Contents

5. Adding a Sphere

5.1 Ray-Sphere Intersection

Equation of sphere centered at $\mathbf{C} = (C_x, C_y, C_z)$

$$(x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2 = r^2$$

$$(x - C_x)(x - C_x) + (y - C_y)(y - C_y) + (z - C_z)(z - C_z) = r^2$$

$$(x - C_x, y - C_y, z - C_z) \cdot (x - C_x, y - C_y, z - C_z) = r^2$$

let $\mathbf{P} = (x, y, z)$, $\mathbf{C} = (C_x, C_y, C_z)$

$$(\mathbf{P} - \mathbf{C}) \cdot (\mathbf{P} - \mathbf{C}) = r^2$$

Intersection is where $\mathbf{P}(t) = \mathbf{A} + t\mathbf{b}$ hits sphere (if anywhere), meaning there may be some t for which:

$$(\mathbf{P}(t) - \mathbf{C}) \cdot (\mathbf{P}(t) - \mathbf{C}) = r^2$$

$$((\mathbf{A} + t\mathbf{b}) - \mathbf{C}) \cdot ((\mathbf{A} + t\mathbf{b}) - \mathbf{C}) = r^2$$

$$(t\mathbf{b} + (\mathbf{A} - \mathbf{C})) \cdot (t\mathbf{b} + (\mathbf{A} - \mathbf{C})) = r^2$$

Using vector algebra:

$$t^2\mathbf{b} \cdot \mathbf{b} + 2t\mathbf{b} \cdot (\mathbf{A} - \mathbf{C}) + (\mathbf{A} - \mathbf{C}) \cdot (\mathbf{A} - \mathbf{C}) = r^2$$

$$t^2\mathbf{b} \cdot \mathbf{b} + 2t\mathbf{b} \cdot (\mathbf{A} - \mathbf{C}) + (\mathbf{A} - \mathbf{C}) \cdot (\mathbf{A} - \mathbf{C}) - r^2 = 0$$

The ray, center, and radius are known, meaning $\mathbf{A}$, $\mathbf{C}$, $\mathbf{b}$, r are all known.

$$t^2(\mathbf{b} \cdot \mathbf{b}) + t(2\mathbf{b} \cdot (\mathbf{A} - \mathbf{C})) + ((\mathbf{A} - \mathbf{C}) \cdot (\mathbf{A} - \mathbf{C}) - r^2) = 0$$

$(\mathbf{b} \cdot \mathbf{b})$, $(2\mathbf{b} \cdot (\mathbf{A} - \mathbf{C}))$, $((\mathbf{A} - \mathbf{C}) \cdot (\mathbf{A} - \mathbf{C}))$ are all scalars, meaning we can apply the quadratic formula. Let:

$a = (\mathbf{b} \cdot \mathbf{b})$
$b = (2\mathbf{b} \cdot (\mathbf{A} - \mathbf{C}))$
$c = ((\mathbf{A} - \mathbf{C}) \cdot (\mathbf{A} - \mathbf{C}))$

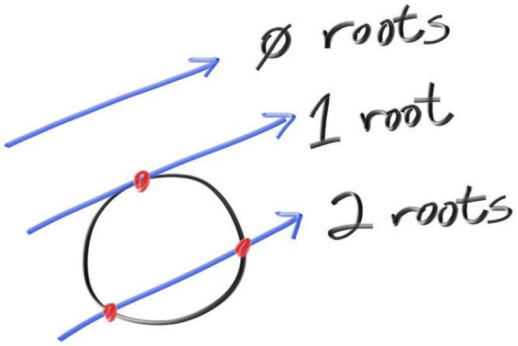
We can solve for t now:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

and the discriminant:

$$\sqrt{b^2 - 4ac}$$

This makes sense geometrically:



5.2. Raytraced Sphere Code

```
def hit_sphere(sphere center, radius, ray):
        A = ray origin, B = ray direction (ray.dir), C = sphere center
        a = B · B
        b = 2b · (A − C)
        c = (A − C) · (A − C) − r^2
        discriminant = b^2 − 4ac
        return true if ray intersects sphere

def ray_color(ray):
        if ray hits sphere(center=(0,0,-1), radius=0.5)
                return red
        otherwise, keep same shading for background
```

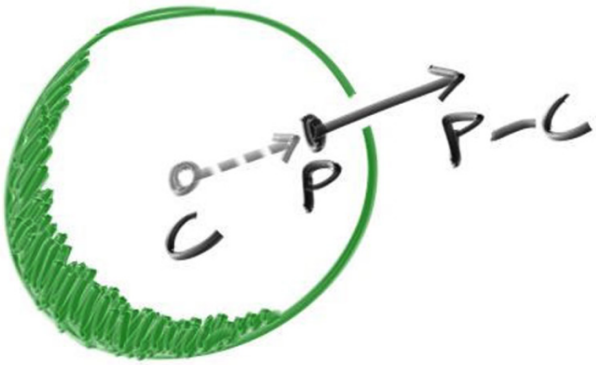
6. Surface Normals and Multiple Objects

6.1. Shading with Surface Normals

3 Reasons for Normalizing Surface Normal Vector

1. Safer to compute once for a surface normal than on per-need basis
2. We require unit normals in several places
3. Unit normals can be generated efficiently using specific geometry class (ex. sphere)

> We are going to normalize our sphere surface normal vector



$\vec{P}$ is the direction of the hit point

$\vec{C}$ is the center of the sphere

$\vec{P} - \vec{C}$ is the outward normal

$\vec{N} = \frac{\vec{P}-\vec{C}}{\|\vec{P}-\vec{C}\|}$ (surface normal vector $\vec{N}$)

```
def hit_sphere(sphere center, radius, ray):
        compute discriminant

        if ray misses sphere
                return -1
        if ray hits sphere
                return contact point: t

def ray_color(ray):
        get solutions of t for (ray intersect sphere)

        if ray hits sphere in front of camera (t>0)
                vector from center to hit (N) = vector to hit - vector to center
                normalize N
                map components of N from [-1, 1] to [0, 1] as rgb

        otherwise, keep same shading for background
```

6.2. Simplifying Ray-Sphere Intersection Code

1. $v \cdot v = \|v\|^2$
2. Consider if $b = 2h$

$$
\begin{aligned}
&\frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \\
=&\frac{-2h \pm \sqrt{(2h)^2 - 4ac}}{2a} \\
=&\frac{-2h \pm 2\sqrt{h^2 - ac}}{2a} \\
=&\frac{-h \pm \sqrt{h^2 - ac}}{a}
\end{aligned}
$$

Update code accordingly:

6.3. An Abstraction for Hittable Objects

`hittable` abstract class: anything a ray might hit

- everything individual object that can be hit is included

`hit` function accepts a `ray` argument
valid interval for t can be created: $t_{min} < t < t_{max}$

```
class hit_record: holds information about a rays hit record
        3 pieces of information:
        - point of contact
        - normal vector at point of contact
```

```
        - t at which contact was made

class hittable(): abstract base class for hittable objects (sphere)
        @abstractmethod
        def hit(ray, t_min, t_max, hit record) -> bool:
```

`sphere.py`:

```
class sphere(inherits from hittable):
        constructor takes center(point3) and radius

        def hit(ray, t_min, t_max, hit record (passed by reference)):
                calculates whether it(a sphere) is hit by ray

                if ray misses it, return false
                otherwise:
                find first possible root
                if root is outside [t_min, t_max]
                        try computing the other root
                        if second root is also out of bounds
                                return false

                fill hit record with information
                - t = roots found
                - p = point of intersection using ray.at(t)
                - normal = normalized surface normal
```

6.4. Front Faces Versus Back Faces

Design Choice: should surface normals always point **outwards** or **against the ray**?

Outwards: we know side of surface at intersection
Against the Ray: we won't know side of surface of ray until coloring, requiring us to store additional information

Matter of preference, both implementations used

> We are choosing the ray to point **against the ray** since we have more material types than geometry types

Meaning that:
if $ray \cdot outward\ normal > 0.0$, ray is inside sphere
if $ray \cdot outward\ normal <= 0.0$, ray is outside the sphere

`hittable.py`

```
class hit_record:
        add instance attribute: front_face: bool

        def set_face_normal(ray, outward normal):
                front_face = dot product of ray and outward normal
                normal = outward normal if ray hits front face
                otherwise, normal is -(outward normal)
```

`sphere.py`

```
outward_normal = (rec.p - self.center) / self.radius
```

```
rec.set_face_normal(r, outward_normal)
```

6.5. A List of Hittable Objects

Add a class that stores a list of `hittable`s
`hittable_list`

```
class hittable_list(hittable):
        objects = []

        def clear():
                clears self.objects

        def add(object(hittable)):
                adds object to self.objects

        @override
        def hit(ray, t_min, t_max, hit_record) -> bool:
                '''returns true if anything in self.objects is hit'''
                temp_hit_record = hit_record()
                hit_anything = False until something is hit
                closest_collosion_t = t_max

                iterate through all objects
                        if any object is hit
                                hit_anything is set to true
                                update closest t of intersection
                                temp_rec updated when hit() called, so update hit_record
using deepcopy
                return hit_anything
```

6.7. New `main.py`

```
def ray_color(ray, hittable):
        hitrecord
```

Bug

Experienced a bug where the `hit()` method in `hittable_list` reassigned the reference `rec` to a new object, instead of modifying the object originally referenced by `rec`.
This was solved by implementing a `copy()` method in `hit_record` which copies the instance attributes from another `hit_record` object `other`.

Tip

However, it will be faster to directly assign the variables, despite being inelegant

```
Scanlines remaining: 177 Traceback (most recent call last):
  File "/Users/caleb/Documents/Projects/raytracer/src/main.py", line 84, in <module>
    main()
```

```
  File "/Users/caleb/Documents/Projects/raytracer/src/main.py", line 75, in main
    pixel_color = ray_color(r, world)
                  ^^^^^^^^^^^^^^^^^^^
  File "/Users/caleb/Documents/Projects/raytracer/src/main.py", line 14, in ray_color
    return 0.5 * (rec.normal + rgb(1, 1, 1))
                  ~~~~~~~~~~~^~~~~~~~~~~~~~
  File "/Users/caleb/Documents/Projects/raytracer/src/vec3.py", line 34, in __radd__
    return self.__add__(v)
           ^^^^^^^^^^^^^^^
  File "/Users/caleb/Documents/Projects/raytracer/src/vec3.py", line 31, in __add__
    raise TypeError(vec3.te.format(op="vector addition") + f": {type(v)}")
TypeError: Unsupported operand type for vector addition.: <class 'NoneType'>
```

6.8. An Interval Class

- manage real-valued intervals with a minimum and maximum

change `hittable`'s `hit()` to

```
@abstractmethod
def hit(_r: ray, ray_t: interval, rec: hit_record) -> bool:
        pass
```

replace all `t_min`, `t_max` with `interval`

7. Moving Camera Code Into Its Own Class

`camera` class has two jobs:

1. Construct and dispatch rays into the world
2. Use results of these rays to construct rendered image

Camera refactor:

- collect `ray_color()`
- image, camera, render sections of `main.py`

Design:

- default constructed with no arguments
- owning code with directly modify camera's public variables
- camera will call `initialize()` at start of `render()`

> Design choices used by this section are questionable

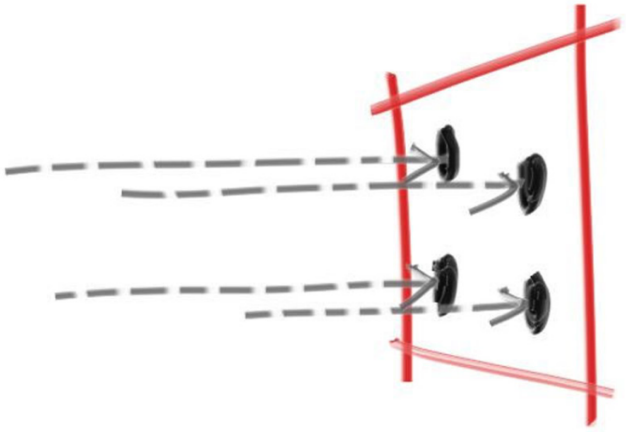
8. Antialiasing

- A true image of the world has effectively infinite resolution. So far, our ray tracer
- raytracer should ideally integrate neighboring colors just as our eyes would

    - black-and-white checkerboard example
- Our point sampling does not take into account any other point except the one it intersects, but it should to model a "real-world" rendering

Modifying Raytracer Point Sampling

- sample the square region centered at the pixel that extends halfway to each of the four neighboring pixels
- approximate result of initial pixel using the results



8.1. Random Number Utilities

Random number generator returns $0 \leq n < 1$

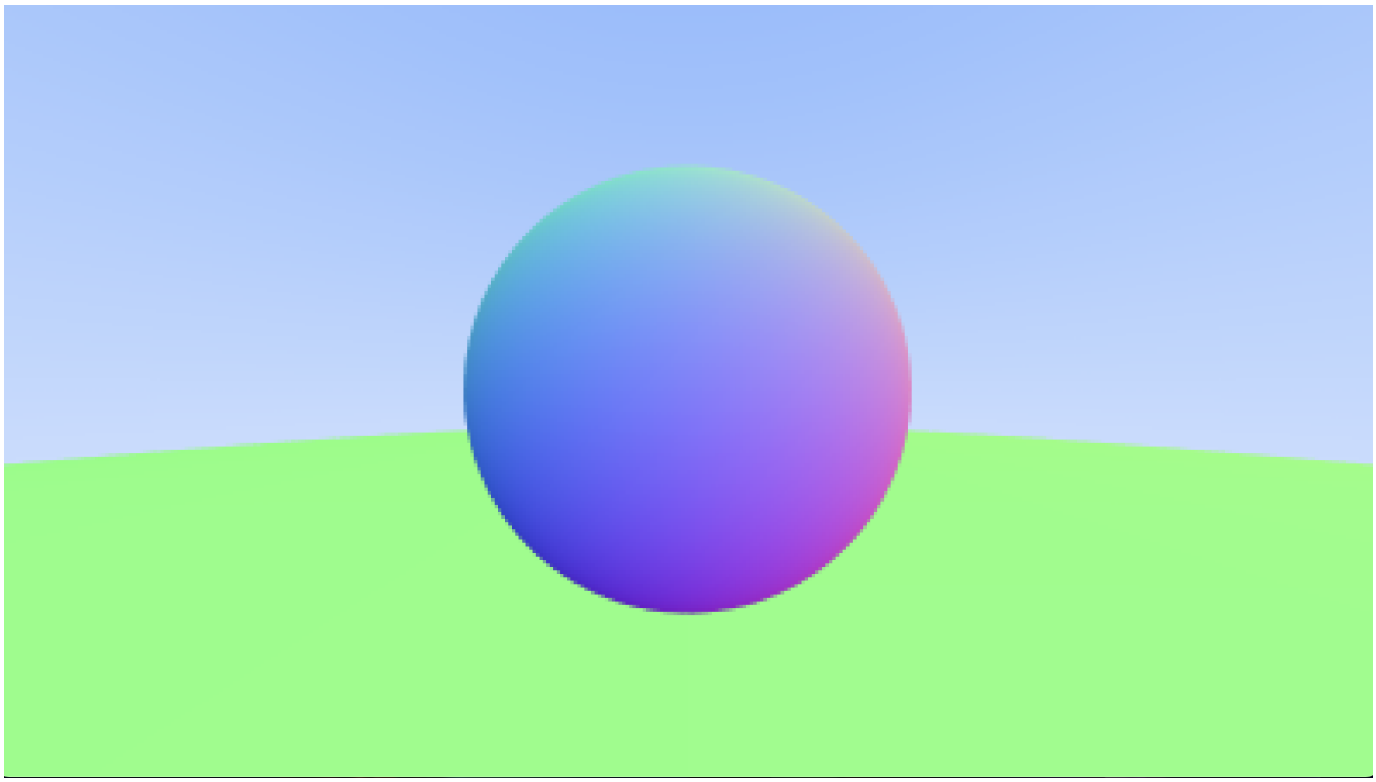
8.2. Generating Pixels with Multiple Samples

Method:

1. Collect samples from surrounding area described
2. Average resulting color values

> Updated `interval`: added `clamp`
> Updated `write_color`
> Change `camera.render` to include new methods `get_ray` and `pixel_sample_square`

Gradient sphere render:



Bug

Experienced bug with `+=` operator between two instances of `vec3`. `vec3.` In `camera.py`, the line `pixel_color += rc` is calling `__radd__` in `vec3.py` which in turn calls `__add__` (line 28).

Unsure (as of now) why the wrong method is being called. Also unsure why even after calling the wrong method, the second operand becomes `NoneType` somewhere along the line.

```
(venv) caleb@Calebs-MBP-75 raytracer % python src/main.py > image.ppm Scanlines remaining:
225 <class 'vec3.vec3'> <class 'NoneType'> Traceback (most recent call last): File
"/Users/caleb/Documents/Projects/raytracer/src/main.py", line 28, in <module> main() File
"/Users/caleb/Documents/Projects/raytracer/src/main.py", line 25, in main cam.render(world)
File "/Users/caleb/Documents/Projects/raytracer/src/camera.py", line 42, in render
pixel_color += rc File "/Users/caleb/Documents/Projects/raytracer/src/vec3.py", line 36, in
__radd__ return self + v ~~~~~^~~ File
"/Users/caleb/Documents/Projects/raytracer/src/vec3.py", line 32, in __add__ raise
TypeError(vec3.te.format(op="vector addition") + f": {type(v)}") TypeError: Unsupported
operand type for vector addition.: <class 'NoneType'>
```

9. Diffuse Materials

We begin creating materials, starting with diffuse materials (*matte*)

As for the approach, do we:

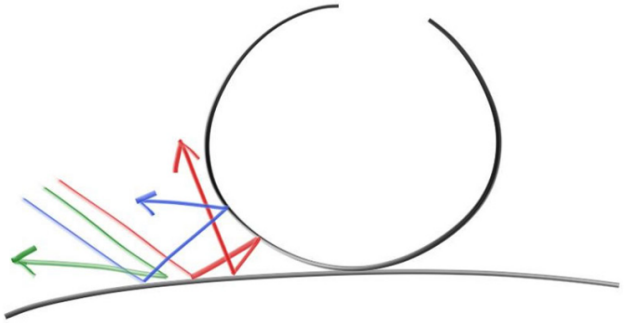
1. Mix and match geometry and materials (so we can assign a material to multiple spheres or vice versa)?

2. Keep geometry and materials tightly bound (useful for procedural objects where geometry and material are linked)?

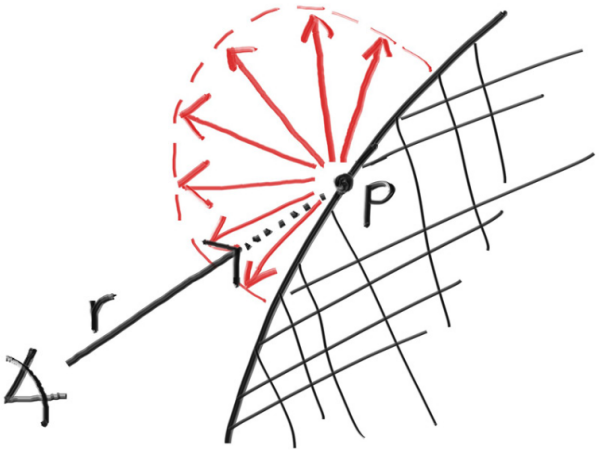
> We will pick the first option (common in most renderers)

9.1. Simple Diffuse Material

- non-light-emitting diffuse objects take on color of surroundings, modulated by own intrinsic color
- light reflecting off diffuse surface has direction randomized



- ray can also be absorbed instead of reflected
  - darker surface → more likely ray is absorbed
- any algorithm that randomizes direction will produce surface that look matte



> add `random()` function to `vec3` using `helper.rand_double`

Next task is to manipulate random vector → only get results on surface of a hemisphere. Analytical methods for this are complicated both to understand and implement.

We will use a rejection method:

1. Generate a random vector inside of unit sphere
2. Normalize this vector
3. Invert if it falls onto wrong hemisphere

1:

```
def rand_in_unit_sphere() -> vec3:
        while True:
                p: vec3 = vec3.random(-1, 1)
                if p.length_squared() < 1: # within unit sphere
                        return p
```

2:

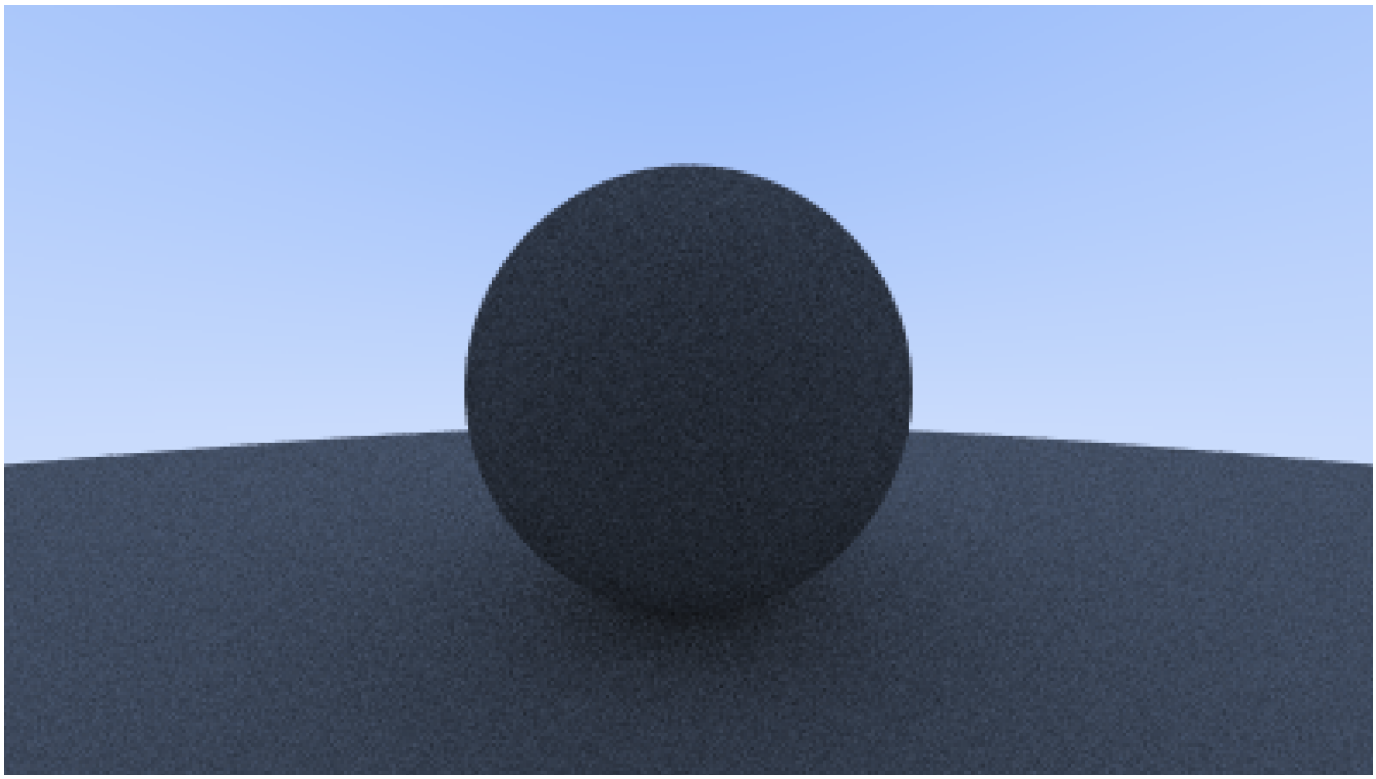
```
def rand_unit_vec() -> vec3:
    '''returns random vector ON unit sphere'''
    return normalize(rand_in_unit_sphere())
```

3:

```
def rand_on_hemisphere(normal: vec3) -> vec3:
    on_unit_sphere: vec3 = rand_unit_vec()
    return on_unit_sphere if (dot(on_unit_sphere, normal) > 0.0) else -on_unit_sphere
```

Update `camera.ray_color`:

```
if ray hits object in world:
        random_direction
        shoot new random_direction ray into world # recursive
```

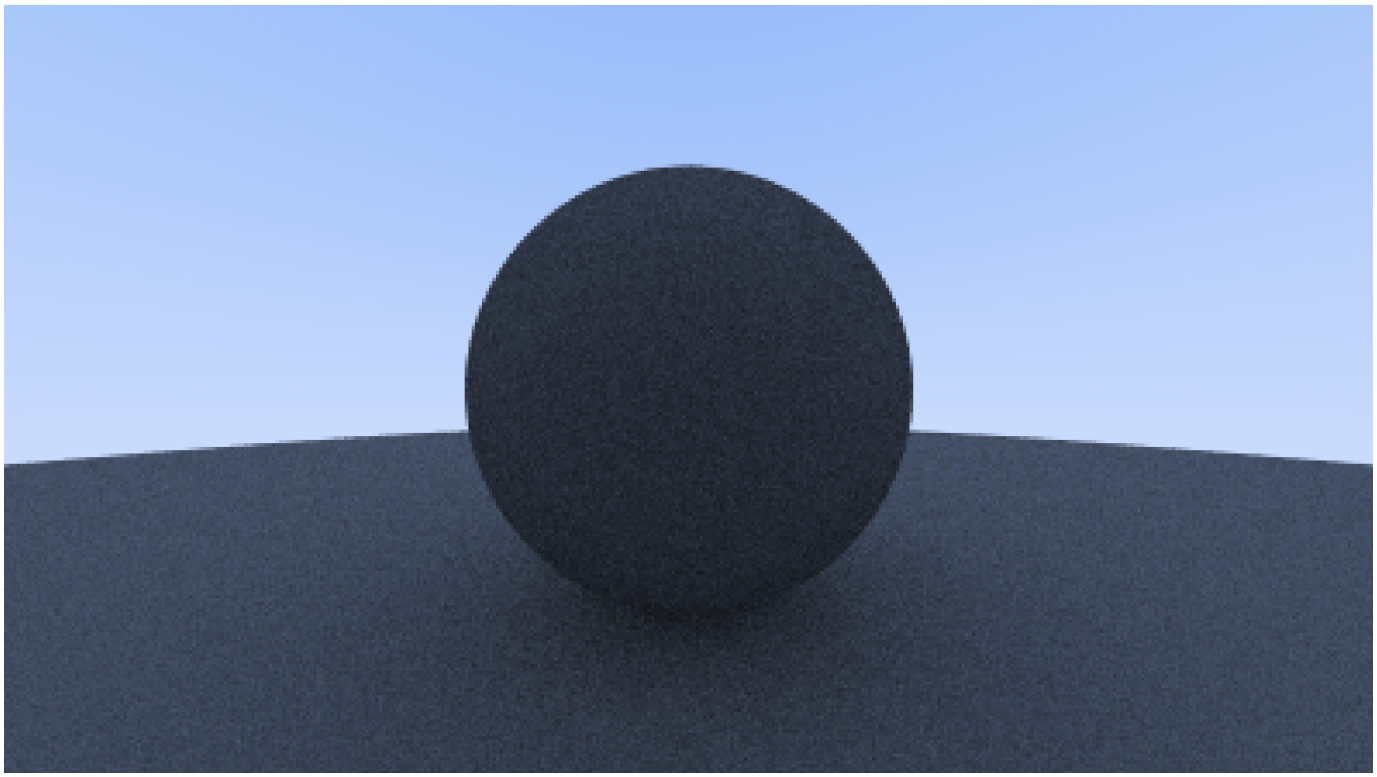


9.2. Limiting the Number of Child Rays

Recursion maximum depth

```
class camera:

    def __init__(self) -> None:
        ...
        self.max_depth: int = 10
    ...
    ...
    def render(world):
            ...
            rc = self.ray_color(r, self.max_depth, world)
        ...
        ...
        def ray_color(ray, depth, world):
                ...
                if world.hit(r, interval(0, inf), rec)
                        rand_dir = rand_on_hemisphere
                        return 0.5 * ray_color(ray(rand_dir), depth - 1, world)
```



9.3. Fixing Shadow Acne

The problem

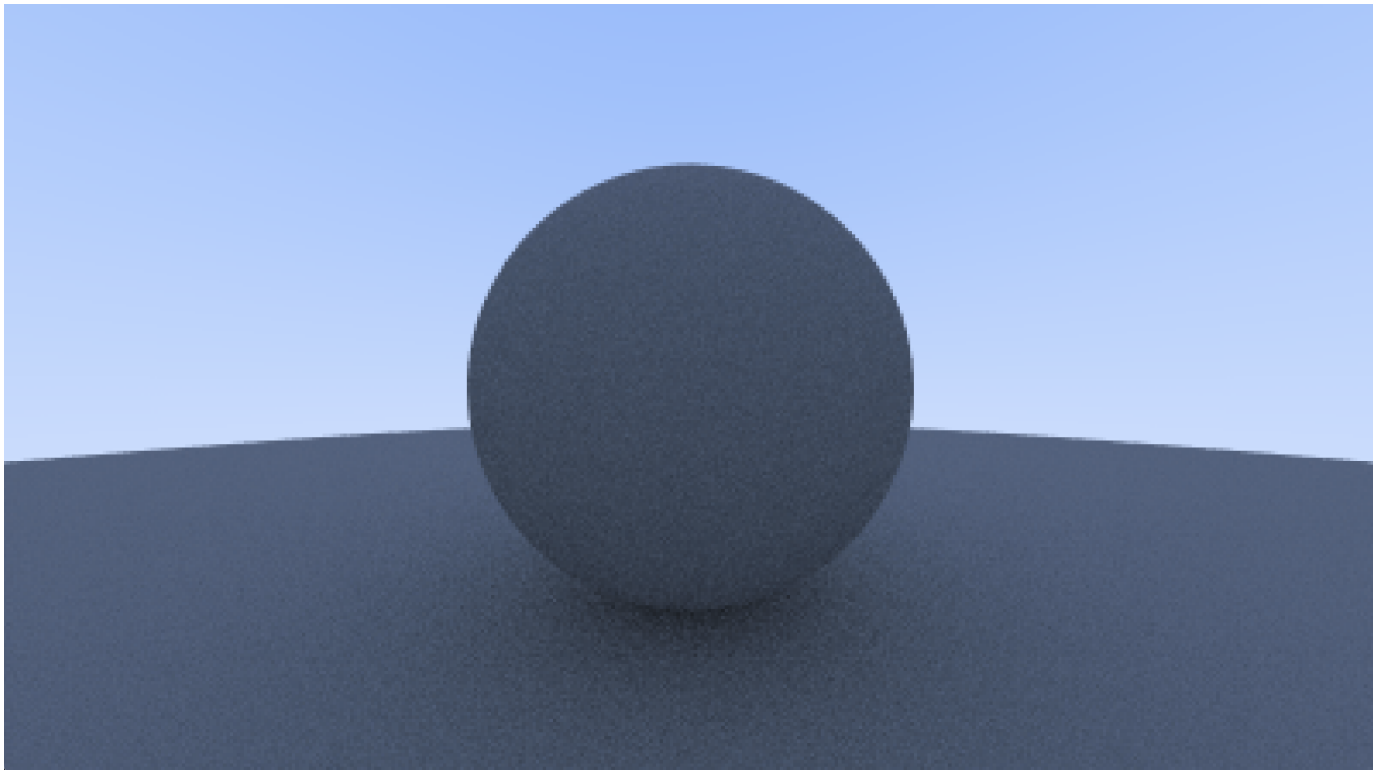
- Floating point arithmetic errors may cause ray's origin to be slightly above/below surface
- origin of next ray is not perfectly aligned w/surface
- randomly scattered ray may intersect with same surface again due to small errors $\rightarrow$ incorrect shading/lightling calculations

Solution

- ignore hits that are very close to calculated intersection point
- avoids treating small errors as valid intersections, reducing likelihood of shadow acne and improving accuracy of subsequent calculations

change lower bound of interval in `camera.ray_color` to `0.001`:

```
class camera:
        ...
        def ray_color(ray, depth, world):
                ...
                if world.hit(r, interval(0.001, inf), rec)
                        ...
```



9.4. True Lambertian Reflection

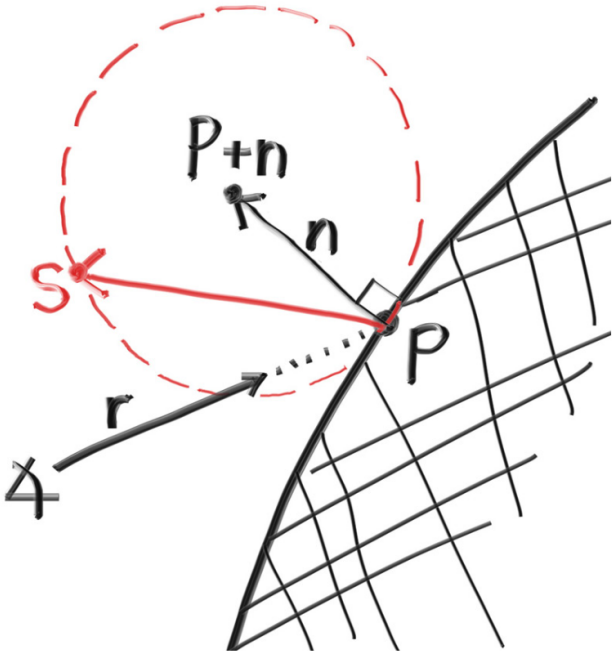
Current Approach: even scattering about hemisphere

More Accurate Approach: *Lambertian* distribution scatters reflected rays proportional to $cos(\phi)$, (ϕ is angle between reflected ray and normal)

Method

We achieve this by adding a random unit vector to the **surface normal** which is already oriented in the "*natural*" hemisphere direction. Therefore, even if the normal vector direction is randomized, the amount of possible reflected rays will be **densest** close to the direction of the normal

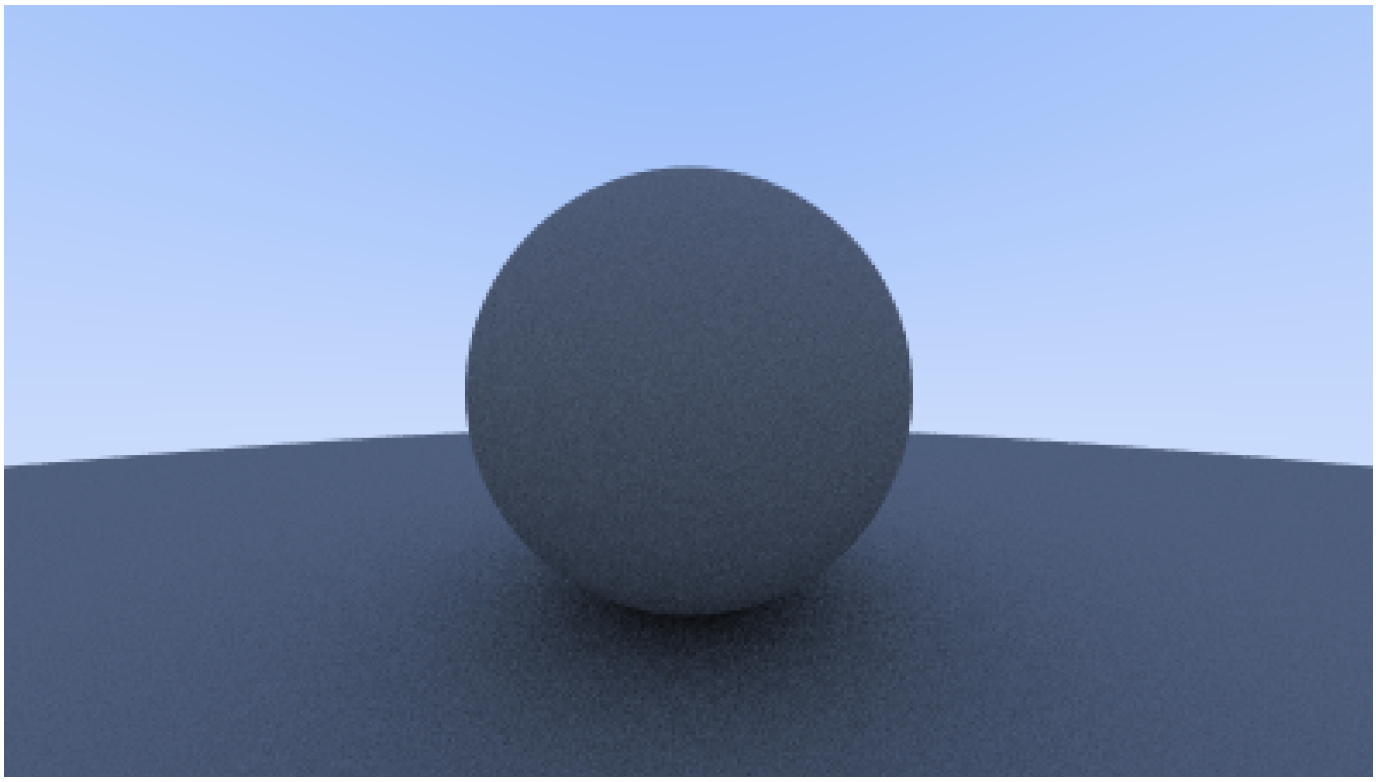
- Two unit spheres placed tangent to point of intersection, one with center $(\mathbf{p} + \mathbf{n})$ (outside surface) and the other with center $(\mathbf{p} - \mathbf{n})$ (inside surface)
- Select tangent unit sphere on same side of surface as ray origin
- Pick random point $\mathbf{S}$ on unit sphere and send ray from hit point, the new ray will have the direction vector $(\mathbf{S} - \mathbf{P})$



(vector generation according to Lambertian distribution)

```
class camera:
        ...
        def ray_color(ray, depth, world):
                ...
                if world.hit(r, interval(0.001, inf), rec)
                        new_dir = rec.normal + random_unit_vector()
                        ...
```

Result:



1. Shadows are more pronounced after the change

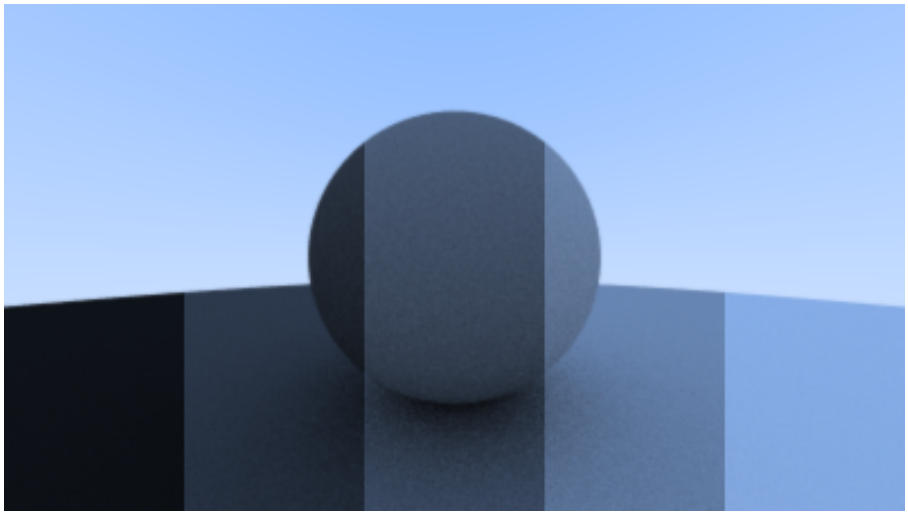
2. Both spheres are tinted blue from the sky after the change

More rays scattered towards the normal, meaning that diffuse objects will appear darker
For shadows, more light bounces straight up $\rightarrow$ area underneath sphere is darker

9.5. Using Gamma Correction for Accurate Color Intensity

Render for different diffuse ray reflection percentages

- 10%, 30%, 50%, 70%, 90%



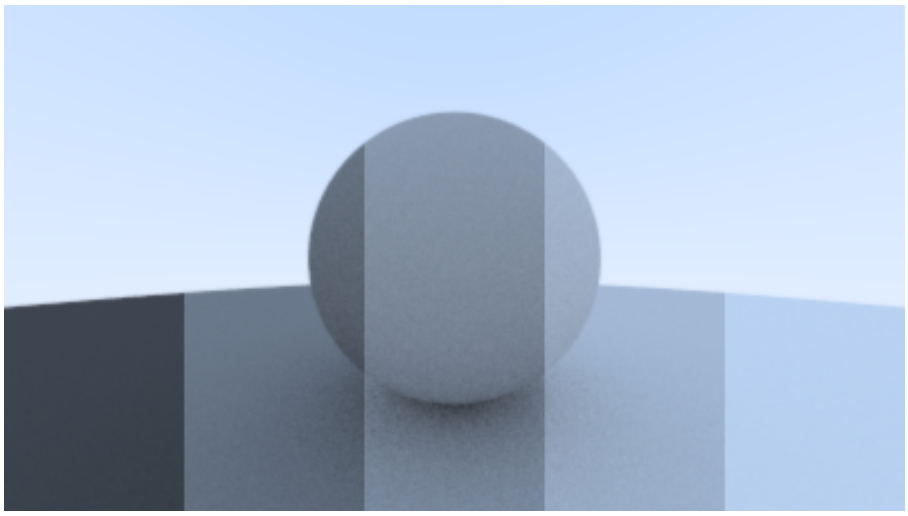
The middle section, which has a 50% diffuse ray reflection is too dark to be half-way between white and black

70% is close to middle-gray

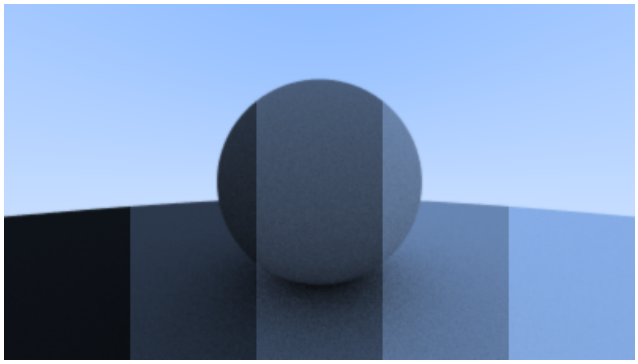
Why?

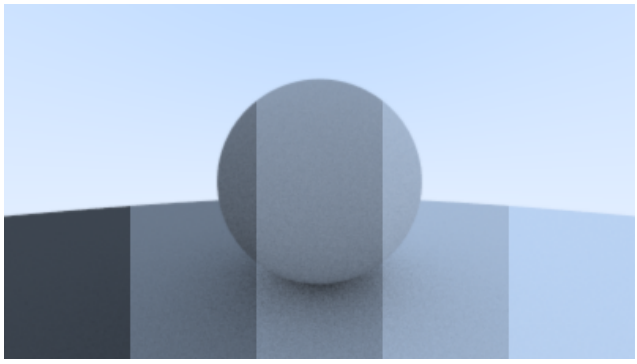
- computer programs assume image is *gamma corrected* before being written into image file
  - meaning 0-1 values have some transform applied before being stored as byte
  - *linear space*: untransformed
  - *gamma space*: transformed
- why gamma space is used? who cares for now.

Gamma space overlayed render:



Linear (first) and gamma (second) side-by-side comparison:





10. Metal

10.1. An Abstract Class for Materials

Question -- should we:

1. Have a universal material type which accepts lots of parameters, where individual material types can simply ignore parameters that don't affect it
2. Create abstract material class that encapsulates unique behavior

> Second approach is the more object-oriented approach.

Every material has two behaviors:

1. Produce a scattered ray (or absorbed incident ray)
2. If scattered, say how much the ray should be attenuated

10.2. A Data Structure to Describe Ray-Object Intersections

> **Notice**
>
> To avoid a possible circular import, `hit_record` never imports `material` and simply initializes an instance attribute `mat` to `None` by default
>
> The abstract base class `material` does import `hit_record` from `hittable`

How it's going to work:

- `hit_record`'s job is simply to package multiple pieces of information about a certain hit point
- At contact, `hit_record`'s `material` should be set to the `hittable`'s material (assigned at the beginning)
- `ray_color` can then call member functions of whatever `material` is stored in `hit_record`

> add `mat` to `hit_record`
> updated sphere to have a `material` attribute

10.3. Modeling Light Scatter and Reflectance

Our Lambertian (diffuse case) can either:

1. always scatter and attenuate by reflectance (which is actually a probability) which we'll call R
2. sometimes scatter (w/probability $1-R$) with not attenuation (ray is only absorbed and not scattered)
3. some mixture of both, for example scatter w/fixed probability p and have attenuation be `albedo`$/p$

> We will choose to always scatter:

```
class lambertian(material):
        ...
        def scatter(self, ray_in, hit_record, &attenuation, &scattered):
                calculate scatter_dir using same method as before
                create the scatter ray using the scatter_dir
                attenuation is the albedo of this surface
                always return true for now
```

Note that we are free to implement any of the 3 options above

> **Careful**
>
> There is a small chance for the random unit vector to be directly oposite of the normal vector in `scatter_dir = _rec.normal + rand_unit_vec()`. `scatter_dir` will be a zero vector which could lead to unwanted behavior (infinity, NaN)

> create new vector method `vec3.near_zero()` which returns true if vector is very close to being $\vec{0}$ (how necessary is this, really?)

WORKING ON CURRENTLY:

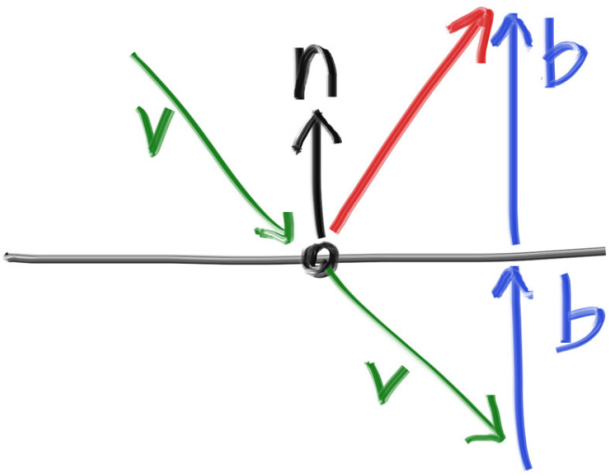
- [x] ~~work on implementing copy functions for `vec3.py` and `ray.py` or using `dataclasses` (or maybe just stick to directly assigning the attributes for now)~~
- [x] ~~this is so you can correctly and elegantly implement `scatter` in `material.py`~~

FOCUS ON LATER (MAYBE AFTER TUTORIAL):

- [ ] reduce method calls across modules (or method calls in general) to speed up code (**KEEP IN MIND**)
- [x] ~~make all class attributes public~~
- [x] ~~set up timer in `camera.render` to measure performance of code~~
- [ ] consider precomputing `vec3` values such as `length` for speedups (**SAVE FOR LATER**)
    - rays will be shot out per pixel
    - `render` calls `ray_color`
        - if `world.hit`
            - `hittable_list.hit` → `hittable.hit` (sphere)
                - `length_squared()` (x2)
            - `rand_unit_vec()` (calls **`length`**) and (**STOPPED HERE**)
            - recursive with max recursion depth of 10 by default (will likely be set to more)
        - for no-contact rays:
            - `normalize` will still be called to create the sky
                - will call **`length`**
- [x] ~~in `camera.py`, remove `initialize()` completely and move all functionality to `__init__`~~
- [x] ~~in `vec3.py`, condense random_unit_sphere functions into one or two~~

10.4 Mirrored Light Reflections

Metals will reflect a ray instead of scatter it. How?



To think about this, let the contact point be (0, 0)

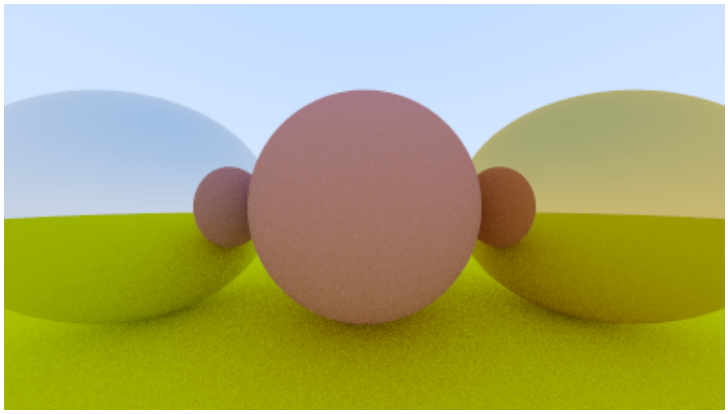
- $\mathbf{v}$ is the bottom right green vector originating from the origin

- assume $\mathbf{n} = (0, 1)$. $\mathbf{v} \cdot \mathbf{n}$ would be negative
- so the $\mathbf{b}$ shown is actually $-(\mathbf{v} \cdot \mathbf{n})\vec{n}$ translated
- Therefore, both blue vectors are really $-2(\mathbf{v} \cdot \mathbf{n})\vec{n}$
- And the red is $\mathbf{v} - 2(\mathbf{v} \cdot \mathbf{n})\vec{n}$

In code: `v - 2*dot(v, n) * n`

10.5. A Scene with Metal Spheres

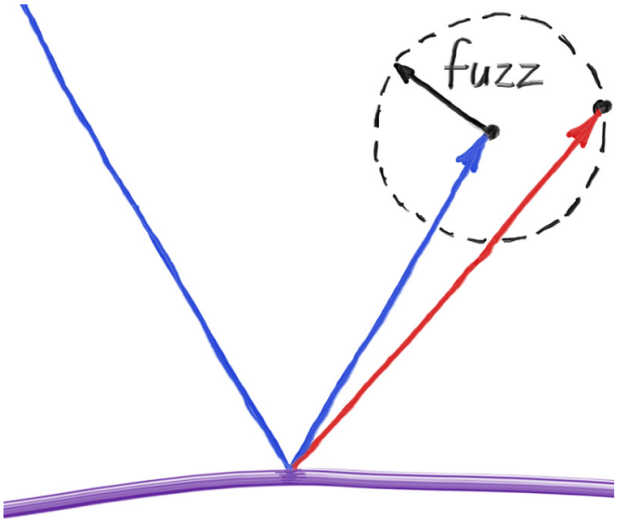
Result (137.5 seconds):



10.6. Fuzzy Reflection

Can slightly randomize reflected direction using a small sphere $\rightarrow$ new endpoint for ray

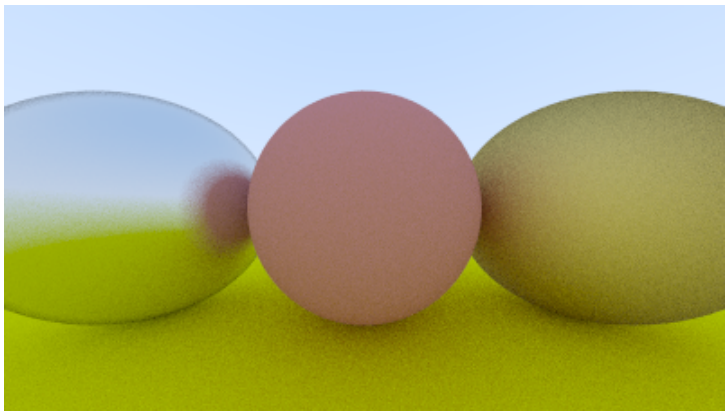
- sphere centered on original endpoint
- sphere radius is fuzz factor



Implementation is relatively simple:

> add `fuzz` to `metal`
> update `main`

Render:



11. Dielectrics

Include:

- water, glass diamond

11.1. Refraction

Light rays that intersect split into reflected and refracted (transmitted ray)

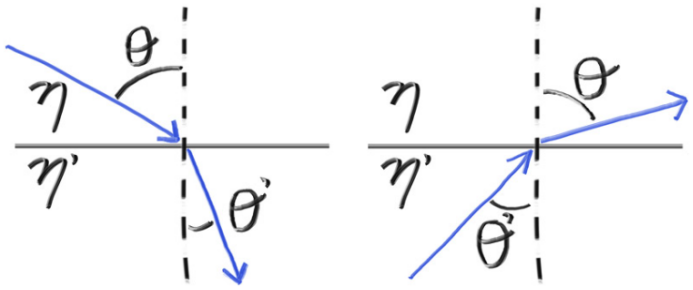
- handle by randomly choosing between reflection and refraction (one scattered ray per intersection)

11.2. Snell's Law

Refraction is described by Snell's Law

$$\eta \cdot \sin\theta = \eta' \cdot \sin\theta'$$

θ, θ' : angles from normal (before and after intersection)
η, η' : refractive indices (before and after intersection)



Solve for the direction of the refracted ray:

$$\sin\theta' = \frac{\eta}{\eta'} \cdot \sin\theta$$

On refracted side of surface, there is the refracted ray $\mathbf{R}'$ and normal $\mathbf{n}'$ with θ' separating them.

Since $\mathbf{R}'$ is a vector, we can split it into the parts of the ray that are perpendicular to $\mathbf{n}'$ and parallel to $\mathbf{n}'$

$$\mathbf{R}' = \mathbf{R}'_{\perp} + \mathbf{R}'_{\parallel}$$

Solving for $\mathbf{R'}_\perp$ and $\mathbf{R'}_\parallel$:

$$\mathbf{R'}_\perp = \frac{\eta}{\eta'}(\mathbf{R} + \cos\theta\mathbf{n})$$

$$\mathbf{R'}_\parallel = -\sqrt{1 - \|\mathbf{R'}_\perp\|^2}\mathbf{n}$$

(I don't really understand this part, but I want to finish this raytracer for now)

However, every term on the right-hand side is known save $\cos\theta$
We know:

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\|\|\mathbf{b}\|\cos\theta$$

Restricting $\mathbf{a}$ and $\mathbf{b}$ to be unit vectors:

$$\mathbf{a} \cdot \mathbf{b} = \cos\theta$$

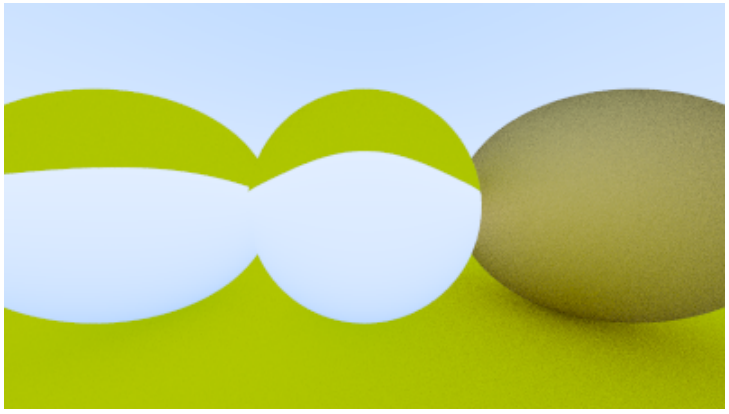
$\mathbf{R'}_\perp$ can now be rewritten as:

$$\mathbf{R'}_\perp = \frac{\eta}{\eta'}(\mathbf{R} + (-\mathbf{R} \cdot \mathbf{n})\mathbf{n})$$

Now, we can write a function to calculate $\mathbf{R'}$:

> create `refract` in `vec3` module
> create `dielectric` material class in `material.py`

Render (136.7 seconds):



(glass sphere which always refracts)

11.3. Total Internal Reflection

The render is obviously inaccurate because glass looks nothing like that in the real world

On reason is because when the ray is in a solution with the higher refractive index, there is no solution to Snell's law:

$$\sin\theta' = \frac{\eta}{\eta'} \cdot \sin\theta$$

In case of air ($\eta = 1.5$) $\rightarrow$ glass ($\eta' = 1.0$):

$$\sin\theta' = \frac{1.5}{1.0} \cdot \sin\theta$$

The value of $\sin\theta'$ cannot be greater than 1:

$$\frac{1.5}{1.0} \cdot \sin\theta > 1.0$$

No solution, clearly.
The glass therefore cannot refract the ray and must reflect it instead

```
if refraction_ratio * sin_theta > 1.0:
        # must reflect
else:
        # can refract
```

Let's solve for `sin_theta` using `cos_theta`:

$$\sin\theta = \sqrt{1 - \cos^2\theta}$$

and

$$\cos\theta = \mathbf{R} \cdot \mathbf{n}$$

```
sin_thet = square root of (1 - cos_theta squared)
```

> update dielectric material to perform check on whether material can refract or not

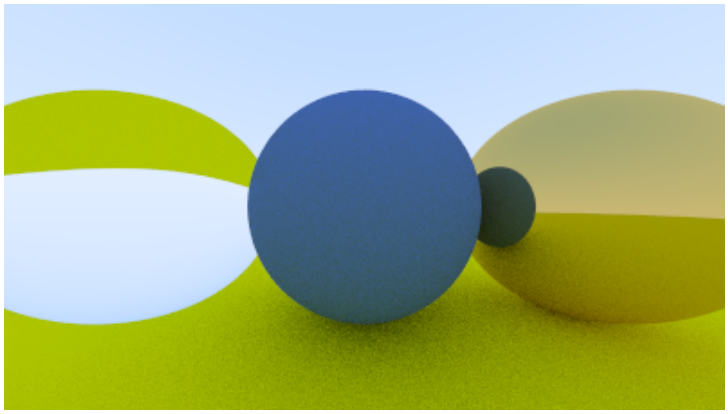
```
calculate sin_theta
calculate cos_theta

cannot_refract (bool) checks if refraction_ratio*sin_theta > 1.0

if we cannot refract:
        direction calls reflect()
else
        direction calls refract()

assign scattered (passed in) according to contact point and direction
return True
```

Render:



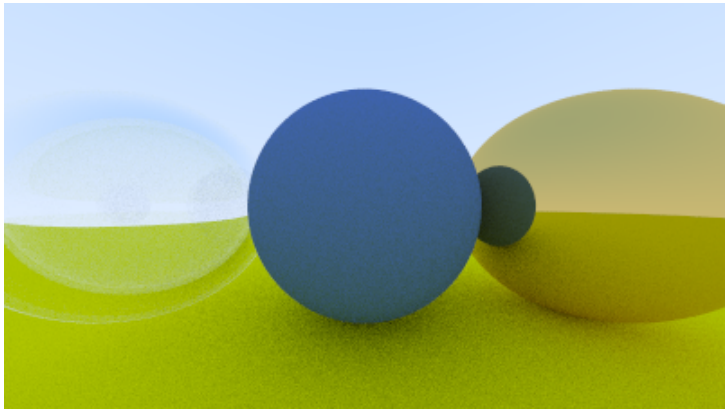
11.4. Schlick Approximation

In reality, reflectivity varies with angle. Our model can be made more accurate using the **Schlick Approximation**.

11.5. Modeling a Hollow Glass Sphere

Using a negative radius inverts the surface normals (to point inward) which can be used as a bubble to make a hollow glass sphere

Render (193.8 seconds):

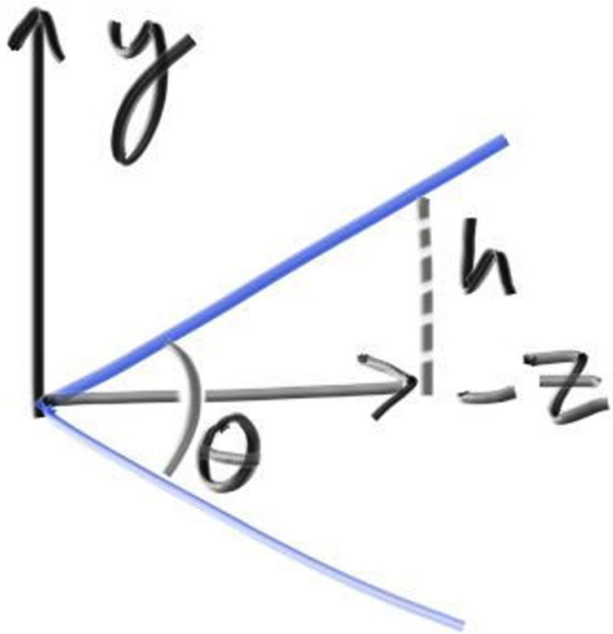


12. Positionable Camera

Incrementally develop camera:

1. Adjustable FOV
2. Positioning an Orientation

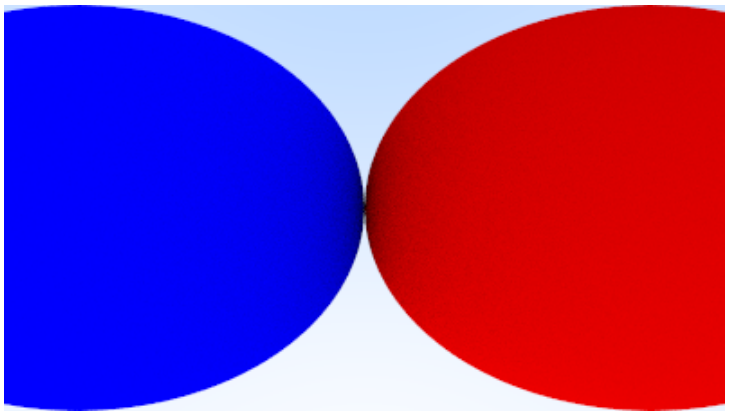
12.1. Camera Viewing Geometry



$h = \tan(\frac{\theta}{2})$

> Adjust code so that viewport height and width vary according to the FOV which sets a ratio

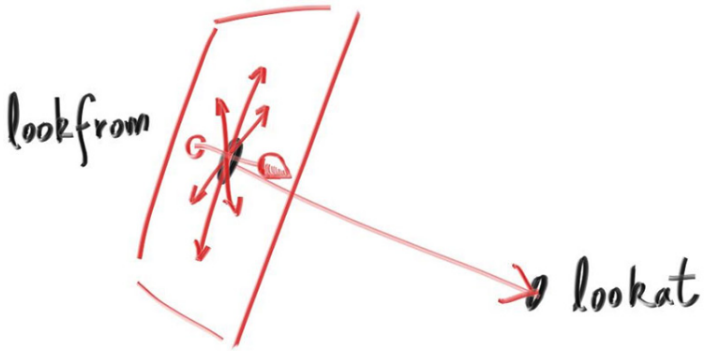
Rendering two spheres (76.8 seconds):



12.2. Positioning and Orienting the Camera

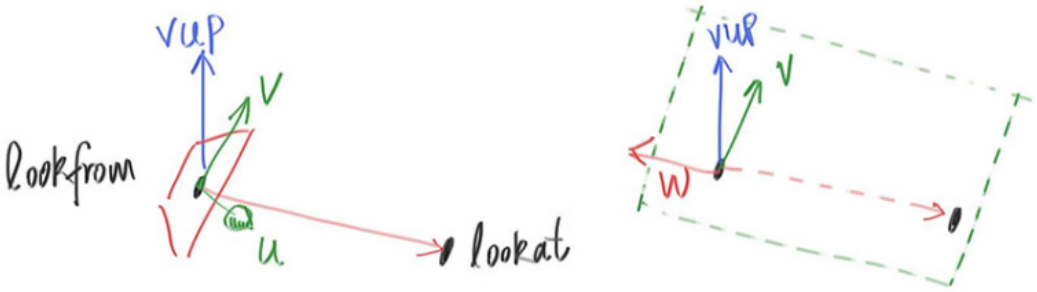
lookfrom: camera position
lookat: point we look at

Camera roll/sideways tilt: rotation around the lookat-lookfrom axis



We need to specify an "up" vector for our camera (non-parallel to view direction)

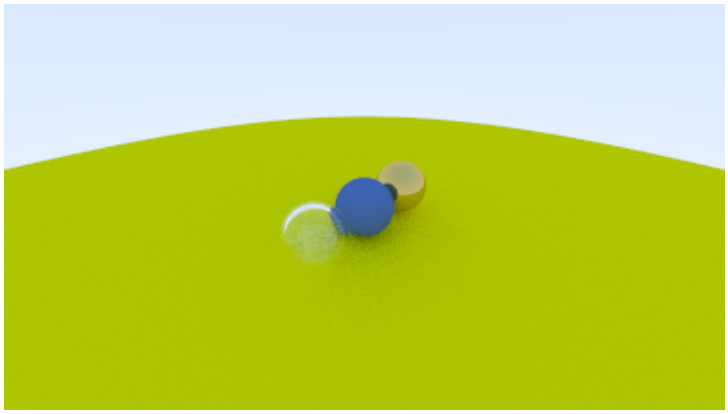
- projected onto plane orthogonal to view direction
- camera-relative up vector (`vup`)
- can use `vup` cross product and vector normalizations to have complete orthonormal basis (u, v, w) to describe camera orientation
  - $\mathbf{u}$ will point to camera right
  - $\mathbf{v}$ is unit vector pointing up
  - $\mathbf{w}$ is unit vector pointing opposite view direction (right hand coords)
  - camera center at origin



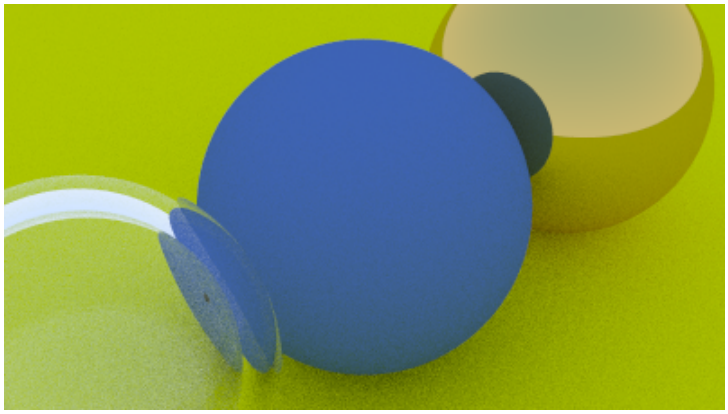
(camera faces $-\mathbf{w}$)

> modified `camera.py` according to new camera orientation and positioning system
> **DO LATER:** annotate all the new code
> updated `main.py` with old scene and new camera FOV and orientation

90 FOV Render (97.9 seconds):



20 FOV Render (219.7 seconds):



13. Defocus Blur

Also known as *depth of field* (among photographers)

Cameras in the real world need a big hole through which to gather light

- a large hole defocuses everything, so a lens is put in front of film/sensor to where there is a certain distance at which everything is in focus
- objects will become linearly blurrier from that distance

focus distance: distance between camera center and plane where everything is in perfect focus (not the same as *focal length*)

- controlled by length between lens and film/sensor

In our program, it will just so happen that our *focus distance* is the same as our *focal length*

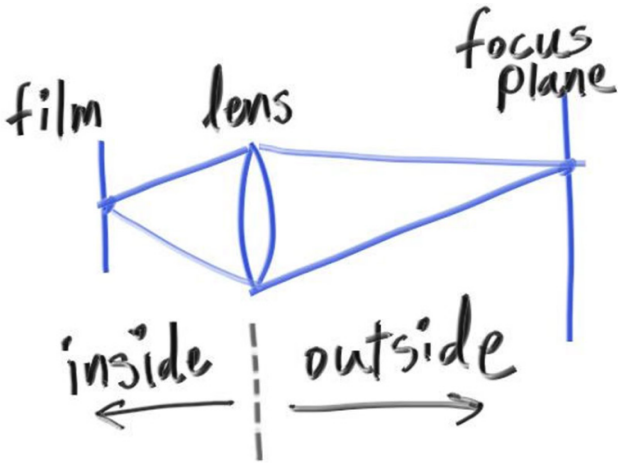
We will use an *aperture* for defocus blur, but we don't actually need one for more light

13.1. A Thin Lens Approximation

There is the option to simulate the compound lens order of a real camera:

1. sensor
2. lens
3. aperture
   and then determine where to send rays and flip image after computation

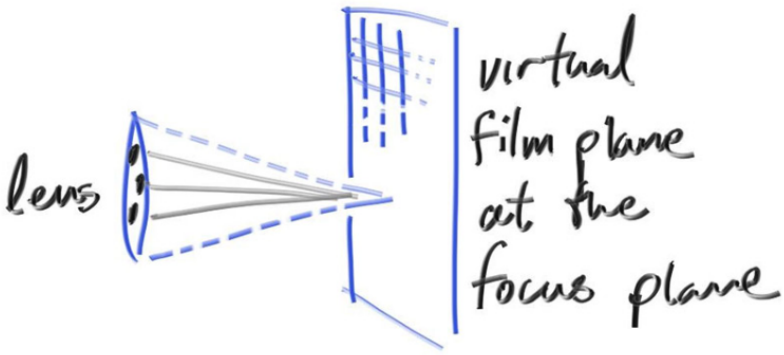
However, a thin lens approximation is usually the graphics approach



- simulating the inside of a camera is unnecessarily complex

Our procedure: start rays from infinitely thin circular "lens" and send towards pixel of interest on focus plane (`focal_length` away from the lens) in perfect focus. In practice the viewport is placed on this plane:

1. *focus plane* is orthogonal to *camera view direction*
2. *focus distance* is distance between *camera center* and *focus plane*
3. *viewport* lies on *focus plane*, centered on camera view direction vector
4. grid of pixel locations lies in side viewport (located in 3D world)
5. random image sample locations are chosen from region around current pixel location
6. camera fires rays from random points on the lens through the current image sample location



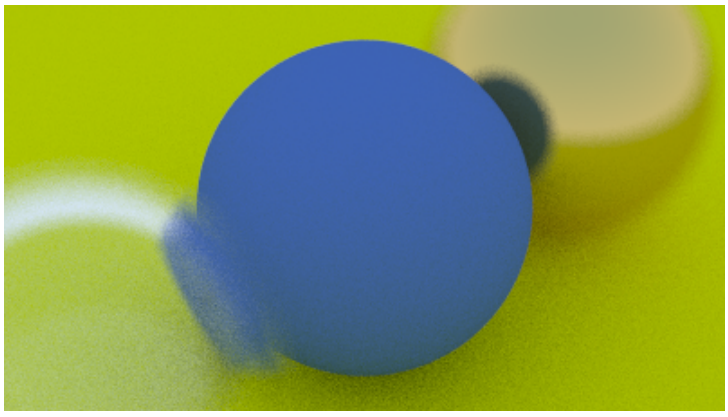
13.2. Generating Sample Rays

Before: all scene rays originate from camera center (`lookfrom`)
Now: disk centered at camera center (larger radius = greater defocus blur) (originally radius = 0)

- We will make defocus disk size a parameter of `camera` class
  - blur varies depending on projection distance: there must be a certain proportion between the defocus disk radius and the projection distance
  - we should pass the defocus disk size in the form of the angle of the cone with apex at viewport center and base (defocus disk) at camera center
  - this should give more consistent results even while varying focus distance for a shot
- will need `random_in_unit_disk()` to choose random points from defocus disk

> add `random_in_unit_disk()` to `vec3` module
> added `defocus_disk_sample()` to `camera`
> changed `focus_length` to new parameters, updated accordingly
> **DO LATER:** annotate all the new code **AND** understand everything

Render (229.5 seconds):



14. Final Render

- generate a bunch of random spheres with random material types and colors
- 3 large center spheres
- new FOV

Final Render (2867.155879020691 seconds w/ `pypy3`):