Ray Tracing

EDAN35: Seminar 1

Overview

- Ray tracing overview (recap from EDAF80)
- swTracer C++ framework overview
- Lab 1 - Whitted Ray Tracing

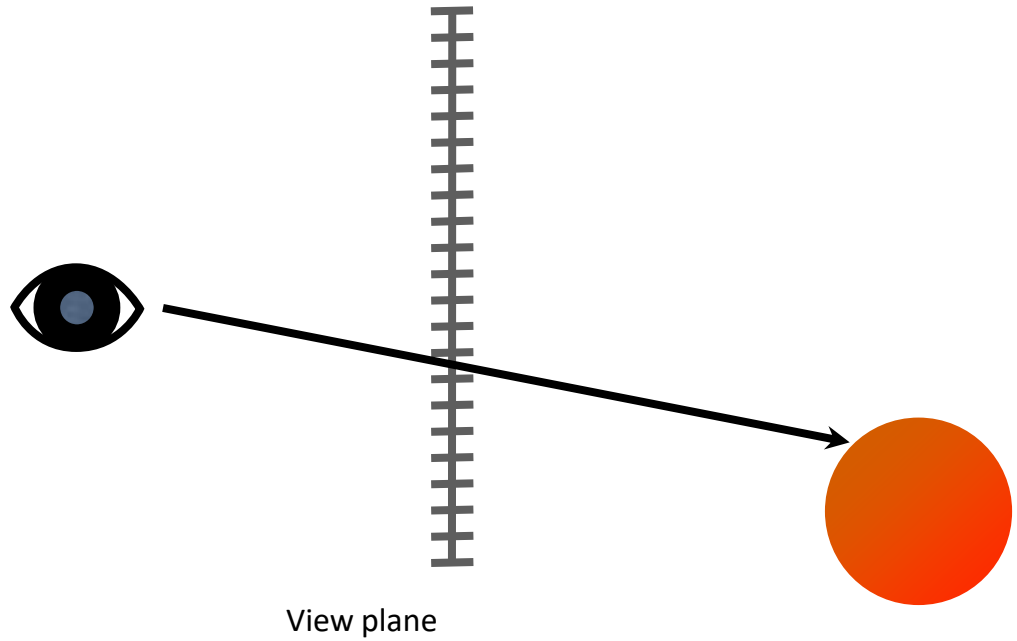
Ray Tracing

- For each pixel trace a ray
  - Find the nearest object along the ray
- Global sampling for lighting and visibility
- "Rasterization is fast, but needs cleverness to support complex visual effects. Ray tracing supports complex visual effects, but needs cleverness to be fast.”

  - David Luebke, Nvidia

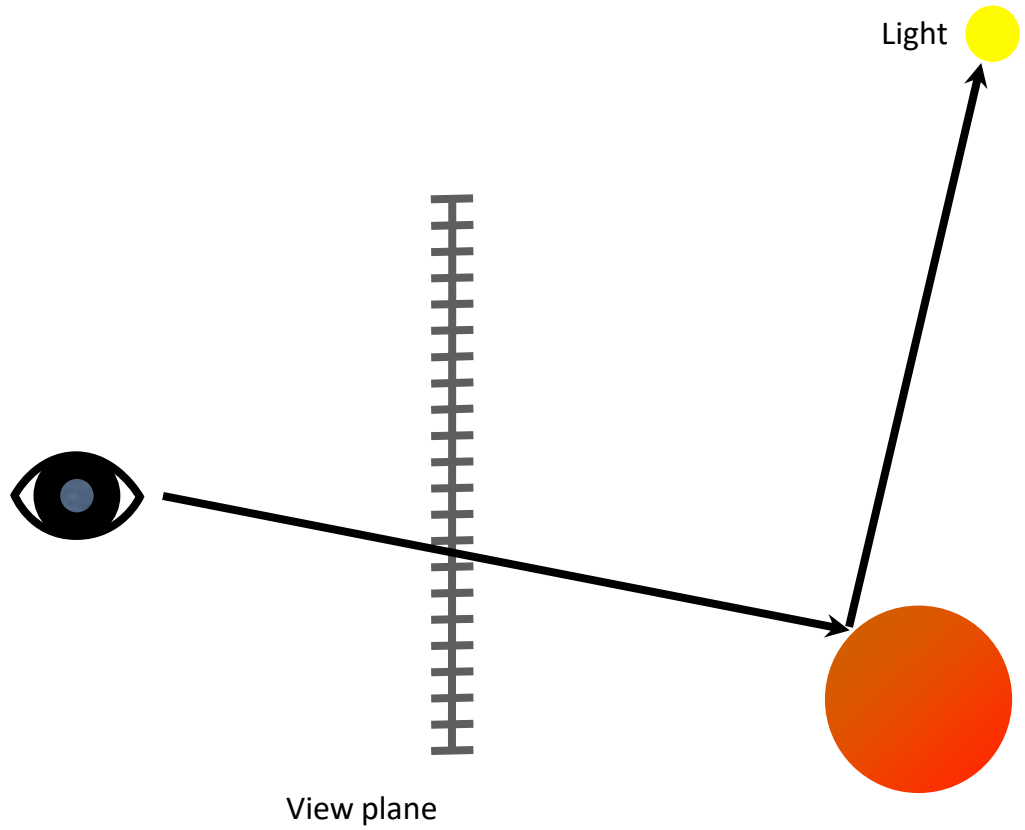
Ray Tracing

- Construct a line (ray) from the eye
  - through the view plane and into the scene
- Find intersection with objects

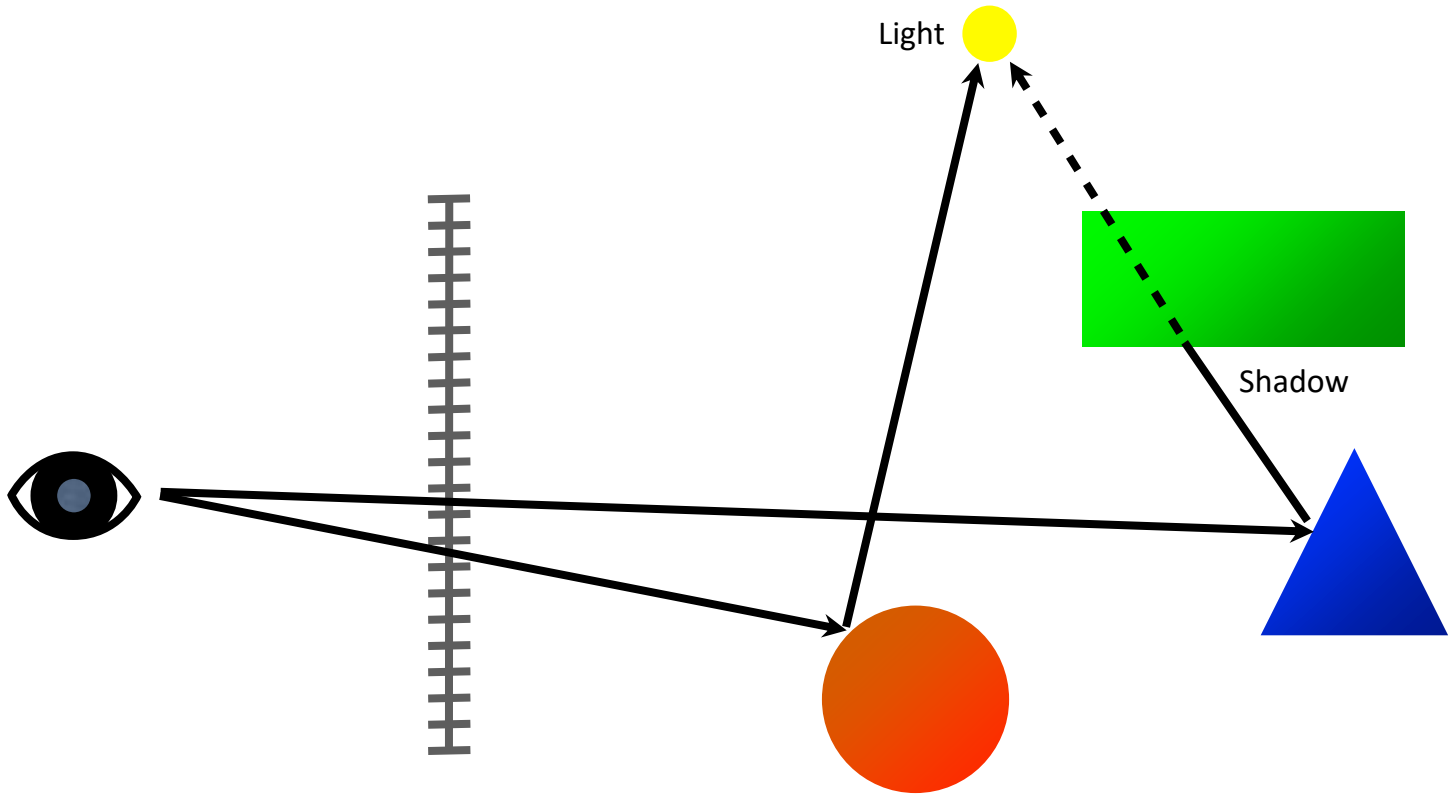


Shadows

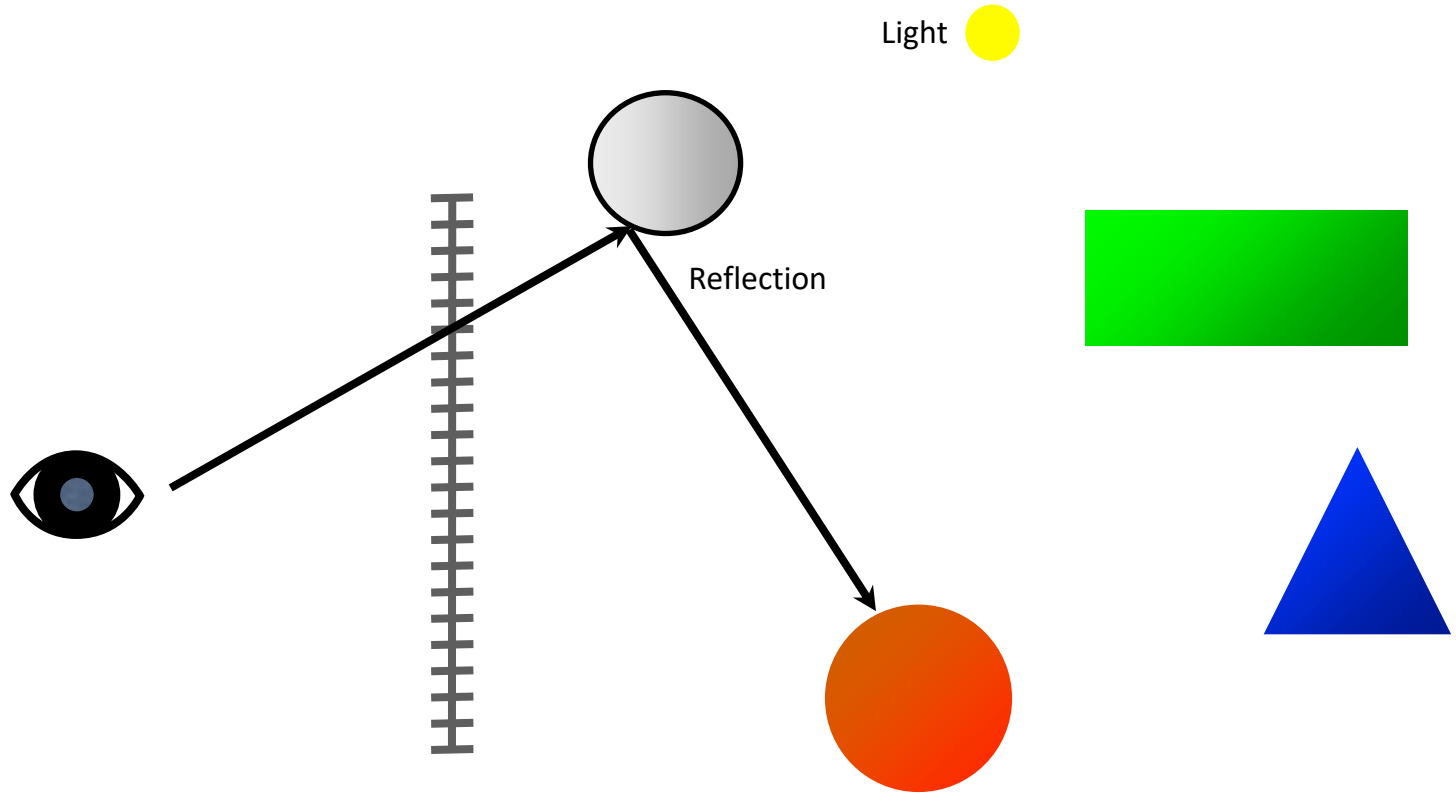
- Trace a ray from intersection to every light to check for shadows and shade surface



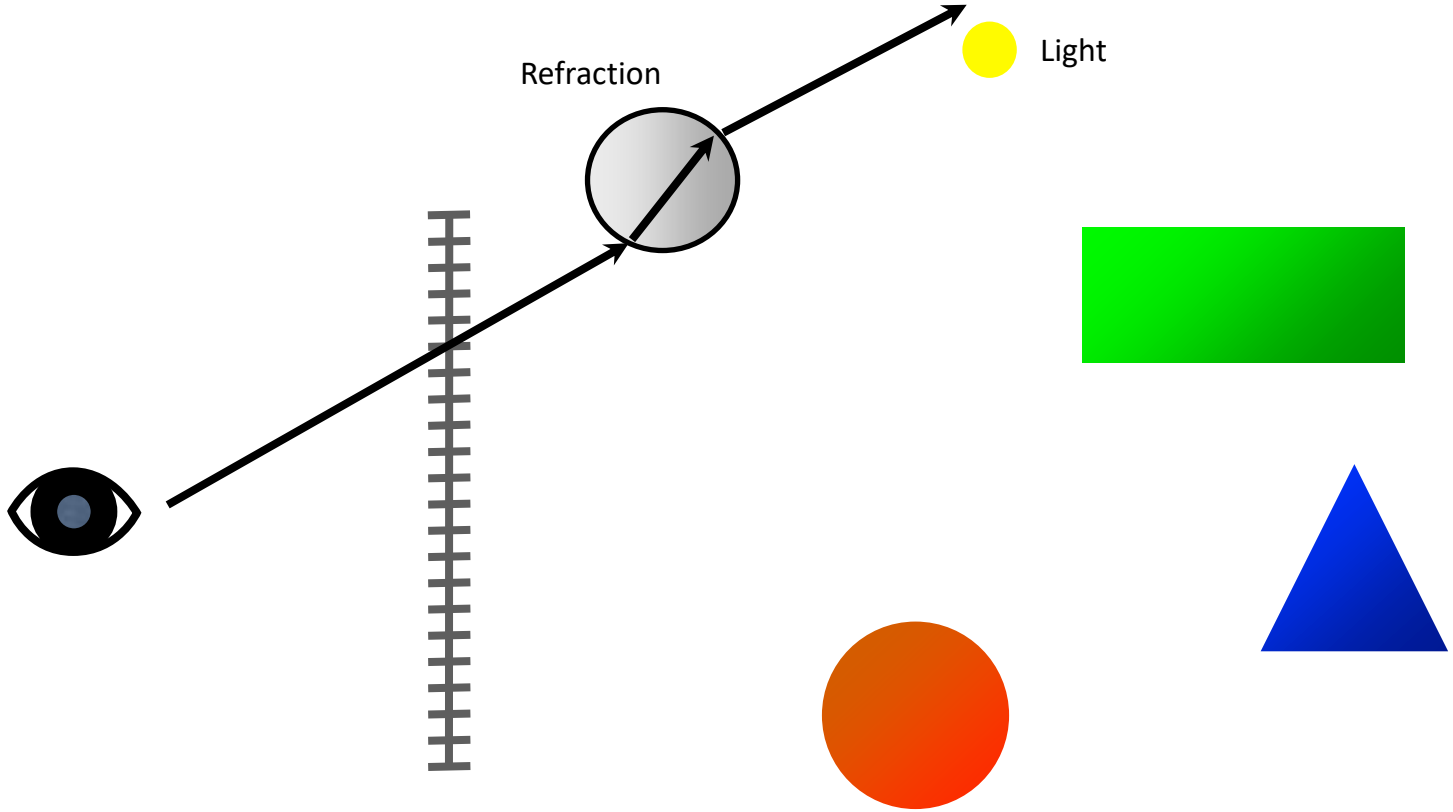
Shadows



Reflection

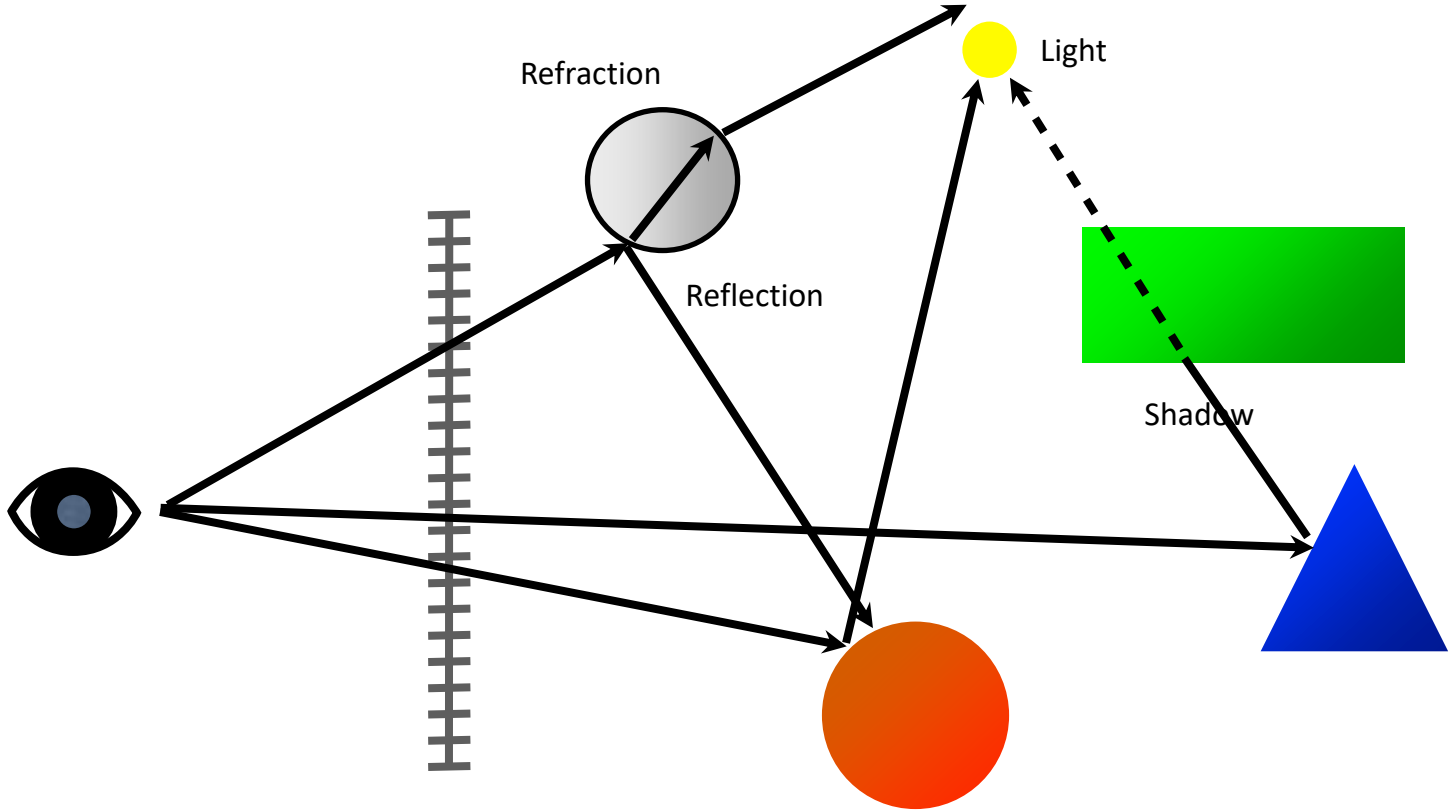


Refraction



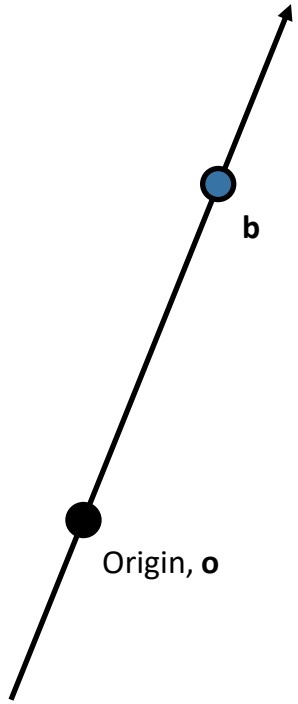
Recursive Ray Tracing

- At each intersection, trace shadow, reflection, and refraction rays

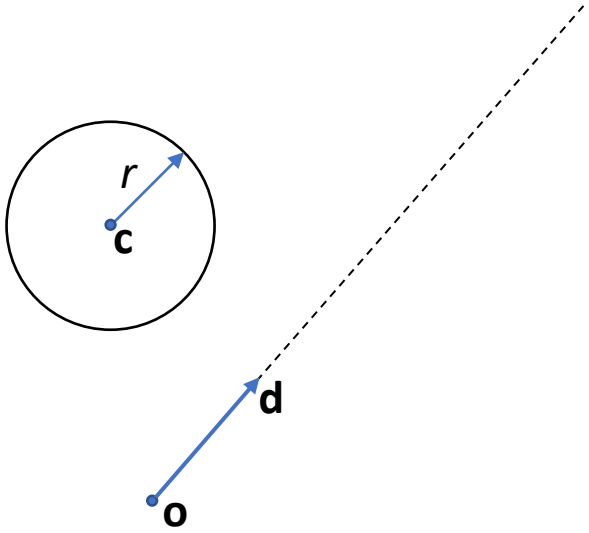


Ray definition

- Origin and direction
- Position on ray is represented using a parameter, *t*
  - **c** = (1-t)**o** + *t***b**
  - **c** = **o** + *t*(**b** - **o**) = **o** + *t***d**
  - **c** is a point along the ray
- t >= 0, ray is a half line
- ϵ < t < FLT_MAX
  - Small epsilon avoids intersecting surface due to numerical imprecisions



Ray Sphere Intersection



- Analytical solution
- Sphere centre: **c**, and radius *r*
- Ray: $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$
- Sphere: $\|\mathbf{p} - \mathbf{c}\| = r$
- Replace **p** by **r**(t), and square it:

$$(\mathbf{r}(t) - \mathbf{c}) \cdot (\mathbf{r}(t) - \mathbf{c}) - r^2 = 0$$

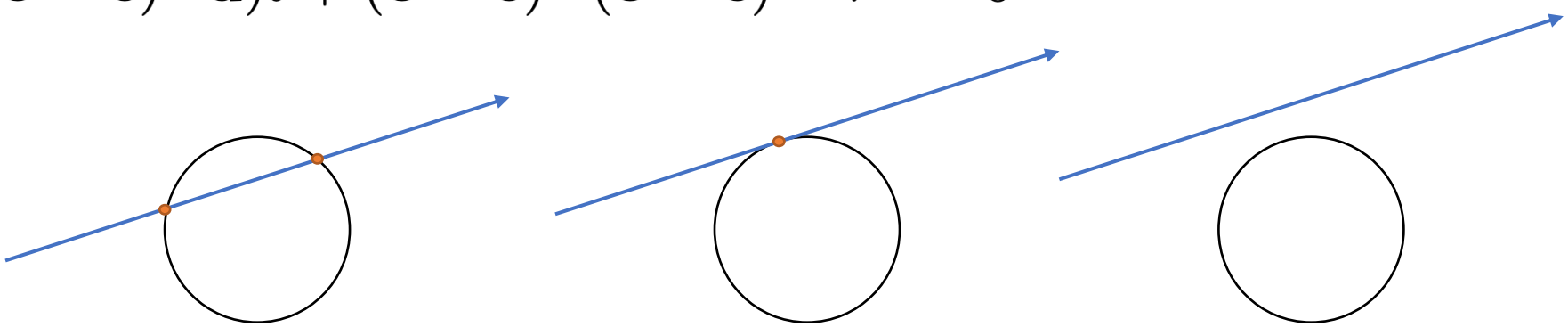
$$(\mathbf{o} + t\mathbf{d} - \mathbf{c}) \cdot (\mathbf{o} + t\mathbf{d} - \mathbf{c}) - r^2 = 0$$

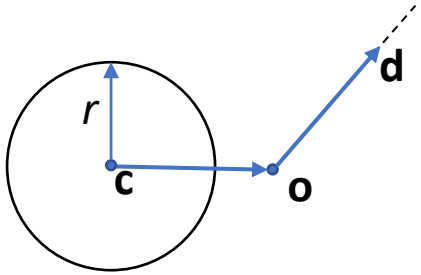
$$(\mathbf{d} \cdot \mathbf{d})t^2 + 2((\mathbf{o} - \mathbf{c}) \cdot \mathbf{d})t + (\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - r^2 = 0$$

$$t^2 + 2((\mathbf{o} - \mathbf{c}) \cdot \mathbf{d})t + (\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - r^2 = 0 \qquad \|\mathbf{d}\| = 1$$

Ray Sphere Intersection

$$t^2 + 2((\mathbf{o} - \mathbf{c}) \cdot \mathbf{d})t + (\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - r^2 = 0$$





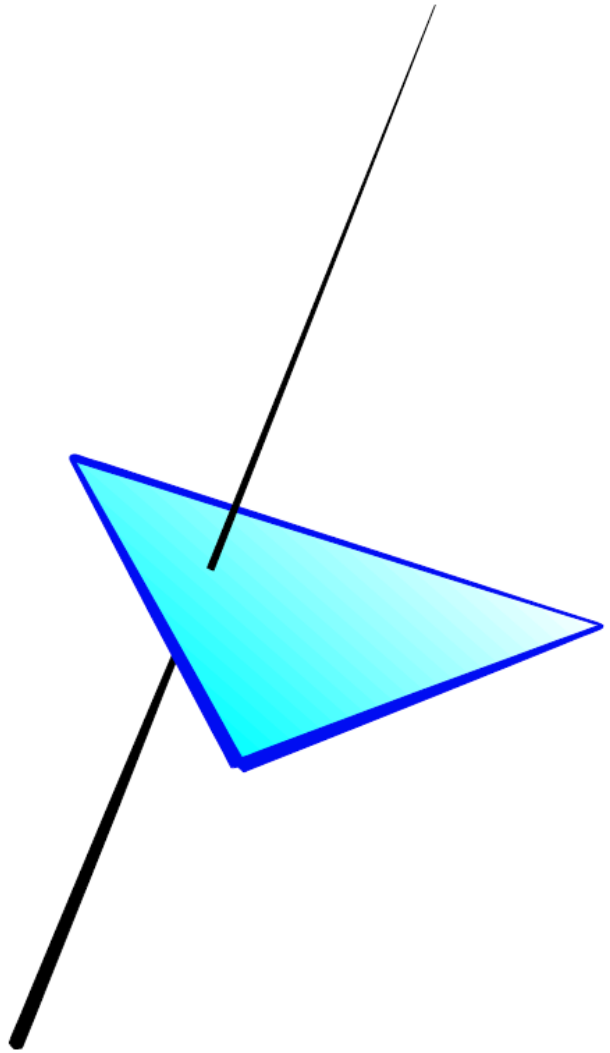
- Some optimisations can be done

  $(\mathbf{o} - \mathbf{c}) \cdot \mathbf{d} > 0?$

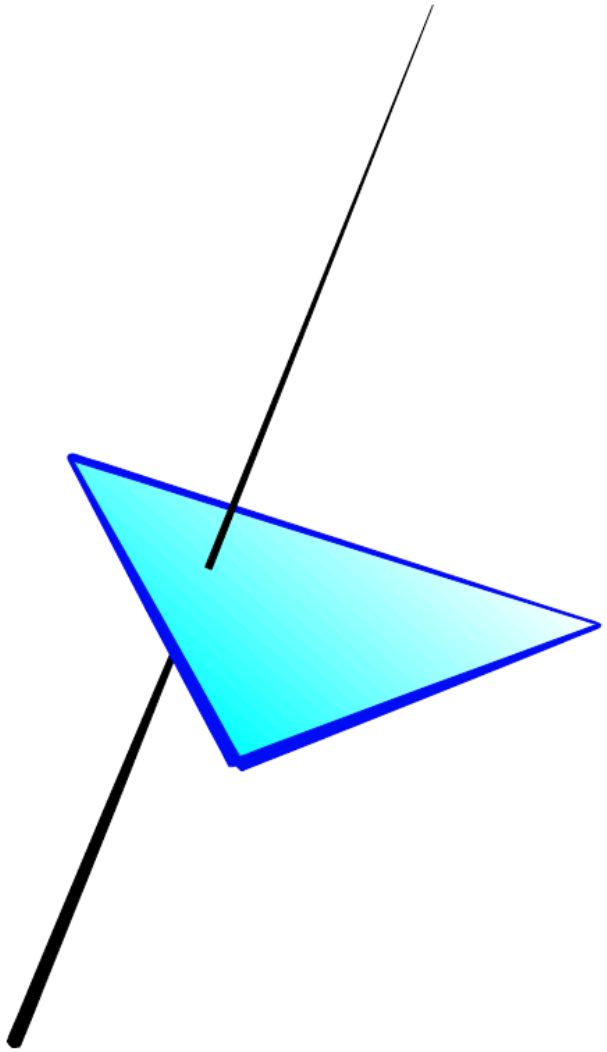
  $(\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - r^2 < 0?$

- Such tests are called “rejection tests”
- This approach can be extended for other shapes

Ray Triangle Intersection



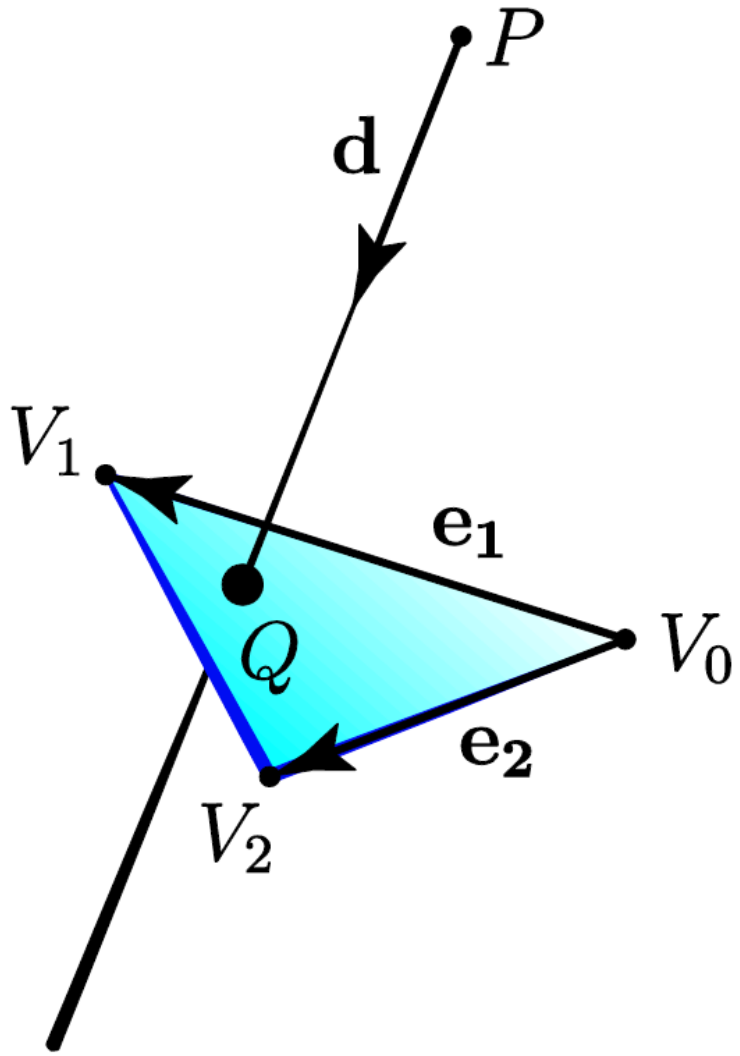
Ray Triangle Intersection



Solve the intersection problem in two steps:

1. Find the intersection point *(Q)* of the ray with the triangle plane
2. Determine if *Q* is inside the triangle bounds using barycentric coordinates

Ray Triangle Intersection



Recall…

A triangle is defined by three vertices

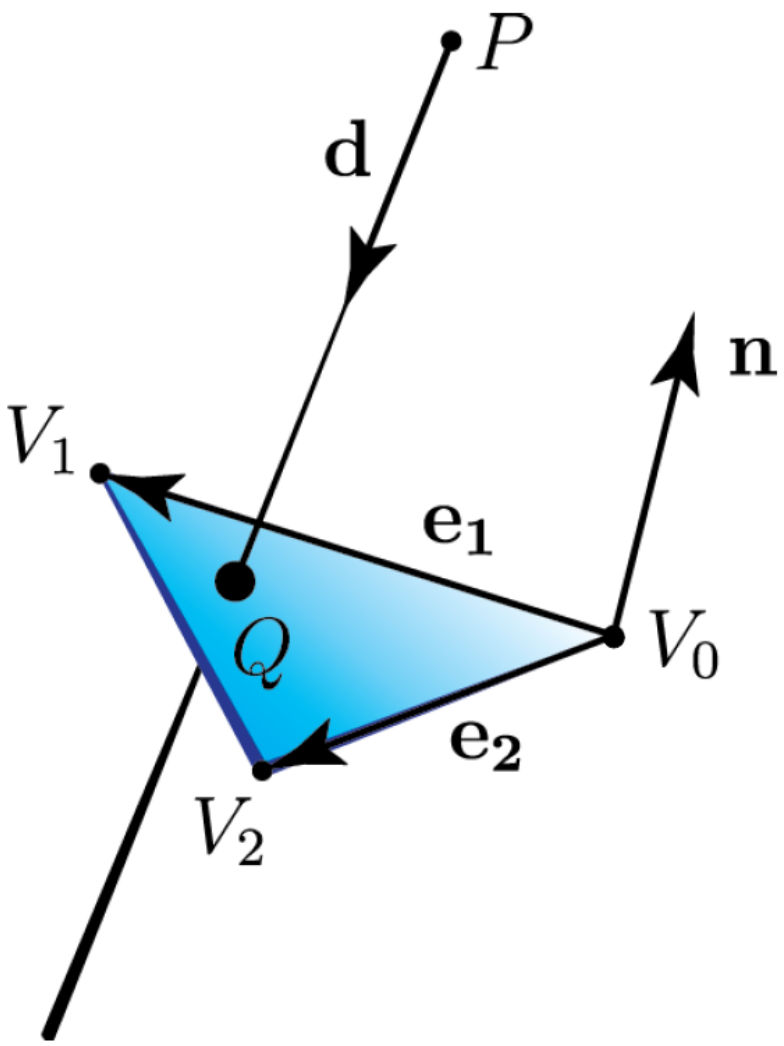
V_i, i = {0, 1, 2}

A ray is defined by some origin *P*, and a direction vector **d**.

An intersection with the triangle plane will occur at some distance *t* along the ray.

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

Ray Triangle Intersection



Step 1: Find Q

Plane normal

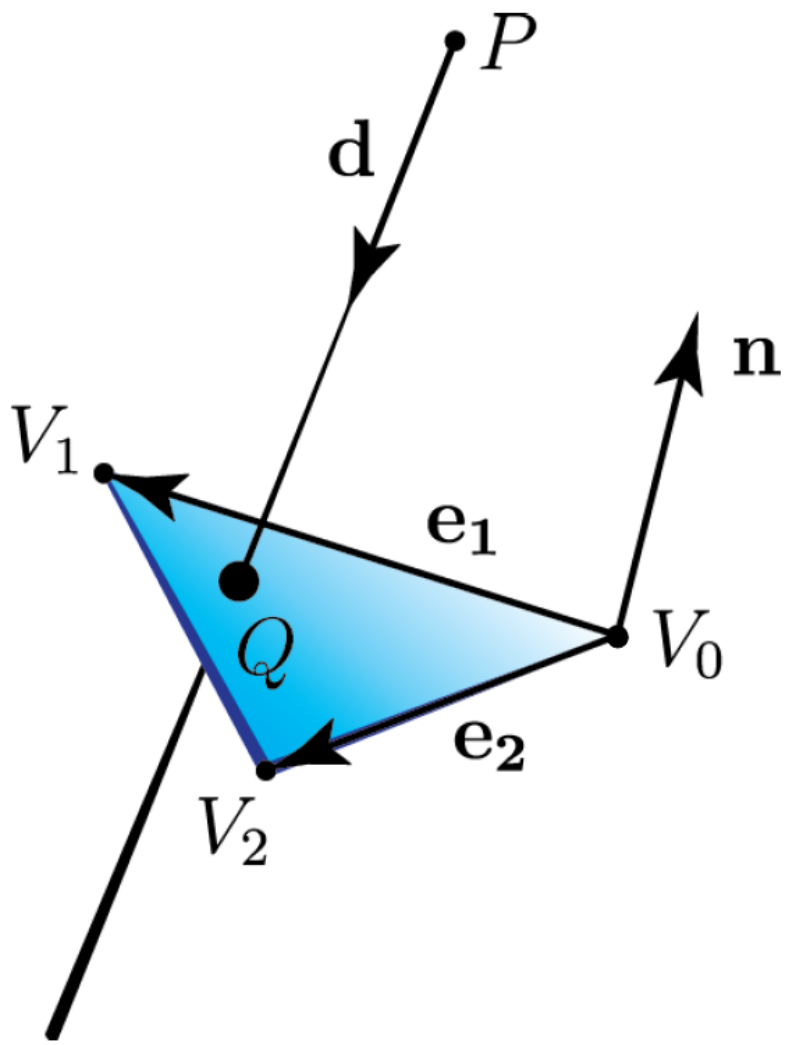
$$\mathbf{n} = \mathbf{e}_1 \times \mathbf{e}_2$$

Plane equation

$$\mathbf{n} \cdot X + m = 0$$
$$m = -\mathbf{n} \cdot V_0$$

Note: The magnitude of **n** corresponds to 2x the area of the triangle.

Ray Triangle Intersection



Step 1: Find Q

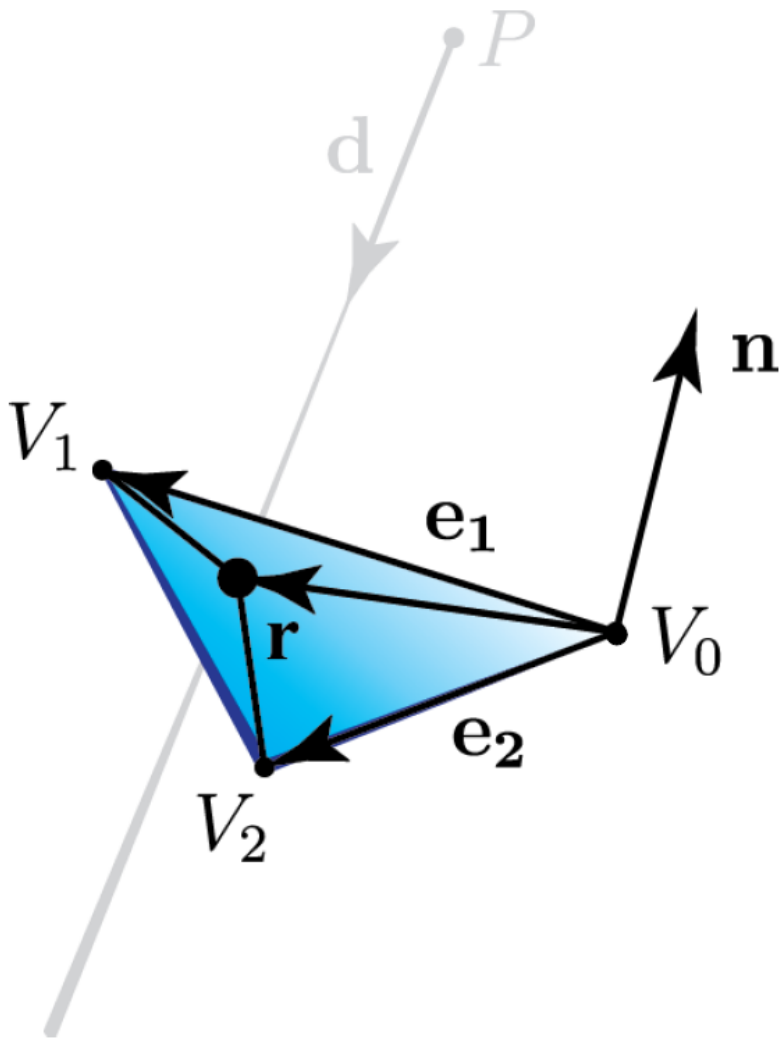
Plane intersection

$$t = \frac{\mathbf{n}\cdot P + m}{-\mathbf{n}\cdot\mathbf{d}}$$

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

Must also make sure that

$$t_{min} < t < t_{max}$$



Step 2: Find barycentric coordinates

Create vector **r**, which is *coplanar* with $\mathbf{e}_1$ and $\mathbf{e}_2$

$$\mathbf{r} = Q - V_0$$

Barycentric *v* and *w*

$$v = \frac{||\mathbf{e}_1 \times \mathbf{r}||}{||\mathbf{n}||} \quad w = \frac{||\mathbf{r} \times \mathbf{e}_2||}{||\mathbf{n}||}$$

Barycentric coordinates can be expressed as the area of a "sub-triangle" divided by the area of the whole triangle

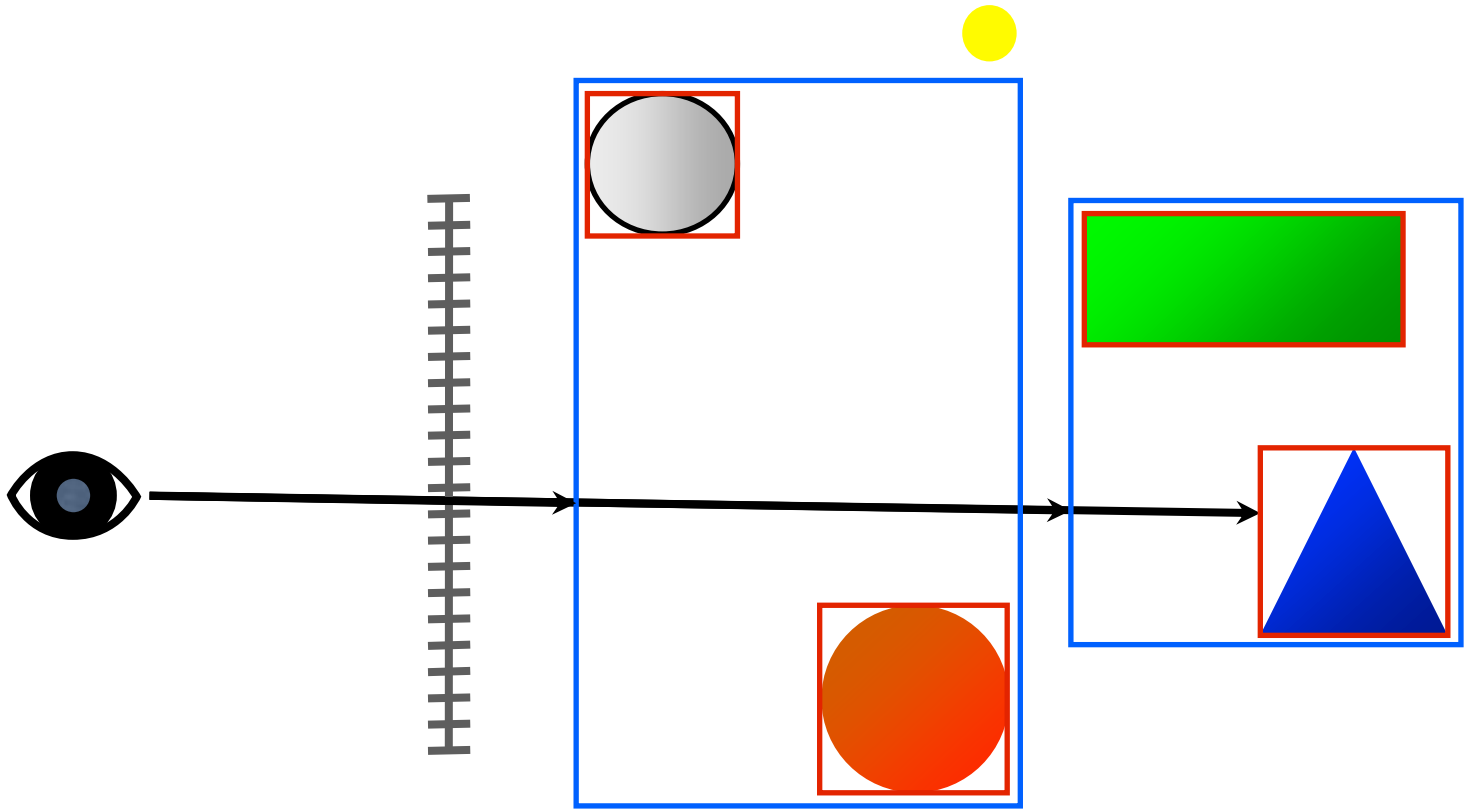
Thus, if the following holds, *Q* is inside the triangle

$$v \geq 0$$
$$w \geq 0$$
$$v + w < 1$$

Ray Triangle Intersection

- This method is just one of many
- Many faster ones exist

Hierarchical data structure



Bounding Volume Hierarchy (BVH) is most the common acceleration structure for ray tracing

swTracer Overview

- Written in C++
- Windows/Linux/Mac via CMake
- Very basic 3 component Vector library swVec3.h
- Using stb_image.h for writing images
  - PNG

sw::Vec3 vector class

- Color is a `sw::Vec3` class
- sw::Vec3 supports various math operations (+, -, *, +=, etc)
  - `w = u * v;  // dot product`
  - `w = u % v;  // cross product`
  - `v.normalize();`
  - `v.m[0], v.m[1], v.m[2], v.x(), v.y(), v.z()`

sw::Ray class

- Class describing a ray in 3D
  - Origin (`sw::Vec3`) `orig`
  - Direction (`sw::Vec3`) `dir`
- Constructor
  - `sw::Ray(origin, direction, minT, maxT)`

sw::Camera class

- Class describing a camera
- Constructor
  - `sw::Camera(origin, lookAt, up, fov, aspect);`
- `void sw::Camera::setup(width, height);`
  - Run at start of program
- `sw::Ray sw::Camera::getRay(x, y);`
  - Returns ray for given (x,y) in image plane, floats

sw::Primitive class

- Abstract class describing an object in the scene
- `virtual bool intersect(ray, isect);`
- `Material material;`

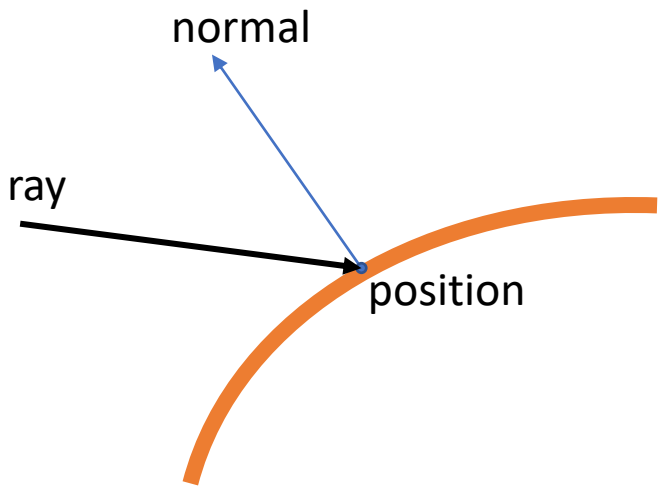
- Inherited by `sw::Sphere` and `sw::Triangle`

sw::Scene class

- Stores a list of primitives – `sw::Primitive` class
  - Sphere class includes sphere intersection code already
- Primitives have materials – `sw::Material` class
  - color, reflectivity, transparency, refractiveIndex
- Find closest intersection, returns true if intersection found
  - `bool Scene::intersect(const Ray* r, Intersection &isect)`
  - returns intersection information, slower
- Find any intersection
  - `bool Scene::intersect(const Ray* r, Intersection &isect, true)`
  - Used for shadow rays, can be faster

sw::Intersection class

- `getShadowRay(lightPos);`
- `getReflectedRay();`
- `getRefractedRay();`

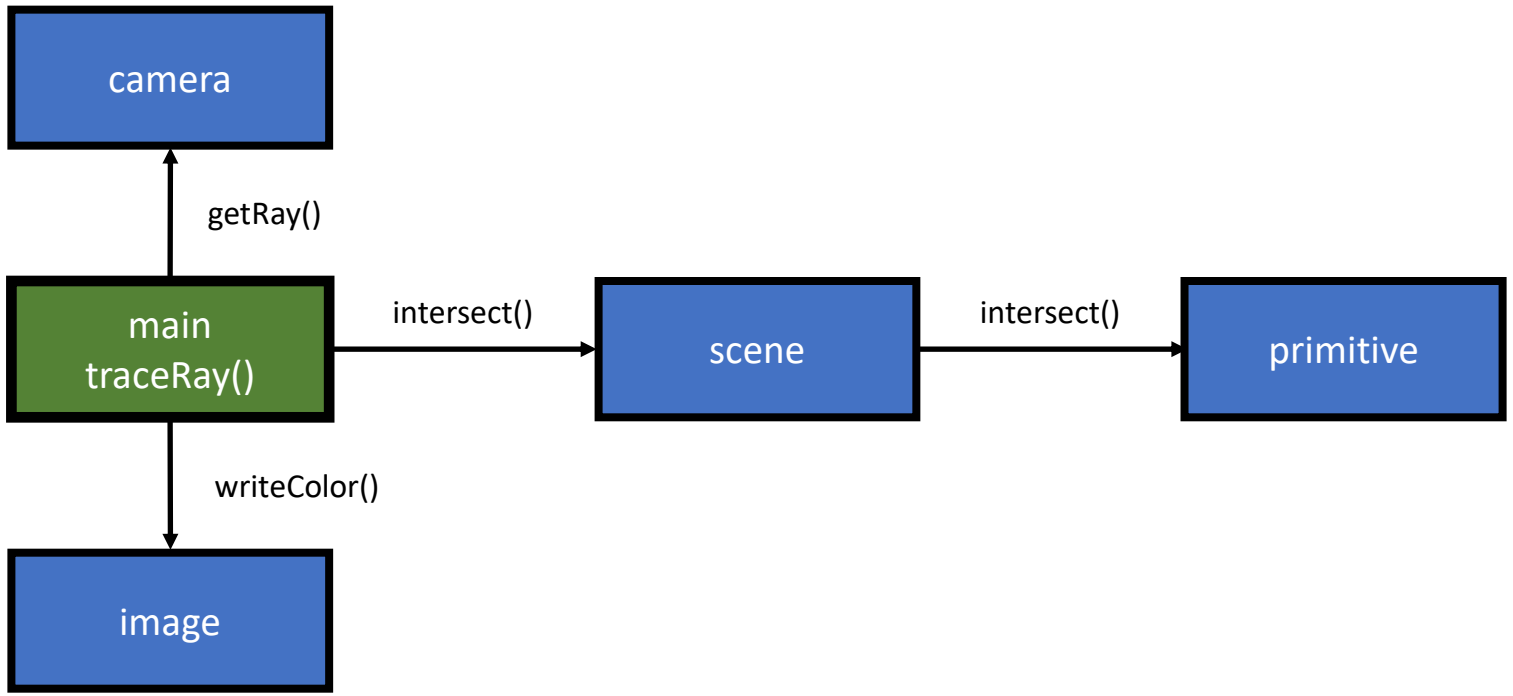


- `position` `// Position of hit point`
- `normal` `// Surface normal at hit point`
- `ray` `// Incoming ray direction`

Main function

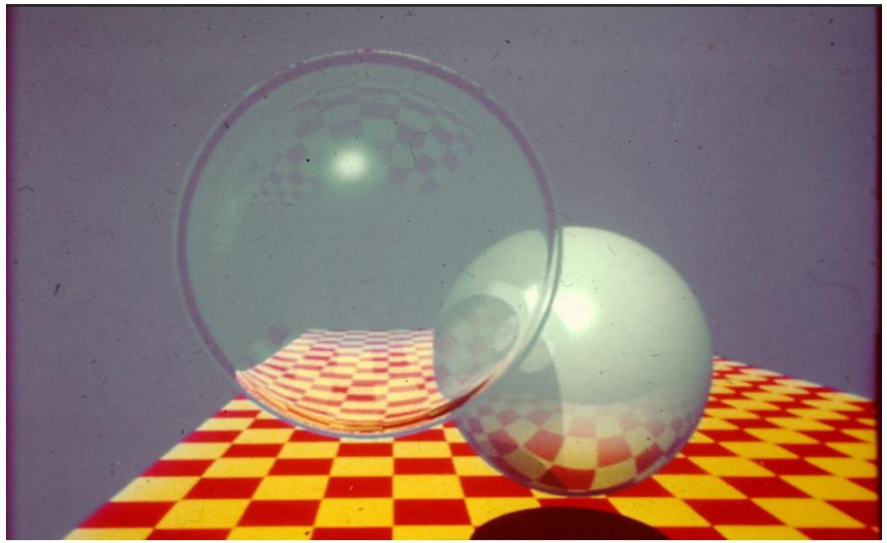
- Create image buffer (allocates memory)
- Create scene
  - Creates materials, add primitives to scene
- Setup camera
- Ray traces pixels
  - Loops over each pixel in scene
  - Calls traceRay() for rays each pixel, at center 0.5, 0.5
- Creates output image

swTracer flow overview



Whitted Ray Tracing

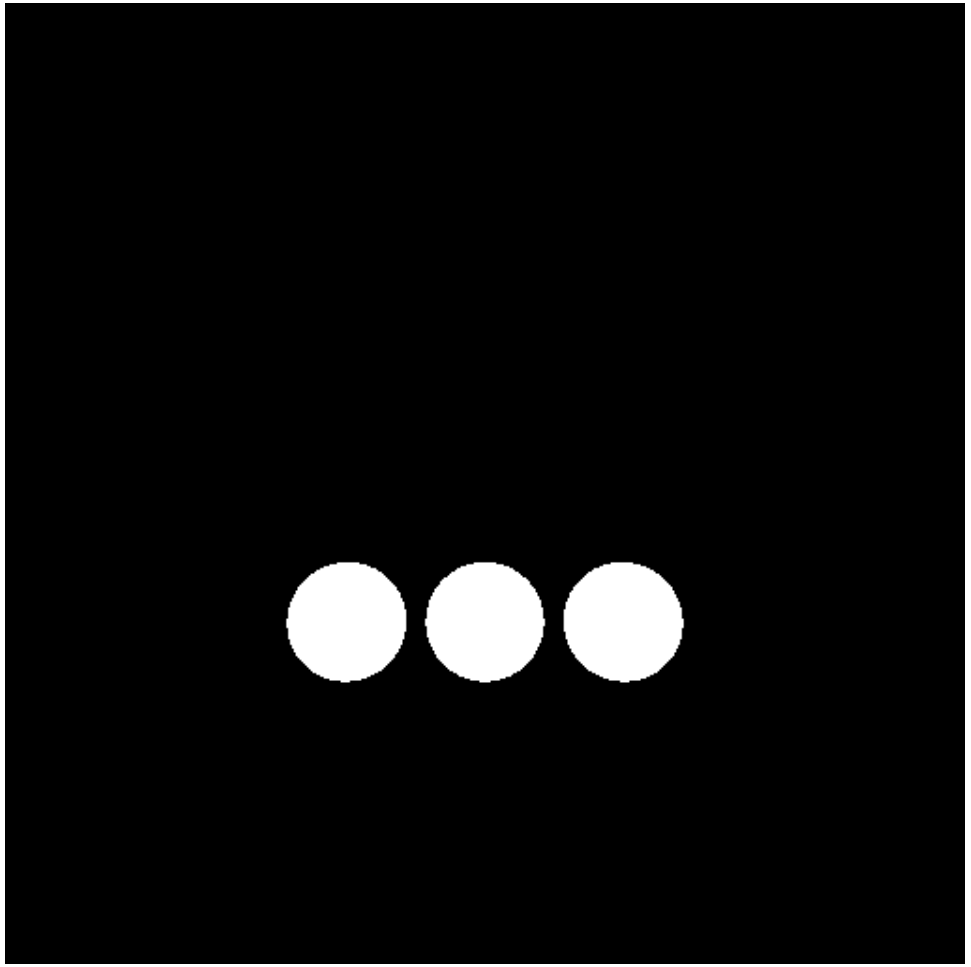
- 1980, “An Improved Illumination Model for Shaded Display”, Turner Whitted, CACM
- Simple surface model, perfect reflection
- Shadow rays trace to point light sources



Lab 1 - Whitted Ray Tracing

- Diffuse Shading
- Ray-triangle intersection
- Whitted Ray Tracing
  - Shadows
  - Reflections
  - Refractions
- Supersampling

Start Up



- White if the ray hits anything, black if background

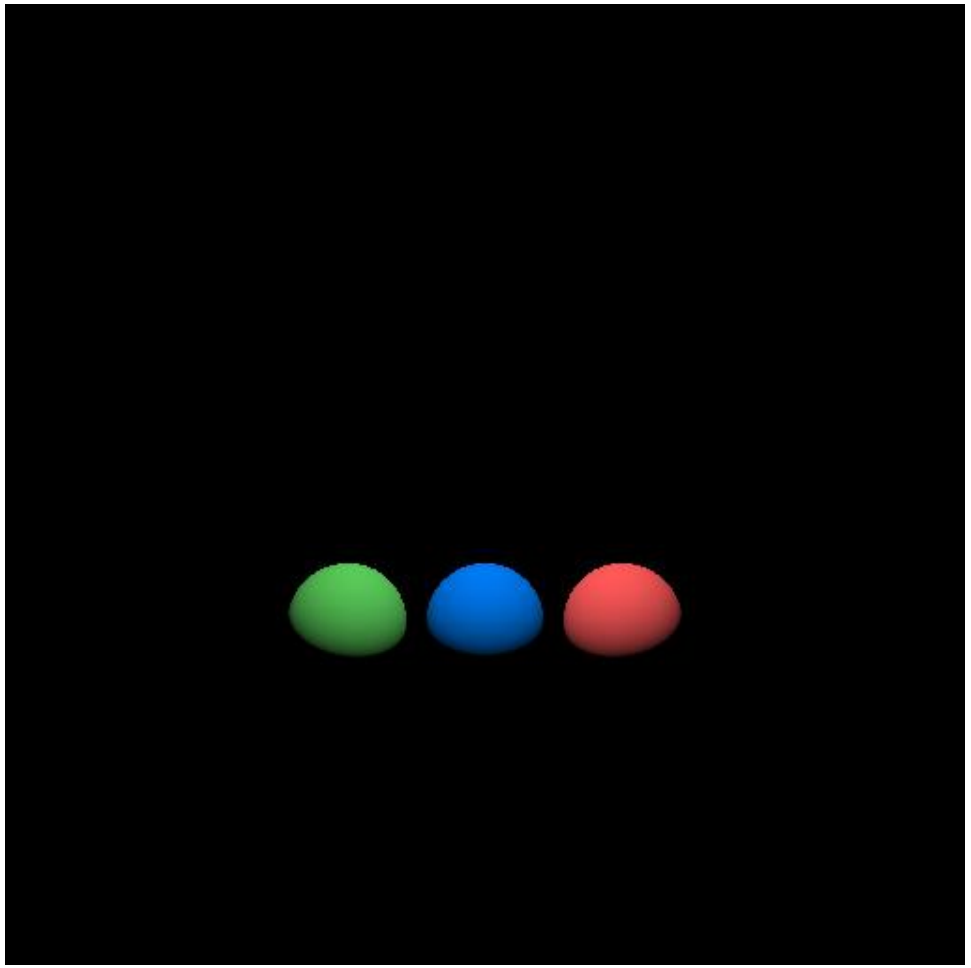
Diffuse Shading

- Need to modify `traceRay()` in `main.cpp`

- Change white return color to
  - material.color * N * L
  - “N dot L”, consider the cases

- Light vector is already calculated in the code: `lightDir`

Diffuse Shading

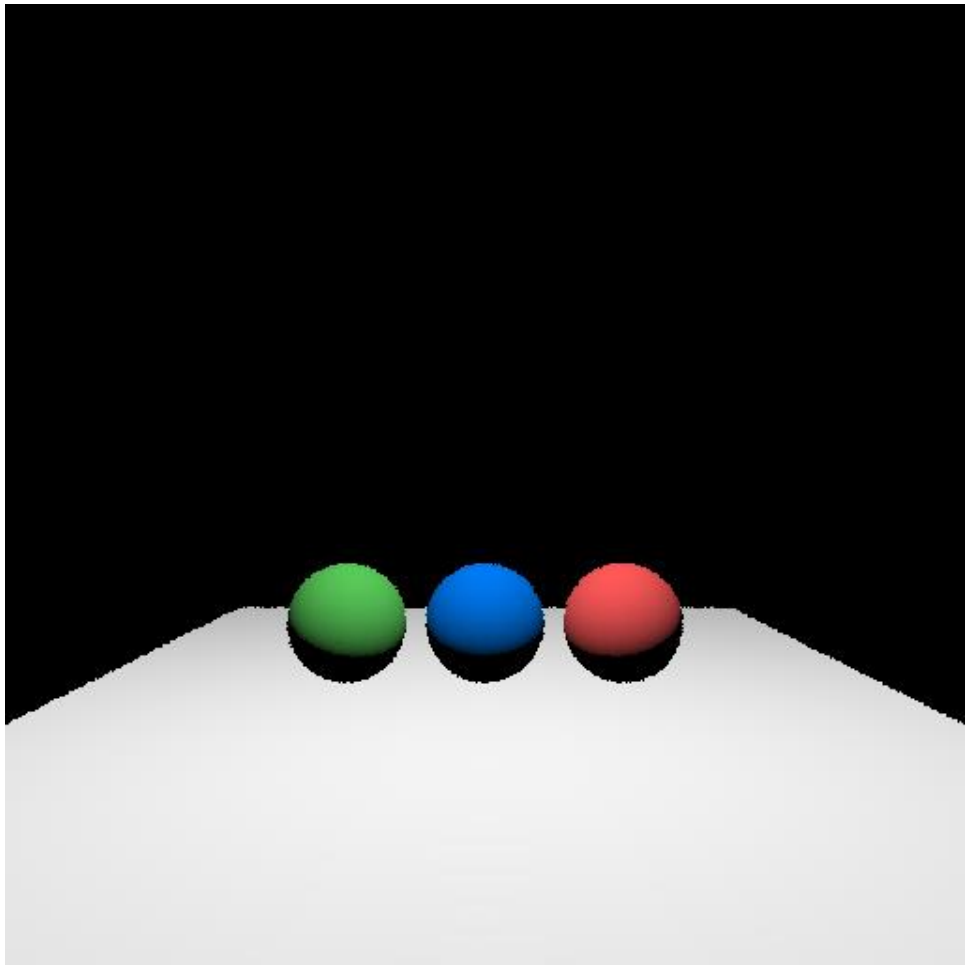


Ray Triangle Intersection

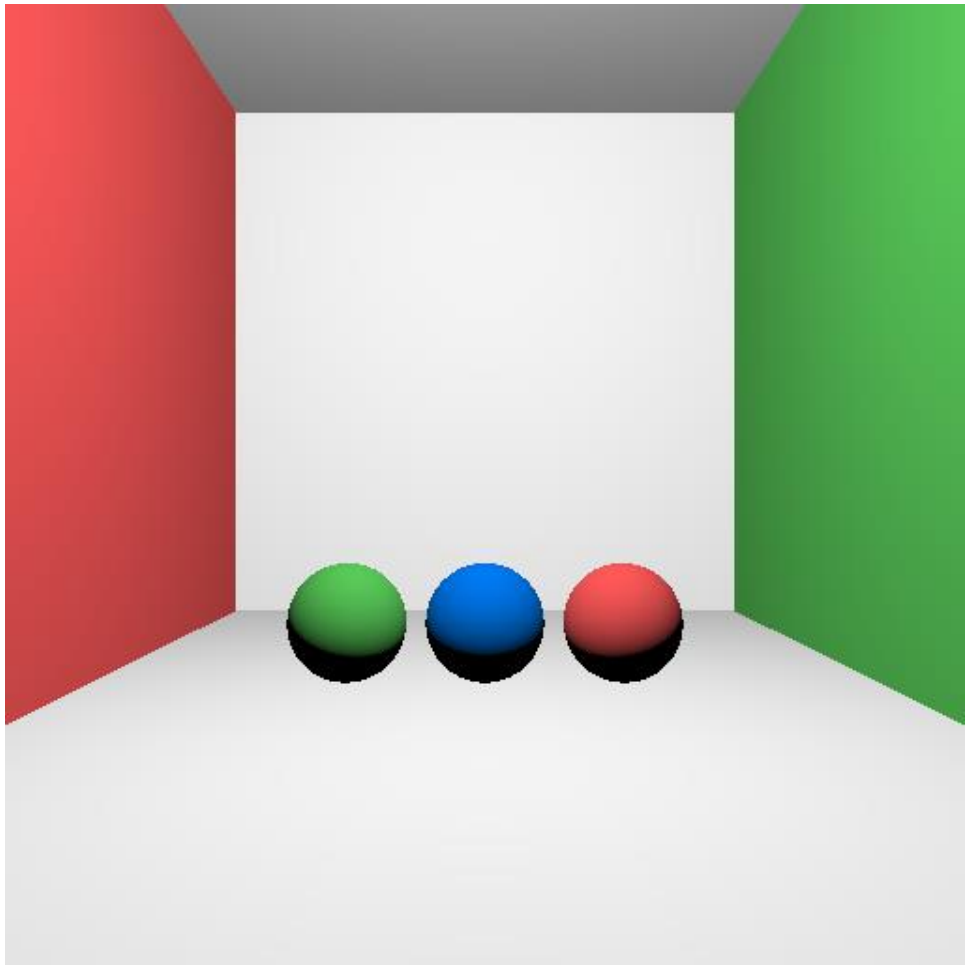
- Add triangles to scene (uncomment floor of the box)
- Need to implement `intersect()` in `swTriangle.cpp`

- Add in walls and ceiling when ray triangle intersection works

Ray Triangle Intersection



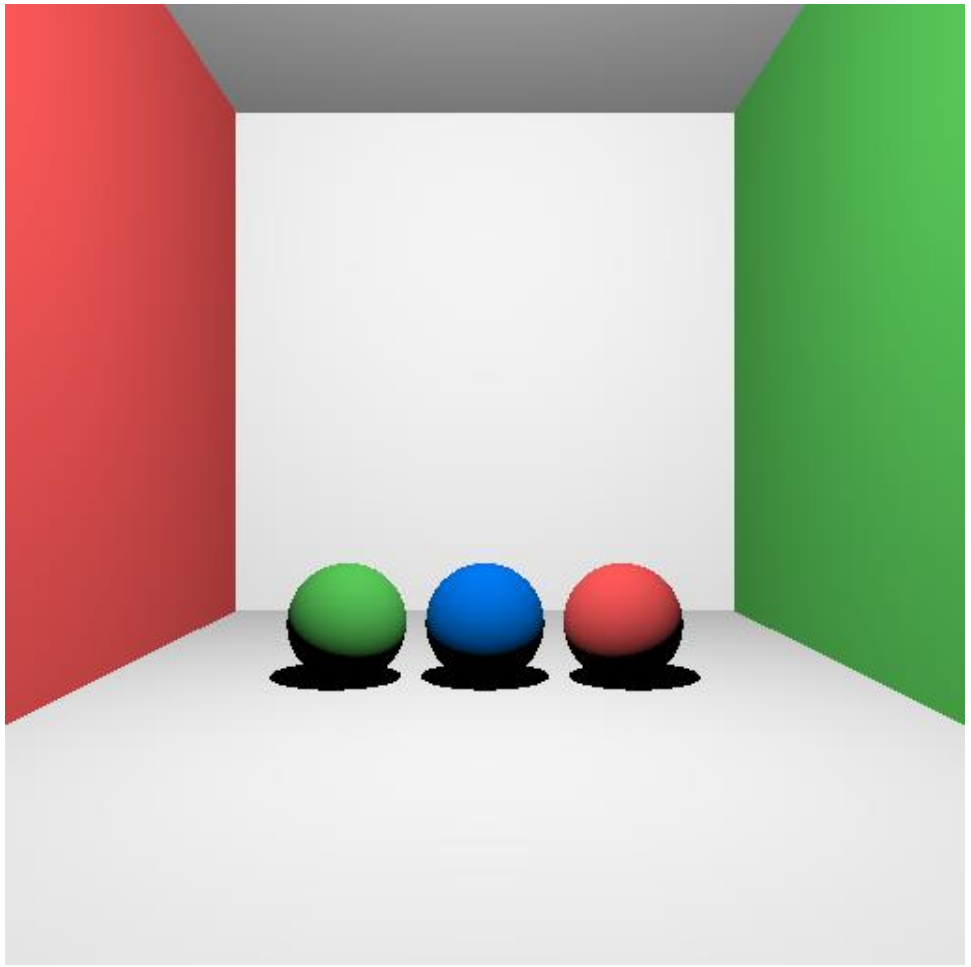
Ray Triangle Intersection



Shadows

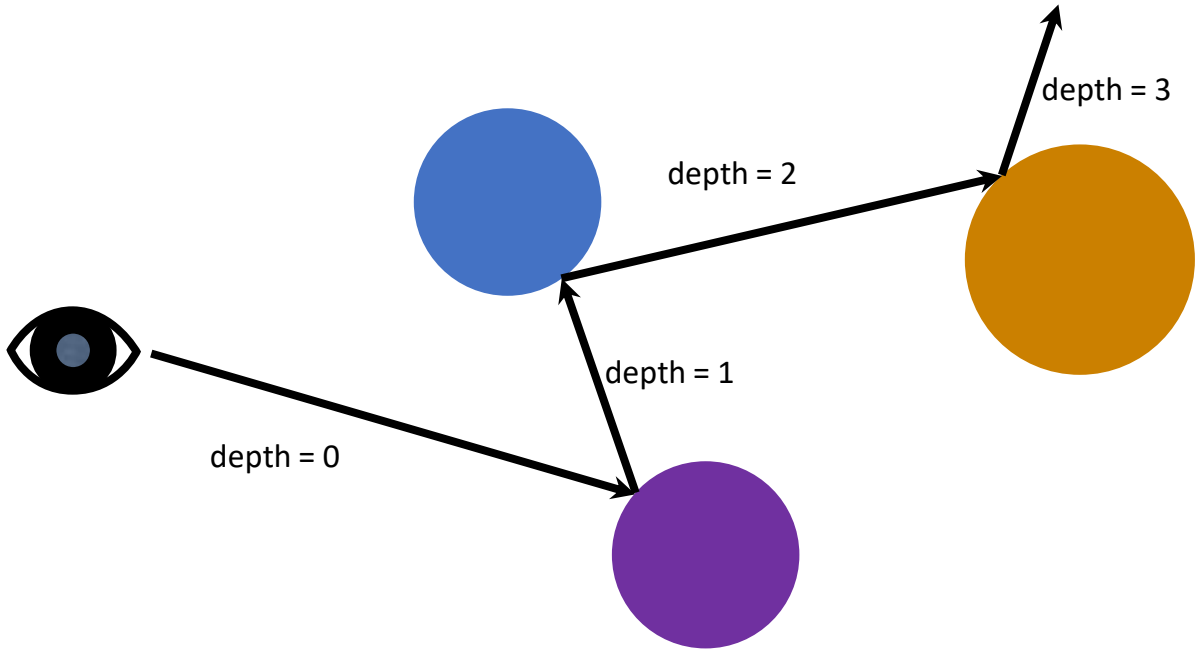
- If not in shadow, return shaded
- Use hit.getShadowRay()
  - hit is the hit point (`sw::Intersection`) returned by `scene.intersect()`
- Only 1 light source in `swTracer`
- New rays, starting from a surface need a small epsilon
  - 0.01 is used in `swTracer`
  - This is needed so the ray doesn't intersect with the surface

Shadows



Reflection

- Reflection rays are traced recursively
- Set a recursion depth, start with 2 or 3



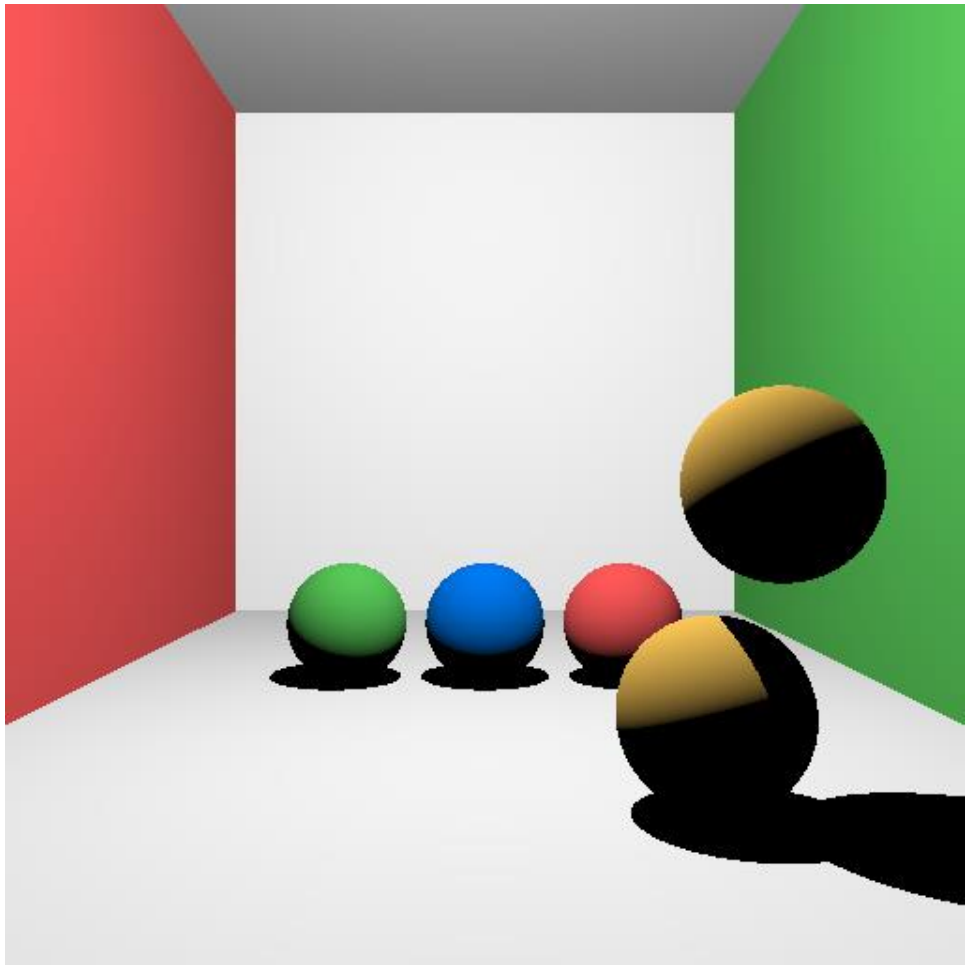
Reflection in main.cpp

- Add reflective spheres to scene (uncomment code)
- Recursively call traceRay()
- Decrease depth each time traceRay() is called
  - Only trace recursively if depth > 0
- Use hit.getReflectedRay() to get the reflected ray
- Also check reflectivity > 0
- Linearly interpolate reflected color with diffuse color

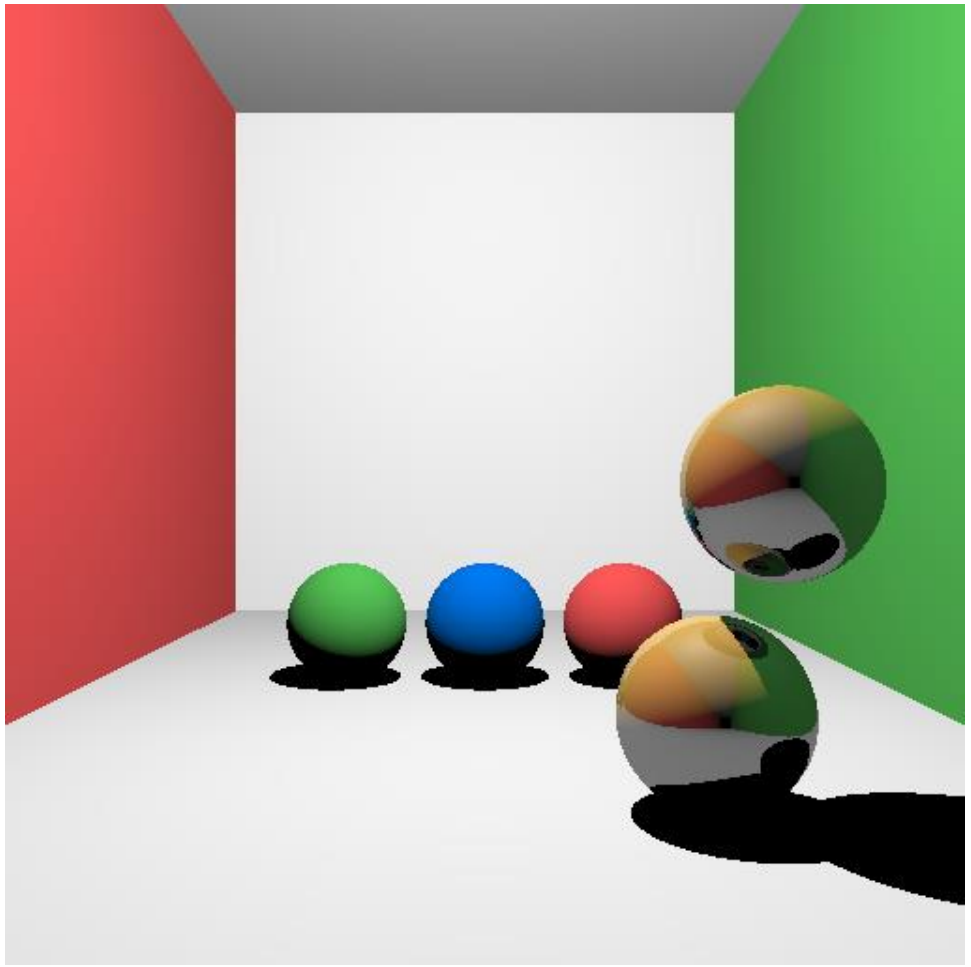
Reflection in swIntersection.cpp

- In `sw::Intersection::getReflectedRay()`
  - Compute the reflected vector using N and ray direction
  - Remember GLSL `refect(i,n)` = **i** - 2(**n**·**i**)**n**

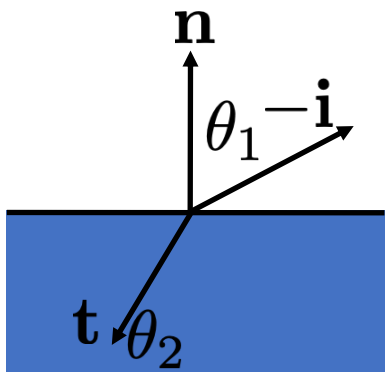
Reflection - without



Reflection - with



Refraction in swIntersection.cpp



$$\mathbf{t} = \eta\mathbf{i} + (\eta r - \sqrt{c})\mathbf{n}$$

$$r = -\mathbf{i} \cdot \mathbf{n}$$

$$c = 1 - \eta^2(1 - r^2)$$

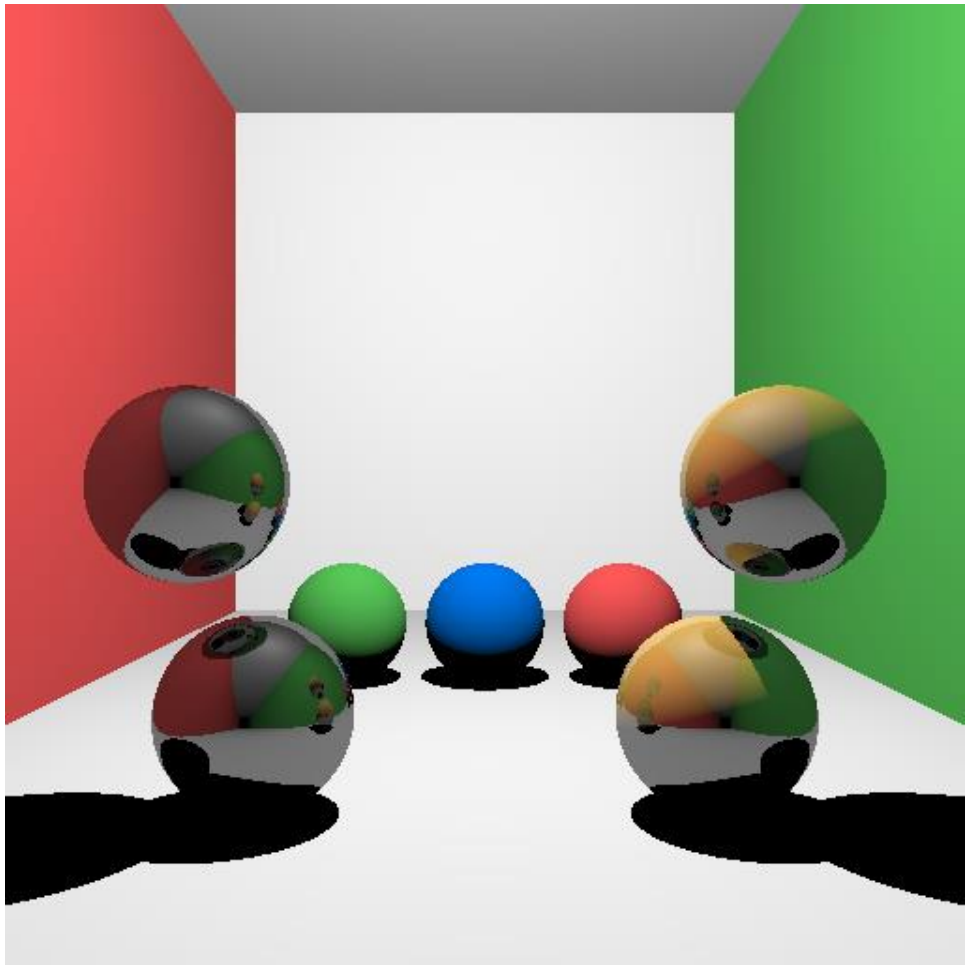
η **Eta, ratio of refractive indexes**

- Use in `sw::Intersection::getRefractedRay()`
  - Compute the outgoing refracted vector
  - Check if c < 0
    - Use reflection direction instead

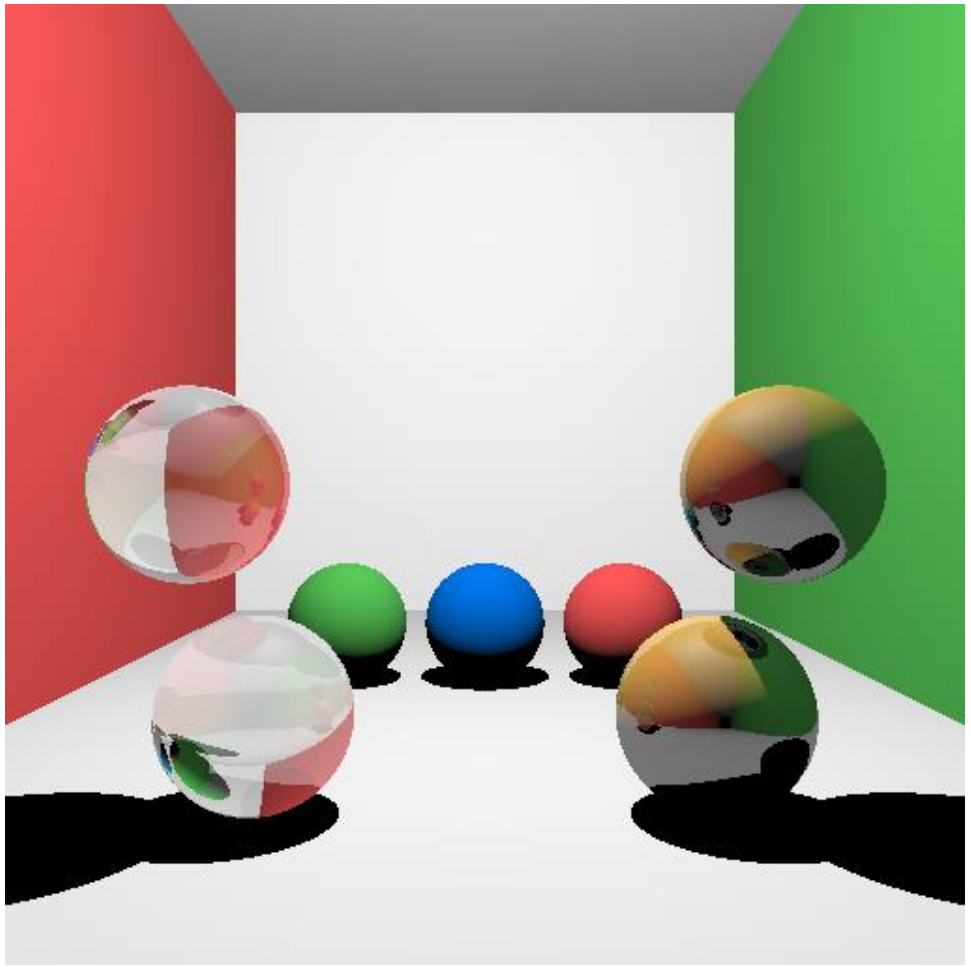
Refraction in main.cpp

- Add refractive spheres to scene (uncomment code)
- Check material transparency > 0 and depth > 0
- Trace a ray using `hit.getRefractedRay()`
- Linearly interpolate transparency color with diffuse color using transparency coefficient

Refraction - without

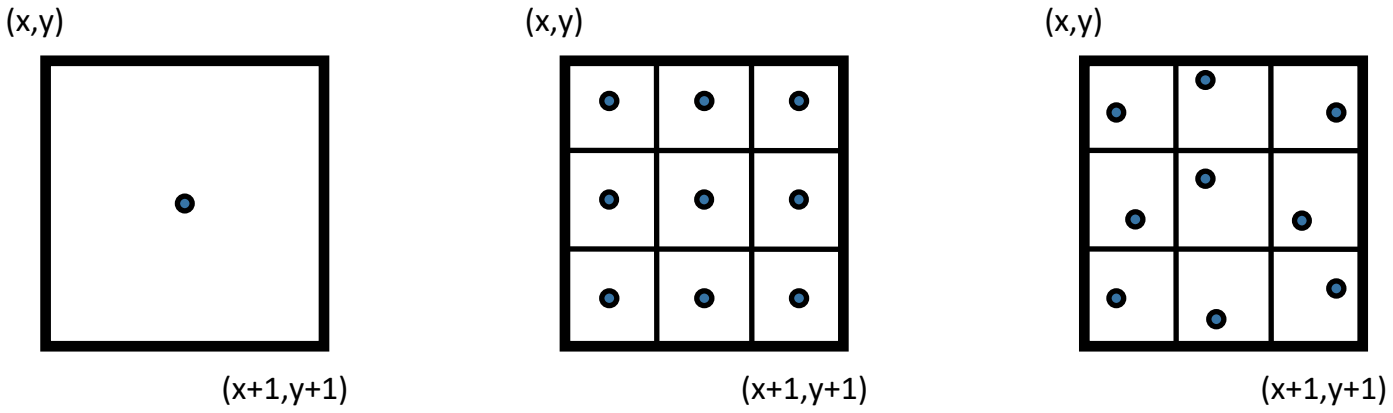


Refraction - with



Supersampling

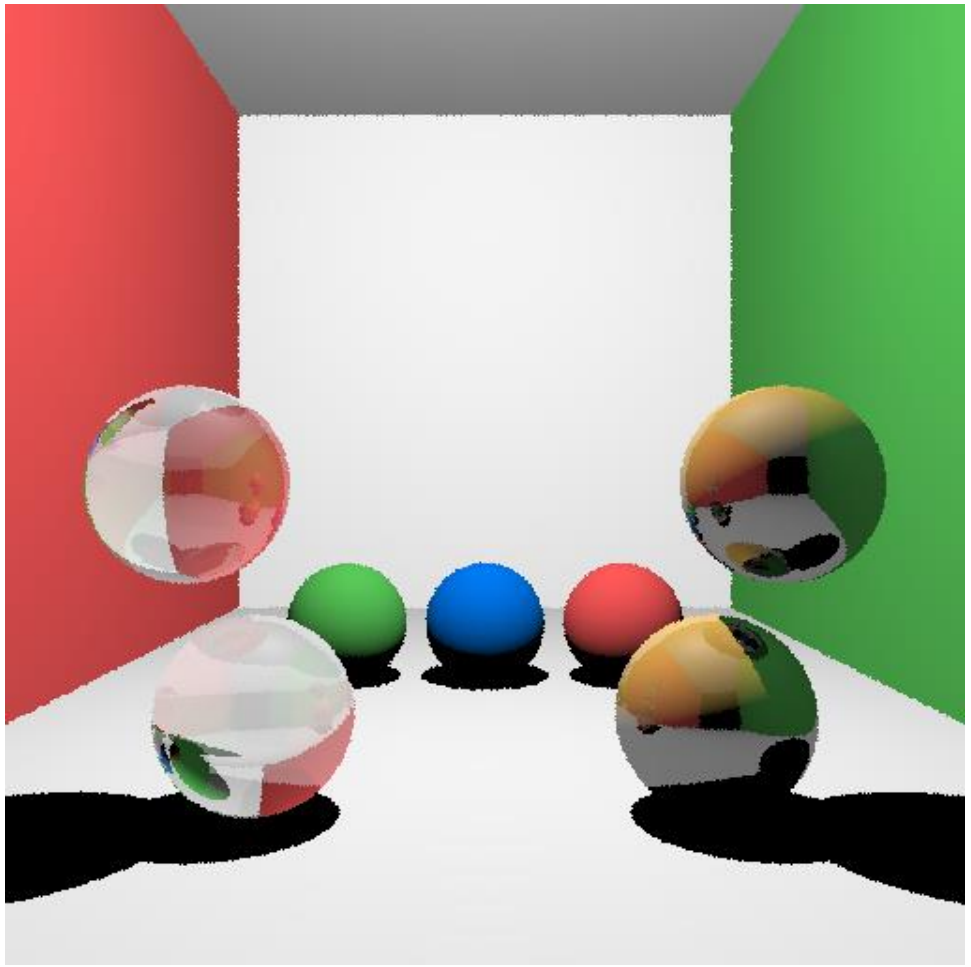
- Increase sample count per pixel
- Use NxN stratified/jittered samples



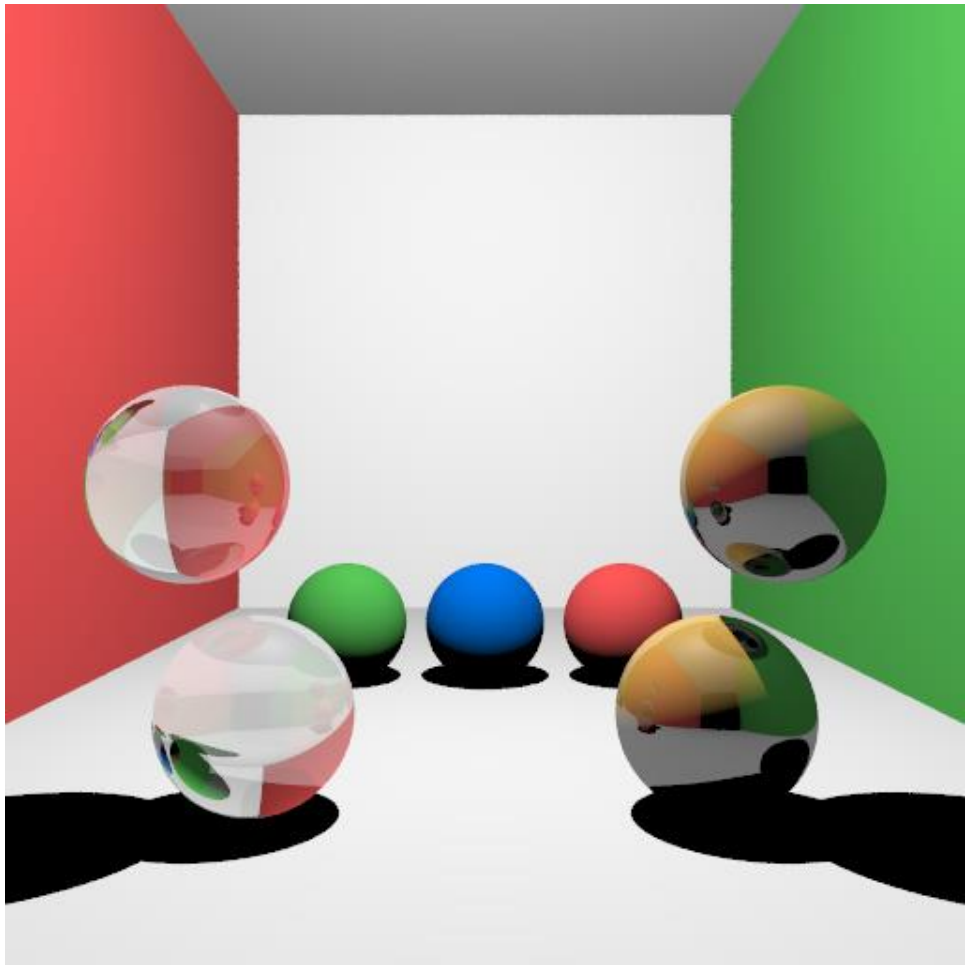
Supersampling

- Change pixel tracing in `main.cpp`
- Add X,Y loops around pixel coordinate creation
- Use uniform() to get value between [0,1)
  - Add to sub-pixel grid position
- Sum all sub-pixel rays together
- Divide by total number of sub-pixel rays

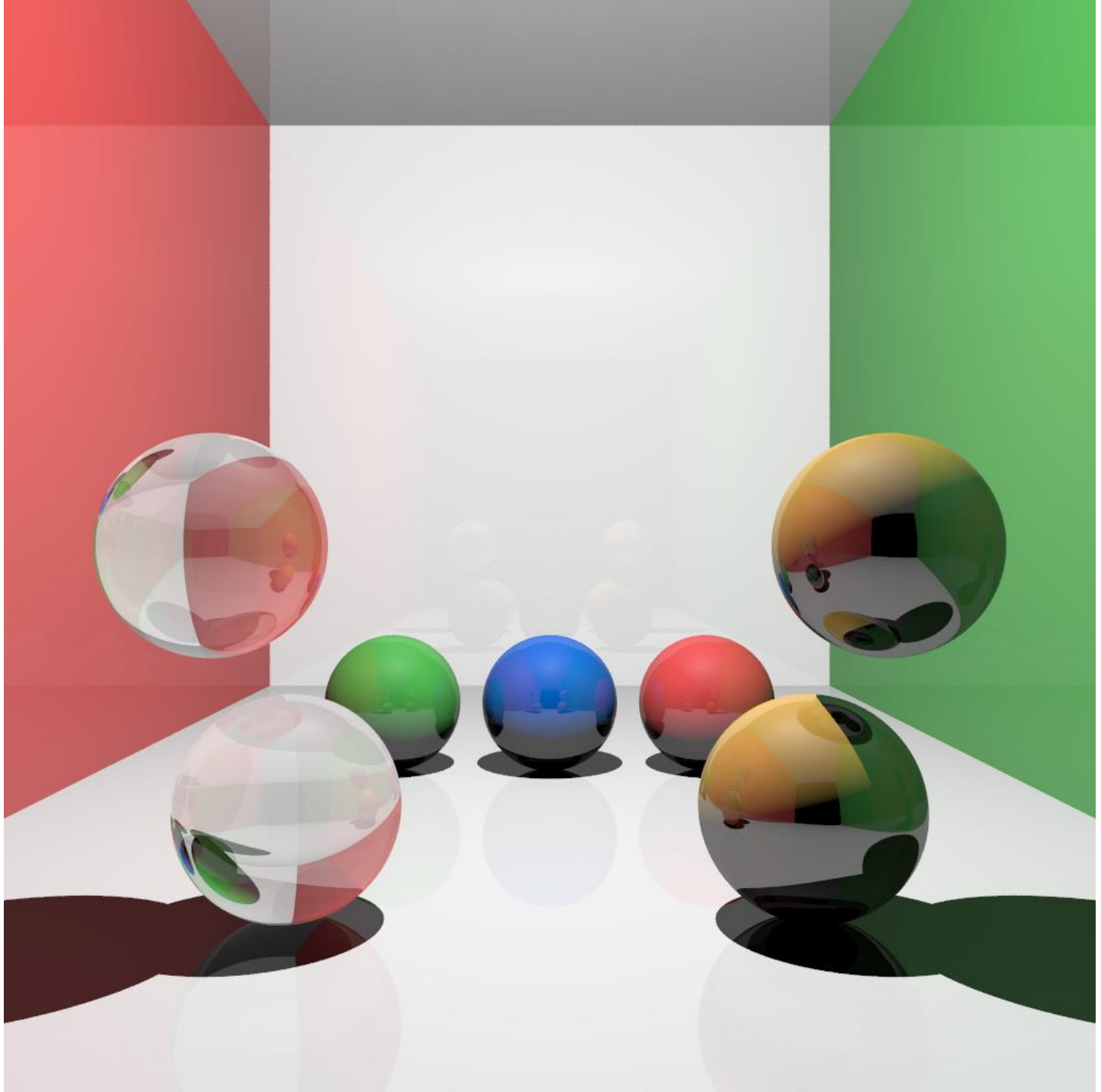
Supersampling – 1 SPP



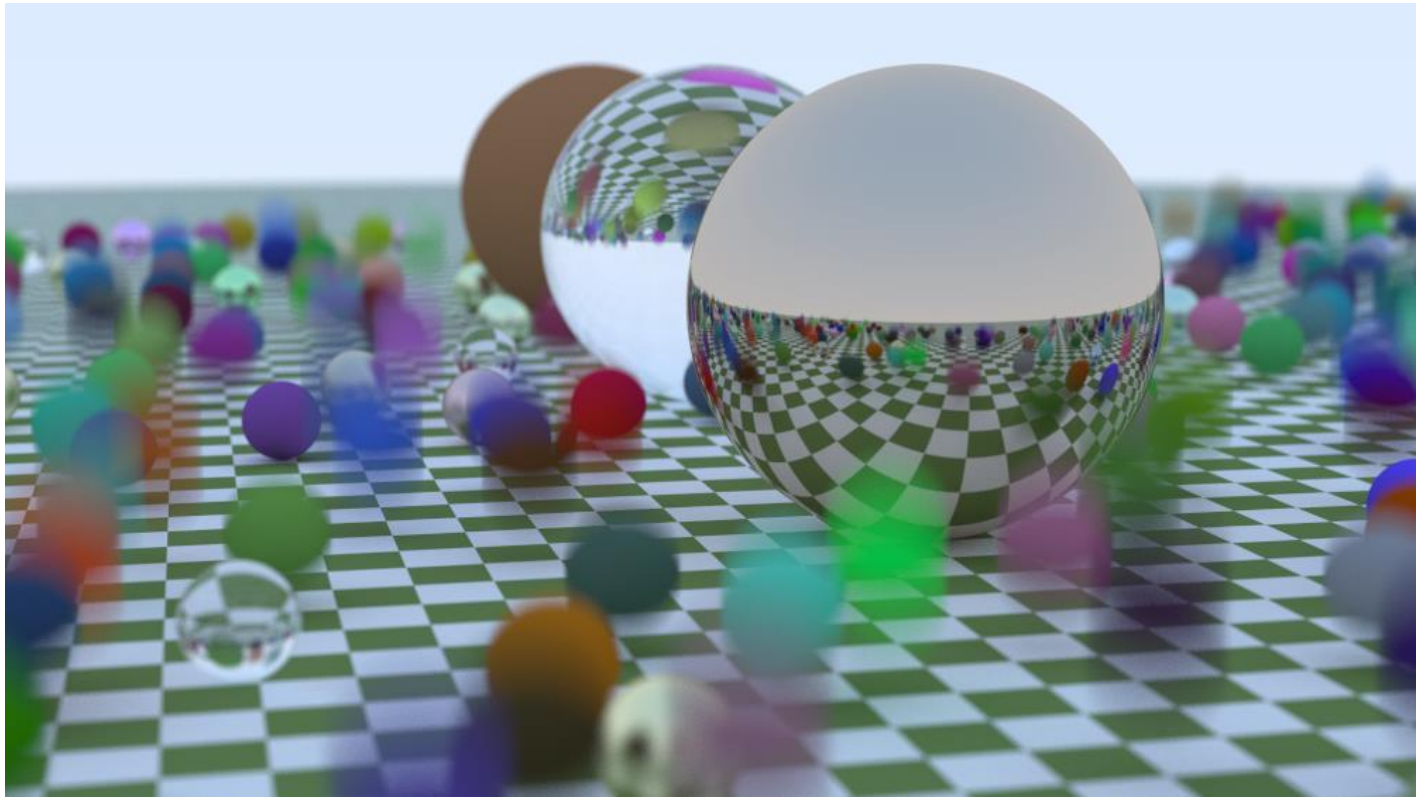
Supersampling – 9 SPP

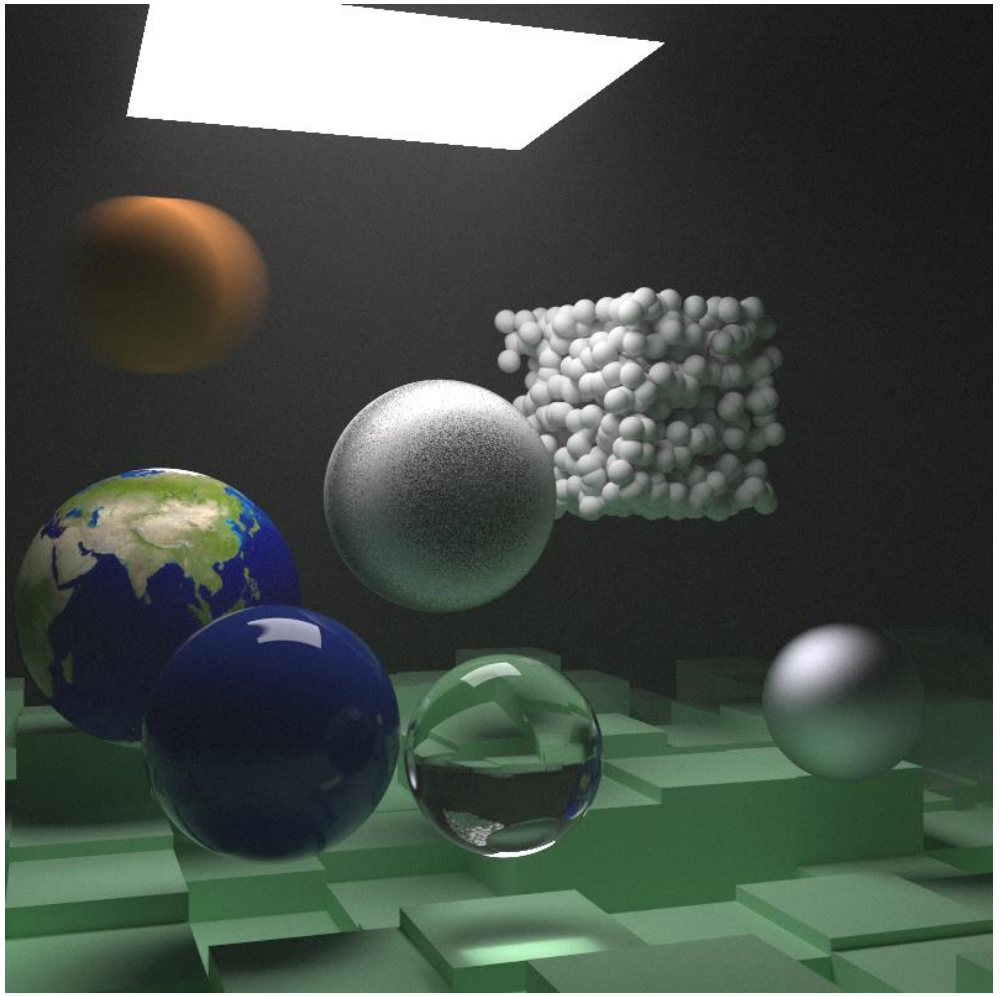


1024^2
9 SPP



Ray Tracing in One Weekend





https://raytracing.github.io/

Ray Tracing SW

- Ray Tracing in One Weekend https://raytracing.github.io
- Classic
  - POV-Ray http://www.povray.org
- Path Tracers
  - PBRT https://www.pbr-book.org
  - Mitsuba 2 https://www.mitsuba-renderer.org

- Start now!
- On webpage
  - Link to GitHub repo with code
  - Lab instructions