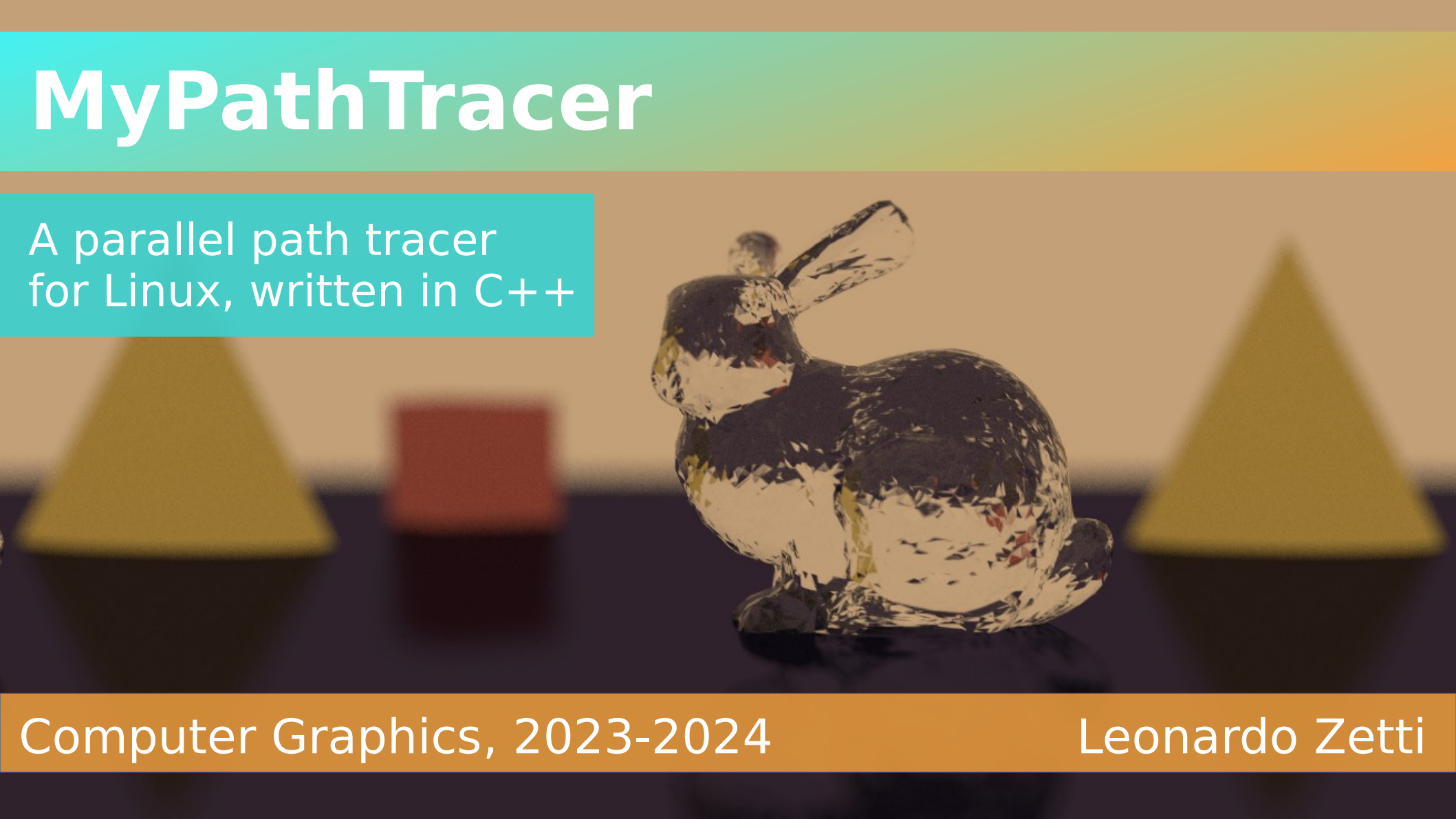
MyPathTracer

A parallel path tracer
for Linux, written in C++

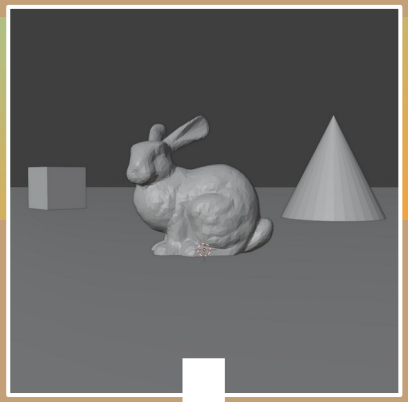


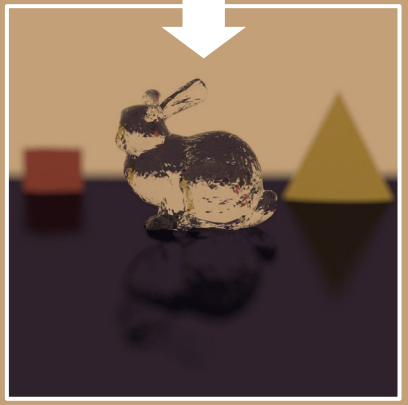
Computer Graphics, 2023-2024

Leonardo Zetti

INTRODUCTION

What's a ***Path Tracer*** ?



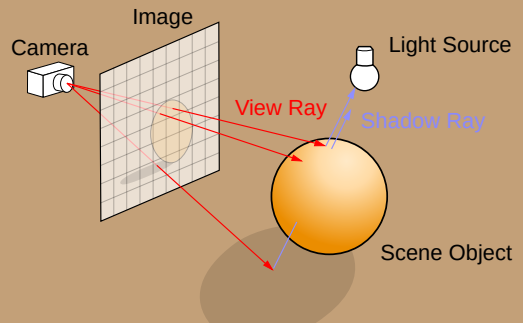


Ray Tracer **:**

- Realistic rendering of 3D models
- Simulates **interactions** between **light** and **objects** in **virtual scene**
- Realistic **reflections**, **refractions**, **shadows**, **indirect lighting**
- Find **paths** of **light** rays in scene from **light source** to eye (**camera**)
- Done in **reverse**: rays are "sent out" from the camera
- Calculates how rays travel back to light sources
- Intersections between rays and objects in scene

Path Tracer **:**

- Type of ray tracer
- Follows **rays** as they "**bounce around**" **numerous times**
- Allows complex **global illumination** effects (*direct + indirect illumination*)
- Also statistical techniques (**Monte Carlo integration**) to solve ***the rendering equation*** (for accurate light scattering)



INTRODUCTION

We present a path tracer based on the ***Ray Tracing in One Weekend*** book series (which is also where a lot of the images in the slides come from), with some minor modifications.

- For example, the ***OpenGL Mathematics*** (***GLM***) library was used for vector operations.

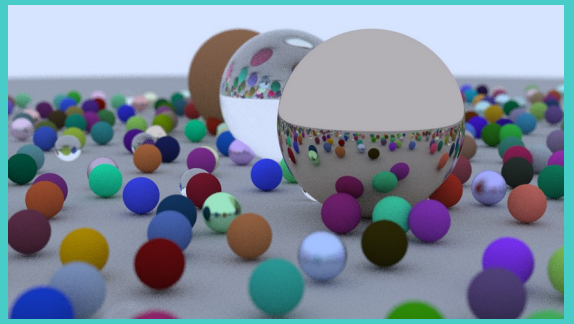


Image from *Ray Tracing in One Weekend*

The presentation will be split in 3 sections:

1) ***Ray Tracing in One Weekend*** implementation.
2) **Additional features**, including
   - parallelism,
   - ray-triangle intersection,
   - personalization of scenes through an input *.txt* file,
   - 3D Model Loading.
3) An **OpenGL** "**scene explorer**" program, which can be used to
   - explore the virtual scene,
   - position the camera,

   before rendering ray-traced images.

```
using Vec3 = glm::vec3;
using Point3 = Vec3;
using Color = Vec3;
```

The *glm* library is used for vectors

3D model loading and the OpenGL program are based on the *Learn OpenGL* tutorial series and BennyQBD's *Modern OpenGL Tutorial*.

RAY TRACING IN ONE WEEKEND

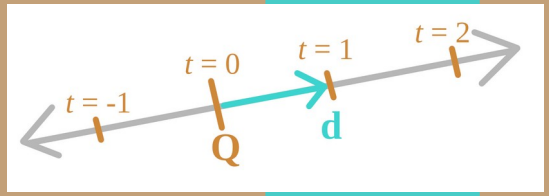
1) RAY TRACING IN ONE WEEKEND

Basics: **Rays** and the ***viewport***

Think of a ray as a function

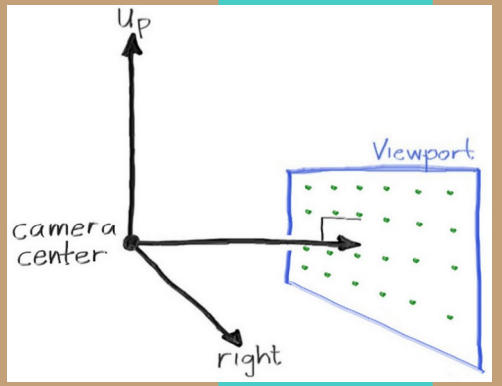
$$\mathbf{P}(t) = \mathbf{Q} + t\mathbf{d}$$

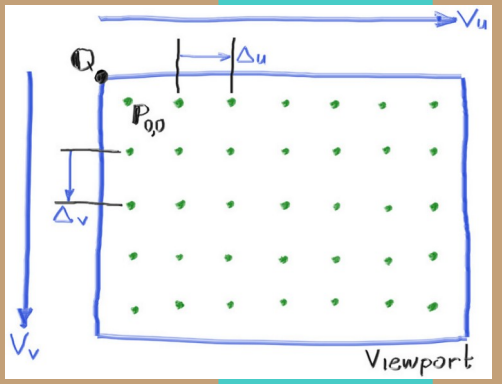
$\mathbf{P}$ → point along the ray
$\mathbf{Q}$ → ray **origin**
$\mathbf{d}$ → ray **direction**
t → how far you move along the ray
Rays originate from the **camera center** (***look-from position***) and pass through the ***viewport***.



Viewport **:**

- **Virtual rectangle** in 3D world with grid of image **pixel locations**
- In front of camera (perpendicular to/centered on ***look-at*** direction)
- At user-specified distance (***focal length***)
- Left-to-right direction $\mathbf{Vu}$
- Up-down direction $\mathbf{Vv}$



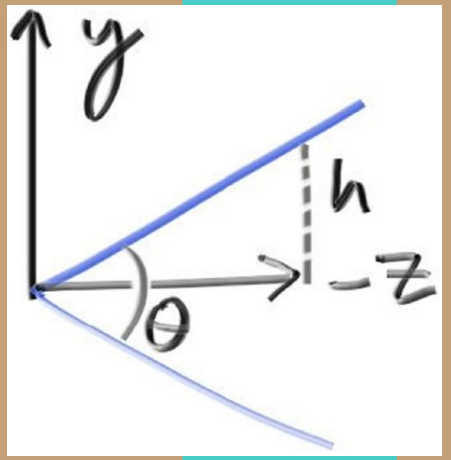


To create an image, "**scan**" each of the **pixels** in the viewport:

- We scan row-by-row, top-to-bottom, and left-to-right across each row
- Send rays through pixel centers
- Determine which objects it intersects
- Give pixel the color of closest intersected object

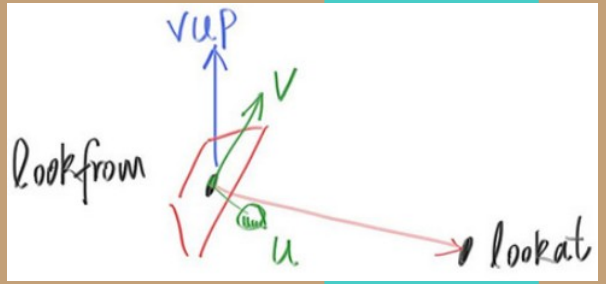
1) RAY TRACING IN ONE WEEKEND

The viewport (like the output image) has an **aspect ratio** and a **number of pixels** defined by the user, but we still need to give it a *size* in our 3D, virtual space. We do this by making the user specify a **vertical field of view** (an angle, θ) for the camera.



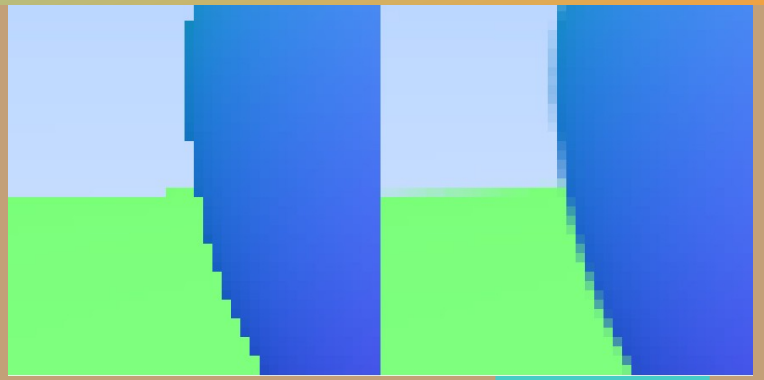
In code:

```
void Camera::initialize() {
    // Determine viewport dimensions
    float theta = glm::radians(vfov); // from degrees to radians
    float h = tan(theta/2);
    float viewportHeight = 2 * h * focusDist;
    float viewportWidth = viewportHeight * (float(imageWidth)/imageHeight);

    // Calculate the u,v,w unit basis vectors for the camera coordinate frame
    Vec3 w = glm::normalize(lookFrom - lookAt);
    Vec3 u = glm::normalize(glm::cross(up, w));
    Vec3 v = glm::cross(w, u);

    // Calculate the vectors across the horizontal and down the vertical viewport edges
    Vec3 viewportU = viewportWidth * u;
    Vec3 viewportV = viewportHeight * -v;

    // Calculate the horizontal and vertical delta vectors from pixel to pixel
    pixelDeltaU = viewportU / float(imageWidth);
    pixelDeltaV = viewportV / float(imageHeight);

    // Calculate the location of the upper left pixel
    Point3 viewportUpperLeft = cameraCenter - (focusDist * w) - viewportU/2.0f - viewportV/2.0f;
    pixel00 = viewportUpperLeft + 0.5f * (pixelDeltaU + pixelDeltaV);
```



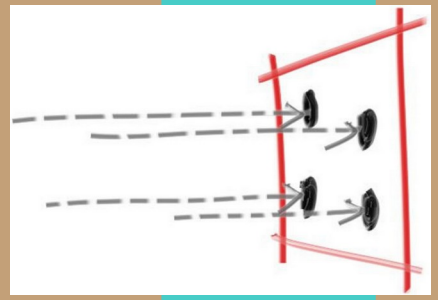
1) RAY TRACING IN ONE WEEKEND

Getting better images through ***Antialiasing***



Sending a single ray through each pixel center (*point sampling*) creates “stair steps” on object edges (and, in general, ***noise***).
Solution: taking **multiple samples** for each pixel.

We sample the square region centered at the pixel that extends halfway to each of the four neighboring pixels.
getRay constructs a ray originating from the camera and directed at a **randomly sampled point around the pixel** location (i,j)



```
Ray Camera::getRay(int i, int j) const{
    Vec3 offset = sampleUnitSquare();
    Point3 pixelSample = pixel00 + ((float(i)+offset.x)*pixelDeltaU) + ((float(j)+offset.y)*pixelDeltaV);

    Point3 rayOrigin = (defocusAngle <= 0) ? cameraCenter : defocusDiskSample();
    Vec3 rayDirection = pixelSample - rayOrigin;

    return Ray(rayOrigin, rayDirection);
}
```

(more on this later)

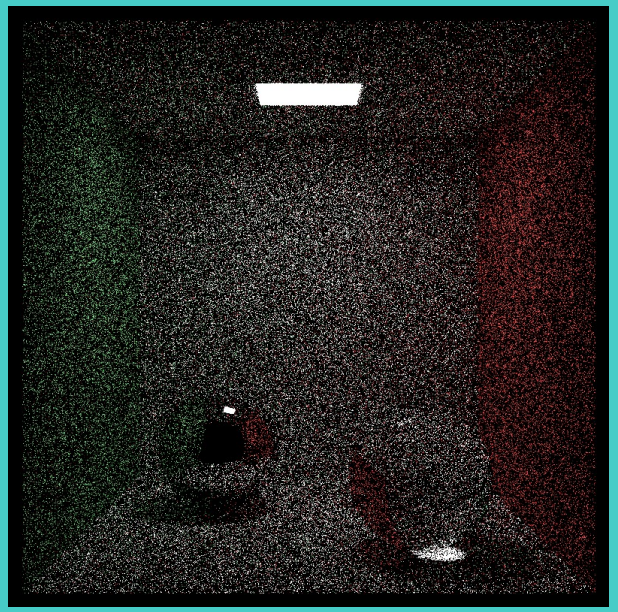
```
Vec3 Camera::sampleUnitSquare() const {
    return Vec3(randomFloat() - 0.5, randomFloat() - 0.5, 0);
}
```

1) RAY TRACING IN ONE WEEKEND

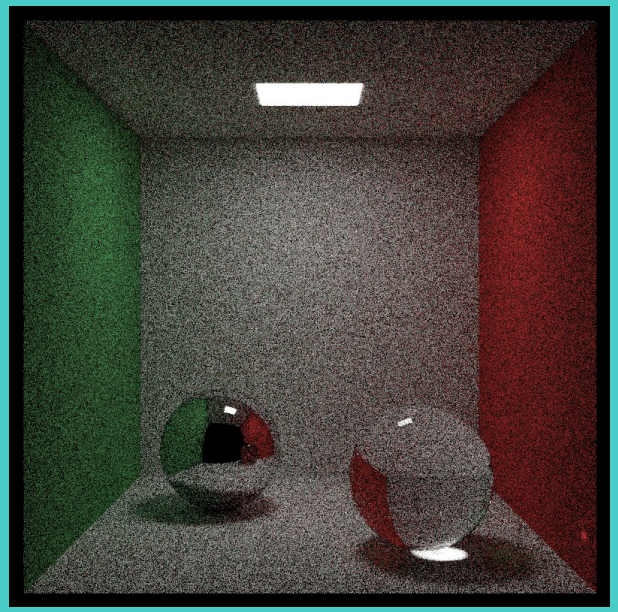
Let’s do a ***Cornell Box*** test for various values of per-pixel samples.

(Tested on *AMD Ryzen 7 5700U*)

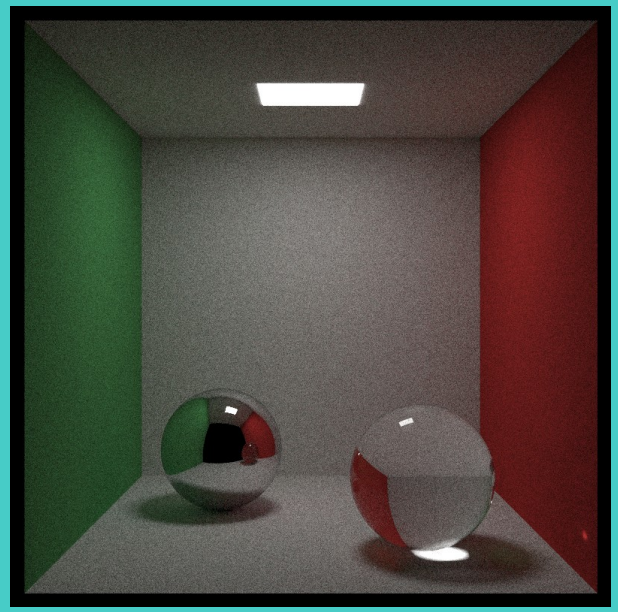
800x800 pixels, max depth of ray bounces 7 (16 threads, 160 “sub-images”)



10 samples-per-pixel
0h 0m 42s

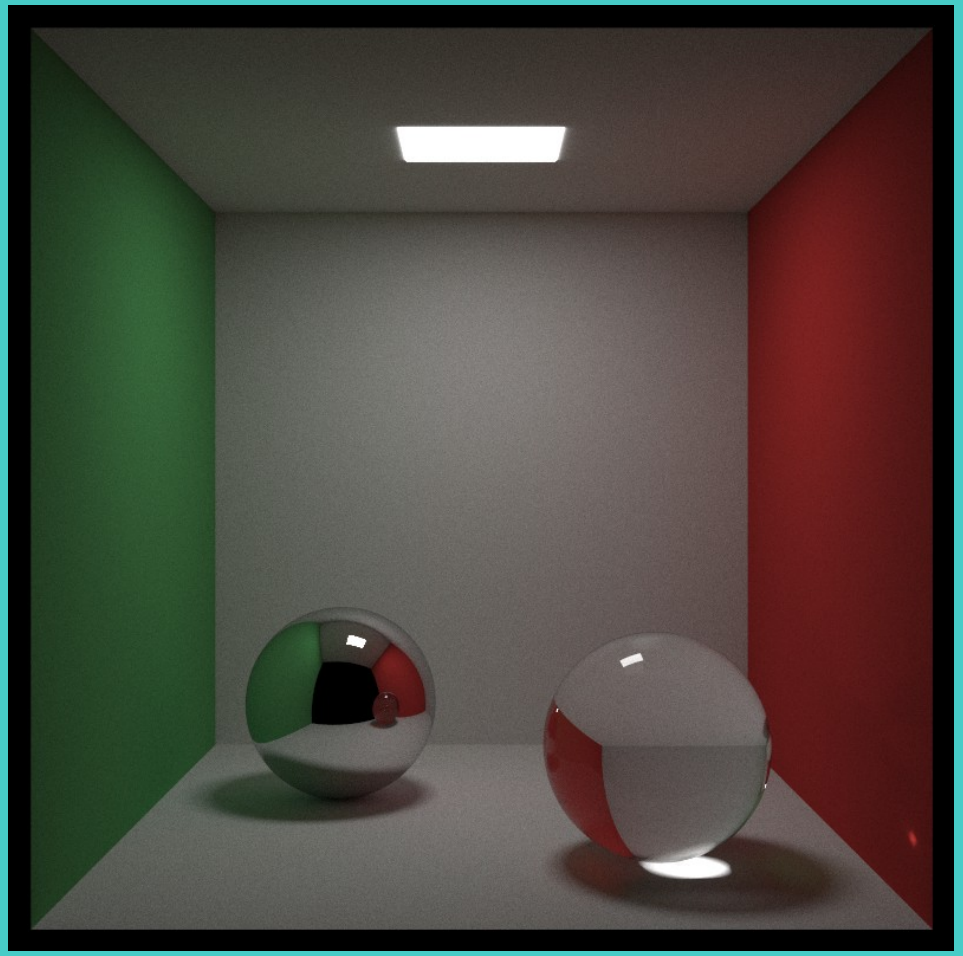


100 samples-per-pixel
0h 7m 3s



1000 samples-per-pixel
1h 11m 38s

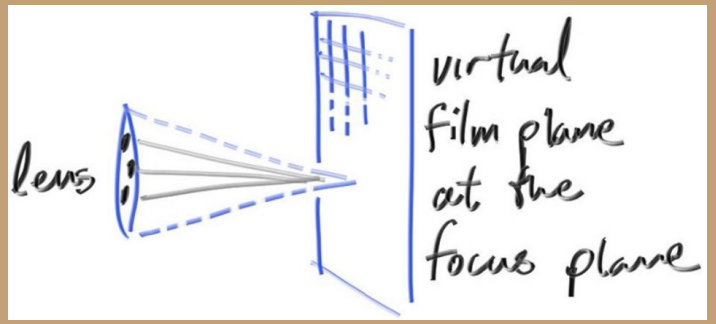
1) RAY TRACING IN ONE WEEKEND



10000 samples-per-pixel
10h 27m 30s

1) RAY TRACING IN ONE WEEKEND

Simulating a real camera with ***Defocus Blur***

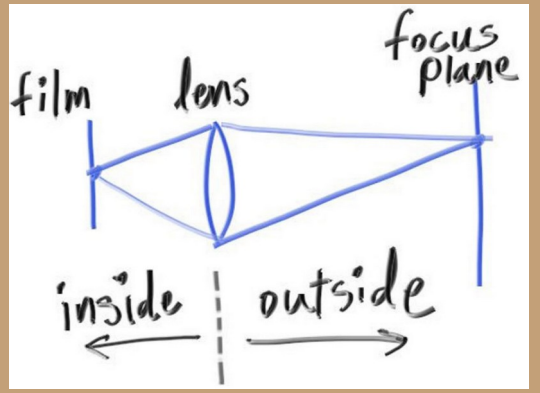


Let's explain these two lines of code now:

```
Point3 rayOrigin = (defocusAngle <= 0) ? cameraCenter : defocusDiskSample();
Vec3 rayDirection = pixelSample - rayOrigin;
```

A real camera has:

- A finite aperture through which light passes
- A lens in front of the aperture which gathers ("organizes") light. We wont consider the lens' thickness (***thin lens model***)



Thanks to the lens, there will be a certain distance at which everything is in focus: the ***focus distance***.
We simplify by making the ***focus distance*** the same as the ***focal length*** (distance between camera center and viewport).

We make this defocus effect visible by having the camera fire the **ray from** a **random point on** the **lens** through the current image sample location.

```
Point3 Camera::defocusDiskSample() const {
    Vec3 v = randomInUnitDisk();
    return cameraCenter + (v.x * defocusDiskU) + (v.y * defocusDiskV);
}
```

(more on these in next slide)

1) RAY TRACING IN ONE WEEKEND

Where we've added

```
// Calculate the camera defocus disk radius
float defocusRadius = focusDist * tan(glm::radians(defocusAngle/2));
defocusDiskU = u * defocusRadius;
defocusDiskV = v * defocusRadius;
```

in `void Camera::initialize()`

Note: Complex objects in our scene are broken down into simpler geometrical shapes. For now, we only use **spheres**, next, we'll have **triangles** as well.

Basics: ray-sphere intersection

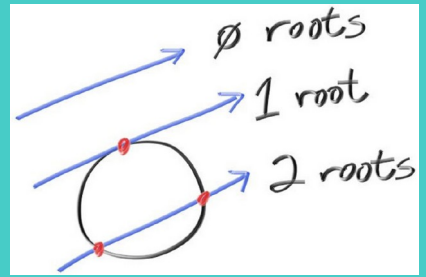
Equation of sphere:

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

$\mathbf{P}$ → point on sphere
$\mathbf{C}$ → sphere center
r → sphere radius

Ray-sphere intersection:

$$\mathbf{P}(t) = \mathbf{Q} + t\mathbf{d}$$
$$(\mathbf{C} - \mathbf{P}(t)) \cdot (\mathbf{C} - \mathbf{P}(t)) = r^2$$
$$(\mathbf{C} - (\mathbf{Q} + t\mathbf{d})) \cdot (\mathbf{C} - (\mathbf{Q} + t\mathbf{d})) = r^2$$
$$(-t\mathbf{d} + (\mathbf{C} - \mathbf{Q})) \cdot (-t\mathbf{d} + (\mathbf{C} - \mathbf{Q})) = r^2$$
$$t^2\mathbf{d} \cdot \mathbf{d} - 2t\mathbf{d} \cdot (\mathbf{C} - \mathbf{Q}) + (\mathbf{C} - \mathbf{Q}) \cdot (\mathbf{C} - \mathbf{Q}) - r^2 = 0$$



We get a quadratic equation for t that we can easily solve

1) RAY TRACING IN ONE WEEKEND

Our scene: A list of Hittable Objects

We'll use an **abstract class** for anything a ray might hit, and call it **Hittable**.

```
class Hittable {
    public:
        virtual ~Hittable() = default;
        virtual bool hit(const Ray& r, Interval rayT, HitRecord& rec) 
        virtual Aabb boundingBox() const = 0;
};
```

(more on this later)

- The HitRecord structure is used by the hit function to memorize information about a potential hit
- rayT is the interval (t_{min}, t_{max}) such that the hit only "counts" if $t \in rayT$
- At the start, we set rayT = (0.001, infinity)

Here's the information we need about a hit:

```
struct HitRecord {
    Point3 p;
    Vec3 normal;
    std::shared_ptr<Material> material;
    float t;
    // u,v texture coordinates
    float u;
    float v;
    bool frontFace;
```

We'll group our objects together in a list, that is, a **HittableList** object (which is also a Hittable):

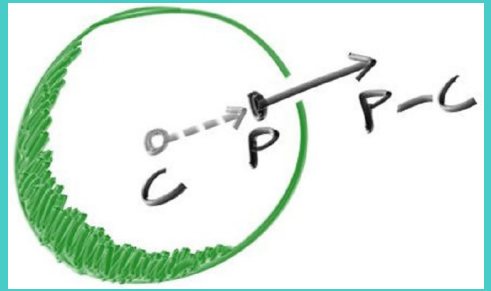
```
class HittableList : public Hittable {
    public:
        std::vector<std::shared_ptr<Hittable>> objects;
```

So, we "send" a ray into the scene. Then what?
The idea, for now, is to call each object's hit function on the interval $rayT = (0.001, t_{closest})$, where $t_{closest}$ is the distance of the closest hit point, updated after every hit.

1) RAY TRACING IN ONE WEEKEND

Applying our strategy, we can write the hit function in HittableList as:

```
bool hit(const Ray& r, Interval rayT, HitRecord& rec) const override {
    HitRecord tempRec;
    bool hitAnything = false;
    float closestSoFar = rayT.max;
    for (const auto& object : objects) {
        if (object->hit(r, Interval(rayT.min, closestSoFar), tempRec)) {
            hitAnything = true;
            closestSoFar = tempRec.t;
            rec = tempRec;
        }
    }
    return hitAnything;
}
```



Sphere surface-normal geometry

$$a = \mathbf{d} \cdot \mathbf{d}$$

$$h = \frac{b}{-2} = \mathbf{d} \cdot (\mathbf{C} - \mathbf{Q})$$

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

$$\frac{h \pm \sqrt{h^2 - ac}}{a}$$

And here's our Sphere class's hit function:

```
bool Sphere::hit(const Ray& r, Interval rayT, HitRecord& rec)
    Vec3 oc = center - r.origin();
    Vec3 d = r.direction();
    float a = glm::dot(d, d);
    float h = glm::dot(d, oc);
    float c = dot(oc, oc) - radius * radius;
    float discriminant = h * h - a * c;

    if (discriminant < 0) return false;

    // Find the nearest root that lies in the acceptable rang
    float sqrtd = sqrt(discriminant);
    float root = (h - sqrtd) / a;
    if (!rayT.surrounds(root)) {
        root = (h + sqrtd) / a;
        if (!rayT.surrounds(root)) {
            return false;
        }
    }
    rec.t = root;
    rec.p = r.at(rec.t);
    Vec3 outwardNormal = (rec.p - center) / radius;
    rec.setFaceNormal(r, outwardNormal);
    getSphereUV(outwardNormal, rec.u, rec.v);
    rec.material = mat;
    return true;
}
```

(sphere texture coordinates)

1) RAY TRACING IN ONE WEEKEND

Finally: Materials!

We create the abstract class Material:

```
class Material {
    public:
    virtual bool scatter(const Ray& in, const HitRecord& rec,
                         Color& attenuation, Ray& scattered)
                         const { return false; }
    // Emitted light (no light by default)
    virtual Color emitted(float u, float v, const Point3& p) const {
        return Color(0.0f,0.0f,0.0f);
    }
    virtual ~Material() = default;
};
```

scatter()

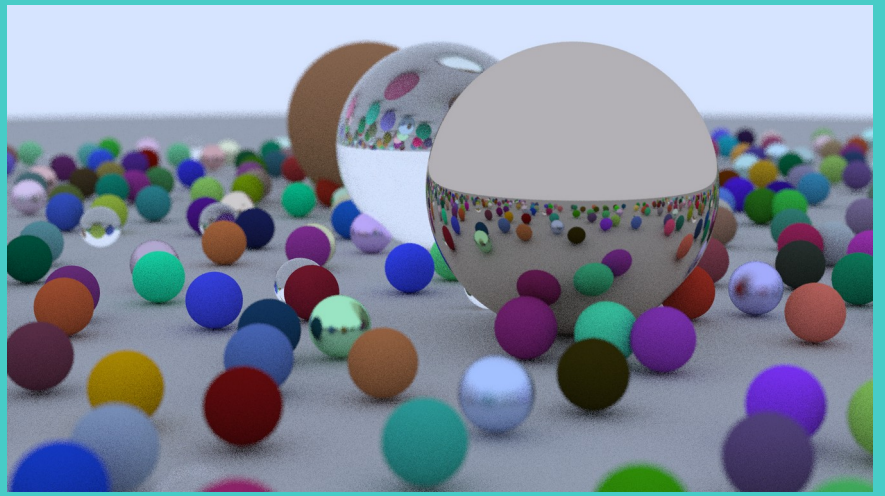
- Produce a scattered ray (or say the incident ray was absorbed)
- If scattered, say how much the ray should be attenuated

emitted()

- returns emitted light when present

Each sphere is assigned a material through its constructor method:

```
class Sphere : public Hittable {
    public:
        Sphere(const Point3& center,
               float radius,
               std::shared_ptr<Material> mat);
```



Lambertian, ***Dielectric*** and ***Metal*** materials

1) RAY TRACING IN ONE WEEKEND

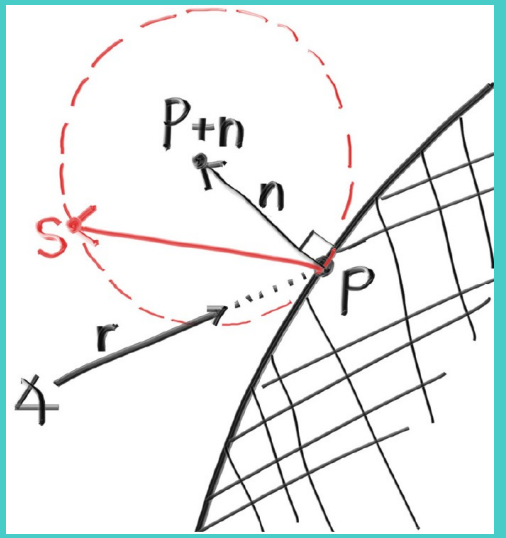
Lambertian materials

Diffuse (***matte***) surfaces are represented through the ***Lambertian distribution*** **:**

- ϕ angle between reflected ray and surface normal
- Rays are scattered in a manner that is proportional to $\cos(\phi)$.
  Meaning that:
- A reflected ray is most likely to scatter in a direction near the surface normal.

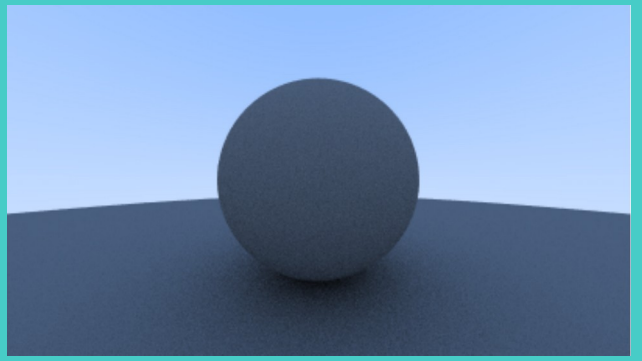
We can create this distribution by adding a random unit vector to the normal vector.

- Pick a random point $\mathbf{S}$ on unit sphere
- send a ray from the hit point $\mathbf{P}$ to $\mathbf{S}$ ($(\mathbf{S}-\mathbf{P})$ vector)




```
bool Lambertian::scatter(const Ray& in, const HitRecord& rec, Color& attenuation,
                         Ray& scattered) const {
    Vec3 scatterDirection = rec.normal + randomUnitVector();

    // Catch degenerate scatter direction (all vector components near zero)
    if (nearZero(scatterDirection)) scatterDirection = rec.normal;

    scattered = Ray(rec.p, scatterDirection);
    attenuation = tex->value(rec.u, rec.v, rec.p);
    return true;
}
```

(more on this in next slide)



1) RAY TRACING IN ONE WEEKEND

Textures

Lambertian surfaces use **textures** to represent the colors of their points

```
class Lambertian : public Material {
    private:
        std::shared_ptr<Texture> tex;
```

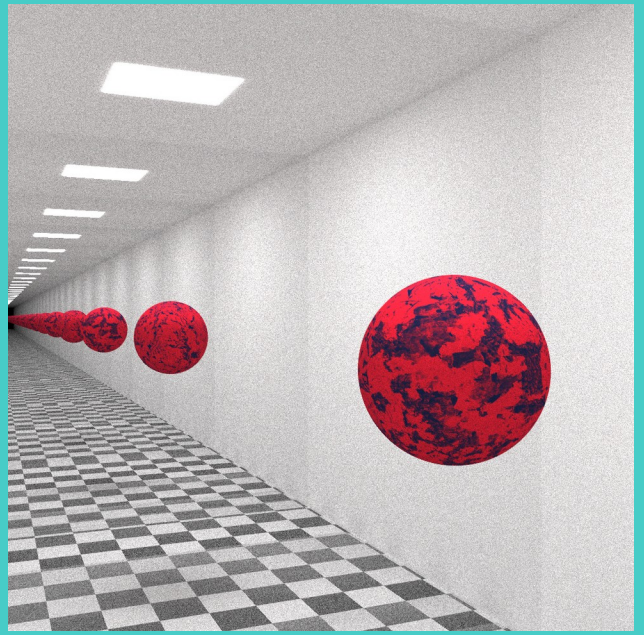
```
class Texture {
    public:
        virtual ~Texture() = default;
        virtual Color value(float u, float v, const Point3& p) 
};
```

```
class SolidColor : public Texture {
    public:
        SolidColor(const Color& albedo) : albedo(albedo) {}

        SolidColor(float r, float g, float b)
            : SolidColor(Color(r, g, b)) {}

        Color value(float u, float v, const Point3& p) const override {
            return albedo;
        }

    private:
        Color albedo;
};
```

```
class ImageTexture : public Texture {
    public:
        ImageTexture(std::shared_ptr<Image> image) : image(image) {}

        Color value(float u, float v, const Point3& p) const override {
            // If we have no texture data, then return solid cyan as a de
            if (image->height() <= 0) return Color(0, 1, 1);

            // Clamp input texture coordinates to [0,1]
            u = Interval(0, 1).clamp(u);
            v = Interval(0, 1).clamp(v);

            #ifdef FLIP_Y_AXIS_TEXTURE
                // Flip v to image coordinates
                v = 1.0 - v;
            #endif

            int i = int(u * image->width());
            int j = int(v * image->height());
            auto pixel = image->pixelData(i, j);

            auto colorScale = 1.0 / 255.0;
            return Color(colorScale * pixel[0], colorScale * pixel[1],
                         colorScale * pixel[2]);
        }

    private:
        std::shared_ptr<Image> image;
};
```

2) ADDITIONAL FEATURES

Example of texture mapped on 3D model of globe.



In this scene an external image has been loaded using texture coordinates for spheres.

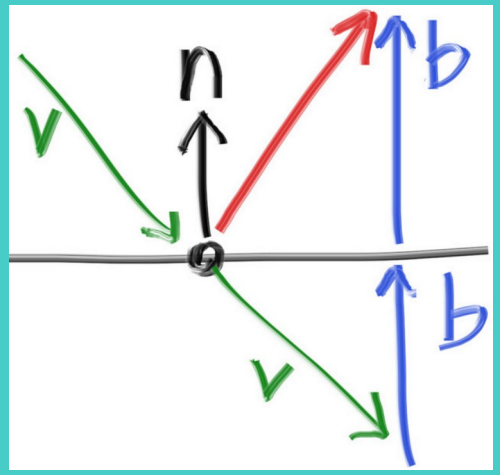
$$u = \frac{\phi}{2\pi}$$

$$v = \frac{\theta}{\pi}$$

1) RAY TRACING IN ONE WEEKEND

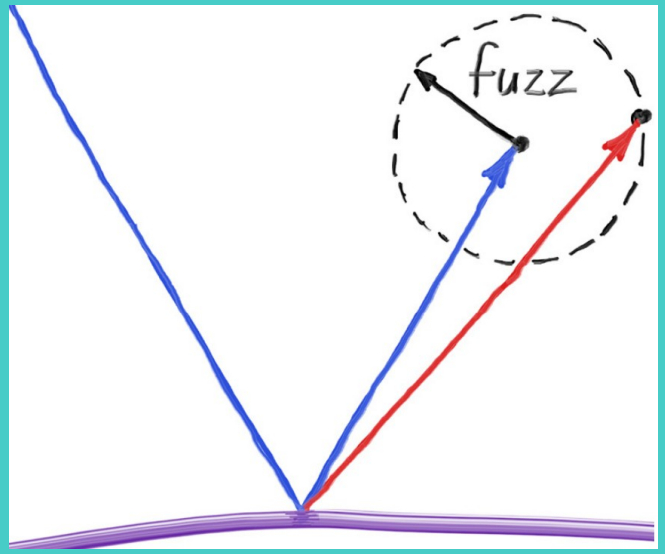
Metal materials

Reflection of ray from a polished metal (**mirror**):



The reflected ray direction is $\mathbf{v} + 2\mathbf{b}$.
This is calculated by the glm::reflect() function

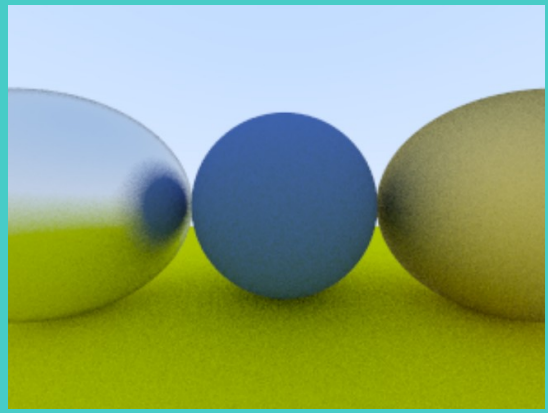
We also want ***fuzzy reflections*** **:** randomize the reflected direction by using a **random point** from a **small sphere** and choosing a **new endpoint** for the **ray**.



The bigger the fuzz sphere, the fuzzier the reflections will be.

```
class Metal : public Material {
    private:
        Color albedo;
        float fuzz;
```

fuzz is the radius of the sphere



1) RAY TRACING IN ONE WEEKEND

Implementing what we just said in code:

```
bool Metal::scatter(const Ray& in, const HitRecord& rec, Color& attenuation,
                    Ray& scattered) const {
    Vec3 reflected = glm::reflect(in.direction(), rec.normal);
    reflected = glm::normalize(reflected) + (fuzz * randomUnitVector());
    scattered = Ray(rec.p, reflected);
    attenuation = albedo;
    return (glm::dot(scattered.direction(), rec.normal) > 0);
}
```

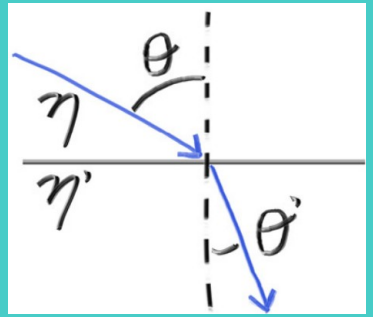
Dielectric materials

Water, **glass**, and **diamond** are examples of dielectrics.

When a light ray hits them, it splits into a **reflected ray** and a ***refracted*** (*transmitted*) ***ray*.**
We'll simply **choose randomly** between **reflection** and **refraction**.

A refracted ray bends as it transitions from a material's surroundings into the material itself.
How much it bends is determined by the material's **refractive index**

Refraction is described by ***Snell's law***



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

1) RAY TRACING IN ONE WEEKEND

$$\eta_1 \sin\theta_i = \eta_2 \sin\theta_t$$

$$\sin\theta_t = \frac{\eta_1}{\eta_2}\sin\theta_i$$

Split the transmitted ray in tangent and normal components:

$$\mathbf{t} = \mathbf{t}_{\parallel} + \mathbf{t}_{\perp}$$

We know that:

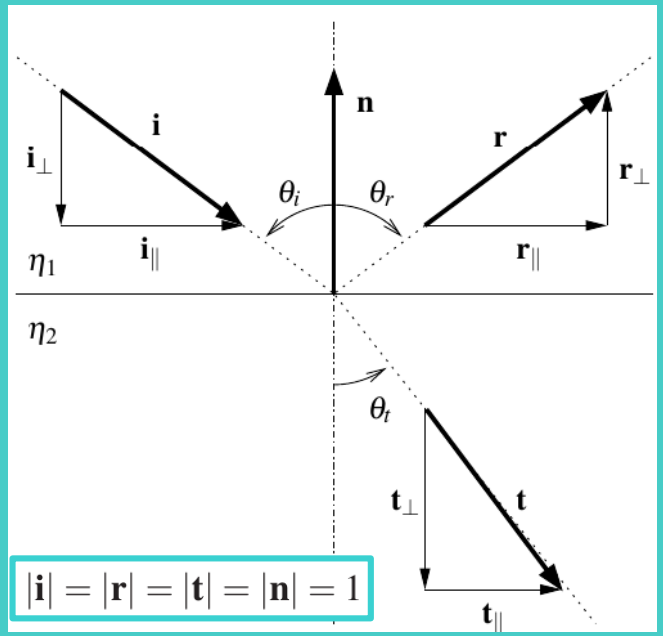
$$\sin\theta_v = \frac{|\mathbf{v}_{\parallel}|}{|\mathbf{v}|} = |\mathbf{v}_{\parallel}|$$

$$|\mathbf{t}_{\parallel}| = \frac{\eta_1}{\eta_2}|\mathbf{i}_{\parallel}|$$

So:

$$\mathbf{t}_{\parallel} = \frac{\eta_1}{\eta_2}\mathbf{i}_{\parallel} = \frac{\eta_1}{\eta_2}[\mathbf{i} + \cos\theta_i \mathbf{n}]$$

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



We give Dielectric materials a refractionIndex value:

```
class Dielectric : public Material {
    private:
        float refractionIndex;
```

refractionIndex can be interpreted as refractive index of object divided by refractive index of enclosing medium

```
bool Dielectric::scatter(const Ray& in, const HitRecord& rec, Color& attenuation,
                         Ray& scattered) const {
    attenuation = Color(1.0f, 1.0f, 1.0f);
    float eta = rec.frontFace ? (1.0 / refractionIndex) : refractionIndex;

    Vec3 unitDirection = glm::normalize(in.direction());
    float cosTheta = fmin(glm::dot(-unitDirection, rec.normal), 1.0);
    float sinTheta = sqrt(1.0 - cosTheta * cosTheta);
```

continues in next slide...

1) RAY TRACING IN ONE WEEKEND

Here are the remaining lines in Dielectric::scatter()

```
bool cannotRefract = eta * sinTheta > 1.0;
Vec3 direction;

if (cannotRefract || reflectance(cosTheta, eta) > randomFloat()) {
    direction = glm::reflect(unitDirection, rec.normal);
} else {
    direction = glm::refract(unitDirection, rec.normal, eta);
}

scattered = Ray(rec.p, direction);
return true;
}
```

The glm::refract() function calculates

$$\mathbf{t} = \frac{\eta_1}{\eta_2}\mathbf{i} + \left(\frac{\eta_1}{\eta_2}\cos\theta_i - \sqrt{1 - |\mathbf{t}_{\parallel}|^2}\right)\mathbf{n}$$

But what about the other lines of code?

1 Sometimes no solution is possible using Snell's law:

- The ray enters a medium of lower refraction index
- The ray has a sufficiently large angle of incidence (larger than ***critical angle***)

In this case, the equation

$$\sin\theta_t = \frac{\eta_1}{\eta_2}\sin\theta_i$$

is broken, and the ray must **reflect**.

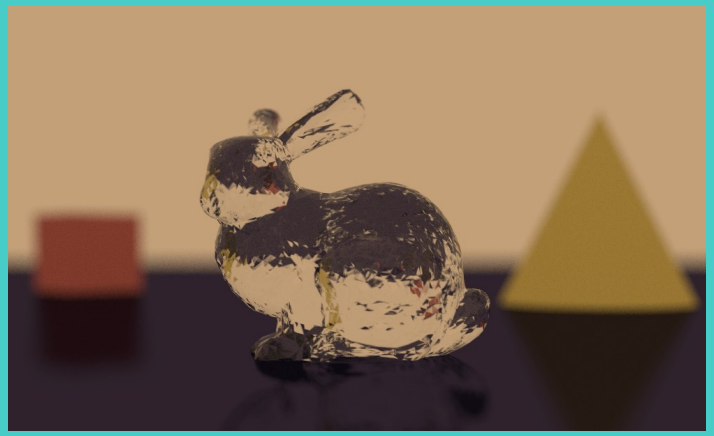
2 Real glass has **reflectivity** that **varies with angle** (***Fresnel equations***). Usually a **polynomial approximation** by **Christophe Schlick** is used for the **reflection coefficient** :

$$R(\theta) = R_0 + (1 - R_0)(1 - \cos\theta)^5$$

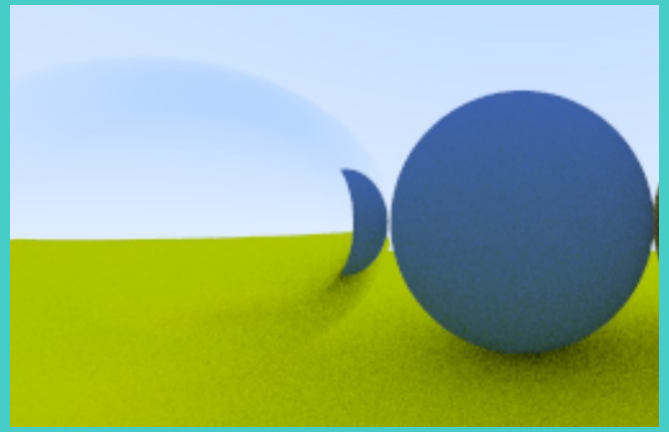
$$R_0 = \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2$$

```
float Dielectric::reflectance(float cosine, float refractionIndex) {
    // Use Schlick's approximation for reflectance
    float r0 = (1.0 - refractionIndex) / (1.0 + refractionIndex);
    r0 = r0 * r0;
    return r0 + (1 - r0) * pow((1 - cosine), 5);
}
```

1) RAY TRACING IN ONE WEEKEND



The bunny here is made out of **glass** (index of refraction 1.5). (Also, notice the ***defocus blur*** effect.)

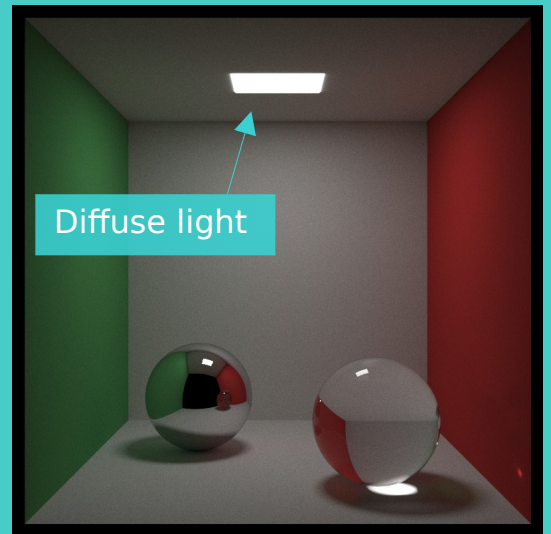


Air bubble (IOR 1.0) in water (IOR 1.33). Sometimes refracts, sometimes reflects.

Diffuse Lights

Lights are very simple to add:

```
class DiffuseLight : public Material {
    public:
        DiffuseLight(const Color& emit) : emit(emit) {}

        Color emitted(float u, float v, const Point3& p) const override {
            return emit;
        }
    private:
        Color emit;
};
```

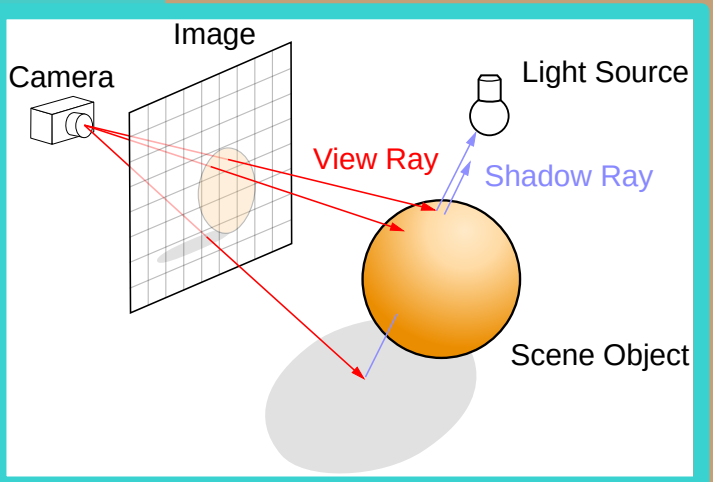


1) RAY TRACING IN ONE WEEKEND

Let’s put everything together by writing the rendering loop in Camera:

```
for (int j = firstRow; j <= lastRow; j++) {
   for (int i = 0; i < imageWidth; i++) {
        Color pixelColor(0.0f,0.0f,0.0f);
        for (int sample = 0; sample < samplesPerPixel; sample++) {
            Ray r = getRay(i, j);
            pixelColor += rayColor(r, maxDepth, world);
        }
        writeColor(outFile, pixelColor * (1.0f/samplesPerPixel));
    }
}
```

```
Color Camera::rayColor(const Ray& r, int depth, const Hittable& world) const {
    if (depth <= 0){ // ray bounce limit exceeded
        return Color(0.0f, 0.0f, 0.0f);
    }
    HitRecord rec;
    // If the ray hits nothing, return the background color
    if (!world.hit(r, Interval(0.001, infinity), rec)){
        return background;
    }
    Ray scattered; Color attenuation;
    Color colorFromEmission = rec.material->emitted(rec.u, rec.v, rec.p);
    if (!rec.material->scatter(r, rec, attenuation, scattered)){
        return colorFromEmission;
    }
    Color colorFromScatter = attenuation * rayColor(scattered, depth-1, world);
    return colorFromEmission + colorFromScatter;
}
```



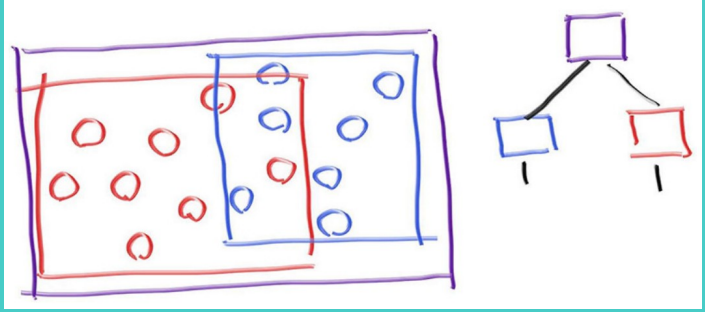
1) RAY TRACING IN ONE WEEKEND

Faster intersections: ***Bounding Volume Hierarchies***

Ray-object intersection is the main **time-bottleneck** in a **ray tracer**.
The run time is linear with the number of objects.
We ought to be able to make it a **logarithmic search**

***Bounding Volume Hierarchies* :**

- For a set of primitives, find a volume that **fully encloses** (bounds) all the **objects**.
- Use these bounding volumes to group the **objects** in the scene into **subgroups**.
- An object will be in just one bounding volume, though bounding volumes can overlap.



```
if (hits purple)
    hit0 = hits blue enclosed objects
    hit1 = hits red enclosed objects
    if (hit0 or hit1)
        return true and info of closer hit
return false
```

We need:

- A way to make good divisions
- A way to intersect a ray with a bounding volume

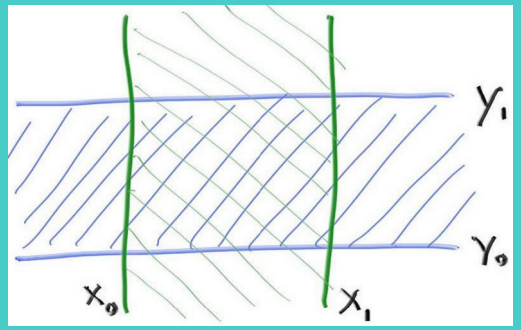
We'll use ***Axis-Aligned Bounding Boxes***, or **AABB**s.

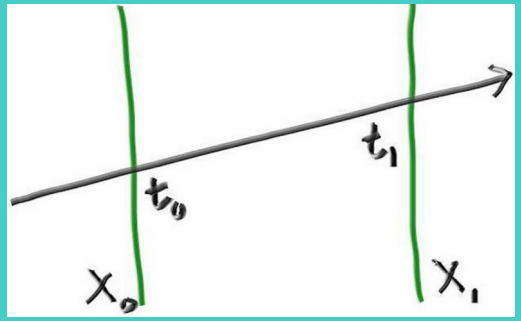
1) RAY TRACING IN ONE WEEKEND

```
class Aabb {
    public:
        Interval x, y, z;
```

For the intersections, we'll use the ***slab method*** **:**
An n-dimensional AABB is just the intersection of n axis-aligned intervals ("slabs").

```
Aabb::Aabb(const Point3& a, const Point3& b) {
    x = (a.x <= b.x) ? Interval(a.x, b.x) : Interval(b.x, a.x);
    y = (a.y <= b.y) ? Interval(a.y, b.y) : Interval(b.y, a.y);
    z = (a.z <= b.z) ? Interval(a.z, b.z) : Interval(b.z, a.z);
    padToMinimums();
}
```





$$\mathbf{P}(t) = \mathbf{Q} + t\mathbf{d}$$

$$x(t) = Q_x + td_x$$

Intersect at $x=x_0$:

$$x_0 = Q_x + t_0 d_x$$

$$t_0 = \frac{x_0 - Q_x}{d_x}$$

Similarly for x_1:

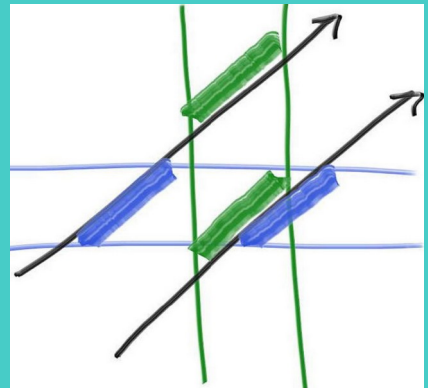
$$t_1 = \frac{x_1 - Q_x}{d_x}$$

To avoid reversed intervals:

$$t_{x0} = \min(\frac{x_0 - Q_x}{d_x}, \frac{x_1 - Q_x}{d_x})$$

$$t_{x1} = \max(\frac{x_0 - Q_x}{d_x}, \frac{x_1 - Q_x}{d_x})$$

Key observation:
If a ray intersects the box bounded by all pairs of planes, then all t-intervals will overlap.



1) RAY TRACING IN ONE WEEKEND

Using our observation:

```
bool Aabb::hit(const Ray& r, Interval rayT) const {
    const Point3& rayOrig = r.origin();
    const Vec3& rayDir = r.direction();

    for (int axis = 0; axis < 3; axis++) {
       const Interval& ax = axisInterval(axis);
        float dinv = 1.0f / rayDir[axis];
        float t0 = (ax.min - rayOrig[axis]) * dinv;
        float t1 = (ax.max - rayOrig[axis]) * dinv;

        if (t0 < t1) {
            if (t0 > rayT.min) rayT.min = t0;
            if (t1 < rayT.max) rayT.max = t1;
        } else {
            if (t1 > rayT.min) rayT.min = t1;
            if (t0 < rayT.max) rayT.max = t0;
        }
        if (rayT.max <= rayT.min) return false;
    }
    return true;
}
```

Each Hittable has a bounding box:

```
class Hittable {
    public:
        virtual ~Hittable() = default;
        virtual bool hit(const Ray& r, Interval r
        virtual Aabb boundingBox() const = 0;
};
```

In the case of spheres:

```
Sphere::Sphere(const Point3& center, float radiu
            : center(center), radius(std::fmax(0
    // Initialize bounding box
    Vec3 rvec = Vec3(radius, radius, radius);
    bbox = Aabb(center - rvec, center + rvec);
}
```

```
Aabb boundingBox() const override {return bbox;}
```

1) RAY TRACING IN ONE WEEKEND

Another constructor for AABBs:

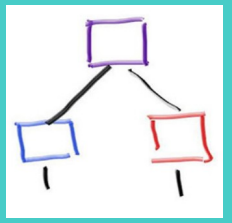
```
Aabb::Aabb(const Aabb& box0, const Aabb& box1) {
    x = Interval(box0.x, box1.x);
    y = Interval(box0.y, box1.y);
    z = Interval(box0.z, box1.z);
}
```

```
// Create the interval tightly enclosing the two input intervals
Interval(const Interval& a, const Interval& b) {
    min = a.min <= b.min ? a.min : b.min;
    max = a.max >= b.max ? a.max : b.max;
}
```

This is used, for example, in a HittableList to create the AABB that encloses all its objects.

Now we create a new class for nodes in the **hierarchy** of bounding volumes (also a Hittable):

```
class BvhNode : public Hittable {
  private:
    std::shared_ptr<Hittable> left;
    std::shared_ptr<Hittable> right;
    Aabb bbox;
```



We want the BvhNode constructor method to create a **BVH** by **splitting** a **list** of **Hittables**. We'll:

1. randomly choose an axis
2. sort the primitives (using std::sort)
3. put half in each subtree

This is what we'll use to sort primitives w.r.t. a given axis:

```
static bool boxCompare(const std::shared_ptr<Hittable> a,
                       const std::shared_ptr<Hittable> b, int axisIndex) {
    auto aAxisInterval = a->boundingBox().axisInterval(axisIndex);
    auto bAxisInterval = b->boundingBox().axisInterval(axisIndex);
    return aAxisInterval.min < bAxisInterval.min;
}
```

For example:

```
static bool boxXCompare (const s
    return boxCompare(a, b, 0);
}
```

1) RAY TRACING IN ONE WEEKEND

The BvhNode constructor:

```
BvhNode(std::vector<std::shared_ptr<Hittable>>& objects, size_t start, size_t end) {
    // Build the bounding box of the span of source objects
    bbox = Aabb::empty;
    for (size_t objectIndex=start; objectIndex < end; objectIndex++) {
        bbox = Aabb(bbox, objects[objectIndex]->boundingBox());
    }
    int axis = bbox.longestAxis();

    auto comparator = (axis == 0) ? boxXCompare
                    : (axis == 1) ? boxYCompare
                                  : boxZCompare;

    size_t objectSpan = end - start;
    if (objectSpan == 1) {
        left = right = objects[start];
    } else if (objectSpan == 2) {
        left = objects[start];
        right = objects[start+1];
    } else {
        std::sort(std::begin(objects) + start, std::begin(objects) + end, comparator);

        size_t mid = start + objectSpan/2;
        left = std::make_shared<BvhNode>(objects, start, mid);
        right = std::make_shared<BvhNode>(objects, mid, end);
    }
}
```

Instead of choosing a random splitting axis, it makes sense to split along the longest axis of the enclosing bounding box.

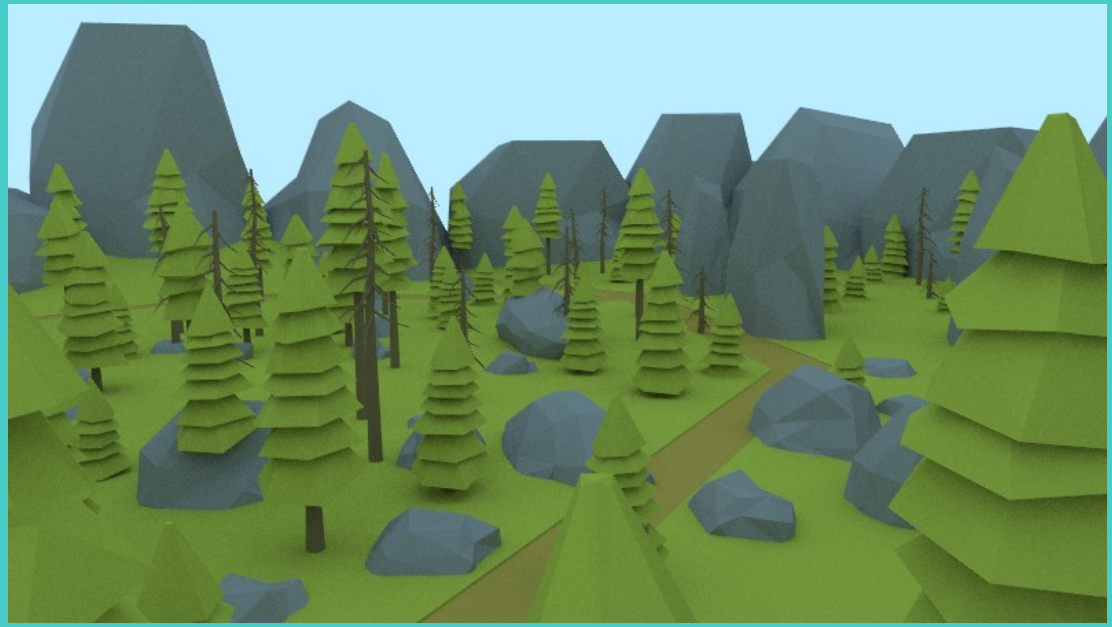
(end of recursion)

1) RAY TRACING IN ONE WEEKEND

Finally, here's the hit function for the BvhNode class:

```
bool hit(const Ray& r, Interval rayT, HitRecord& rec) const override {
    if (!bbox.hit(r, rayT)) return false;

    bool hitLeft = left->hit(r, rayT, rec);
    bool hitRight = right->hit(r, Interval(rayT.min, hitLeft ? rec.t : rayT.max), rec);

    return hitLeft || hitRight;
}
```

Image of complex scene, rendered in ***8h 46m 39s*** with Bounding Volume Hierarchies (and multiple threads).



ADDITIONAL FEATURES

2) ADDITIONAL FEATURES

Even faster: ***Parallelism***

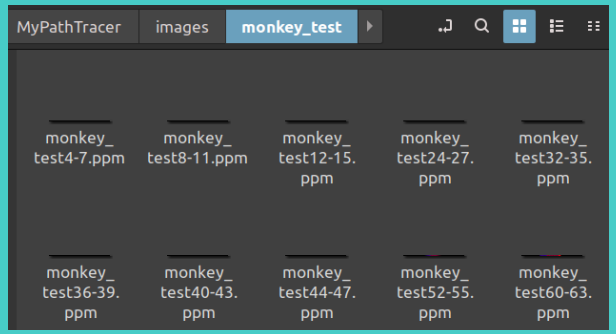
To make the path tracer ***parallel***, we split the image into "**sub-images**" (groups of rows), and make each thread render a separate one.

```
struct SubImage {
    Interval rows;
    std::string path;
};
```

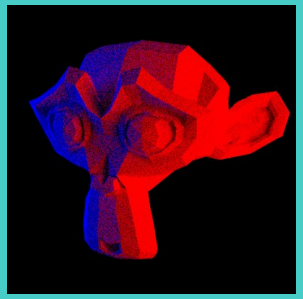
These structs are used by the threads.

```
struct SubImageList {
    // elements of list
    std::vector<SubImage> data;
    // this mutex controls the access to `data`
    std::mutex mtx;
};
```

The sub-images are kept in a directory and then put together at the end.




```
Running 16 threads, with 100 sub-images to be rendered

Number of meshes in scene: 3
Mesh #1 - Triangles: 968
Mesh #2 - Triangles: 2
Mesh #3 - Triangles: 2
Total number of triangles in scene: 972

Rendering sub-images in images/monkey_test
0 sub-images out of 100 have been rendered
Thread #1 has rendered sub-image images/monkey_test/monkey_test304-307.ppm
7 sub-images out of 100 have been rendered
Thread #1 has rendered sub-image images/monkey_test/monkey_test156-159.ppm
52 sub-images out of 100 have been rendered
Thread #1 is 50% done with current sub-image (2 rows left)
```



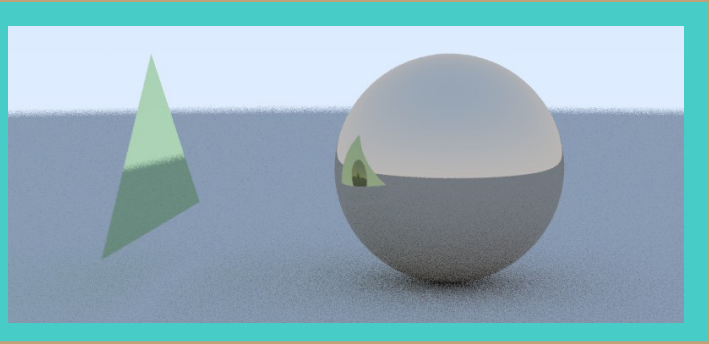
monkey_test.ppm

The rendering loop that we saw earlier is actually inside renderTask:

```
// Task for concurrent threads:
// Renders a sub-image from the `notRendered` list,
// Ends when there are no longer sub-images that need rendering.
void renderTask(const Hittable& world, SubImageList& notRendered,
                SubImageList& rendered, bool first) const;
```

2) ADDITIONAL FEATURES

Adding **Triangles**



A triangle is made of 3 vertices:

```
struct Vertex {
    Point3 position;
    Vec3 normal;
    glm::vec2 texCoords;
};
```

```
class Triangle : public Hittable {
    private:
        // Triangle vertices
        Vertex v0;
        Vertex v1;
        Vertex v2;
        Vec3 e1; // v1 - v0
        Vec3 e2; // v2 - v0

        std::shared_ptr<Material> mat;
        Aabb bbox;
```

The minimum Bounding Box has its corners at:
$(x_{min}, y_{min}, z_{min})$, $(x_{max}, y_{max}, z_{max})$

```
void Triangle::setBoundingBox() {
    // Compute the triangle's bounding box
    Point3 p0 = v0.position;
    Point3 p1 = v1.position;
    Point3 p2 = v2.position;

    float minX = fmin(p0.x, fmin(p1.x, p2.x));
    float minY = fmin(p0.y, fmin(p1.y, p2.y));
    float minZ = fmin(p0.z, fmin(p1.z, p2.z));

    float maxX = fmax(p0.x, fmax(p1.x, p2.x));
    float maxY = fmax(p0.y, fmax(p1.y, p2.y));
    float maxZ = fmax(p0.z, fmax(p1.z, p2.z));

    bbox = Aabb(Point3(minX, minY, minZ), Point3(maxX, maxY, maxZ));
}
```

2) ADDITIONAL FEATURES

For ray-triangle intersection, we use the **Möller-Trumbore algorithm.**

Using ***barycentric coordinates*** **:**

$$P = wA + uB + vC$$

$$w = 1 - u - v$$

$$P = (1 - u - v)A + uB + vC$$

$$P = A - uA - vA + uB + vC = A + u(B - A) + v(C - A)$$

Where (B-A) and (C-A) are the edges AB and AC of the triangle ABC.
Parametric equation of ray:

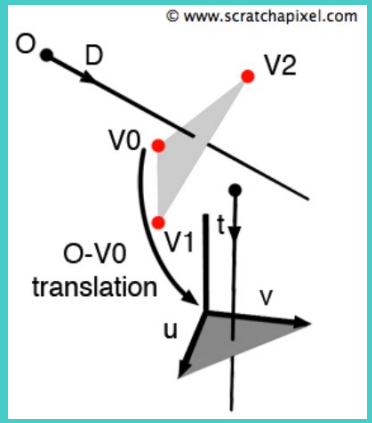
$$P = O + tD$$

Intersection:

$$\begin{aligned} O + tD &= A + u(B - A) + v(C - A) \\ O - A &= -tD + u(B - A) + v(C - A) \end{aligned}$$

t, u and v are the unknown values we are looking for.

$$\begin{bmatrix} -D & (B - A) & (C - A) \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = O - A$$



P is where the ray intersects the triangle ABC.
The barycentric coordinates of P are invariant to rotation, scaling, and translation.
The Möller-Trumbore algorithm takes advantage of this property by solving the ray-triangle intersection using barycentric coordinates.

2) ADDITIONAL FEATURES

To solve the equation

$$\begin{bmatrix} -D & (B-A) & (C-A) \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = O - A$$

We use **Cramer's rule:**

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\det(M)} \begin{bmatrix} \det(M_t) \\ \det(M_u) \\ \det(M_v) \end{bmatrix}$$

Where $M = [-D, B-A, C-A]$ and:

- M_t is M with its first column replaced by O-A
- M_u is M with its second column replaced by O-A
- M_v is M_t with its third column replaced by O-A

We re-write this as:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\begin{vmatrix} -D & E_1 & E_2 \end{vmatrix}} \begin{bmatrix} \begin{vmatrix} T & E_1 & E_2 \end{vmatrix} \\ \begin{vmatrix} -D & T & E_2 \end{vmatrix} \\ \begin{vmatrix} -D & E_1 & T \end{vmatrix} \end{bmatrix}$$

$T = O - A$
$E_1 = B - A$
$E_2 = C - A$

The determinant of a $3x3$ matrix $[A\ B\ C]$ can be calculated as (*scalar triple product*):

$$|A\ B\ C| = -(A \times C) \cdot B = -(C \times B) \cdot A$$

So:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(D \times E_2) \cdot E_1} \begin{bmatrix} (T \times E_1) \cdot E_2 \\ (D \times E_2) \cdot T \\ (T \times E_1) \cdot D \end{bmatrix} =$$

$$= \frac{1}{P \cdot E_1} \begin{bmatrix} Q \cdot E_2 \\ P \cdot T \\ Q \cdot D \end{bmatrix}$$

Where $P = (D x E_2)$ and $Q = (T\ x\ E_1)$

The denominator $(D\ x\ E_2) \cdot E_1$ can be expressed as $D \cdot (E_1\ x\ E_2)$. The cross product of E_1 and E_2 gives the **normal** of the **triangle**. If the **dot product** of the **ray direction** and the **triangle's normal** is **zero**, then the triangle and the ray are **parallel**, resulting in **no intersection**.

2) ADDITIONAL FEATURES

The hit function (part 1):

```
bool Triangle::hit(const Ray& r, Interval rayT, HitRecord& rec) const {
    // Check if ray hits triangle using the Muller-Trumbore method
    Point3 p0 = v0.position;
    Vec3 pvec = glm::cross(r.direction(), e2);
    float det = glm::dot(pvec, e1);

    if (std::fabs(det) < 1e-8) return false; // triangle and ray are parallel
    float invDet = 1.0f / det;

    Vec3 tvec = r.origin() - p0;
    float u = glm::dot(pvec, tvec) * invDet;
    if (u < 0.0f || u > 1.0f ) return false;

    Vec3 qvec = glm::cross(tvec, e1);
    float v = glm::dot(qvec, r.direction()) * invDet;
    if (v < 0.0f || u + v > 1.0f) return false;

    float t = glm::dot(qvec, e2) * invDet;
    if (!rayT.contains(t)) return false;

    Point3 intersection = r.at(t);
    rec.t = t;
    rec.p = intersection;
    rec.material = mat;
```

2) ADDITIONAL FEATURES

The hit function (part 2):

```
    // Interpolate normal values from vertices
    Vec3 normal = v0.normal*(1-u-v) + v1.normal*u + v2.normal*v;
    rec.setFaceNormal(r, normal);
    // Interpolate texture coordinates (u and v)
    // (different meaning from barycentric coordinates)
    rec.u = v0.texCoords.x*(1-u-v) + v1.texCoords.x*u + v2.texCoords.x*v;
    rec.v = v0.texCoords.y*(1-u-v) + v1.texCoords.y*u + v2.texCoords.y*v;
    return true;
}
```

We also use the barycentric coordinates to interpolate between the normal values and the texture coordinates at the three vertices.

A **file** for **user input**

There's a file called *ptInput.txt* which can be used to **set** various **parameters** of the **path tracer**.

```
// Reads the parameter from current line and moves
// to the next line in `inputStream`
template <typename T>
T readParameter(std::ifstream& inputStream);
```

Example of function used to read user input from file.

ptInput.txt ×

```
--------SCENE SETTINGS---------

Scene Number : 0

3D Model Name : monkey

--------CAMERA SETTINGS--------

- Output Image Name : monkey_test

- Aspect Ratio (width/height) :
  - width : 1
  - height : 1

- Image Width (pixels) : 400

- Vertical Field of View : 40

- "Look-From" Vector :
  - x : 1.50235
  - y : 0.84
  - z : 3.5796

- "Look-At" Vector :
  - x : 1.15043
  - y : 0.60995
  - z : 2.67228

- "Up" Vector :
  - x : -0.0831916
  - y : 0.973179
  - z : -0.214481

- Per-Pixel Samples : 100

- Defocus Angle : 0.0
```

Input file

2) ADDITIONAL FEATURES

More interesting scenes: **3D model loading**

The path tracer can load 3D models in the *wavefront .obj* format, along with a *.mtl* file for materials.
All the conventions that need to be followed for correct material loading are listed in the *README.md* file of the project.

For model loading, the ***Open Asset Import Library*** (***Assimp***) was used.

```
class Model {
    private:
        // A Model is a list of meshes
        std::vector<Mesh> meshes;
```

```
class Mesh {
    private:
        // A mesh is represented as a list of triangles,
        // all sharing the same material.
        std::vector<std::shared_ptr<Triangle>> triangles;
        std::shared_ptr<Material> material;
```

Assimp splits our 3D model in meshes, which are split into triangles.

Material Conventions

The *.mtl* files are expected to follow various conventions, loosely based on the MTL File Format documentation (here, here and here)
.
Here's how the parameters are interpreted:

- `Ns` represents the **shininess** (from 0 to 1000)
- `Ka` is ignored
- `Kd` represents the **diffuse color**
- `Ks` represents the **specular color**
- `Ke` represents the **emissive color**
- `Ni` represents the **index of refraction**
- `d` is ignored
- `illum` represents the **illumination model**
- `map_Kd` is the **texture image file** for **lambertian materials**

The path tracer works with the following types of materials:

- **Lambertian**, which can be specified in the *.mtl* file with:
  - `illum 2` (*base shading*) as the illumination model
  - `Kd` as the color, or
  - `map_Kd` as the texture (in this case, `Kd` is ignored) (**NOTE**: if `Ke` is different from zero, the material will be treated as a **diffuse light**)
- **Metal**, which can be specified with:
  - `illum 3` (*reflection on*) as the illumination model

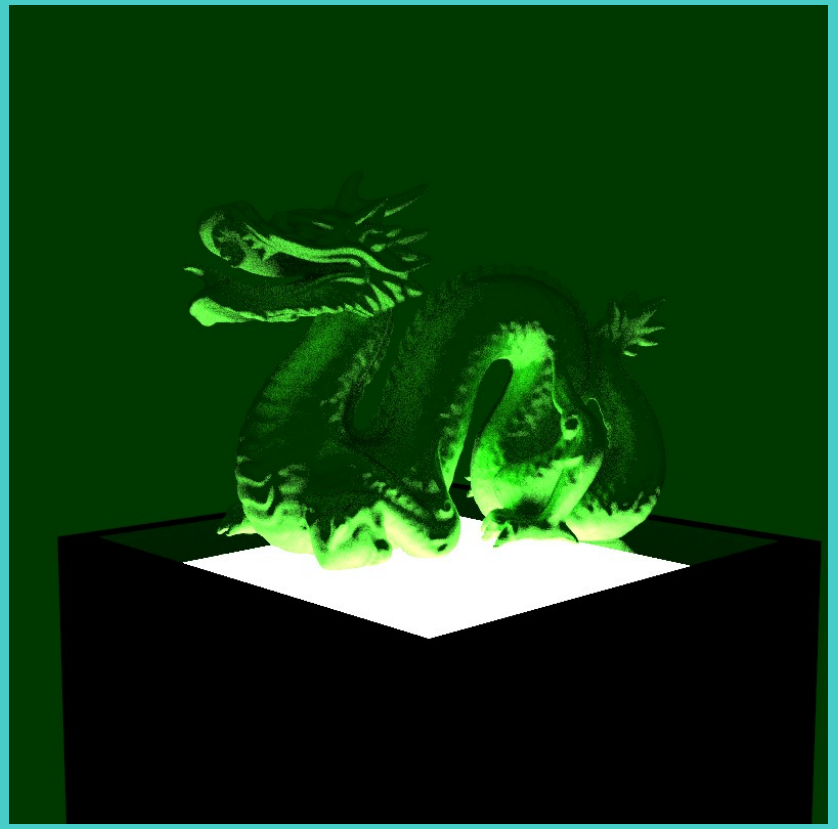
README.md

2) ADDITIONAL FEATURES

We can build a Bounding Volume Hierarchy out of a loaded 3D model:

```
std::shared_ptr<BvhNode> Model::buildBvh(){
    // Bounding Volume Hierarchies of meshes
    HittableList meshBvhs;
    for (unsigned int i = 0; i < meshes.size(); i++) {
        meshBvhs.add(meshes[i].buildBvh());
    }
    return make_shared<BvhNode>(meshBvhs);
}
```

```
std::shared_ptr<BvhNode> Mesh::buildBvh(){
    HittableList list;
    for (unsigned int i = 0; i < triangles.size(); i++) {
        list.add(triangles[i]);
    }
    return std::make_shared<BvhNode>(list);
}
```

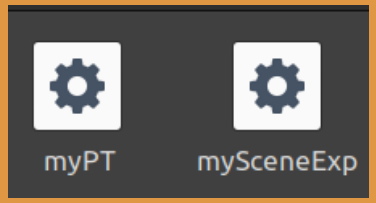


***871,478* triangles**,
rendering time ***36h 37m 37s***,
800x800px, 200 spp pixel, max depth 5,
16 threads, 400 sub-images.

SCENE EXPLORER

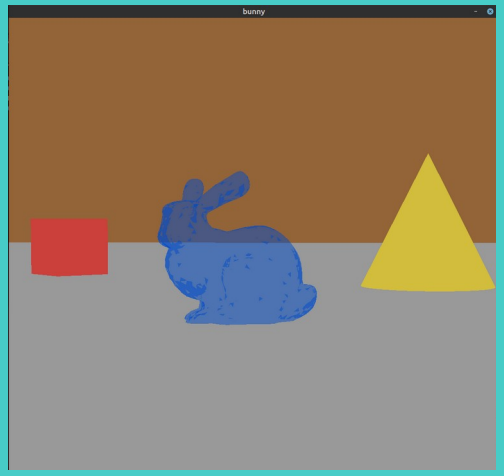
3) SCENE EXPLORER

Along with the path tracer executable *myPT*, there's also the *mySceneExp* program, which can be used to **explore scenes** loaded through *.obj* files, and **position** the **camera** before running the path tracer on them.



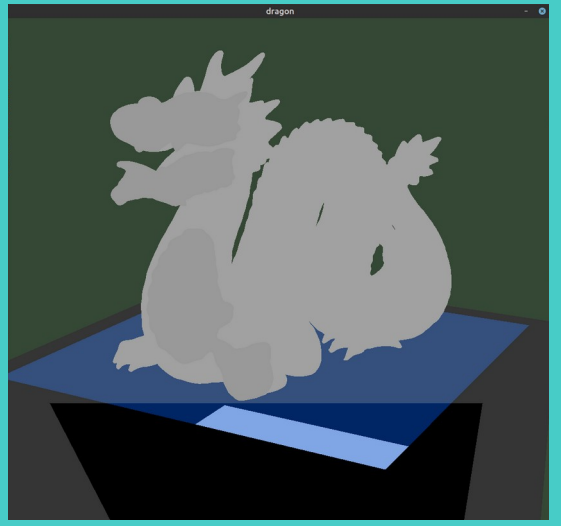
The scene explorer uses fixed colors for the following materials:

- Metal → grey
- Dielectric → transparent blue
- Light → white



The user can move around the scene (mouse, *WASD, SPACEBAR, E*) and then press *ENTER* to have the current **camera position** be **written** on the **path tracer input file**.

The **focus plane** can be seen by typing *F*, and its distance changed with the *UP/DOWN* arrows.



The scene explorer also **reads** the camera position (not orientation) from the **input file** *ptInput.txt*
Input utilities are in source folder *src/common*

3) SCENE EXPLORER

How the camera is moved around: an example

- The user presses *W*
- *SDL2* generates a corresponding event
- processInput() is called in the rendering loop
- The *SDL2* event is recognized as the *W* key being pressed down, and the element at index FORWARD in the global variable cameraIsMoving is set to true

```
// Direction of Camera Movement
enum CamMovement {
    FORWARD,
    BACKWARD,
    LEFT,
    RIGHT,
    UP,
    DOWN,
    NUM_DIRECTIONS,
    NO_MOVEMENT
};
```

```
bool cameraIsMoving[NUM_DIRECTIONS];
```

- In the rendering loop, updateCamera() checks cameraIsMoving and calls Camera::move() if there's any movement

```
// Rendering loop
while(!window.isClosed()){
    // per-frame time logic
    float currentFrame = SDL_GetTicks();
    deltaTime = currentFrame - lastFrame;
    lastFrame = currentFrame;

    // Process user input
    processInput();

    // Move around camera
    updateCamera();
    mvp.view = camera.getViewMatrix();
    updateMVP(ourShader, mvp);

    // Draw to the screen
    window.clear(background.r,background.g,background.b,1.0f);
    ourModel.draw(ourShader);

    if (focusPlaneOn) {
        planeMVP.model = glm::translate(
                                glm::vec3(0.0f,0.0f,-focusPlaneDistance));
        updateMVP(ourShader, planeMVP);
        focusPlane.draw(ourShader);
    }
    // Show new rendered image
    window.update();
}
```

Rendering loop of *mySceneExp*

3) SCENE EXPLORER

- Camera::move() changes the position vector of the camera
- The rendering loop gets the new *view matrix* of the scene through Camera::getViewMatrix()

```
glm::mat4 getViewMatrix() const{
    return glm::lookAt(position, position + front, up);
}
```

- The rendering loop updates the *MVP matrix* used in the *vertex shader* by calling updateMVP()

```
void updateMVP(const Shader &program, MVP mvp){
    program.setUniformMatrix("model", mvp.model);
    program.setUniformMatrix("view", mvp.view);
    program.setUniformMatrix("projection", mvp.projection);
}
```

- The rendering loop uses the shader to draw the new, modified scene on the window
- [Similar for when *W* is released]

Mouse movements are based on the "Camera" tutorial at *learnopengl.com*

```
void Camera::move(bool cameraIsMoving[], float deltaTime){
    float movementScale = movementSpeed * deltaTime;
    vec3 movement(0.0f,0.0f,0.0f);

    if (cameraIsMoving[FORWARD]) movement +=
                    normalize(vec3(front.x,0.0f,front.z));
    if (cameraIsMoving[BACKWARD]) movement -=
                    normalize(vec3(front.x,0.0f,front.z));
    if (cameraIsMoving[LEFT]) movement -= right;
    if (cameraIsMoving[RIGHT]) movement += right;
    if (cameraIsMoving[UP]) movement += WORLD_UP;
    if (cameraIsMoving[DOWN]) movement -= WORLD_UP;

    position += movement*movementScale;
}
```

Camera::move() function

```
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main() {
    TexCoords = aTexCoords;
    gl_Position = projection * view * model * vec4(aPos, 1.0);
}
```

Vertex shader

References

- Ray Tracing / Path Tracing:
  https://developer.nvidia.com/discover/ray-tracing
  https://blogs.nvidia.com/blog/what-is-path-tracing
  https://commons.wikimedia.org/w/index.php?curid=3869326
- The *Ray Tracing in One Weekend* book series:
  https://raytracing.github.io
- The glm::reflect() and glm::refract() functions:
  https://glm.g-truc.net/0.9.5/api/a00157.html
- *Reflections and Refractions in Ray Tracing* by Bram de Greve:
  https://graphics.stanford.edu/courses/cs148-10-summer/docs/2006--degreve--reflection_refraction.pdf
- Ray Tracing meshes:
  https://anandhotwani.github.io/2020/08/24/ray-tracing-meshes.html
  https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/moller-trumbore-ray-triangle-intersection.html
  https://en.wikipedia.org/wiki/M%C3%B6ller%E2%80%93Trumbore_intersection_algorithm
- The *Learn OpenGL* tutorial series:
  https://learnopengl.com
- BennyQBD's *Modern OpenGL Tutorial*:
  https://github.com/BennyQBD/ModernOpenGLTutorial