\text{piece of hemisphere at latitude } \phi}{\text{surface area of hemisphere}}$$

- Captures the idea that there are more points at lower latitudes.

$$\begin{aligned} f_\Phi(\phi) &= \frac{2\pi\sqrt{x^2+y^2}}{2\pi} \\ &= \sqrt{\sin^2\phi\cos^2\theta + \sin^2\phi\sin^2\theta} \\ &= \sqrt{\sin^2\phi(\cos^2\theta + \sin^2\theta)} \\ &= \sin\phi \end{aligned}$$

- No dependence on θ as expected
- Then, CDF is $F_\Phi(\phi) = \int_0^\phi \sin\phi d\phi = 1 - \cos\phi$

Sanity check:

$$\int_0^{\pi/2} f_\Phi(\phi)d\phi = 1$$

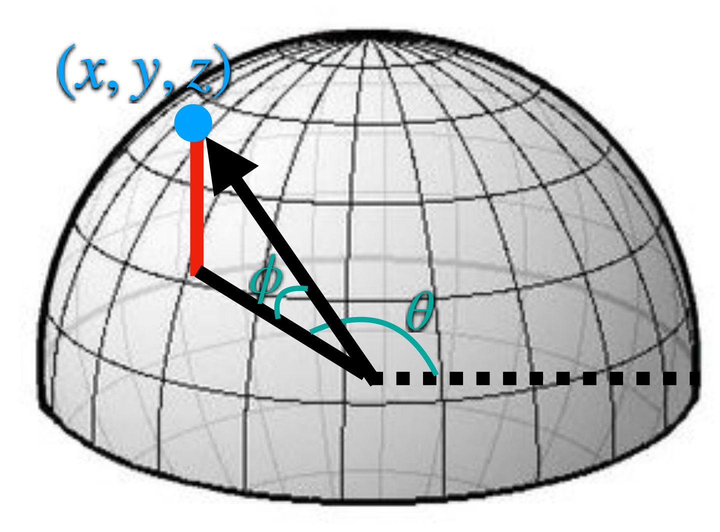
Uniform Hemisphere Sampling Continued

- We now have the CDF $F_\Phi(\phi) = 1 - \cos\phi$
- Inverse CDF is $Q_\Phi(X) = \cos^{-1}(1 - X)$
- How to generate the actual result?
  - Pick a random X, Y in $[0,1]$. Then set $\phi = Q_\phi(X)$, $\theta = 2\pi Y$. Get x, y, z from parameterization of hemisphere.
  - Recall that X is uniform between $[0,1]$, so $Q_\Phi(X) = \cos^{-1}(X)$ works as well.

$$x = \sin\phi\cos\theta$$

$$y = \sin\phi\sin\theta$$

$$z = \cos\phi$$




```
Vector3D UniformHemisphereSampler3D::get_sample() const {
  double Xi1 = (double)(std::rand()) / RAND_MAX;
  double Xi2 = (double)(std::rand()) / RAND_MAX;

  double theta = acos(Xi1);
  double phi = 2.0 * PI * Xi2;

  double xs = sinf(theta) * cosf(phi);
  double ys = sinf(theta) * sinf(phi);
  double zs = cosf(theta);

  return Vector3D(xs, ys, zs);
}
```

Provided in the base code, in sampler.cpp (exactly the same! Well, ϕ and θ are switched)

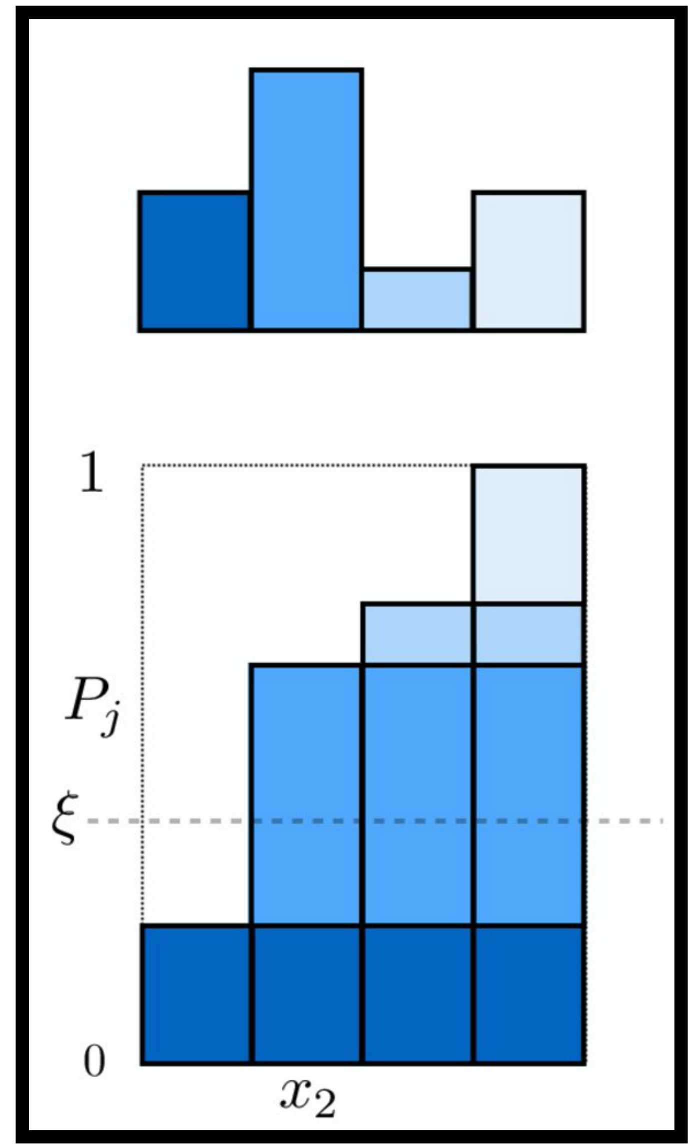
Task 7: Image Based Lighting

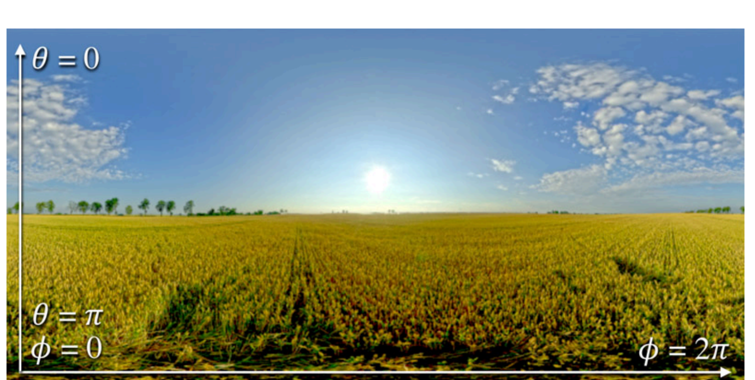
- In task 7, you will be implementing a new type of light, an *Environment Light*. We will use a general technique common in CG called *image based lighting,* which enables us to sample a high-resolution image as our light:

Note: Similar Parameterization to the previous example

Task 7: Importance Sampling environment Maps

- So far we've only talked about inversion sampling with respect to a *uniform* PDF. However, the PDF need not be uniform!
  - Goal: Areas of the environment light with higher radiance should be sampled more often
- Build up a PMF over all *pixels* in the input image, such that the probability of sampling a pixel is proportional to its radiance
  - "Block" analogy: create a stack of blocks, with one block per pixel. The height of each block is the radiance of that pixel. Then select over all heights with uniform probability.





Monte Carlo Methods

Review: Monte Carlo Methods

- Here's one characterization of a Monte Carlo Algorithm you probably have seen:
  - An algorithm A is a $T(n)$-time *Monte Carlo* algorithm with error probability ε if
    - for every input $x \in \Sigma^*$, $A(x)$ gives the wrong answer with probability at most ε, and
    - for every input $x \in \Sigma^*$, $A(x)$ has a worst-case running-time of at most $T(|x|)$.

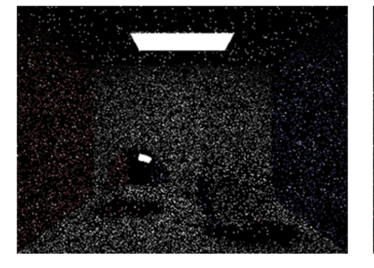
Noise!

Source: 15-251 Recitation Packet

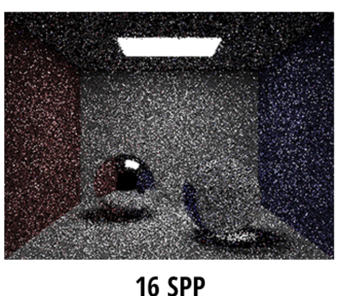
- Main takeaway: We're "gambling" with *correctness*, not running time.
  - Named after the grand casino at Monte Carlo, a ward of Monaco
  - Contrast with a *Las Vegas* algorithm, which gambles with running time

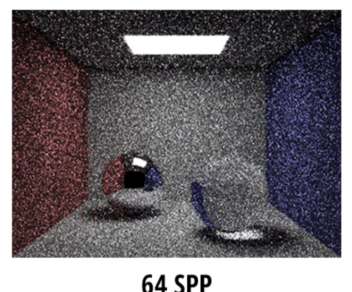
Review: Monte Carlo *Integration*

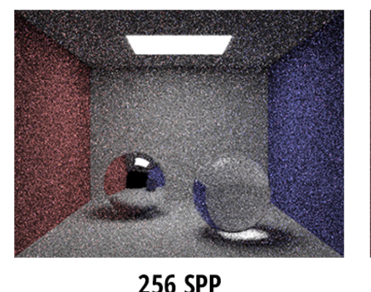
- General class of Monte Carlo algorithms that aim to evaluate a (usually complex) integral numerically
  - Algorithms perspective: A Monte Carlo analog to Reimann Sums (Reimann sums are a *deterministic* algo)
  - Statistical perspective: *Setting up a random event* such that the *Expected Value* is the same as the integral
    - Law of Large Numbers: When taking a large number of samples of a probability distribution, the *average* approaches the *Expected Value* as $\text{samples} \to \infty$.

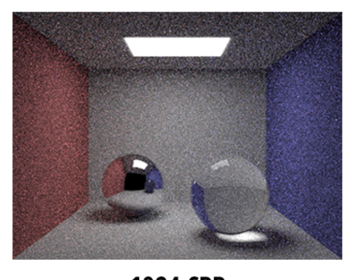

1 SPP
0.1678s

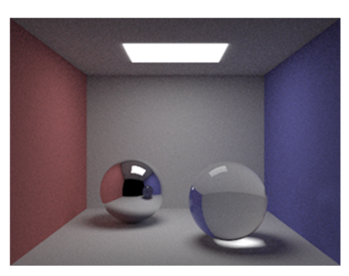
4 SPP
0.6216s


16 SPP
2.7209s


64 SPP
10.7106s


256 SPP
40.9914s


1024 SPP
159.9709s


32769 SPP
6943.6116s

* Images rendered at 400 × 300 on Macbook Pro (2.7GHz Intel Core i5)

Computing π

- How might we compute π using a Monte Carlo algorithm?
  - Area of a circle is πr^2, so if we can find the area of a circle with radius 1, then we have found π
  - Strategy: Use *rejection sampling!*
    1. Draw a Circle with $r = 1$, and a box centered around the circle as tightly as possible ($w = 2$)
    2. Uniformly sample the box (we don't need to know π to do this)
    3. The quantity $4\frac{\text{samples}_{\text{circle}}}{\text{samples}_{\text{total}}}$ is equal to π

Computing π

- But what are we *really doing here?*
  - Beautiful tool lurking in the background:
    "law of the unconscious statistician":
  - $E[F(X)] = \sum F(X)P_X(X)$ (for discrete RV)

    $E[F(X)] = \int F(X)P_X(X)dX$ (for continuous RV)

    for random variable X and (deterministic) function F

Computing π

Law of unconscious statistician:

$$E[F(X)] = \int F(X)P_X(X)dX$$

For computing π, we have

$$X = \text{random point in the box}$$

$$F(X) = \begin{cases} 1 & \text{if } X \text{ is in the circle} \\ 0 & \text{otherwise} \end{cases}$$

$$P_X(X) = \frac{1}{A_{\text{box}}} = \frac{1}{4}$$

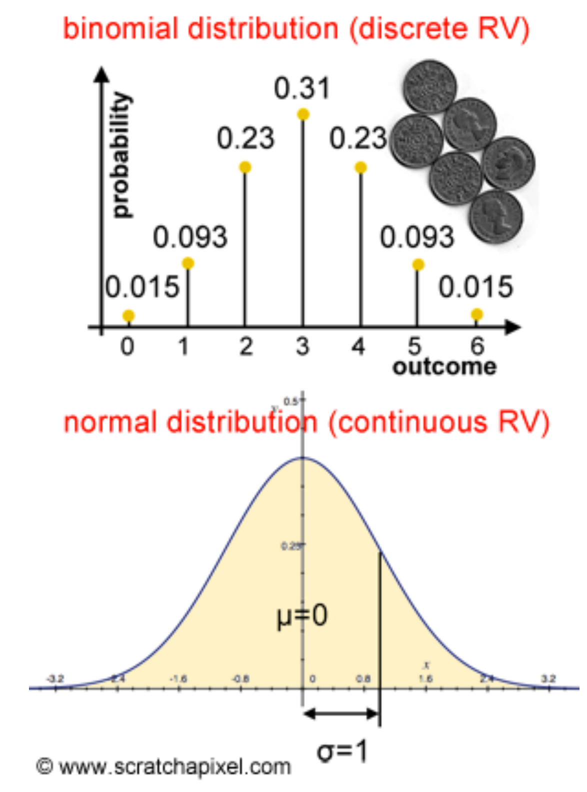
So $E[F(X)] = \int_B \frac{F(X)}{A_{\text{box}}}dX = \frac{1}{4}\int_B F(X)dX = \frac{1}{4}A_{\text{circle}} = \pi/4$

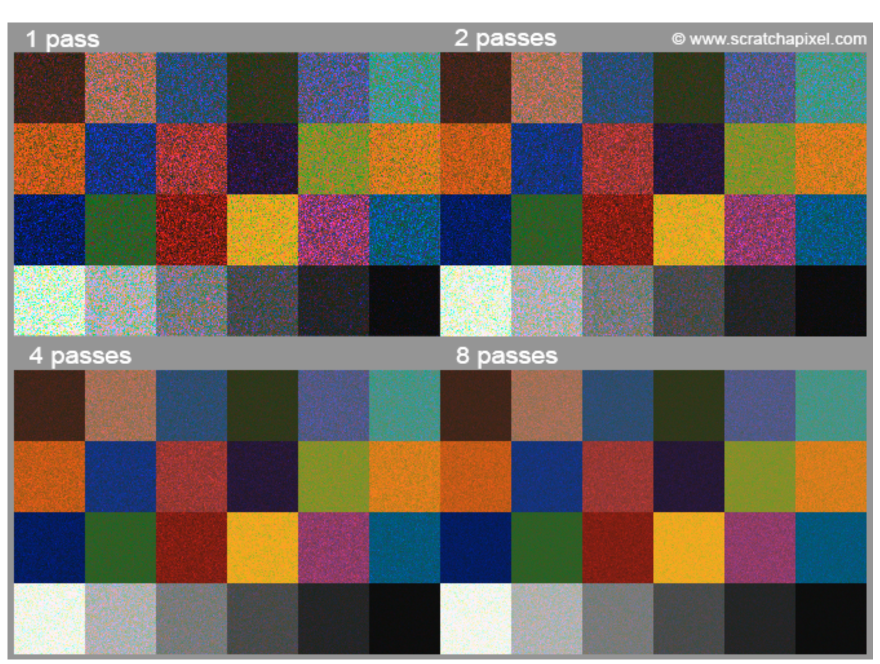
Recall Law of Large Numbers:

$$\Rightarrow E[F(X)] = \frac{\text{samples}_{\text{circle}}}{\text{samples}_{\text{total}}} \text{ as } \text{samples}_{\text{total}} \to \infty$$

More about Monte Carlo Algorithms and Sampling

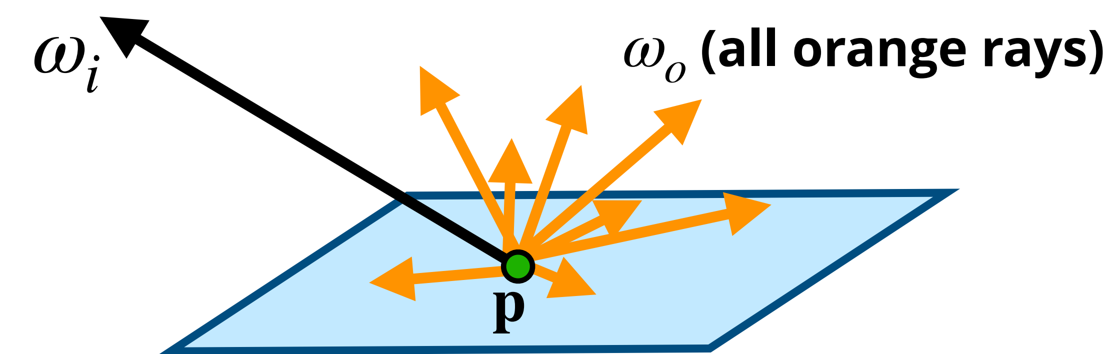
- Scratchapixel has an excellent collection of resources on this topic.
- Scratchapixel, *Mathematical Foundations of Monte Carlo Methods*
  - https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/monte-carlo-methods-mathematical-foundations/quick-introduction-to-monte-carlo-methods
  - Random Variables, PDF/CDF, Expected Value/ Variance, Law of Unconscious Statistician, Inversion Sampling
- Scratchapixel, *Monte Carlo Methods in Practice*
  - https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/monte-carlo-methods-in-practice/monte-carlo-methods
  - Monte Carlo Simulation & Integration, Variance Reduction, Practical Examples / Source Code





The Rendering Equation

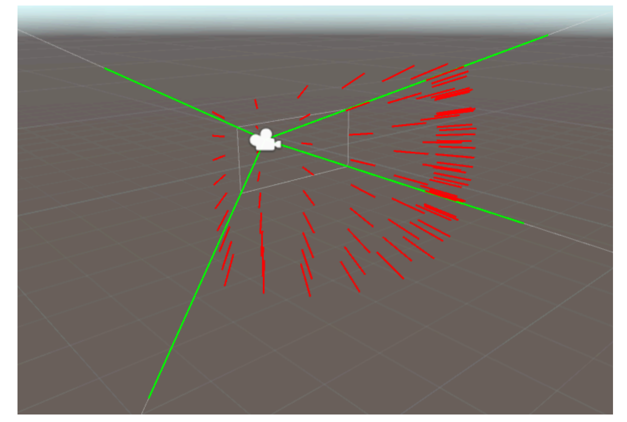
The Rendering Equation



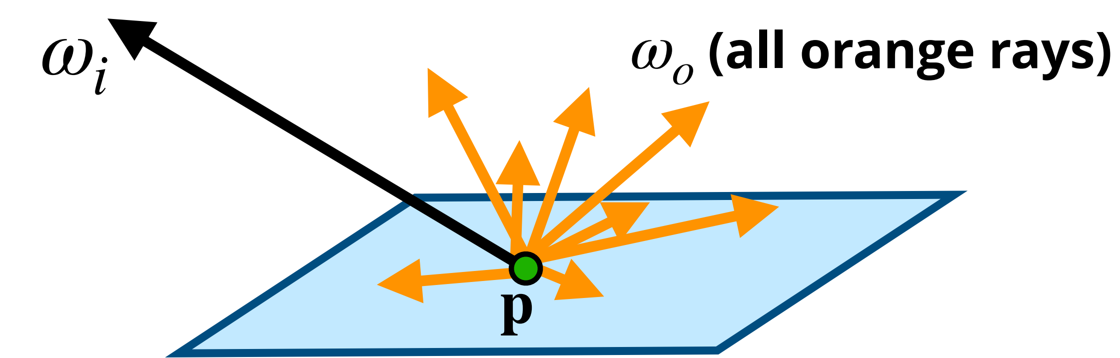
$$L_o(\mathbf{p}, \omega_o) = L_e(\mathbf{p}, \omega_o) + \int_{H^2} f_r(\mathbf{p}, \omega_i \rightarrow \omega_o)\, L_i(\mathbf{p}, \omega_i)\, \cos\theta d\omega_i$$

Output Radiance — *BRDF* — *Incoming Radiance*

- Broadly, your task is to implement a Monte Carlo estimator for the rendering equation.
- The basics: Your inputs are ω_o and $\mathbf{p}$, the <u>outgoing</u> ray direction and the point of intersection.
  - *<u>Common mistake</u>:* Remember that rays start from the *camera!* You implemented this in Part 1 :)



The Rendering Equation



$$L_o(\mathbf{p}, \omega_o) = L_e(\mathbf{p}, \omega_o) + \int_{H^2} f_r(\mathbf{p}, \omega_i \rightarrow \omega_o)\, L_i(\mathbf{p}, \omega_i)\, \cos\theta d\omega_i$$

Output Radiance — *BRDF* — *Incoming Radiance*

- *Important*: In a pathtracer, rays don't start at the light, they start at a camera! How is this possible?
  - Helmholtz Reciprocity: $f_r(\mathbf{p}, \omega_i \rightarrow \omega_o) = f_r(\mathbf{p}, \omega_o \rightarrow \omega_i)$ for most surfaces™ that you find in nature.
  - Counterexample: *Lenticular prints* that change depending on view direction

Source: https://gph.is/13Yglvr

Lighting via "Direct Lighting" (Task 4)

Sample from the rays that hit the light

ω_o (all orange rays)

Area Light

$\mathbf{p}'$

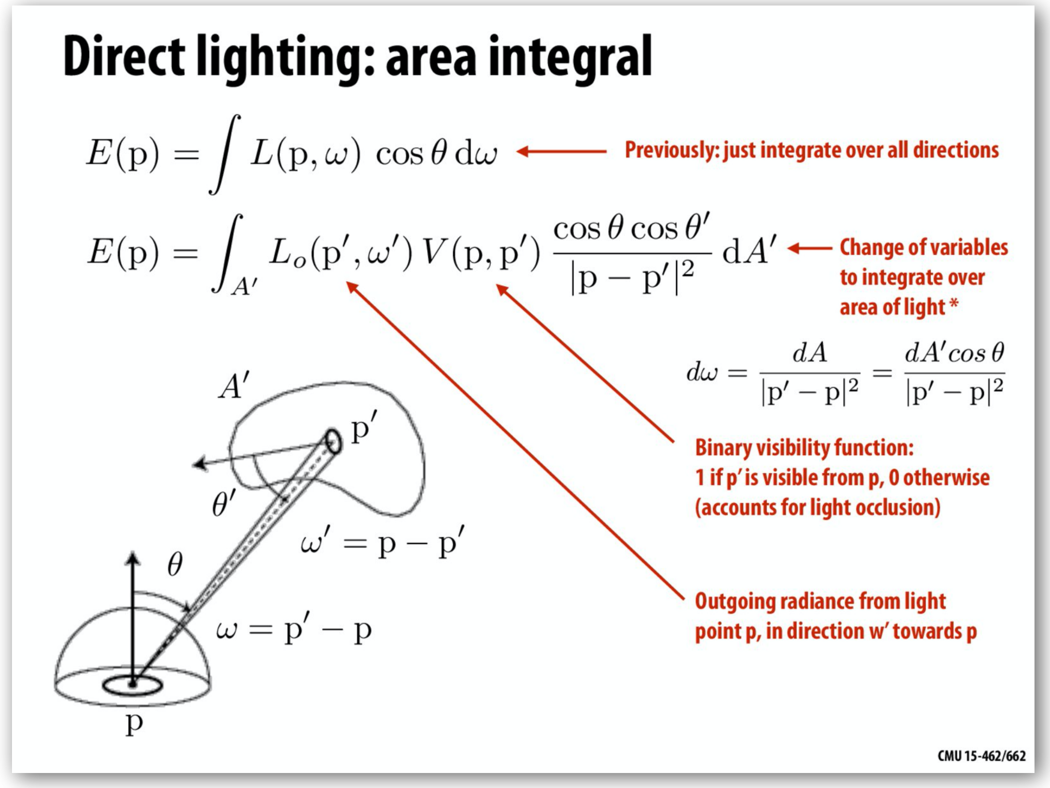
ω_i

$\mathbf{p}$

$$L_o(\mathbf{p},\omega_o) = L_e(\mathbf{p},\omega_o) + \int_{H^2} f_r(\mathbf{p},\omega_i \rightarrow \omega_o)\, L_i(\mathbf{p},\omega_i)\, \cos\theta d\omega_i$$

Direct Lighting Added Here

- In task 4, you added shadows to an existing implementation of *direct lighting*
  - *Idea: Sample/integrate over the light's surface: set $L_i(\mathbf{p},\omega_i)$ to the radiance of the light, accounting for change of terms in integral*



Lighting via Emitted Radiance

Emissive Surface

$\mathbf{p}'$

ω_i

ω_o (all orange rays)

$\mathbf{p}$

$$L_o(\mathbf{p}, \omega_o) = L_e(\mathbf{p}, \omega_o) + \int_{H^2} f_r(\mathbf{p}, \omega_i \to \omega_o)\, L_i(\mathbf{p}, \omega_i)\, \cos\theta d\omega_i$$

Emitted Radiance

- L_e, the *emitted radiance*, is nonzero when the surface itself is a light.
- Above, $L_e(\mathbf{p}, \omega_\mathbf{o}) = 0$ and $L_e(\mathbf{p}', -\omega_i) > 0$
- No surfaces in the provided Scotty3D demo files use an emissive material — still, try to support emission (it is possible in Scotty3D).
  - Useful if you want to both reflect *and* emit light

The Rendering Equation

$$L_o(\mathbf{p},\omega_o) = L_e(\mathbf{p},\omega_o) + \int_{H^2} f_r(\mathbf{p},\omega_i \to \omega_o)\, L_i(\mathbf{p},\omega_i)\ \cos\theta d\omega_i$$

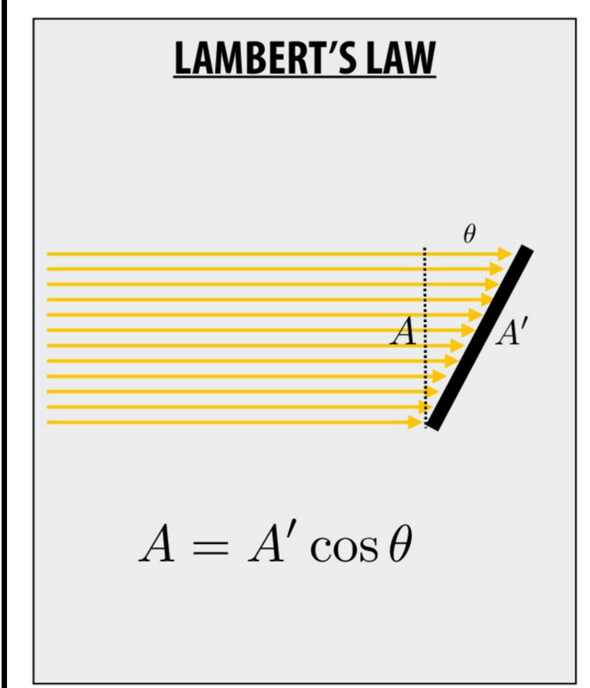
What's this??

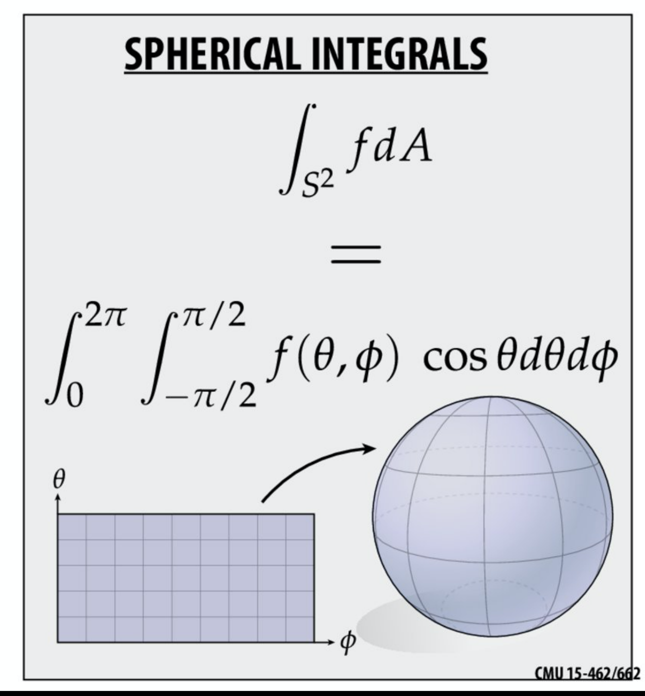
- Note: This cosine is due to parameterization of sphere
- Note: Concerns *solid angles*, not *areas* (Lambert's law)

https://spie.org/publications/fg11_p04_solid_angle_and_projected

Aside: A Tale of Two Cosines

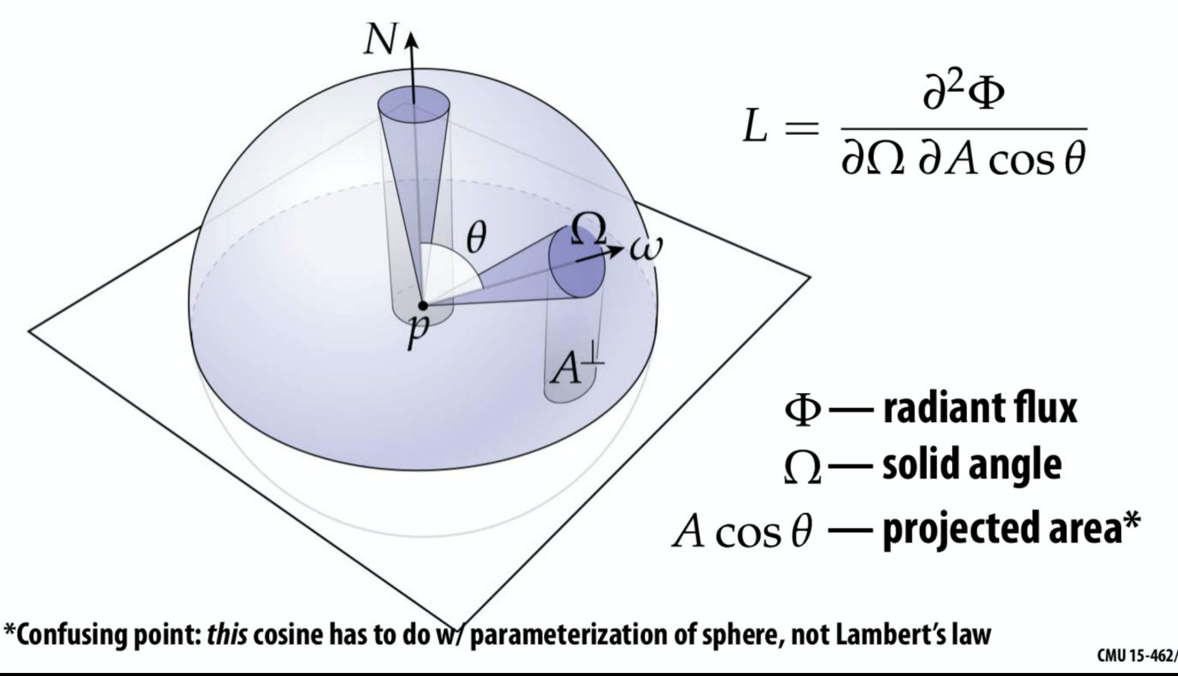
- Confusing point first time you study photorealistic rendering: "cos θ" shows up for two completely unrelated reasons





Recap: What is *radiance?*

- Radiance at point p in direction N is radiant energy ("#hits") per unit time, per solid angle, per unit area perpendicular to N.



*Confusing point: *this* cosine has to do w/ parameterization of sphere, not Lambert's law

CMU 15-462/662

“Delta” BRDFs

$$L_o(\mathbf{p}, \omega_o) = L_e(\mathbf{p}, \omega_o) + \int_{H^2} f_r(\mathbf{p}, \omega_i \to \omega_o)\, L_i(\mathbf{p}, \omega_i)\, \cos\theta d\omega_i$$

- For a Mirror, we could define the BRDF as:

$$f_i(\mathbf{p}, \omega_i \to \omega_o) = \begin{cases} 1/\cos(\theta_i) & \text{if } \omega_o = \text{reflect}(\omega_i) \\ 0 & \text{otherwise} \end{cases}$$

- Written in lecture as a *Dirac delta function* $\dfrac{\delta(\cos\theta_i - \cos\theta_o)}{\cos\theta_i}$

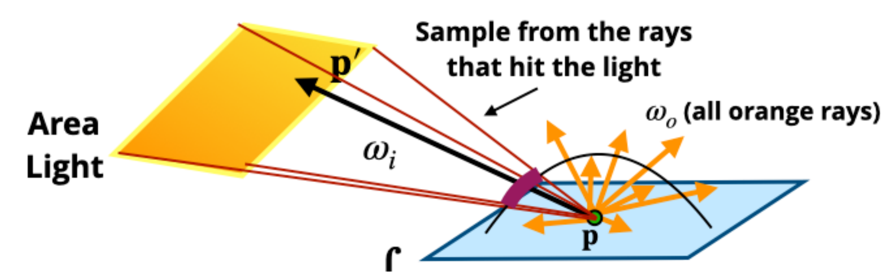
  where θ_i, θ_o are the angles between ω_i, ω_o and the normal

- Wait a minute… ***how do we sample that?!***
  - Our sampler would have to perfectly guess the incoming vector ω_i up to floating point precision
  - Instead we manually provide the reflected vector
- These “Delta BRDFs” also don’t use direct lighting.

Rendering Equation in Direct Lighting

```
if (!isect.bsdf->is_delta()) {
  Vector3D dir_to_light;
  float dist_to_light;
  float pr;

  // ### Estimate direct lighting integral
  for (SceneLight* light : scene->lights) {

    // no need to take multiple samples from a point/directional source
    int num_light_samples = light->is_delta_light() ? 1 : ns_area_light;

    // integrate light over the hemisphere about the normal
    for (int i = 0; i < num_light_samples; i++) {

      // returns a vector 'dir_to_light' that is a direction from
      // point hit_p to the point on the light source.  It also returns
      // the distance from point x to this point on the light source.
      // (pr is the probability of randomly selecting the random
      // sample point on the light source -- more on this in part 2)
      const Spectrum& light_L = light->sample_L(hit_p, &dir_to_light, &dist_to_light, &pr);

      // convert direction into coordinate space of the surface, where
      // the surface normal is [0 0 1]
      const Vector3D& w_in = w2o * dir_to_light;
      if (w_in.z < 0) continue;

        // note that computing dot(n,w_in) is simple
      // in surface coordinates since the normal is (0,0,1)
      double cos_theta = w_in.z;

      // evaluate surface bsdf
      const Spectrum& f = isect.bsdf->f(w_out, w_in);

      // TODO (PathTracer):
      // (Task 4) Construct a shadow ray and compute whether the intersected surface is
      // in shadow. Only accumulate light if not in shadow.
      L_out += (cos_theta / (num_light_samples * pr)) * f * light_L;
    }
  }
}
```

(No delta surfaces)



Generates a random ray dir_to_light from $\mathbf{p}$ to somewhere on the surface of the light.

Need to account for the probability of hitting this patch of the area light, which would bias sampling

Law of unconscious statistician:

$$E[F(X)] = \int F(X)P_X(X)dX$$

$$L_o(\mathbf{p}, \omega_o) = L_e(\mathbf{p}, \omega_o) + \int_{H^2} f_r(\mathbf{p}, \omega_i \rightarrow \omega_o)\, L_i(\mathbf{p}, \omega_i)\, \cos\theta d\omega_i$$

Writing BRDFs in Scotty3D (Task 5, 6)

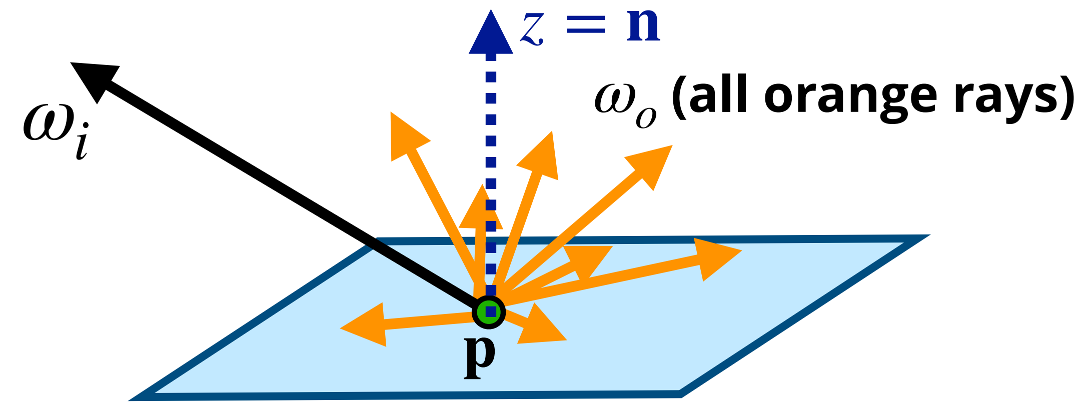
- There are actually two functions to implement each BRDF:

```
Spectrum DiffuseBSDF::f(const Vector3D& wo, const Vector3D& wi) {
  return albedo * (1.0 / PI);
}

Spectrum DiffuseBSDF::sample_f(const Vector3D& wo, Vector3D* wi, float* pdf) {
  // TODO (PathTracer):
  // Implement DiffuseBSDF
  return Spectrum();
}
```

  - `Spectrum f(const Vector3D &wo, const Vector3D &wi)`
    - Input: Ray directions ω_i, ω_o
    - Output: Classical BRDF $f(\omega_i \to \omega_o)$ that returns radiance
    - Called <u>only</u> during direct lighting
  - `Spectrum sample_f(const Vector3D &wo, Vector3D &wi, float* pdf)`
    - Input: <u>Only</u> outgoing ray direction ω_o
    - Output: Generate a random sample ω_i, its probability (in the variable pdf), and return $f(\omega_i \to \omega_o)$
    - You should call this when computing indirect lighting (Task 5)

Writing BRDFs in Scotty3D (Tasks 5,6)

- Common point of confusion: in f() and sample_f(), ω_o and ω_i both point *away* from the intersection point $\mathbf{p}$. Additionally, the *outward* normal is defined to be the z-axis.



- Additionally, in the case of refraction, the z coordinate of ω_i will be negative (z is defined as the *outward* normal)

```
// make a coordinate system for a hit point
// with N aligned with the Z direction.
Matrix3x3 o2w;
make_coord_space(o2w, isect.n);
Matrix3x3 w2o = o2w.T();
```

Towards the top of trace_ray in pathtracer.cpp

Task 5,6 Hints and Tips

- Diffuse BRDF
  - Think carefully about the pdf value.
  - You may optionally implement a Cosine Weighted sampler, which should be straightforward after today.
  - Remember to use the DiffuseBSDF::albedo variable.
- Mirror BRDF
  - Remember to use the MirrorBSDF::reflectance variable
- Glass BRDF
  - Remember to use GlassBSDF::reflectance and GlassBSDF::transmittance
  - The PDF is not 1.0
  - I *highly* recommend using Schlick's approximation (there is a link on the wiki) which is easier to implement and more robust way to compute the fresnel term F_r

Task 6 Grading

- For Task 6, I will be running your code remotely on the Andrew Unix cluster to render the Cornell box with your pathtracer at demanding settings:

```
./scotty3d -w output.png -t 8 -m 4 -s 1024 -d 800x600 ../media/pathtracer/CBspheres.dae
```

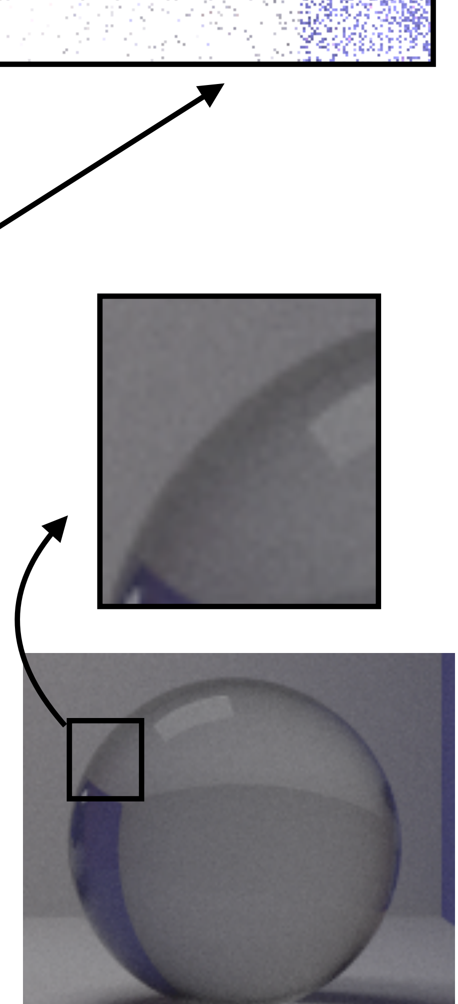
- 8 threads, max 4 light bounces, 1024 samples, 800x600
- I am giving each submission 30 minutes to render (more than double our reference). If your submission doesn't finish in time you will receive a penalty.
- It is important that you test your code! New changes to the base code allow you to remotely render via SSH
  - See Piazza

Performance Improvement Suggestions

- *Use a Profiler!* This will save you loads of headache and can help diagnose slow code, at the level of an individual function call. Come to OH for help, or intro to C++ lecture
  - Most serious performance issues have to do with something silly, like not passing by reference
- Avoid trig functions, use dot and cross products if possible
- Avoid very recursive code (ex: BVH traversal), and convert to a sequential algorithm
- Use the helper functions at the top of bsdf.h
- Remember: *Millions* of rays will be cast in a large render, so make sure your code is as fast as possible with respect to each raycast.

Correctness Improvement Suggestions

- Don't clamp radiances, ever. Your answer will be wrong
  - A common reason for this is that you are seeing white "speckles" in your renders. Try turning up the sample rate first, if that doesn't fix it you have a math error.
  - Note that a Spectrum can have color values > 1
- Make sure that your PDF integrates to 1. Otherwise your rays may gain energy on each bounce and you get this:
- Manage your cosines: if the edges of some objects are too dark, then you have missed a divide by cosine somewhere
- Banging your head against code will not prove fruitful, especially for Task 6. You may want to work out the math on pen and paper.



Questions?