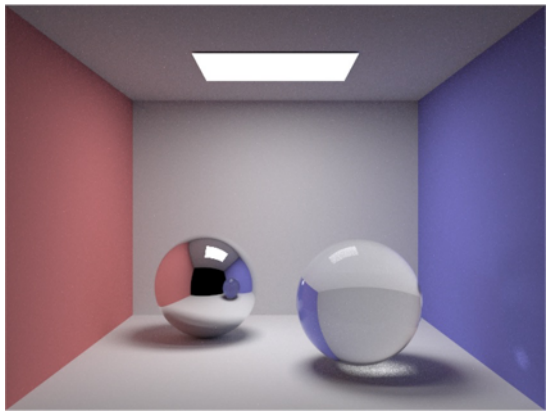
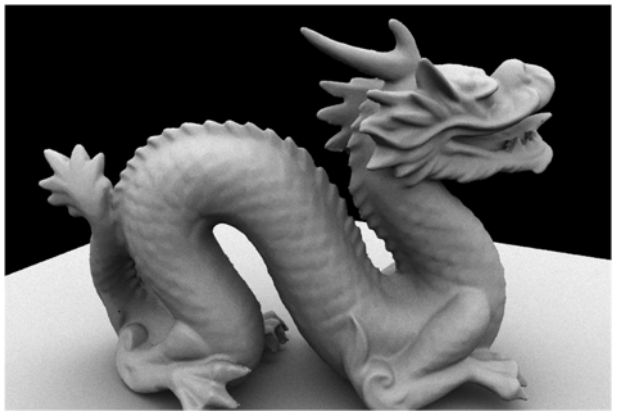
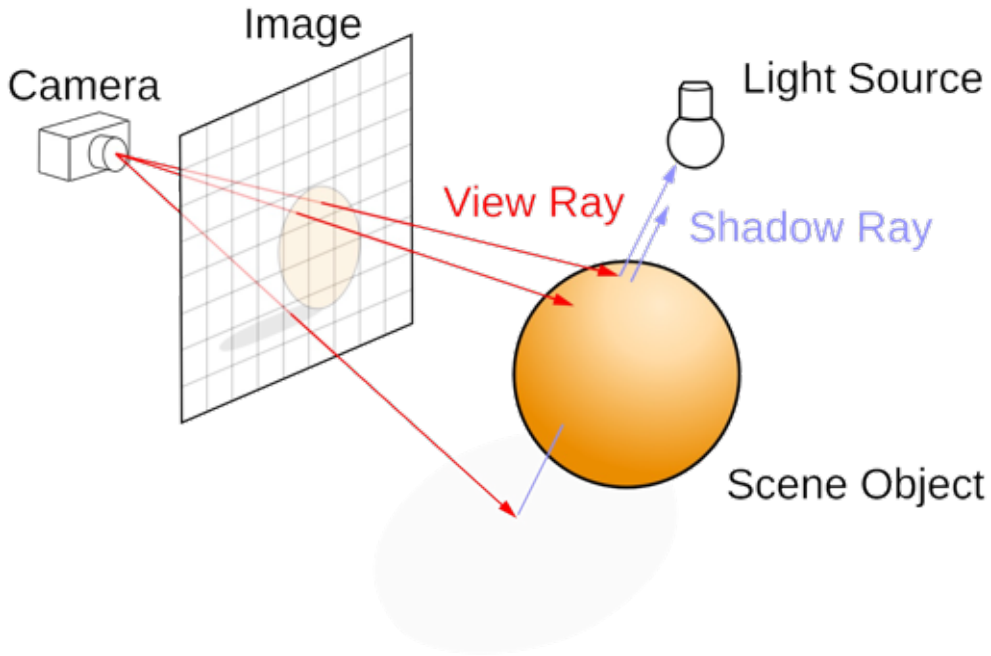
PathTracer

Finally you can put this on your resume


Goal of PathTracer Lab:

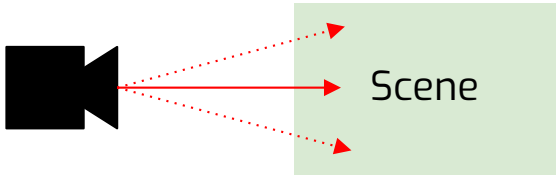
Implement an end-to-end path tracer pipeline!



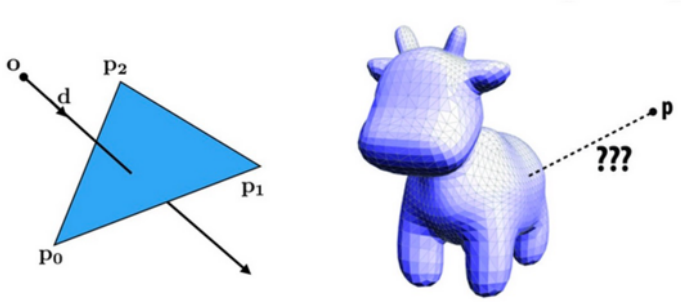
https://en.wikipedia.org/wiki/Ray_tracing_(graphics)

Overview of Tasks (Part I)

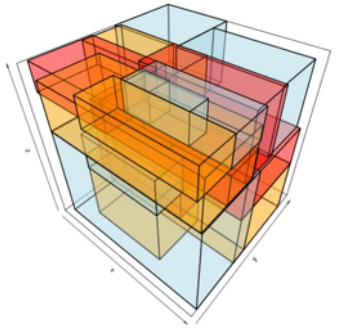
Task 1: Camera Rays



Task 2: Intersecting Primitives



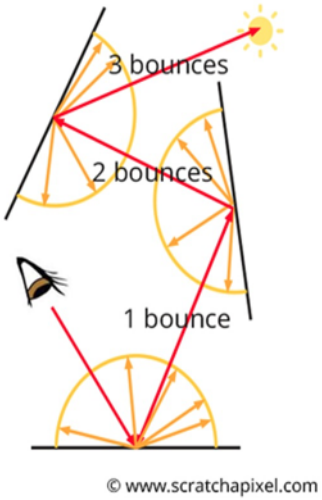
Task 3*: BVH



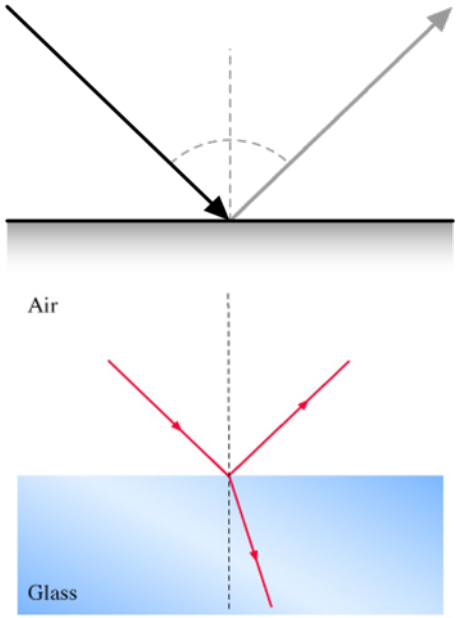
https://github.com/davidmoten/rtree-multi

Overview of Tasks (Part II)

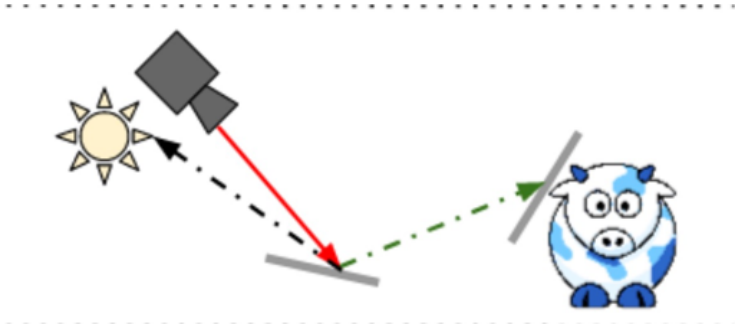
Task 4: Path Tracing



Task 5: Materials



Task 6: Direct Lighting



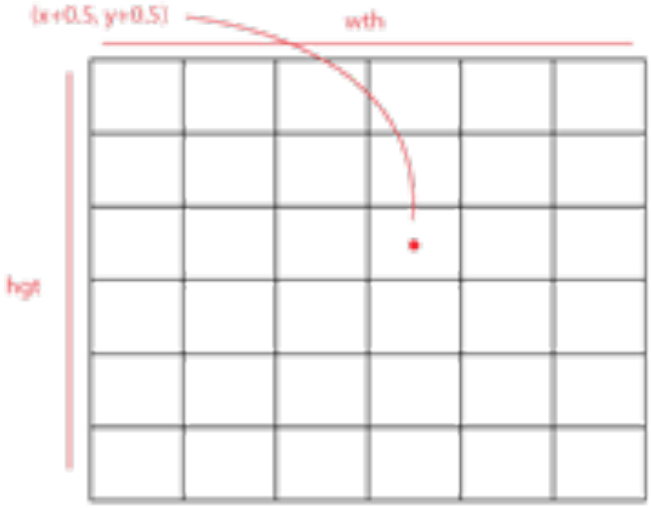
Task 7: Environment Light

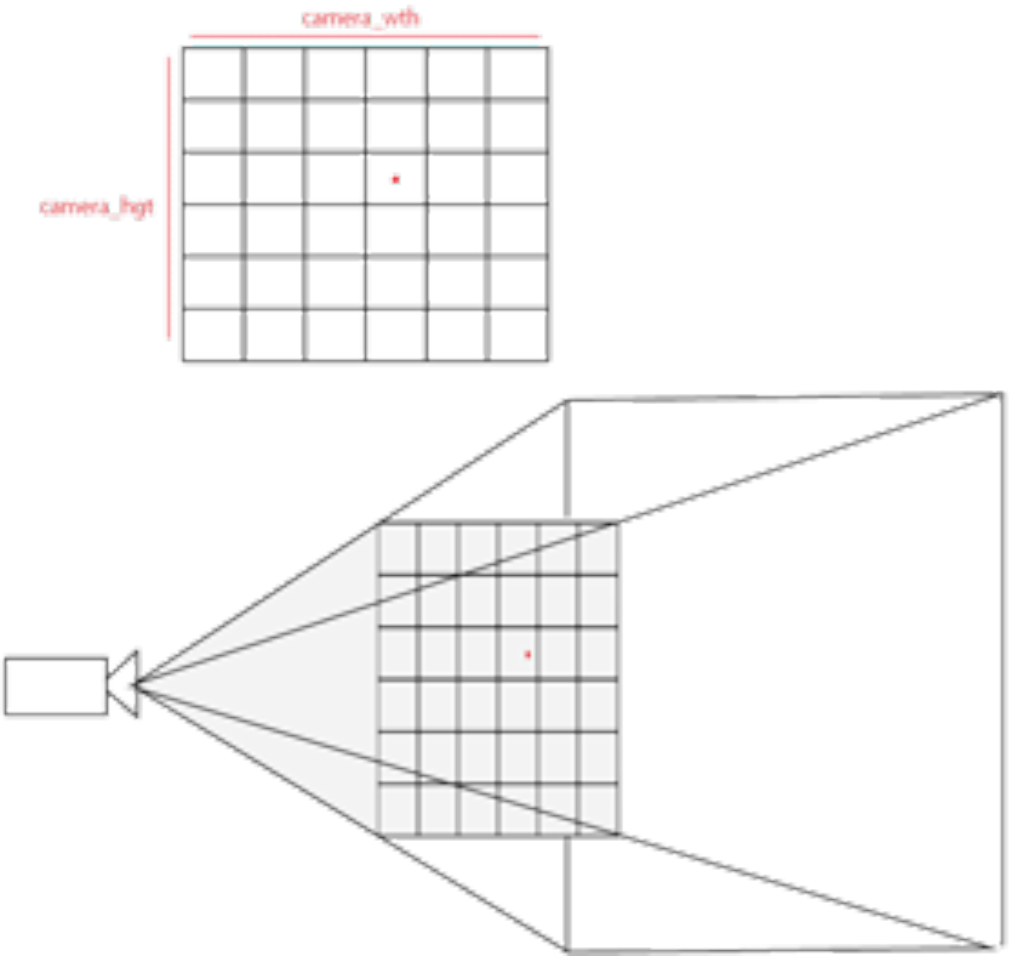
https://en.wikipedia.org/wiki/Bidirectional_reflectance_distribution_function

Part I

Pathtracer::trace_pixel

Task 1: Camera Rays

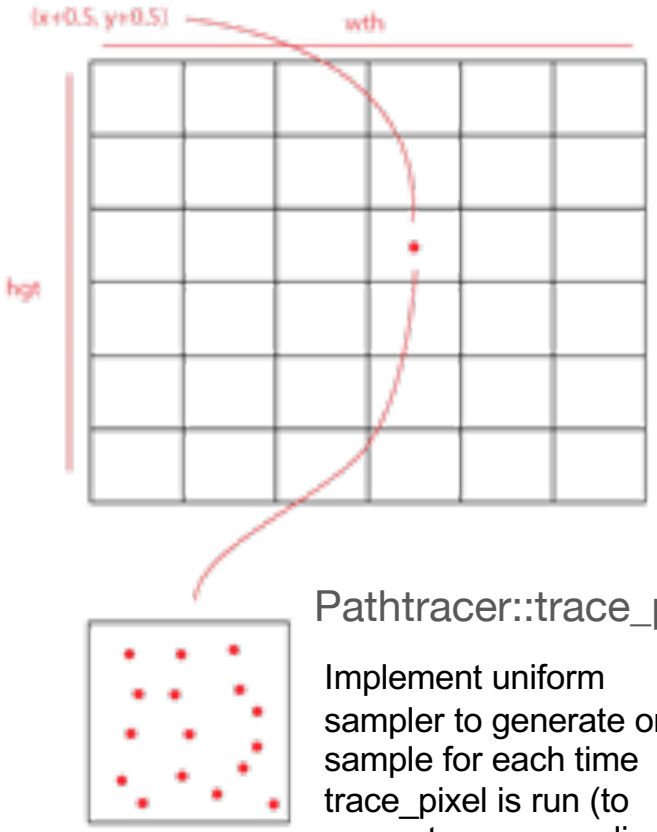




- Normalize x, y coordinates (using half-int coordinates).
- Pass resulting coordinates to generate_r ay().
- generate_ray() will return a Ray object that you can use to get a color by calling trace()

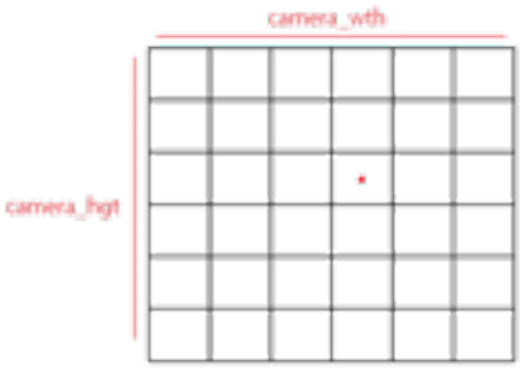
Camera::generate_ray

Task 1: Camera Rays

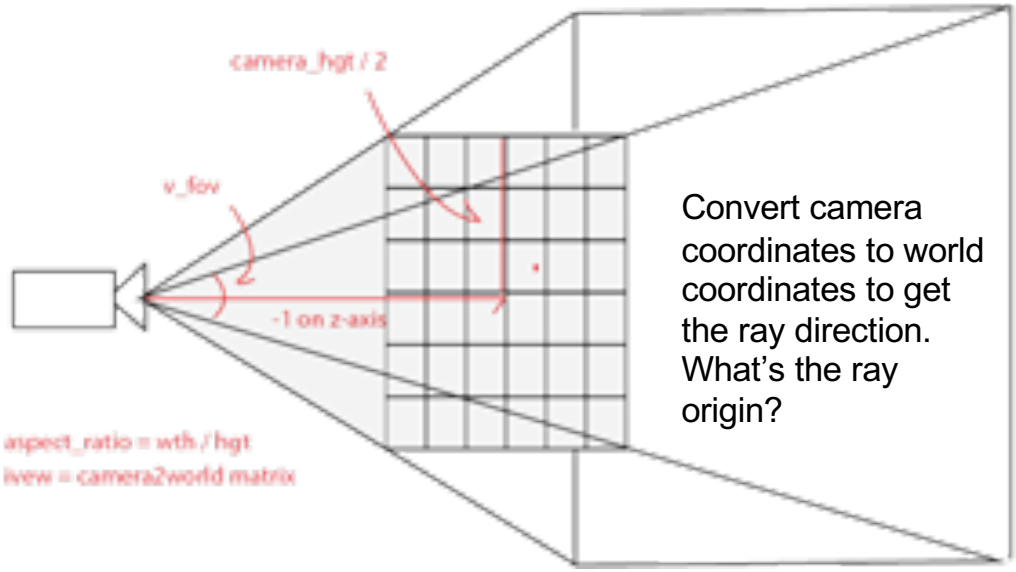


Pathtracer::trace_pixel

Implement uniform sampler to generate one sample for each time trace_pixel is run (to support supersampling)



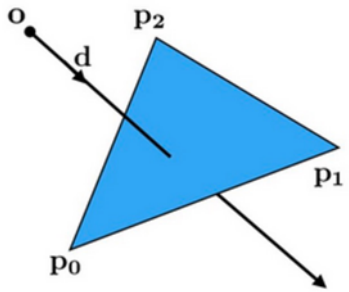
We've given you v_fov, the aspect_ratio, and z = -1. How do we get camera_hgt? camera_wth? Answer: trig!

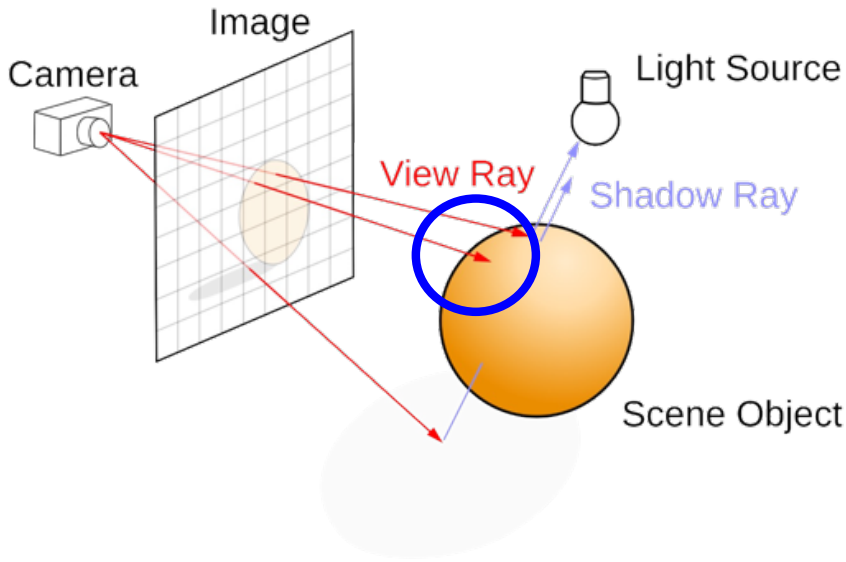


Convert camera coordinates to world coordinates to get the ray direction. What's the ray origin?

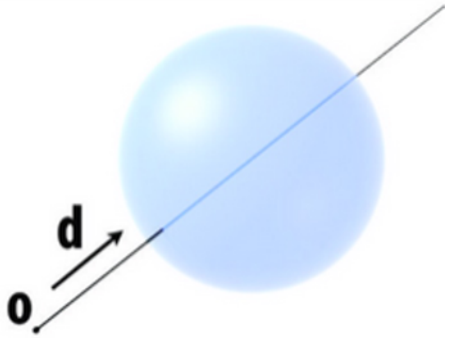
Task 2: Intersecting Primitives

Step 1: Intersecting Triangles



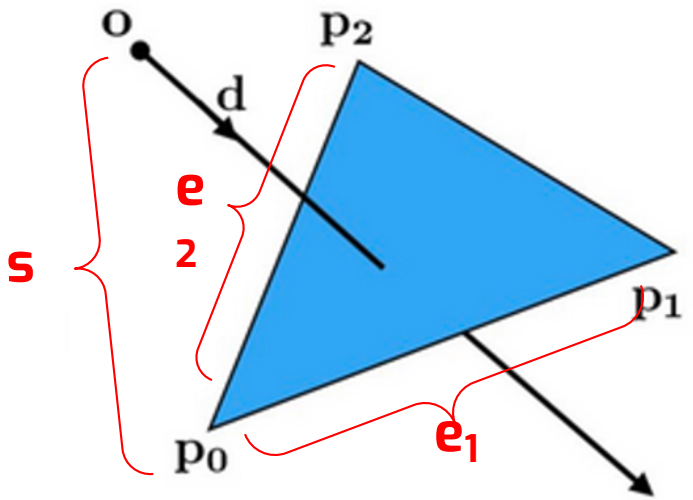


Step 2: Intersecting Spheres



Task 2: Intersecting Primitives

Step 1: Intersecting Triangles



Trace Triangle::hit(const Ray &ray);

in tri_mesh.cpp

Möller-Trumbore algorithm

A fast ray triangle intersection algorithm

Takes advantage of barycentric coordinates parametrization of the intersection point P

$$e_1 = p_1 - p_0$$
$$e_2 = p_2 - p_0$$
$$s = o - p_0$$

System to solve:

Barycentric coordinates of hit point

$$\begin{bmatrix} e_1 & e_2 & -d \end{bmatrix} \begin{bmatrix} u \\ v \\ t \end{bmatrix} = s$$

Solve using *Cramer's Rule*

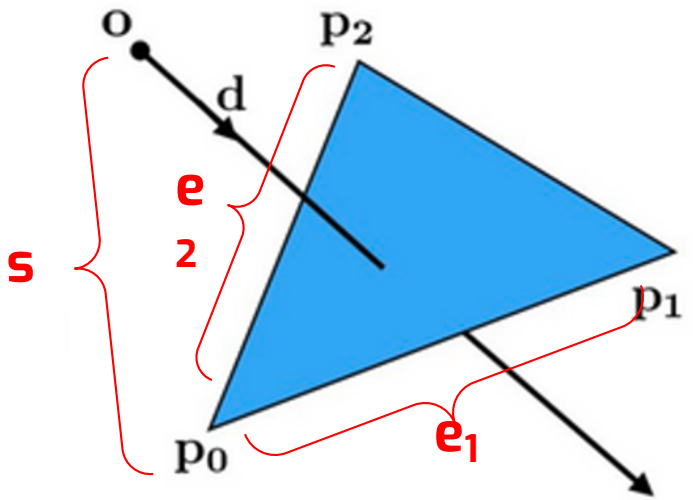
$$\begin{bmatrix} u \\ v \\ t \end{bmatrix} = \frac{1}{(e_1 \times d) \cdot e_2} \begin{bmatrix} -(s \times e_2) \cdot d \\ (e_1 \times d) \cdot s \\ -(s \times e_2) \cdot e_1 \end{bmatrix}$$

Motivation is on Scotty3D

Task 2: Intersecting Primitives

```
ret.origin = ray.point;
ret.hit = false;
ret.distance = 0.0f;
ret.position = Vec3{};
ret.normal = Vec3{};
return ret;
```

Step 1: Intersecting Triangles



Trace Triangle::hit(const Ray &ray);

in tri_mesh.cpp

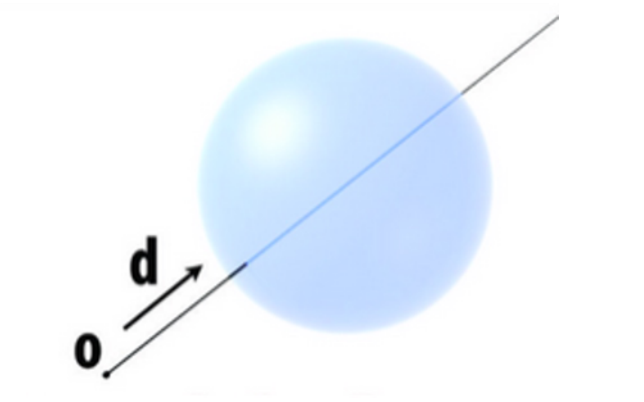
1. Inside Outside Triangle Test using Barycentric coordinates (`ret.hit` = True/False)
   a. Populate `ret.hit`, `ret.distance`
   b. Respect ray.dist_bounds

1. Fill in `ret.position`
   a. `ray.at(t)` may be helpful

1. Fill in `ret.normal`
   c. Use Barycentric coordinates to interpolate the normal at intersection point
   d. `Tri_Mesh_Vert.normal`

Task 2: Intersecting Primitives

Step 2: Intersecting Spheres



Trace Sphere::hit(const Ray &ray);

in shapes.cpp

3D Point Vector [x,y,z]

$$f(\mathbf{x}) = |\mathbf{x}|^2 - 1$$

Equation for unit sphere

$$f(\mathbf{r}(t)) = |\mathbf{o} + t\mathbf{d}|^2 - 1$$

Substitute ray vector

$$\underbrace{|\mathbf{d}|^2}_{a} t^2 + \underbrace{2(\mathbf{o}\cdot\mathbf{d})}_{b} t + \underbrace{|\mathbf{o}|^2 - 1}_{c} = 0$$

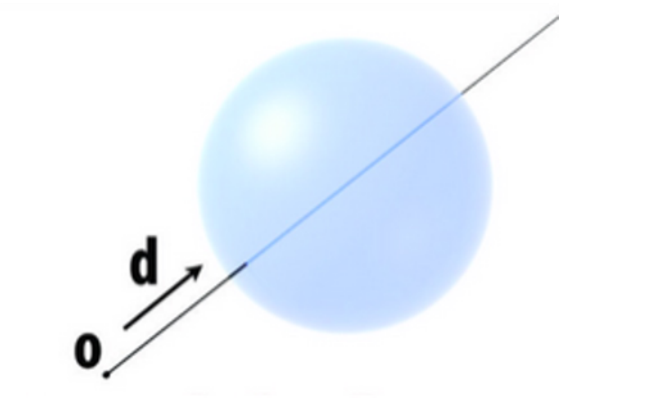
Rearrange equation,
set f(x) = 0

$$t = \boxed{-\mathbf{o}\cdot\mathbf{d} \pm \sqrt{(\mathbf{o}\cdot\mathbf{d})^2 - |\mathbf{o}|^2 + 1}}$$

Two solutions:
one going in, one going out

Task 2: Intersecting Primitives

Step 2: Intersecting Spheres



Trace Sphere::hit(const Ray &ray);

in shapes.cpp

```
ret.origin = ray.point;
ret.hit = false;
ret.distance = 0.0f;
ret.position = Vec3{};
ret.normal = Vec3{};
return ret;
```

1. Solve Quadratic to get t (`ret.hit` = True/False)
   a. Populate `ret.hit` and `ret.distance`
   b. Respect ray.dist_bounds
   c. Warning: what if ray.dir is not unit length?

1. Fill in `ret.position`
   a. `ray.at(t)` may be helpful

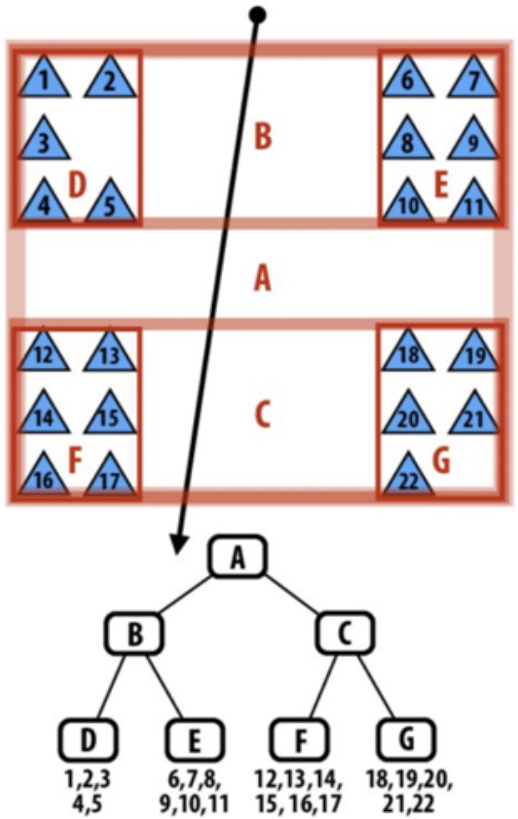
1. Fill in `ret.normal`
   a. Note: You can assume a sphere is centered at the origin because we are in local space

Task 3*: BVH

Bounding Volume Hierarchy

- Spatial Hierarchy
- B(ounding) Boxes + Primitives
- Functions like a tree

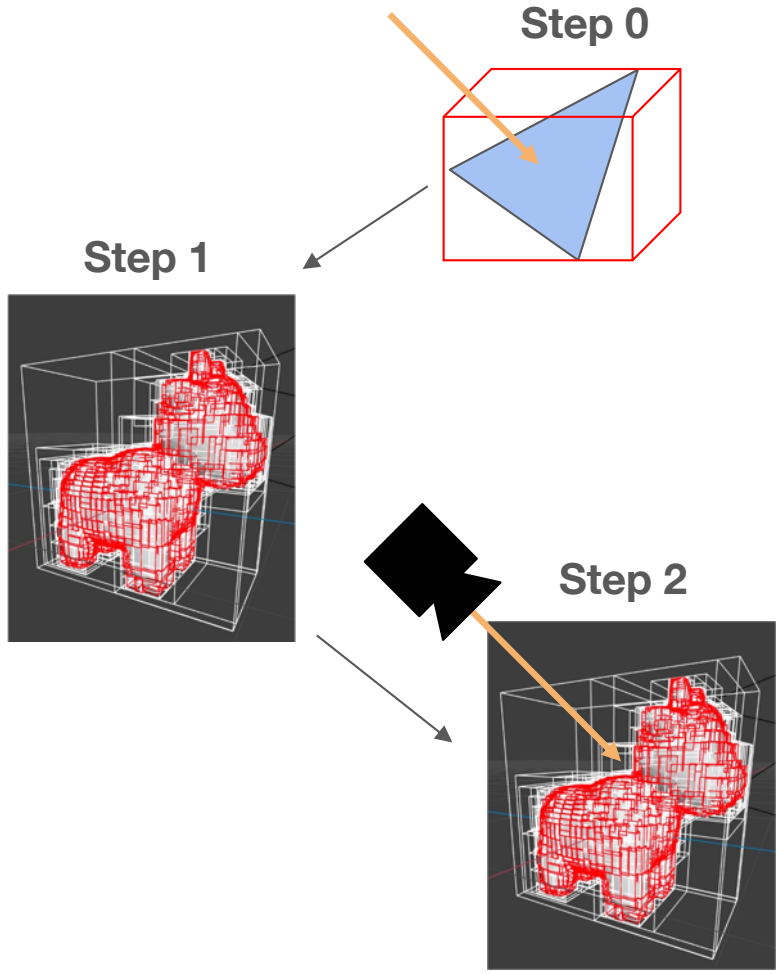
Motivation:
Checking all the primitives at once per ray hit is expensive (O(n)), we want to speed it up. BVH's tree structure makes it O(nlogn)



Task 3*: BVH

High-Level Procedure:

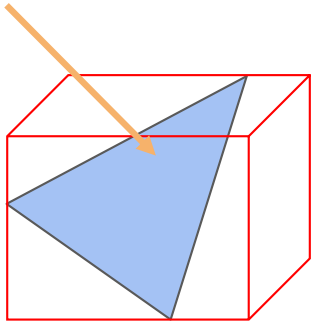
- **Step 0** - Implement BBox Intersection
- **Step 1** - Create BVH acceleration data structure using Surface Area Heuristic (to make Step 2 faster)
- **Step 2** - Implement Ray Tracing with BVH data structure



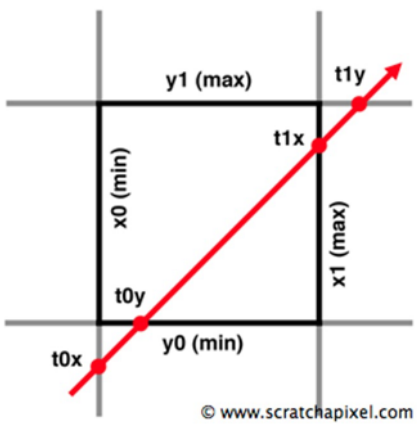
Task 3*: BVH

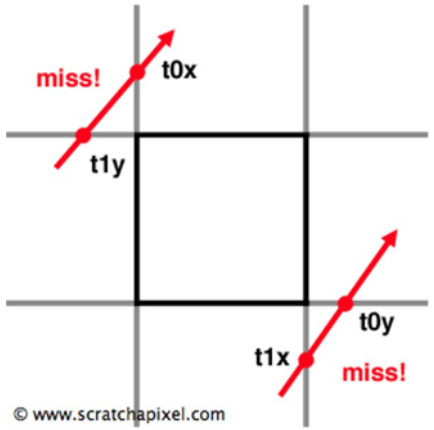
bool BBox::hit(const Ray &ray, Vec2 &times);

Step 0: Ray-BBox Intersection



Implement Ray-BBox intersection in BBox::hit

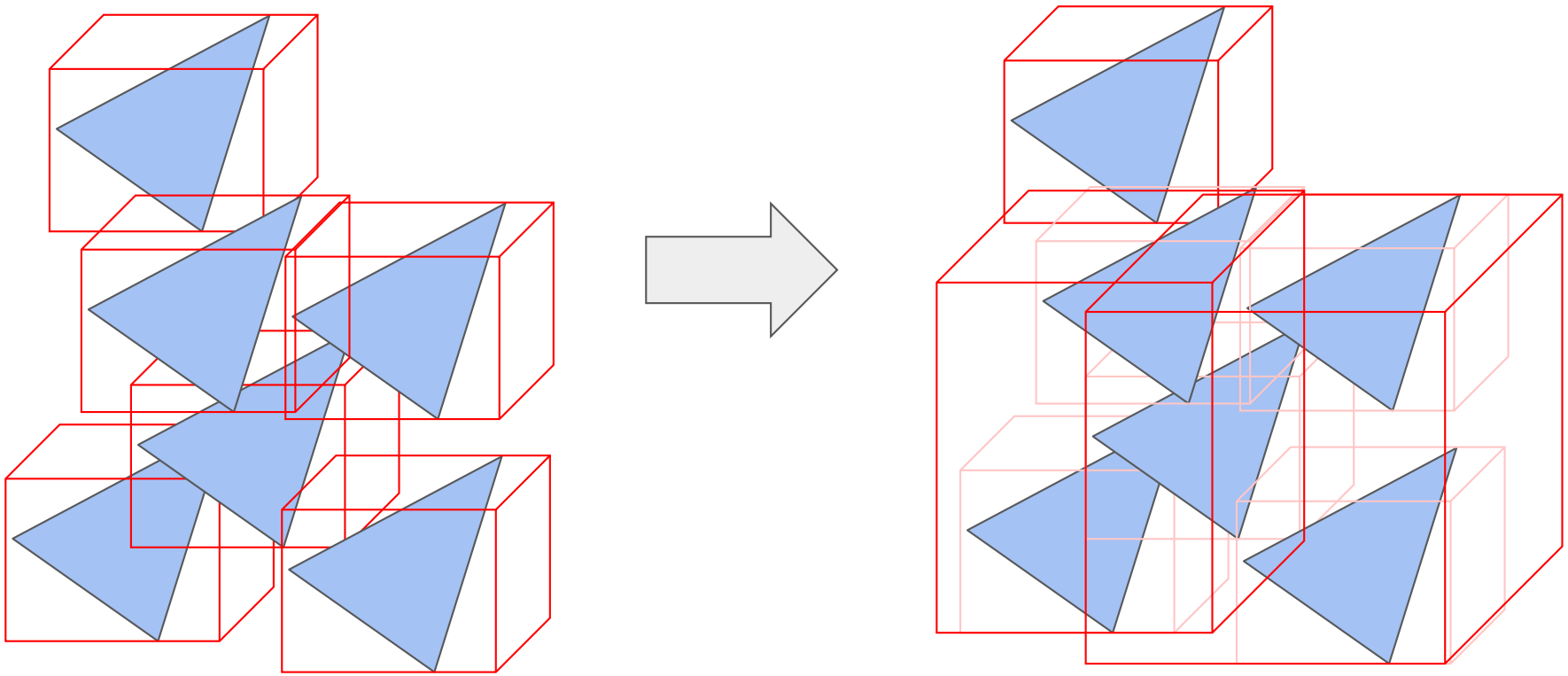




https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-box-intersection

Task 3*: BVH

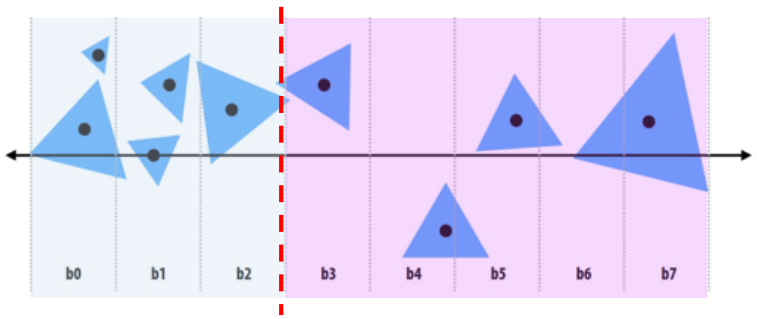
How do I partition the 3D space (and the mesh)?



Task 3*: BVH

Step 1: BVH Construction

Want to optimally sort Primitives into buckets (formed by partitions)



Bucket A Bucket B

Surface Area Heuristic

 # primitives in subtree A

 # primitives in subtree B

SA of *bounding box of* subtree A

SA of *bounding box of* subtree B

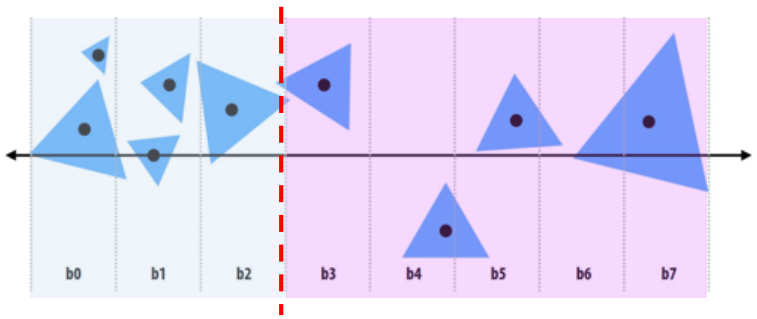
$$C = C_{\text{trav}} + \frac{S_A}{S_N} N_A C_{\text{isect}} + \frac{S_B}{S_N} N_B C_{\text{isect}}$$

SA of *bounding box of* parent

Task 3*: BVH

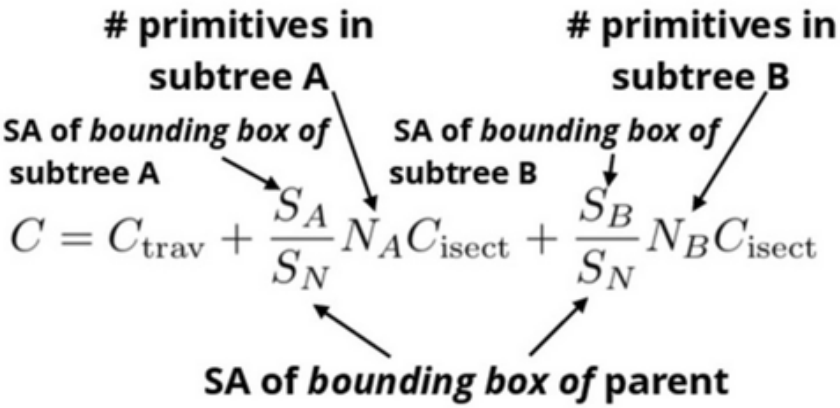
Step 1: BVH Construction

Want to optimally sort Primitives into buckets (formed by partitions)



Bucket A Bucket B

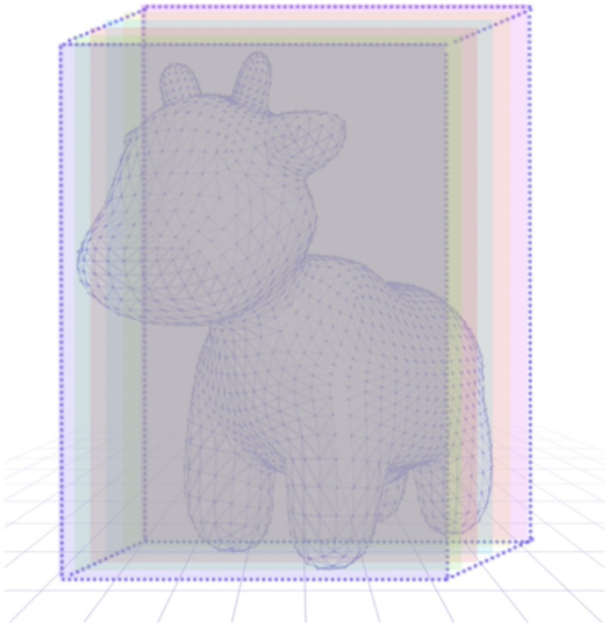
Surface Area Heuristic



We can assume C_{trav} and C_{isect} to be 1 since they are constants (value itself is irrelevant)
So really, just the SAH :)

Task 3*: BVH

High Level Idea:



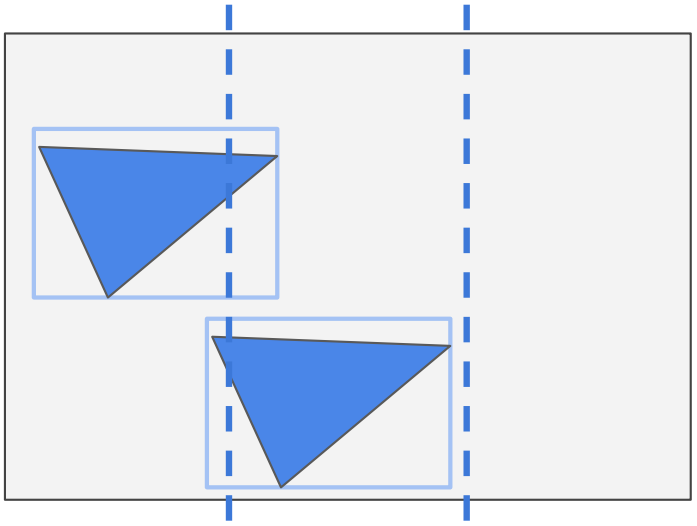
For each partition along the XYZ axis:

1. Define buckets along the line of partition
2. Calculate the BBox for the bucket based on the primitives in the bucket
3. Keep track of the best optimal partition

Construct the BVH based on the lowest cost partition found, and recurse on it(or make node leaf)

Task 3*: BVH

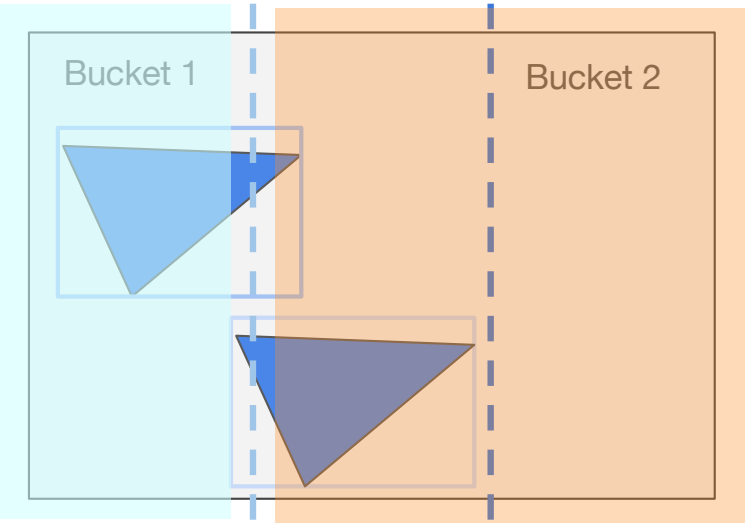
Step 1: BVH Construction



```
For axis x,y,z:
    Initialize buckets
    For each primitive p in node:
        B = compute_bucket(p.centroid)

        B.bbox.enclose(p.bbox)
        B.prim_count++
    For each of |B| - 1 possible partitions
        Evaluate cost (SAH), keep track of lowest cost partition
Recurse on lowest cost partition found (or make node leaf)
```

Task 3*: BVH

Step 1: BVH Construction



Bucket = result from a possible (but maybe not the best) partition

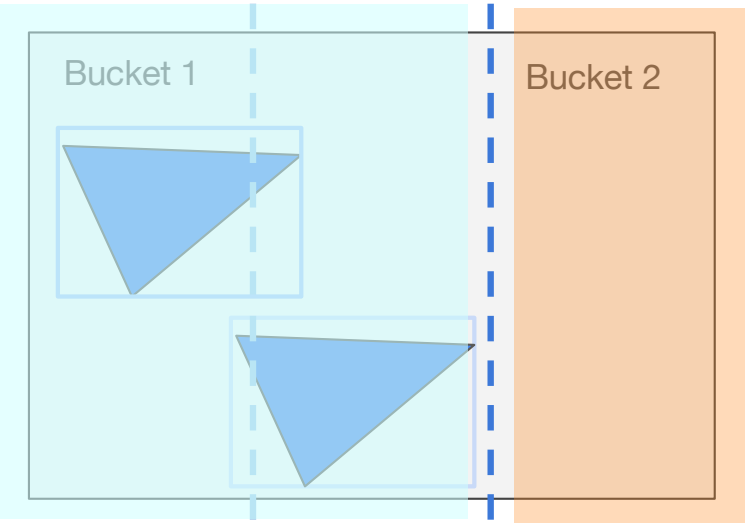
In code: some variable that you will have to keep track of

Partition along x = 1, 2, 3 …

```
For axis x,y,z:
    Initialize buckets
    For each primitive p in node:
        B = compute_bucket(p.centroid)

        B.bbox.enclose(p.bbox)
        B.prim_count++
    For each of |B| - 1 possible partitions
        Evaluate cost (SAH), keep track of lowest cost partition
Recurse on lowest cost partition found (or make node leaf)
```

Task 3*: BVH

Step 1: BVH Construction



Bucket = result from a possible (but maybe not the best) partition

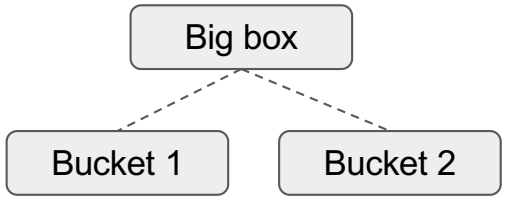
In code: some variable that you will have to keep track of

Partition along x = 1, 2, 3 …

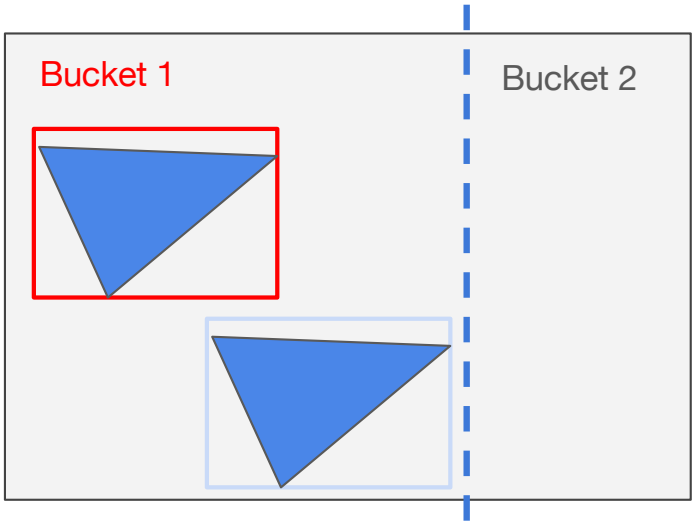
```
For axis x,y,z:
    Initialize buckets
    For each primitive p in node:
        B = compute_bucket(p.centroid)

        B.bbox.enclose(p.bbox)
        B.prim_count++
    For each of |B| - 1 possible partitions
        Evaluate cost (SAH), keep track of lowest cost partition
Recurse on lowest cost partition found (or make node leaf)
```

Task 3*: BVH

Step 1: BVH Construction



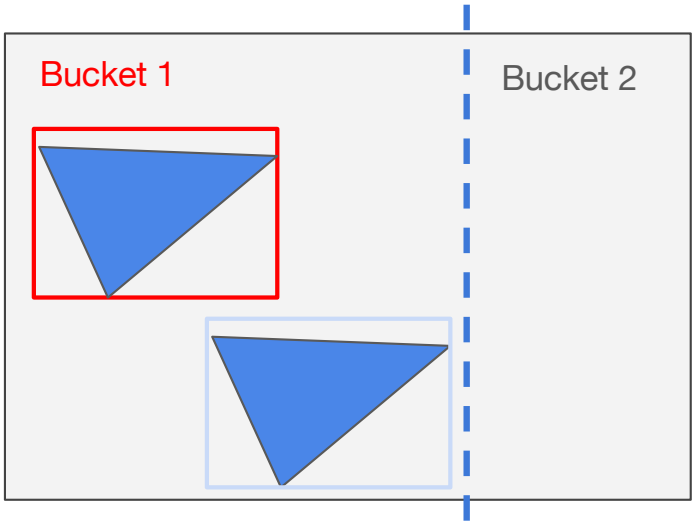
<< nodes




```
For axis x,y,z:
    Initialize buckets
    For each primitive p in node:
        B = compute_bucket(p.centroid)

        B.bbox.enclose(p.bbox)
        B.prim_count++
    For each of |B| - 1 possible partitions
        Evaluate cost (SAH), keep track of lowest cost partition
Recurse on lowest cost partition found (or make node leaf)
```

Task 3*: BVH

Step 1: BVH Construction

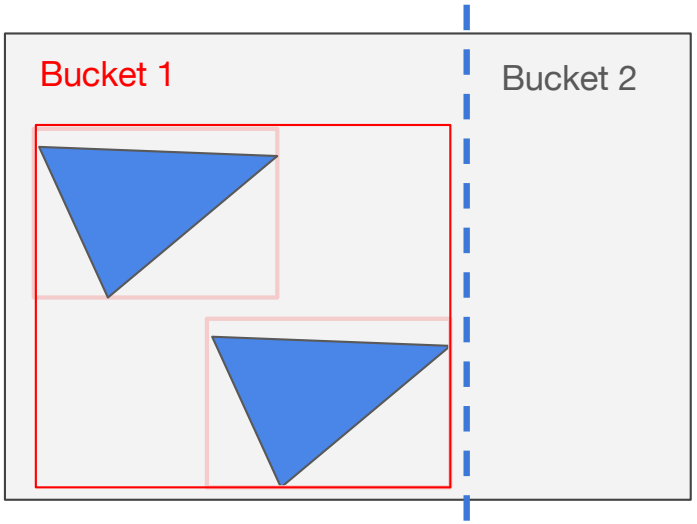


Bucket 1 Count: 1
Bucket 2 Count: 0

```
For axis x,y,z:
    Initialize buckets
    For each primitive p in node:
        B = compute_bucket(p.centroid)

        B.bbox.enclose(p.bbox)
        B.prim_count++
    For each of |B| - 1 possible partitions
        Evaluate cost (SAH), keep track of lowest cost partition
Recurse on lowest cost partition found (or make node leaf)
```

Task 3*: BVH

Step 1: BVH Construction



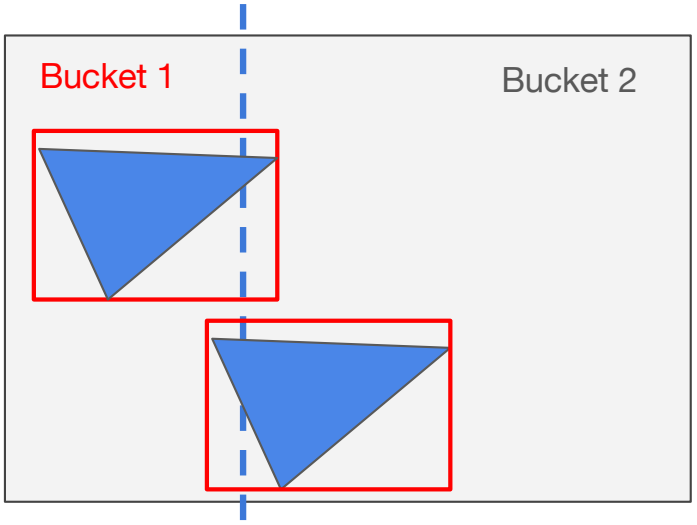
Partition 1

Bucket 1 Count: 2
Bucket 2 Count: 0

```
For axis x,y,z:
    Initialize buckets
    For each primitive p in node:
        B = compute_bucket(p.centroid)

        B.bbox.enclose(p.bbox)
        B.prim_count++
    For each of |B| - 1 possible partitions
        Evaluate cost (SAH), keep track of lowest cost partition
Recurse on lowest cost partition found (or make node leaf)
```

Task 3*: BVH

Step 1: BVH Construction



Partition 1

Bucket 1 Count: 1
Bucket 2 Count: 1

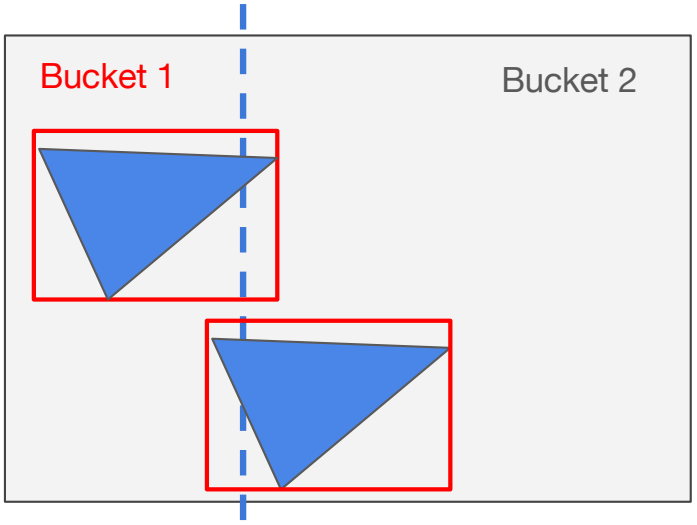
Partition 2

Bucket 1 Count: 2
Bucket 2 Count: 0

```
For axis x,y,z:
    Initialize buckets
    For each primitive p in node:
        B = compute_bucket(p.centroid)

        B.bbox.enclose(p.bbox)
        B.prim_count++
    For each of |B| - 1 possible partitions
        Evaluate cost (SAH), keep track of lowest cost partition
Recurse on lowest cost partition found (or make node leaf)
```

Task 3*: BVH

Step 1: BVH Construction



Partition 1

Bucket 1 Count: 1
Bucket 2 Count: 1
SAH = big

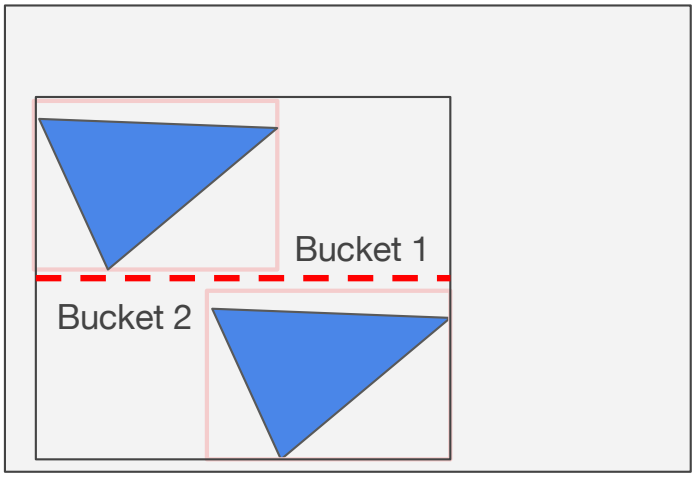
Partition 2

Bucket 1 Count: 2
Bucket 2 Count: 0
SAH = small

```
For axis x,y,z:
    Initialize buckets
    For each primitive p in node:
        B = compute_bucket(p.centroid)

        B.bbox.enclose(p.bbox)
        B.prim_count++
    For each of |B| - 1 possible partitions
        Evaluate cost (SAH), keep track of lowest cost partition
Recurse on lowest cost partition found (or make node leaf)
```

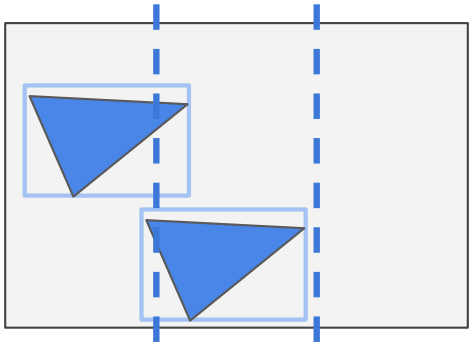
Task 3*: BVH

Step 1: BVH Construction




```
For axis x,y,z:
    Initialize buckets
    For each primitive p in node:
        B = compute_bucket(p.centroid)

        B.bbox.enclose(p.bbox)
        B.prim_count++
    For each of |B| - 1 possible partitions
        Evaluate cost (SAH), keep track of lowest cost partition
Recurse on lowest cost partition found (or make node leaf)

      (stop if number of primitives <= max_leaf_size)
```

Another way to think about it



```
For axis x, y, z:
    For partitions partition along current axis:
        divide primitives into left, right according to partition
        evaluate SAH cost
        keep track of the best partition
Recurse on best partition
```

Helpful functions!

Std::partition

```
auto it = std::partition(v.begin(), v.end(), [](int i){return i % 2 == 0;})
```

```
Original vector:
    0 1 2 3 4 5 6 7 8 9
Partitioned vector:
    0 8 2 6 4  *  5 3 7 1 9
Unsorted list:
    1 30 -4 3 5 -4 1 6 -8 2 -5 64 1 92
```

* is where auto **it** is at

Note that the elements are not sorted within the subgroups themselves. You may want to use std::sort to sort them.

Helpful functions!

Bbox.enclose - lets one box enclose another

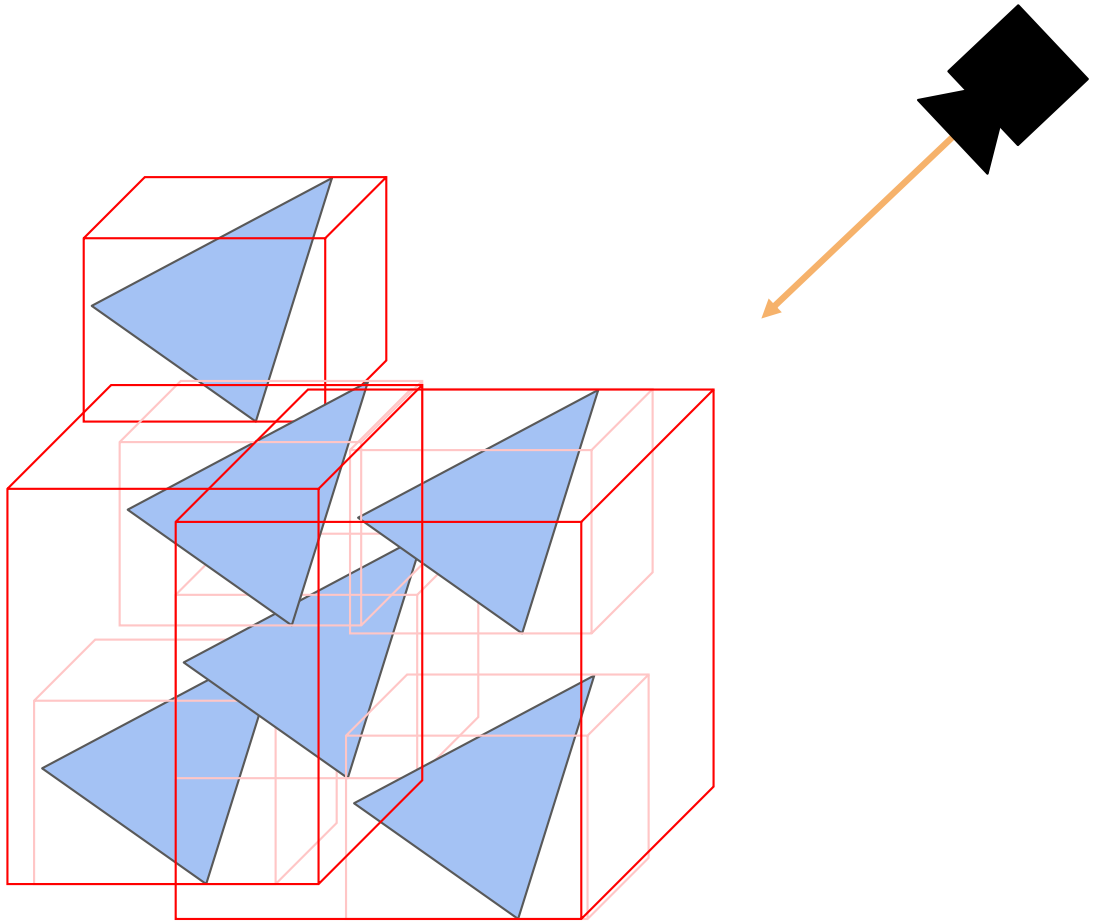
Bbox.center - center of the box

primitives(in build) - vector/array of primitives

Node - used to construct the tree

```
class Node {
    BBox bbox;
    size_t start, size, l, r;
```

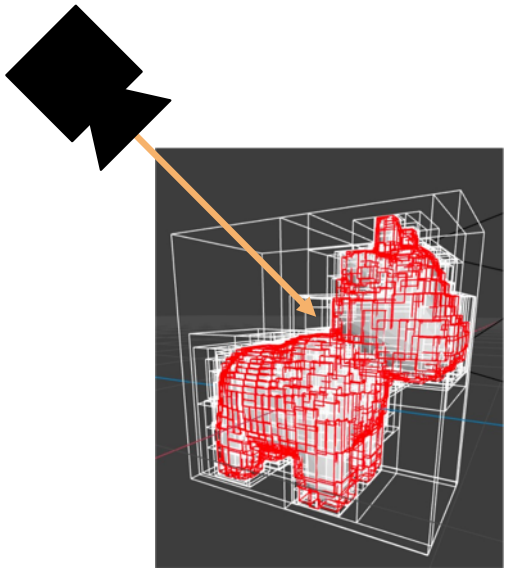
Task 3*: BVH



Task 3*: BVH

Step 2: Ray-BVH Intersection

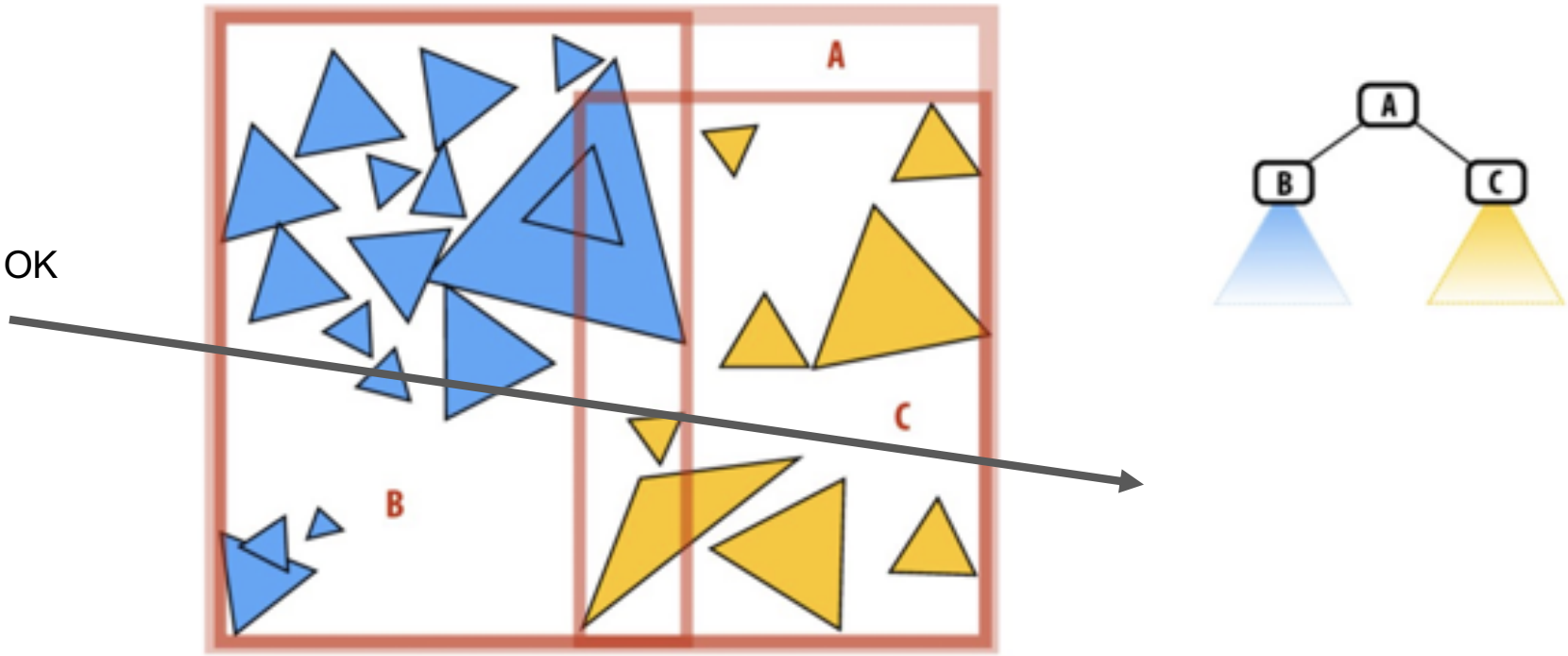
Trace BVH<Primitive>::hit(const Ray& ray);

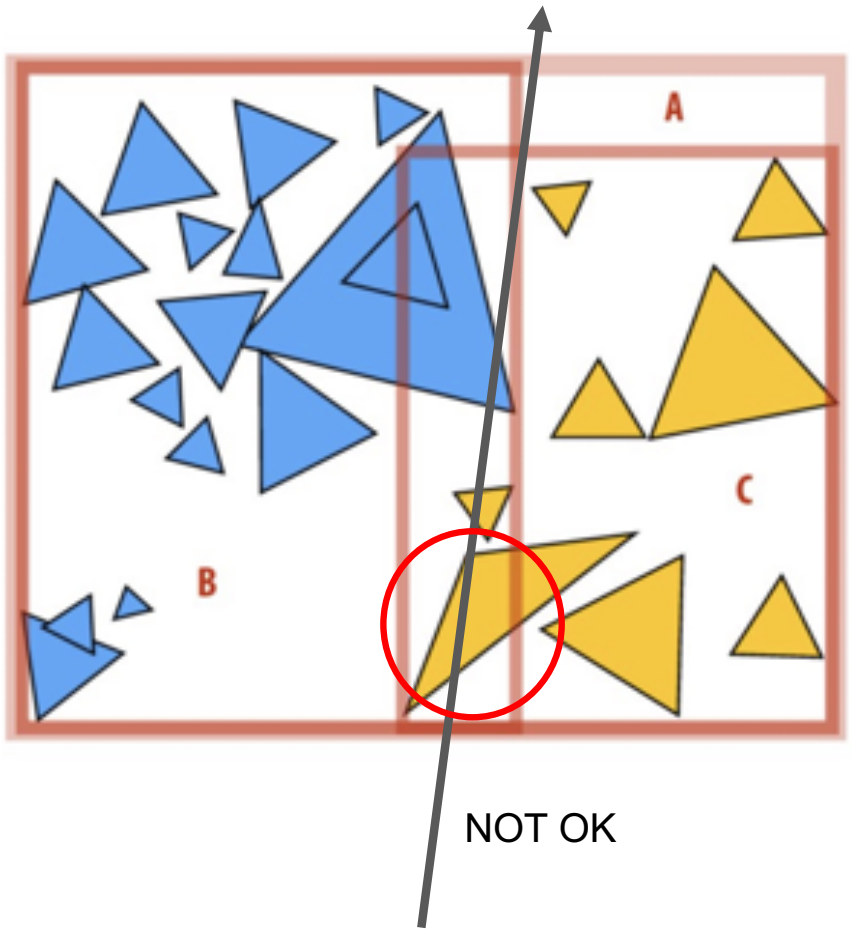


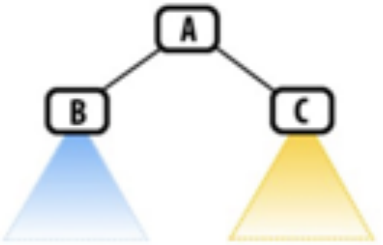
```
void find_closest_hit(Ray* ray, BVHNode* node, HitInfo* closest)
{
   if (node->leaf) {
      // same as before
   } else {
      HitInfo hit1 = intersect(ray, node->child1->bbox);
      HitInfo hit2 = intersect(ray, node->child2->bbox);

      BVHNode* first = (hit1.t <= hit2.t) ? child1 : child2;
      HitInfo* hitsecond = hit1.t <= hit2.t ? &hit2 : &hit1;
      find_closest_hit(ray, first, closest);
      if(hitsecond->t < closest->t)
         find_closest_hit(ray, second, closest);
   }
}
```

"Front to back" traversal. Traverse to closest child node first. Why?

we still need to do this?


OK
A
B
C
A
B
C

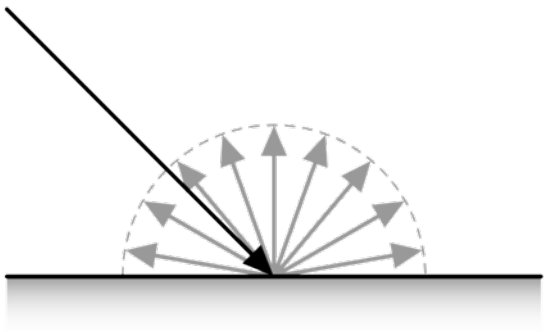

A
B
C
NOT OK


A
B
C

Part II

Task 4: Path Tracing

BSDF_Lambertian



::scatter - returns a Scatter object containing direction and attenuation components of the new ray / surface color

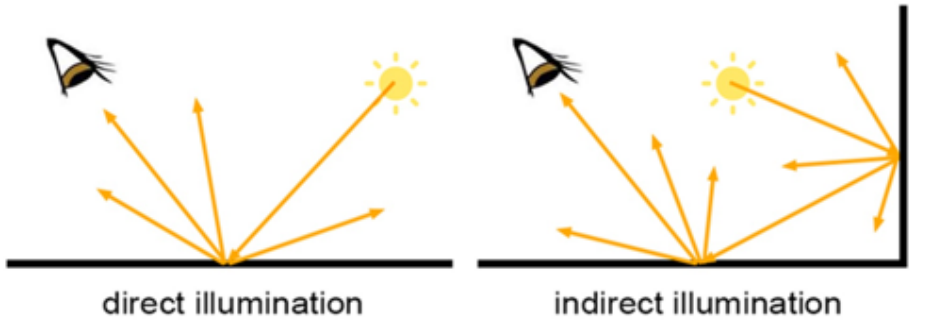
::evaluate - computes the ratio of incoming to outgoing radiance given a pair of directions

::pdf - returns the pdf of a Cosine-weighted sampler

Task 4: Path Tracing

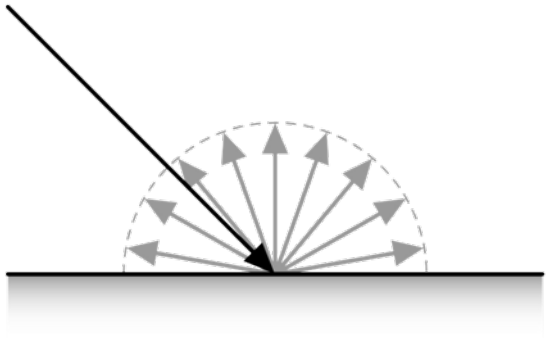
Direct & Indirect Lighting

1) Randomly sample a new ray direction from the BSDF distribution using BSDF::scatter().
2) Create a new world-space ray and call Pathtracer::trace() to get incoming light. You should modify dist_bounds so that the ray does not intersect at time = 0. Remember to set the new depth value.
3) Compute Monte Carlo estimate of incoming light scaled by BSDF attenuation. Use is_discrete() to determine if need to scale by pdf.

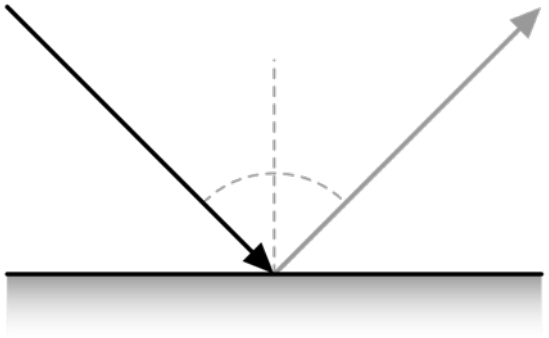


Task 5: Materials

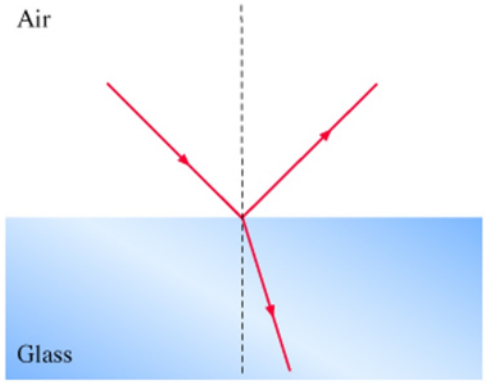
Step 0: Lambertian BSDF



Step 1: Mirror BSDF



Step 2: Glass BSDF



Quick Overview of the code you will implement

`Scatter scatter(Vec3 out_dir)`: given `out_dir`, generates a random sample for `in_dir`. It returns a `Scatter`, which contains both the sampled `direction` and the `attenuation` for the in/out pair.

Remember that we are tracing rays *backwards*, from the camera into the scene. This the reason why the input to your scatter functions is the out_dir, you are calculating the in_dir.

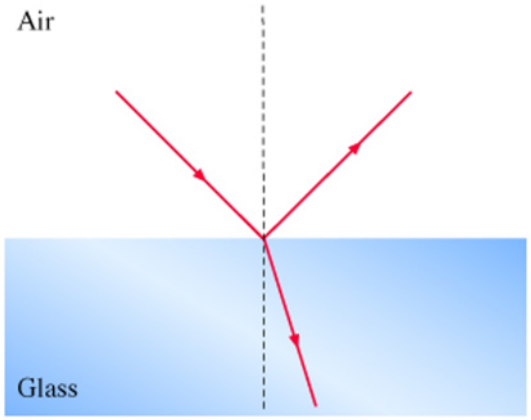
Quick Overview of the code you will implement

`Scatter scatter(Vec3 out_dir)`: given `out_dir`, generates a random sample for `in_dir`. It returns a `Scatter`, which contains both the sampled **`direction`** and the **`attenuation`** for the in/out pair.

You will implement these two helper functions:

`Vec3 reflect(Vec3 dir)`: returns a direction that is the perfect specular reflection of `dir` about `{0, 1, 0}(normal of surface)`.

`Vec3 refract``(Vec3 out_dir, float index_of_refraction, bool& was_internal)`: returns the ray that results from refracting `out_dir` through the surface according to Snell's Law.



Quick Overview of the code you will implement

`Scatter scatter(Vec3 out_dir)`: given `out_dir`, generates a random sample for `in_dir`. It returns a `Scatter`, which contains both the sampled `direction` and the `attenuation` for the in/out pair.

You will get this through:

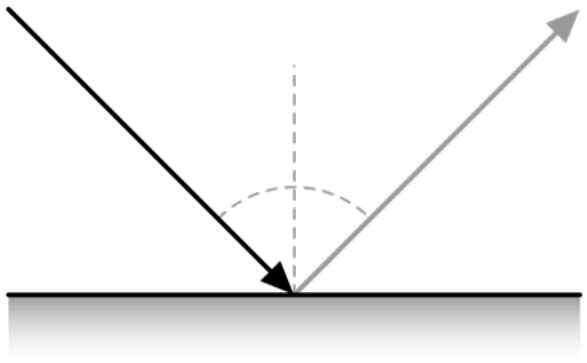
`Spectrum evaluate(Vec3 out_dir, Vec3 in_dir)`: evaluates the BSDF for a given pair of directions. This is only defined for continuous BSDFs.

In the starter code, only defined for Lambertian. You will calculate this value within your function scatter.

This is dependent on the surface property of the material (hint: look in the definition BSDF_glass and BSDF_mirror!)

Task 5: Materials

Step 1: Mirror BSDF

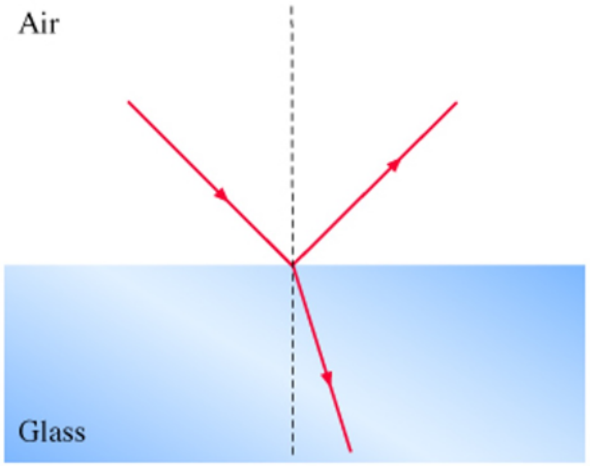


```
struct BSDF_Mirror {
      BSDF_Sample sample(Vec3 out_dir) const;
      Spectrum evaluate(Vec3 out_dir, Vec3 in_dir) const;
            Spectrum reflectance;
}
```

```
struct Scatter{
            Spectrum attenuation; // reflectance
            Vec3 direction;                  // flip it!
(in reflect)
}
```

Task 5: Materials

Step 2: Glass BSDF



General Idea:

Based on the direction of the ray...

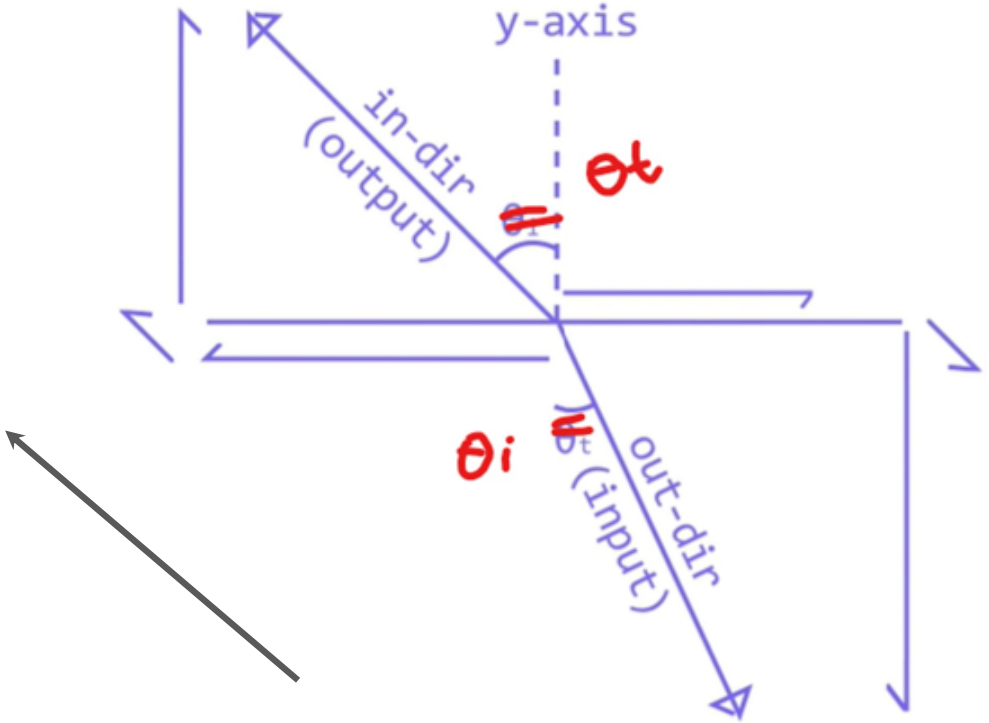
1. Try refraction
2. Determine if total internal reflection is happening
    a. If yes, reflect(identical to mirror)
3. Calculate the Fresnel coefficient(Fr)
4. Reflect or refract probabilistically based on Fr
    a. Reflect with the probability of Fr
    b. Refract/Transmit with probability of (1-Fr)

Refraction

Typo in the diagram making everything confusing ;-;

$\sin\theta$ = x&z-component
$\cos\theta$ = y-component

Snell's Law:
$n_i \cdot \sin\theta_i = n_t \cdot \sin\theta_t$
$\sin\theta_i = n_t/n_i \cdot \sin\theta_t$ (remember to reflect)



How to find x & z direction?

Refraction

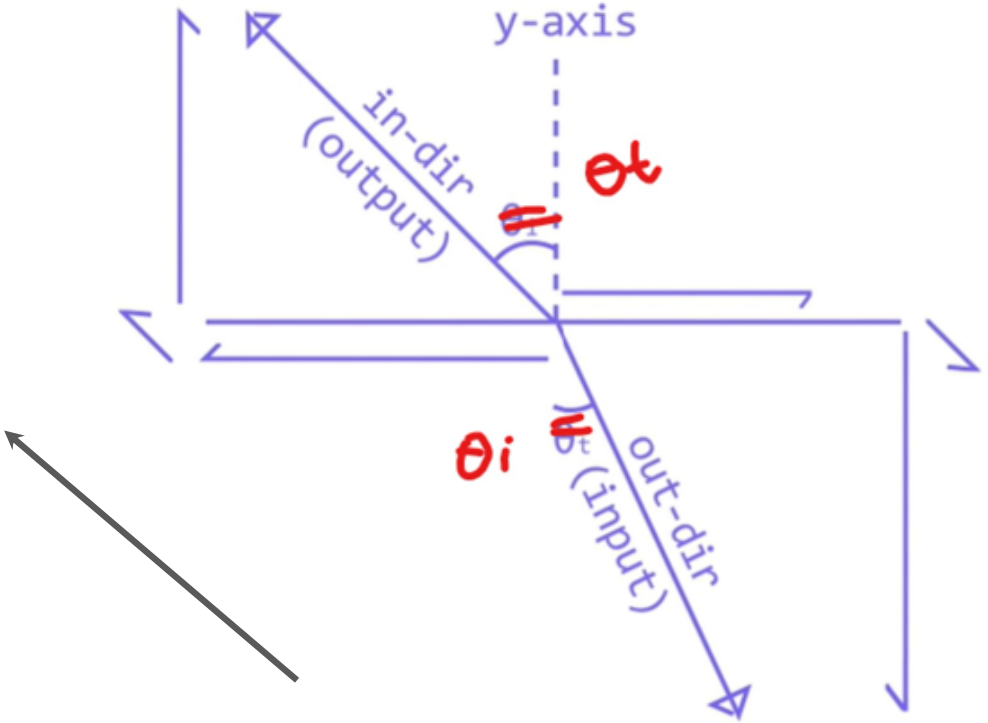
Typo in the diagram making everything confusing ;-;

$\sin\theta$ = x&z-component
$\cos\theta$ = y-component

Snell's Law:
$n_i \cdot \sin\theta_i = n_t \cdot \sin\theta_t$
$\sin\theta_i = n_t/n_i \cdot \sin\theta_t$ (remember to reflect)

Want to find cos(theta)_t
The y-direction of the incoming ray

>> use trig properties!



How to find x & z direction?

Refraction

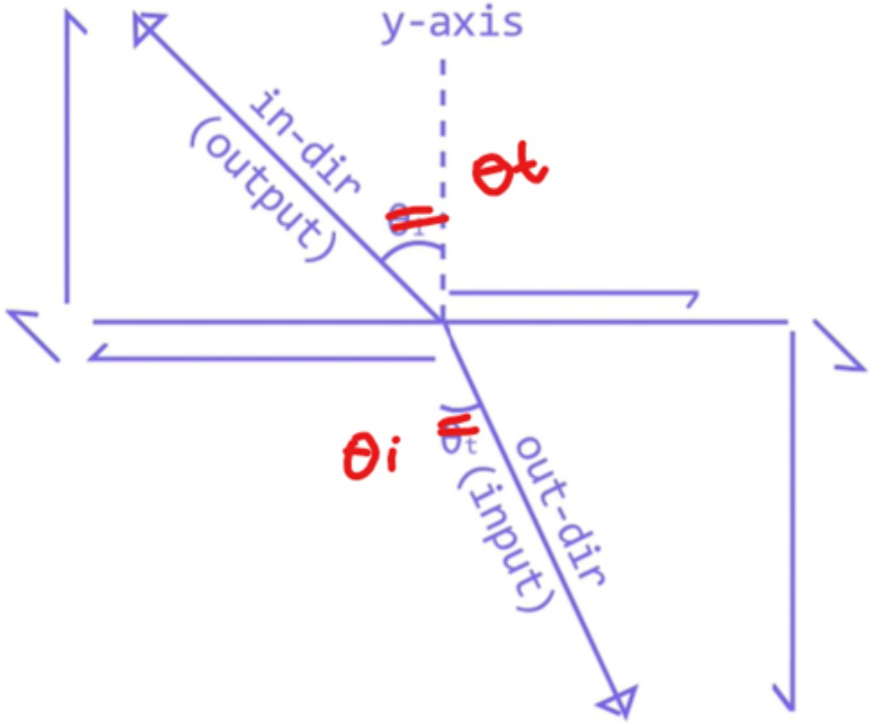
Typo in the diagram making everything confusing ;-;

$\sin\theta$ = x&z-component
$\cos\theta$ = y-component

Snell's Law:
$n_i \cdot \sin\theta_i = n_t \cdot \sin\theta_t$
$\sin\theta_i = n_t/n_i \cdot \sin\theta_t$ (remember to reflect)

$\cos\theta_t = \sqrt{1 - \sin^2\theta_t}$
$\cos\theta_t = \sqrt{1 - (n_i/n_t)^2 \cdot \sin^2\theta_i}$
$\cos\theta_t = \sqrt{1 - (n_i/n_t)^2 \cdot (1 - \cos^2\theta_i)}$

if negative:
total-internal-reflection
just reflect :)

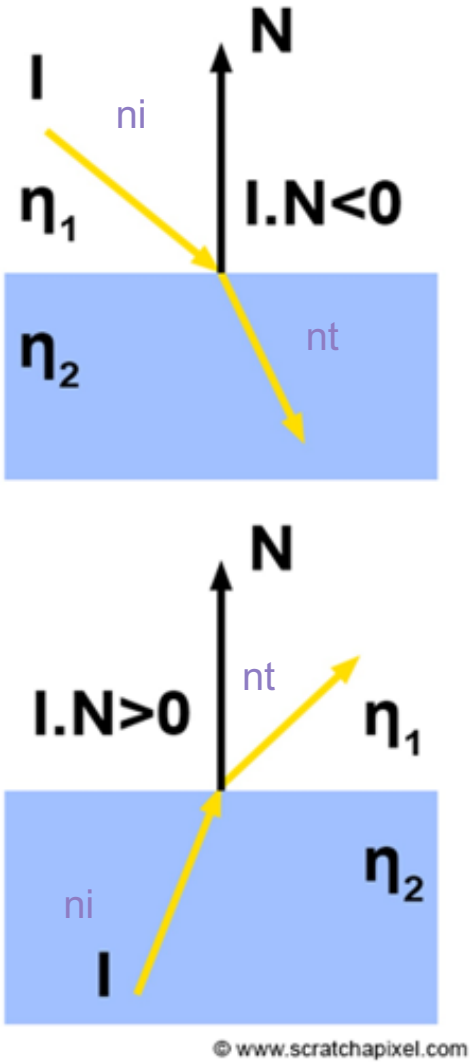


Refraction

Your implementation should assume that when `in_dir` enters the surface (that is, if `cos(theta_out) > 0`) then the ray was previously travelling in a vacuum (i.e. index of refraction = 1.0). If `cos(theta_out) < 0`, then `in_dir` is leaving the surface and entering a vacuum.

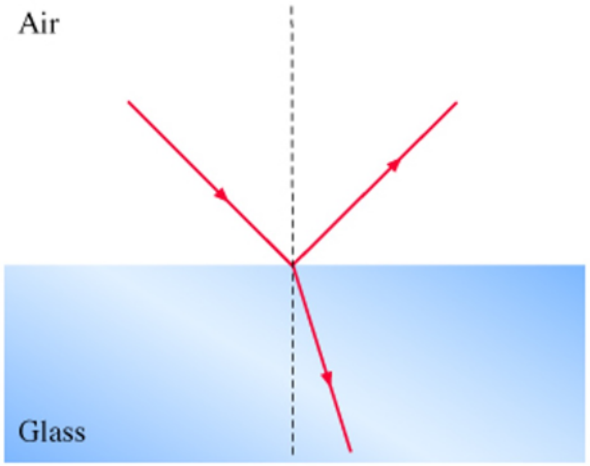
Cos(theta_out) < 0 then
ni = 1, nt = ior

Cos(theta_out) > 0 then
nt = ior, ni =1



Task 5: Materials

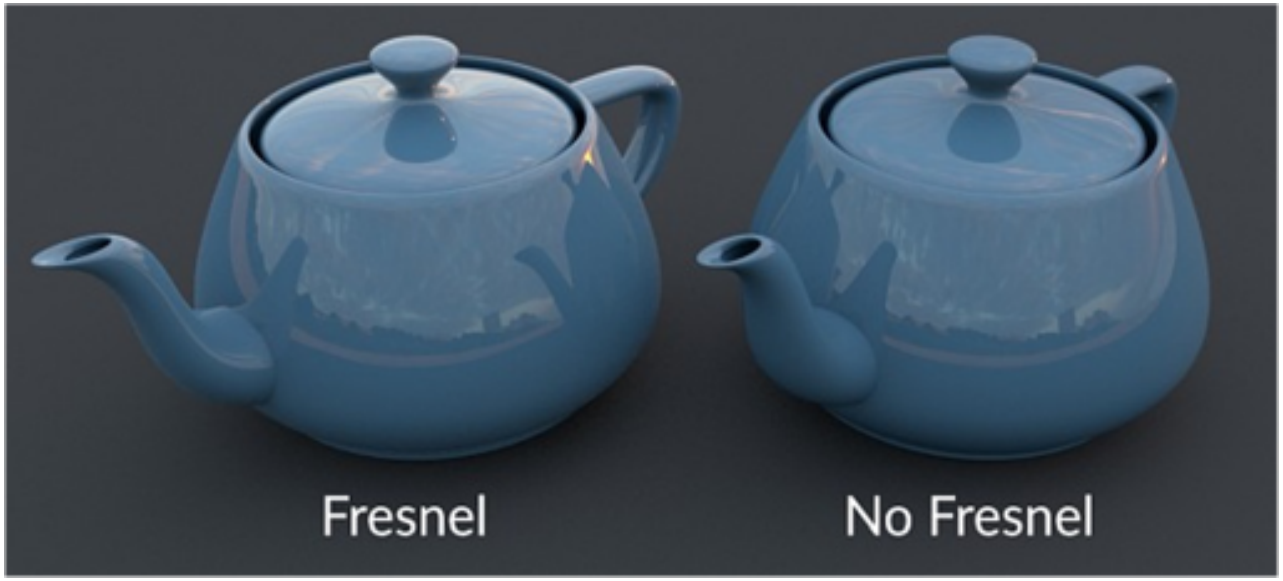
Step 2: Glass BSDF



Use Fresnel Equations to calculate the reflection **Fresnel coefficient** - the proportion of reflected to refracted light

	Schlick's Approximation
F_r = proportion of **reflected** light $1 - F_r$ = proportion of **refracted** light	$R(\theta)$ = proportion of **reflected** light $1 - R(\theta)$ = proportion of **refracted** light
$F_r = \frac{1}{2}(r_{\parallel}^2 + r_{\perp}^2)$	$R(\theta) = R_0 + (1 - R_0)(1 - \cos\theta)^5$
$r_{\parallel} = \frac{\eta_t \cos\theta_i - \eta_i \cos\theta_t}{\eta_t \cos\theta_i + \eta_i \cos\theta_t}$ $r_{\perp} = \frac{\eta_i \cos\theta_i - \eta_t \cos\theta_t}{\eta_i \cos\theta_i + \eta_t \cos\theta_t}$	$R_0 = \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2$

Fresnel



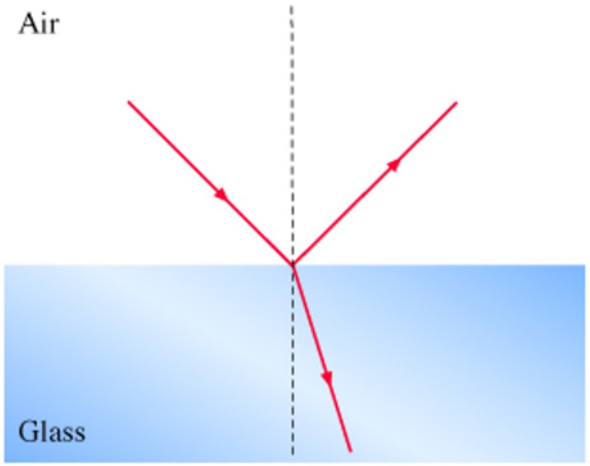
How much light they reflect vs the amount they transmit actually depends on the angle of incidence. The amount of transmitted light increases when the angle of incidence decreases.

source:Introduction to Shading (Reflection, Refraction and Fresnel) (scratchapixel.com)

Task 5: Materials

Don’t forget to account for Total Internal Reflection as Well!

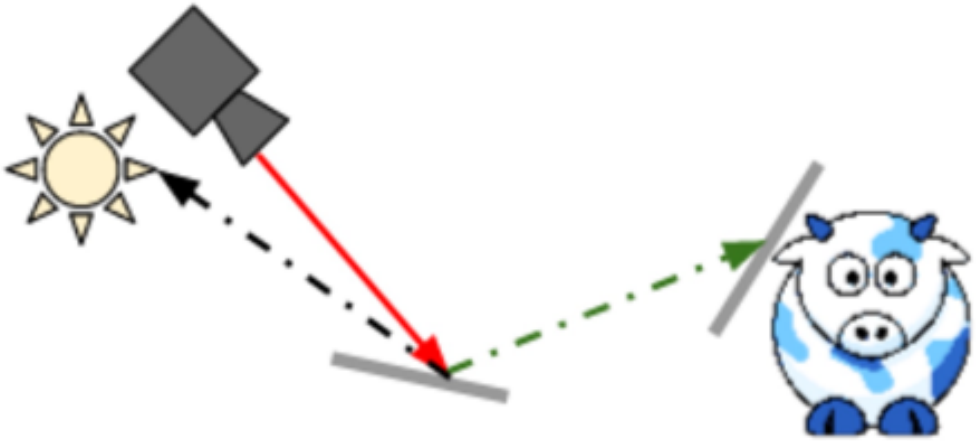
Step 2: Glass BSDF




```
struct BSDF_Glass {
        BSDF_Sample sample(Vec3 out_dir) const;
        Spectrum evaluate(Vec3 out_dir, Vec3 in_dir)
const;

     Spectrum transmittance;
        Spectrum reflectance;
        float index_of_refraction;
}

struct BSDF_sample {
        Spectrum attenuation; // scaled reflectance or
                              // transmittance
        Vec3 direction;                  //
reflect(Vec3 dir) or
                                                    //
refract(Vec3 dir)
}
```

Task 6: Direct Lighting++

1) If the BSDF is discrete, you can ignore the following steps.
2) Equally randomly choose to sample from BSDF::scatter or Pathtracer::sample_area_lights. **Pay attention to inputs and outputs for both functions!**
3) Recompute the new pdf value, given the additional choice of sampling from the area lights - use Pathtracer::area_lights_pdf. **Watch the inputs/outputs!**



Task 7: Environment Light

Implement the following functions:

```
In samplers.cpp

Vec3 Sphere::Uniform::sample() const;           // Uniform sampler

Sphere::Image::Image(const HDR_Image& image) const;         //Set up importance sampling data
structure
Vec3 Sphere::Image::sample() const; //use the data to generate a sampling direction
Float Sphere::Image::pdf(Vec3 dir) const;       //calculate the pdf
```

```
In env_light.cpp

Vec3 Env_Map::sample() const //how to sample(uniform vs importance)
float Env_Map::pdf(Vec3 dir) const  //returns the pdf
Spectrum Env_Map::evaluate(Vec3 dir) const // find incoming light along dir
```

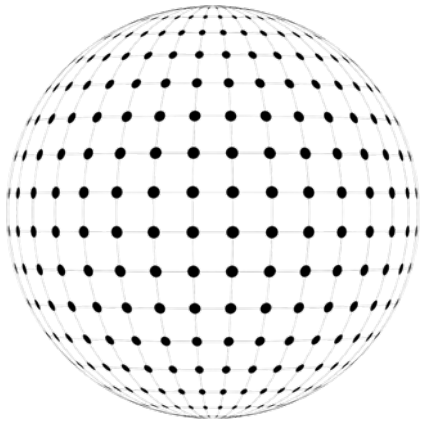
Task 7: Environment Light

Step 1: Uniform Sampling

```
Vec3 Sphere::Uniform::sample() const;          // Uniform sampler
Vec3 Env_Sphere::sample() const
float Env_Sphere::pdf(Vec3 dir)
```

Hint #1: A sphere's surface area is $4\pi r$

Hint #2: Look at `Hemisphere::Uniform::sample` for generating vectors



Task 7: Environment Light

Step 1: Uniform Sampling

```
Spectrum Env_Map::evaluate(Vec3 dir) const // find incoming light along dir
```

General Idea:

1. vector direction → spherical coordinates(theta & phi) → UV coordinates
2. Multiply by image dimensions to get actual pixel coordinates
3. Find 4 nearest pixels, perform bilinear interpolation

(Video explains it pretty well)

Task 7: Environment Light

Step 2: Importance Sampling

```
Sphere::Image::Image(const HDR_Image& image) const;
```

Populate the `pdf` and `cdf` vectors based on flux through each pixel
(flux through pixel is proportional to `L sin(θ)`)

```
Vec3 Sphere::Image::sample() const; // Importance sampler
```

Generate a weighted random phi and theta coordinate pair, convert to xyz coordinates (Vec3)

Task 7: Environment Light(conceptual)

Image I have a 4 pixel image

0.2	0.6	1	0.2

lumen

If I do **uniform sampling**, what's the probability density function(pdf) that I sample each pixel?

0.25 (¼) for all pixels, each pixel has a equal chance of being sampled.

Task 7: Environment Light(conceptual)

Image I have a 4 pixel image

0.2	0.6	1	0.2

lumen

X / 2

0.1	0.3	0.5	0.1

If I do **importance sampling(favoring brighter pixels)**, what's the probability density function(pdf) that I sample each pixel?

**intuitively, we know brighter pixel should have a higher probability

We take each pixel's luminosity and divide it by the total luminous intensity of the image

Task 7: Environment Light(conceptual)

Image I have a 4 pixel image

0.2	0.6	1	0.2

lumen

0.1	0.3	0.5	0.1

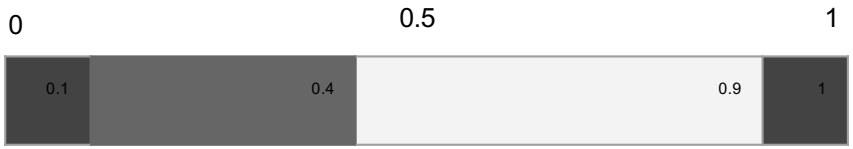
pdf

0.1	0.4	0.9	1

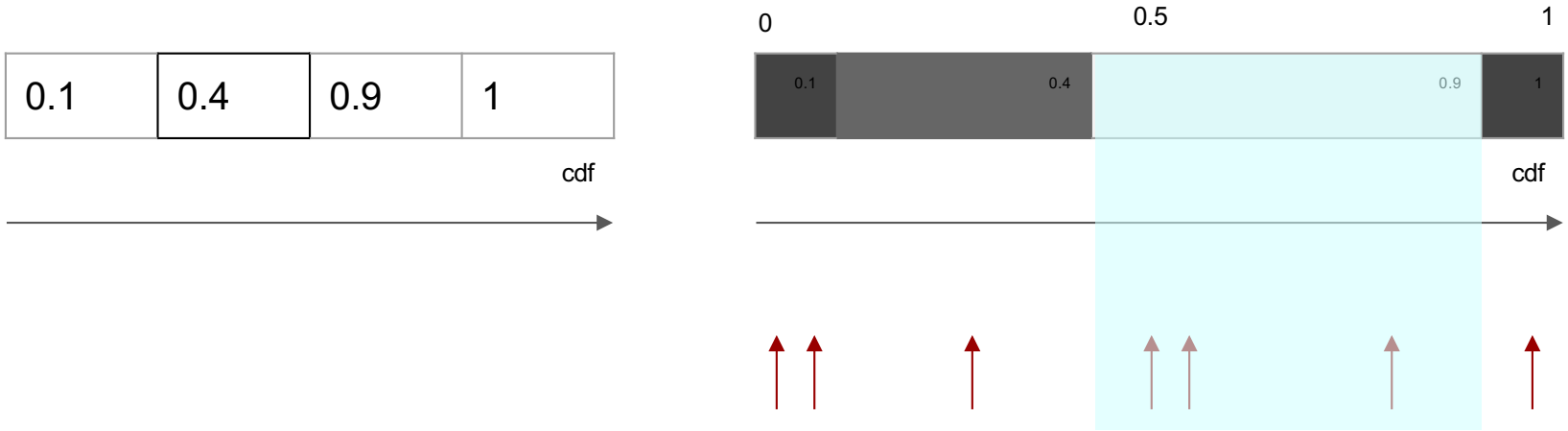
cdf

Now, what if I calculate the Cumulative distribution function(CDF)?

(just add it together!)



Task 7: Environment Light(conceptual)

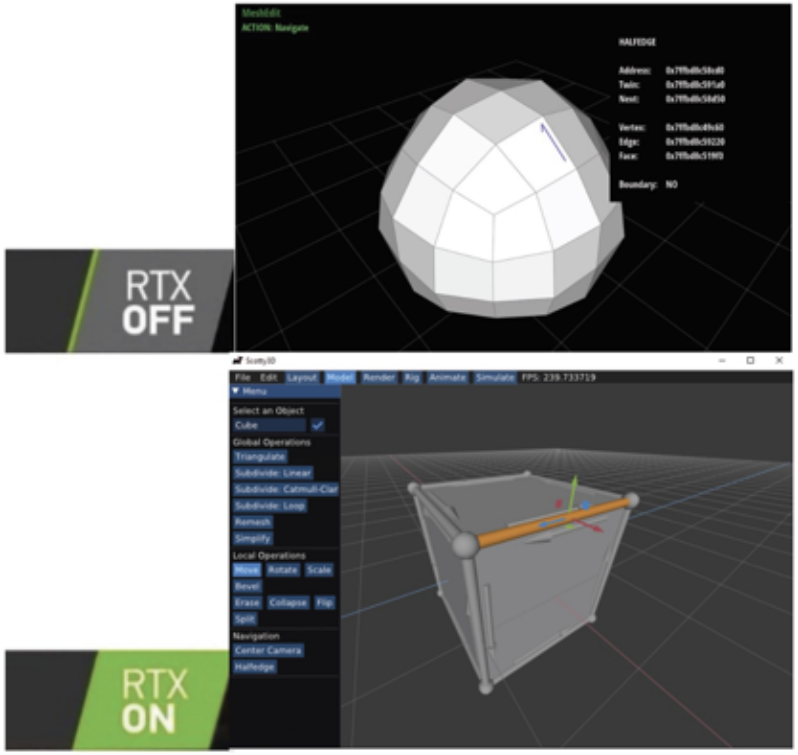


What happens now if I take a random sample between 0 - 1, and find which index contains that within my cdf?

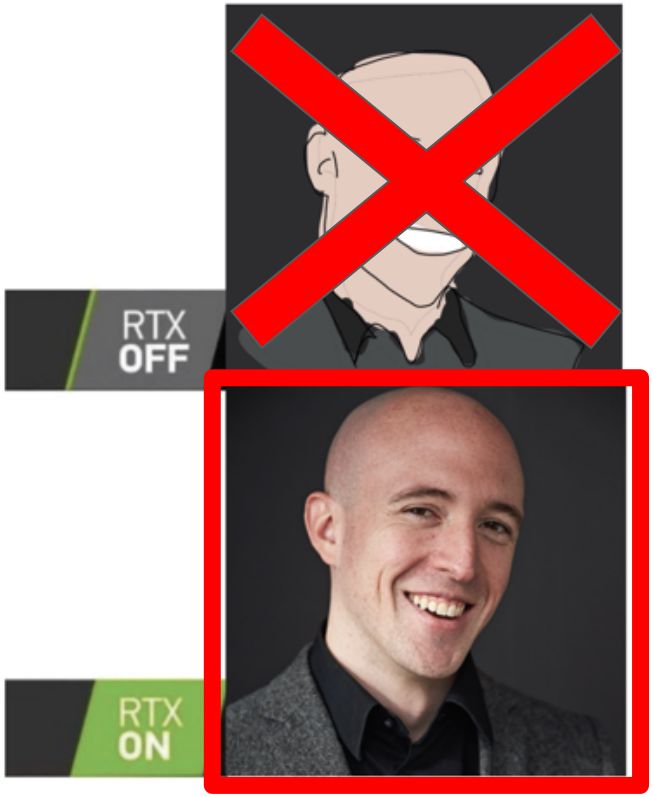
You're more likely to sample the points with higher luminosity! Which is what we want in importance sampling. From here you can find the index of the element you sampled, and retrieve its pdf & luminance values.

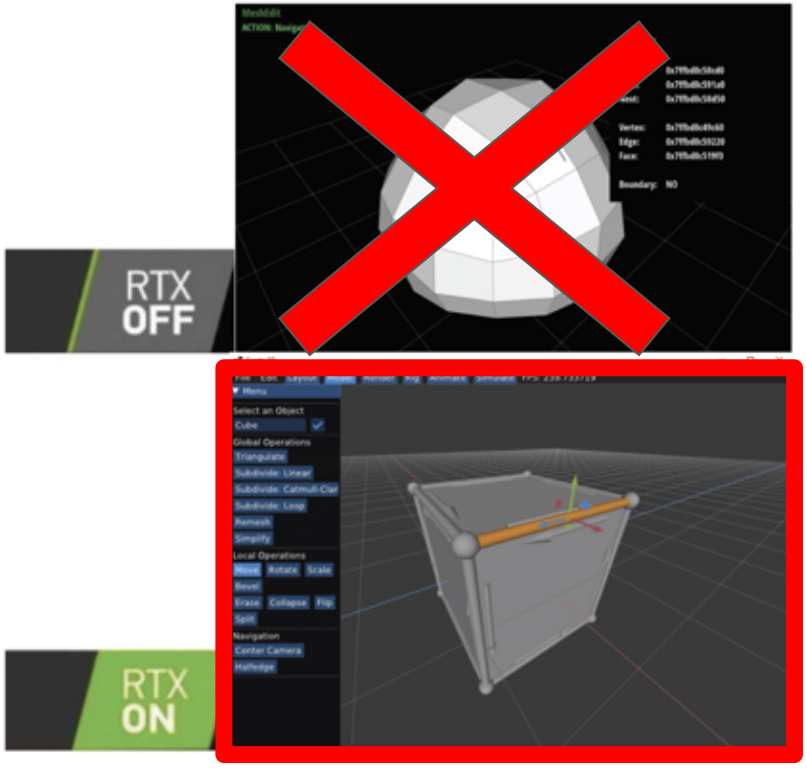
Make a pretty Image!

Make a pretty Image!

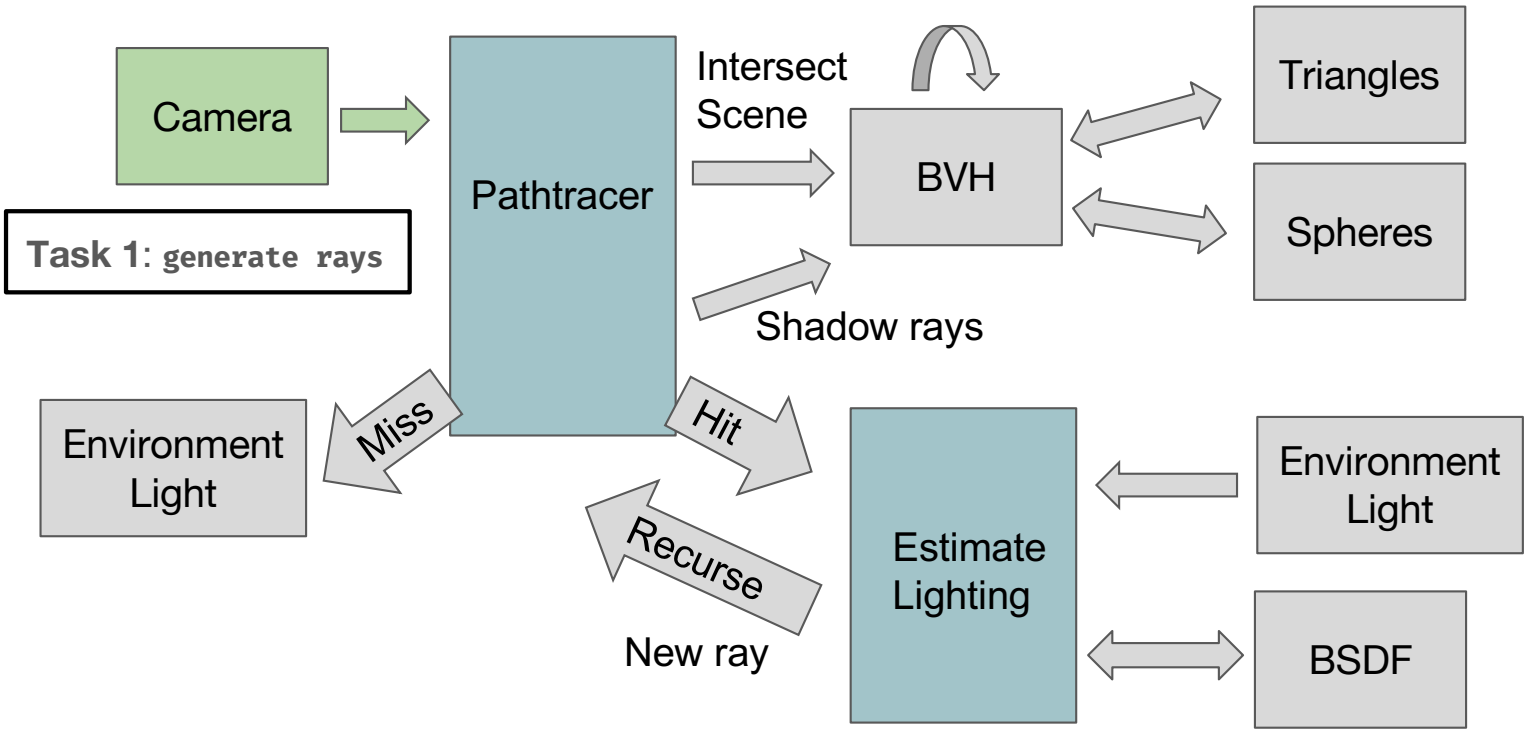


Make a pretty Image!



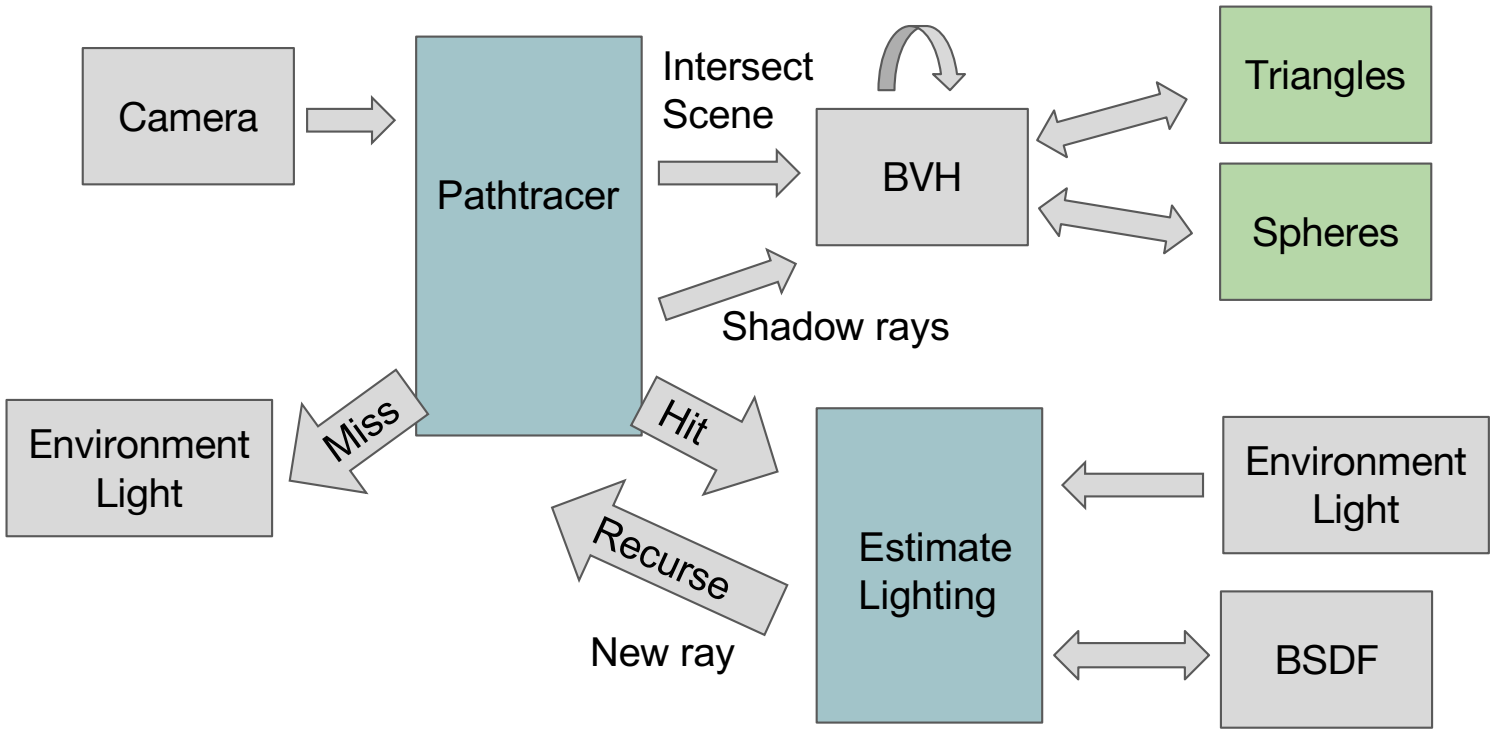


Ray Lifecycle

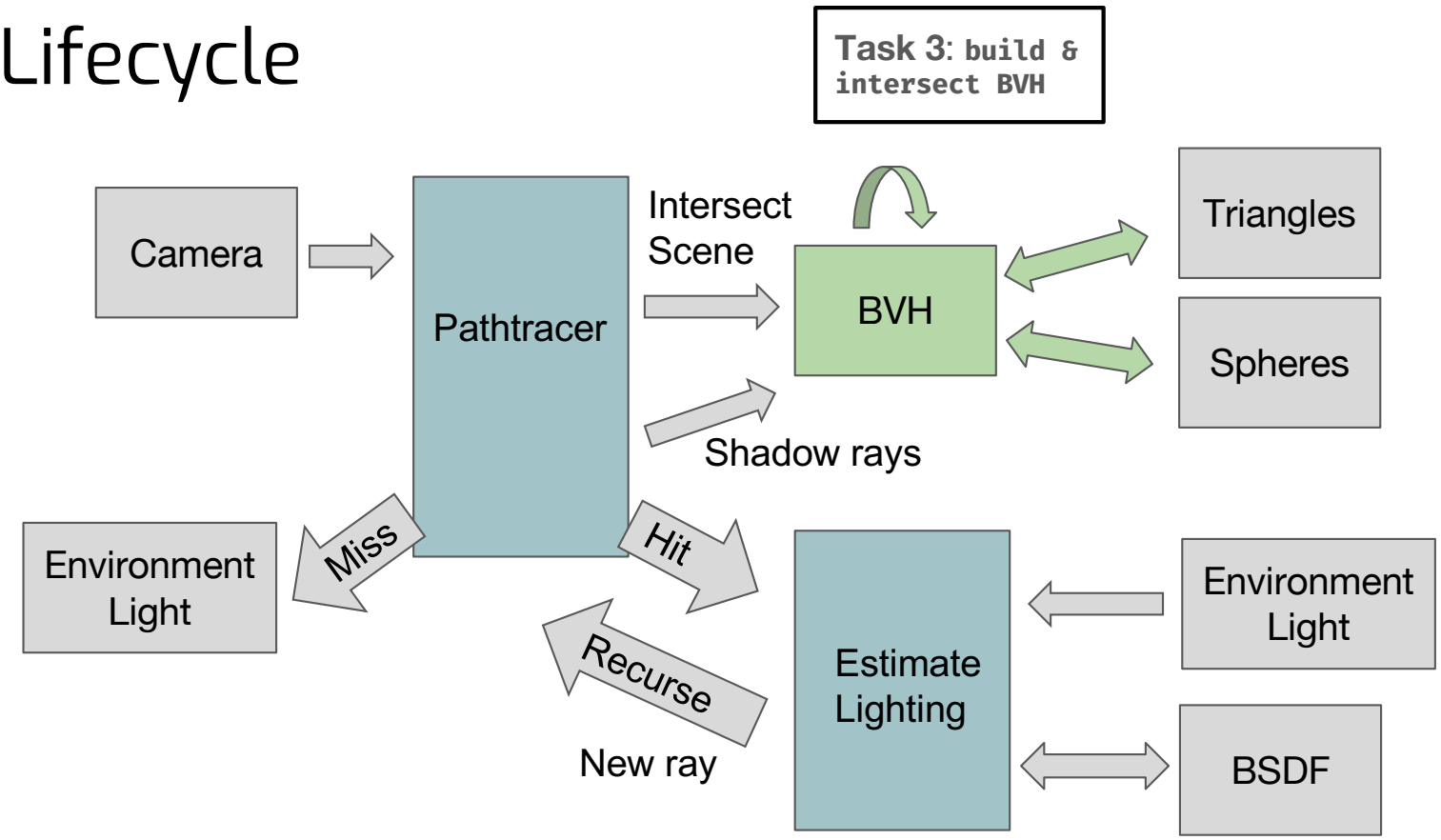


Ray Lifecycle

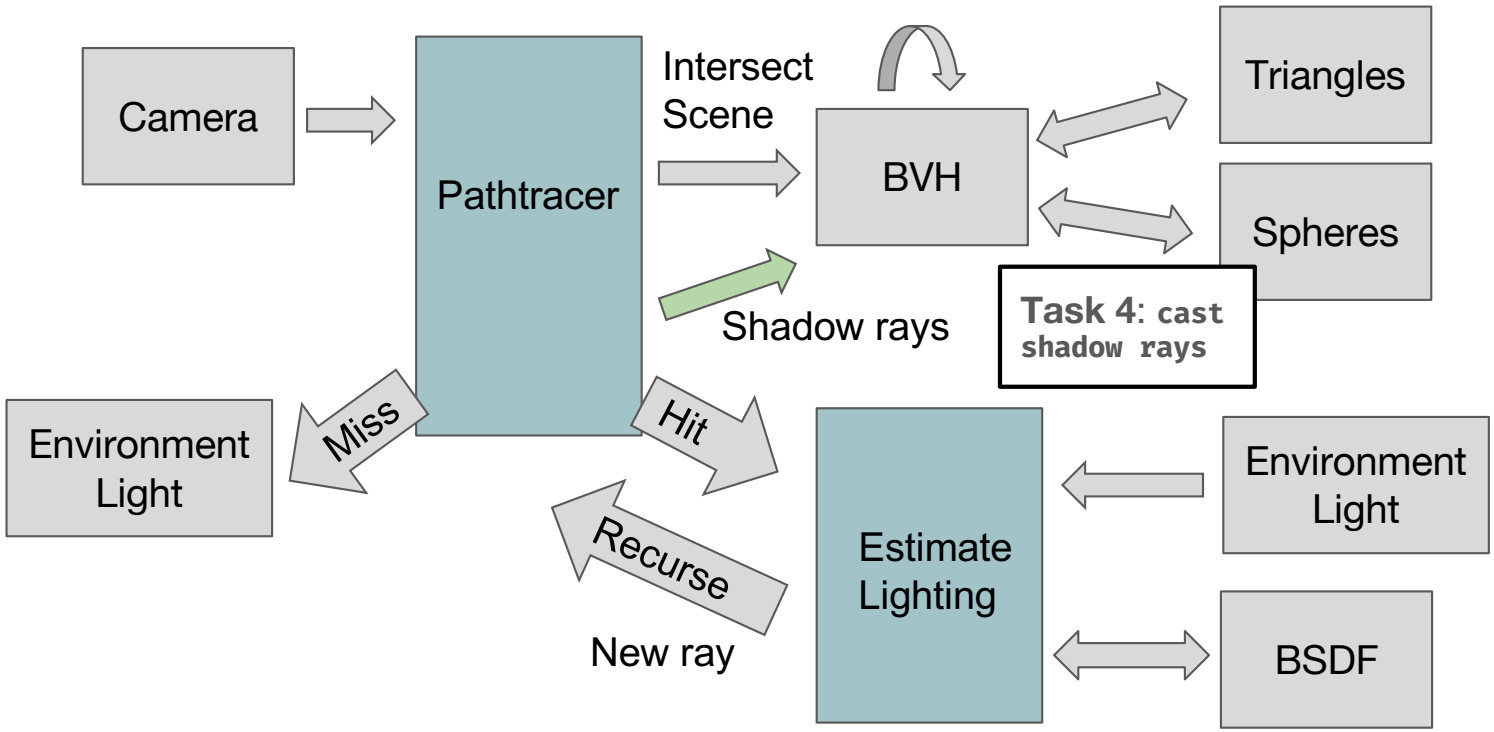
Task 2: `intersect primitives`



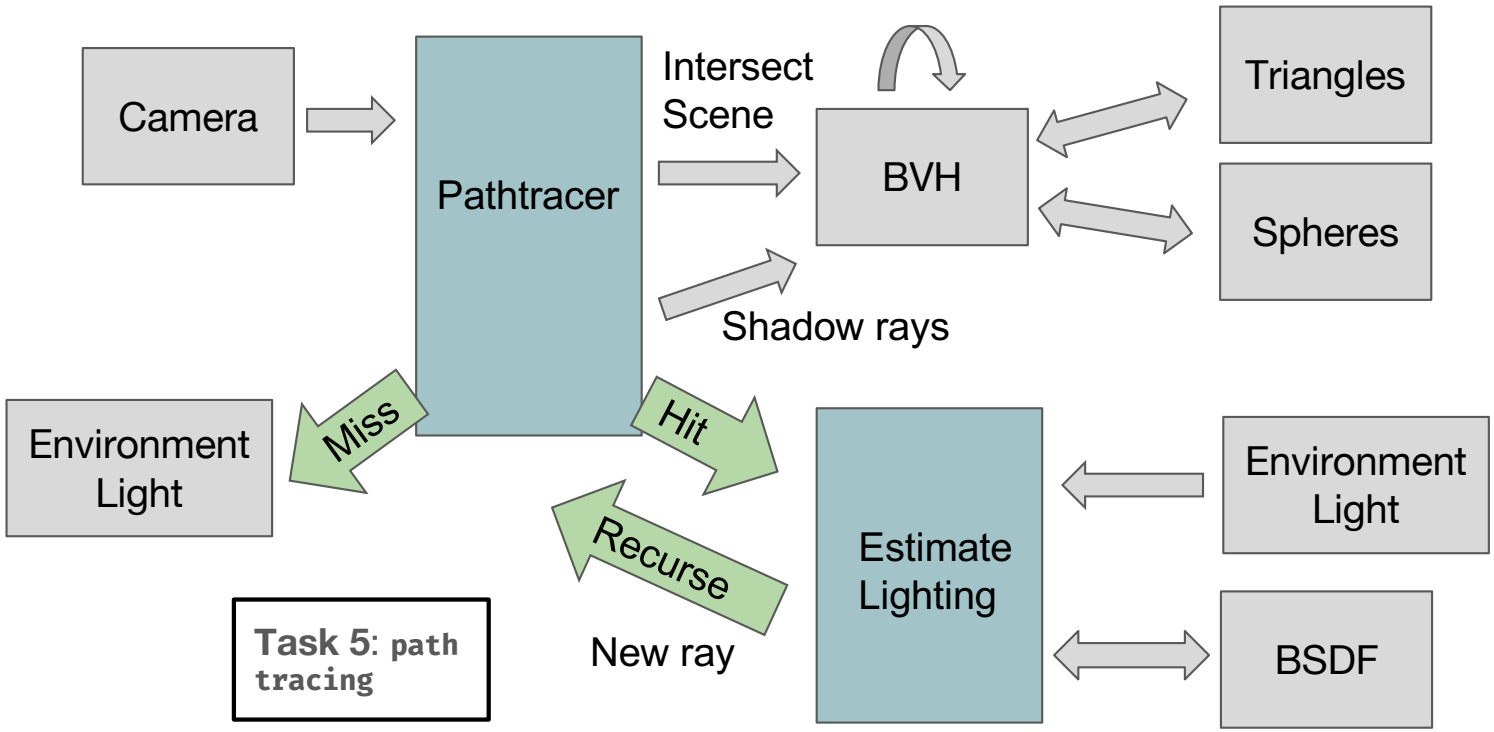
Ray Lifecycle



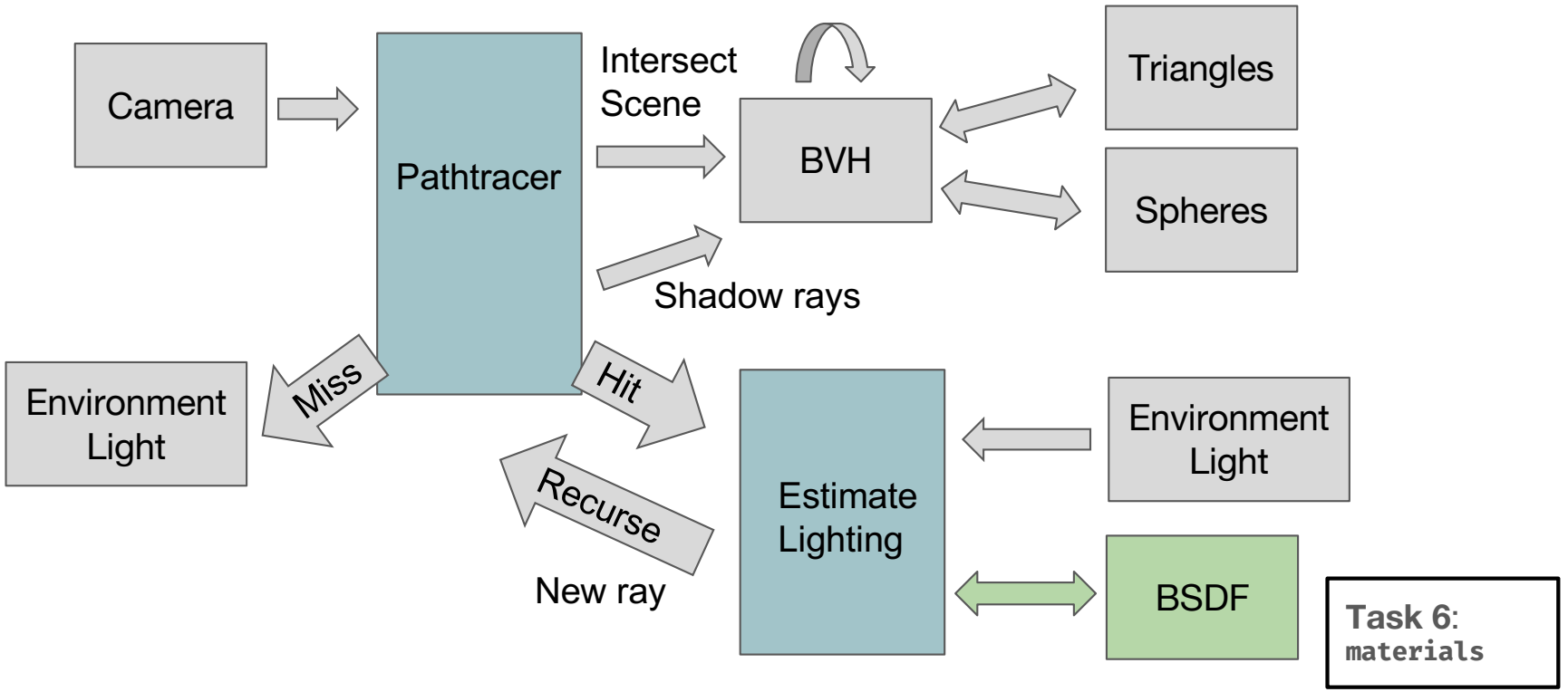
Ray Lifecycle



Ray Lifecycle



Ray Lifecycle



Ray Lifecycle