Ray Tracing – Part 1

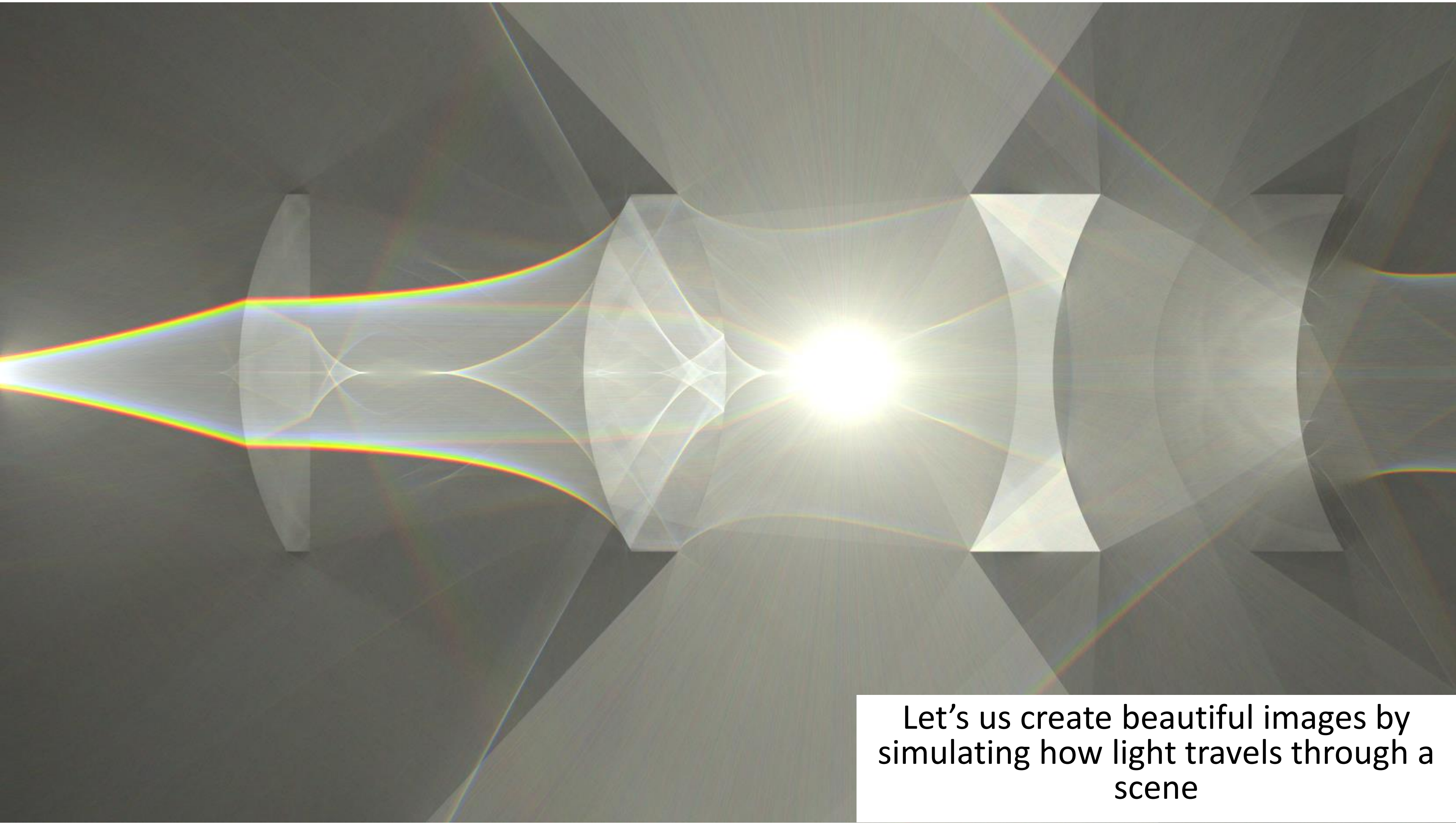
CSC418/2504 – Introduction to Computer Graphics

TA: Muhammed Anwar & Kevin Gibson
Email: manwar@cs.toronto.edu

Overview

- Introduction / Motivation
- Rasterization vs Ray Tracing
- Basic Pseudocode
  - Camera and Ray Casting
  - Calculating Intersections
  - Shading (Lighting, Shadows, Reflections)
- Assignment Examples

Ray Tracing



Let’s us create beautiful images by simulating how light travels through a scene

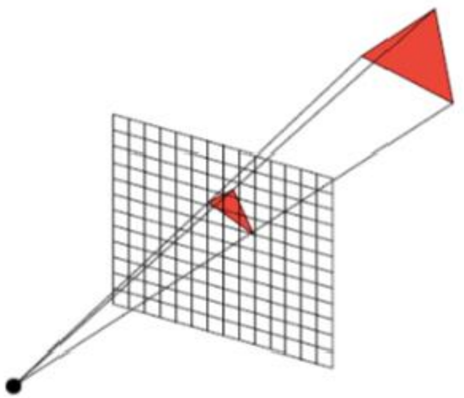
Image generated at https://benedikt-bitterli.me/tantalum/tantalum.html

Drawing a Scene

Projective methods:

Object-order rendering, i.e.

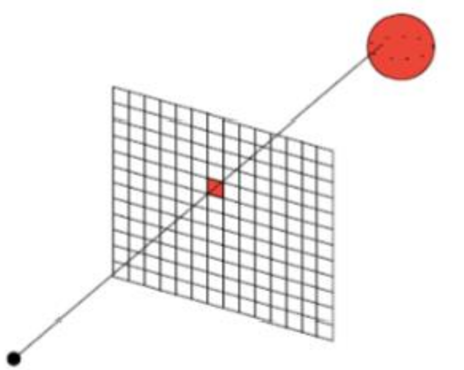
- For each object . . .
- . . . find and update all pixels that it influences and draw them accordingly



Ray tracing:

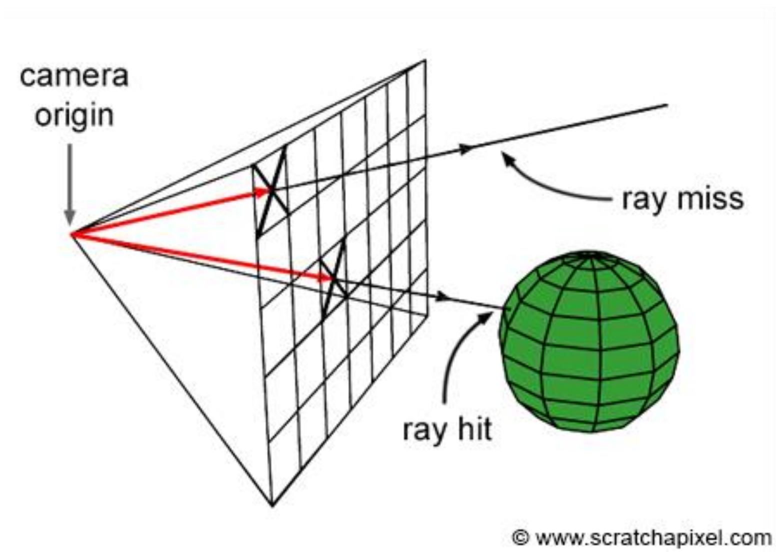
Image-order rendering, i.e.

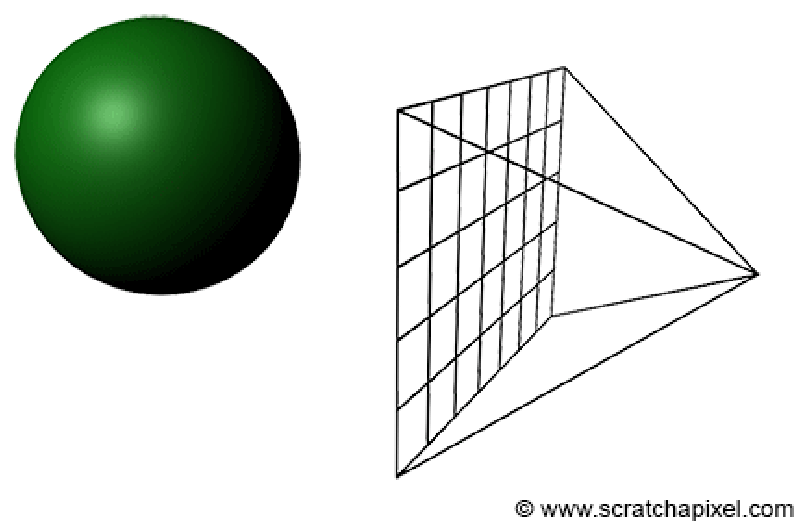
- For each pixel . . .
- . . . find all objects that influence it and update it accordingly



Ray Tracing - Overview

```
for each pixel in image:
- compute and construct viewing ray
-  find the 1st hit object by the ray and compute an intersection
- set pixel colour from the intersection
```

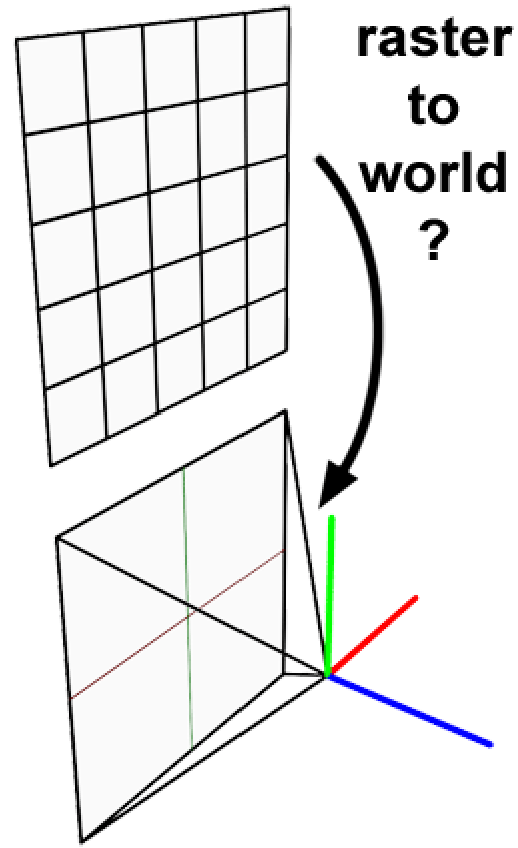




Shooting a Ray

- Suppose we have a camera at the origin, looking down the z-axis.
- We set an image plane a distance “f” away, and limit our view to a certain width and height.
- We slice up this image plane into pixel-sized squares
- “Shoot” a ray from the origin through each square, collect color from whatever it hits.

$$u = l + (r - l)(i + 0.5)/n_x$$
$$v = b + (t - b)(j + 0.5)/n_y$$



© www.scratchapixel.com

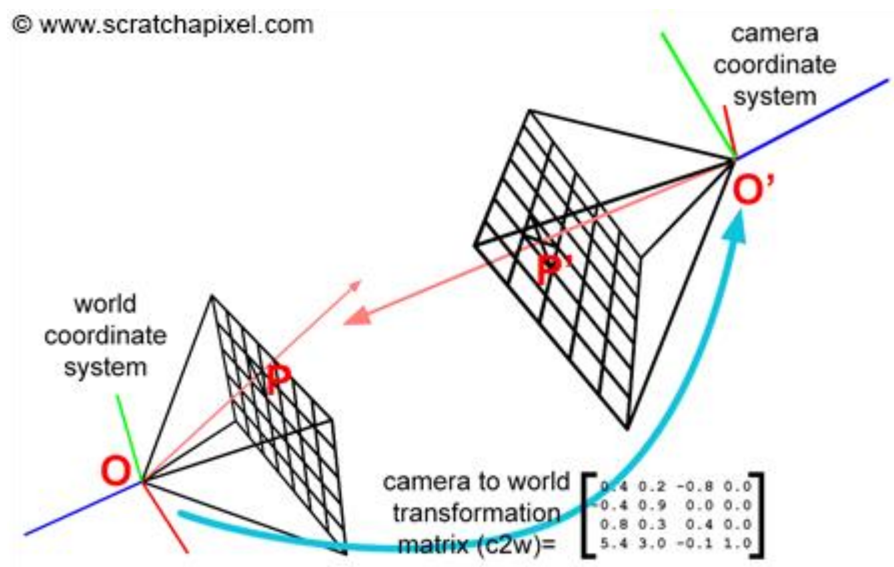
Shooting a Ray

In camera coordinates, if pixel coordinates are $\vec{d} = (u, v, f)$, then the ray we shoot is a line of the form

$$\vec{r}(t) = \vec{0} + t\vec{d}$$

For simplicity later on, we usually convert this to world coordinates by multiplying by our constructed basis.

$$M_{cw} = \begin{bmatrix} u_x & v_x & -view_x & eye_x \\ u_y & v_y & -view_y & eye_y \\ u_z & v_z & -view_z & eye_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Ray Casting Pseudo-Code

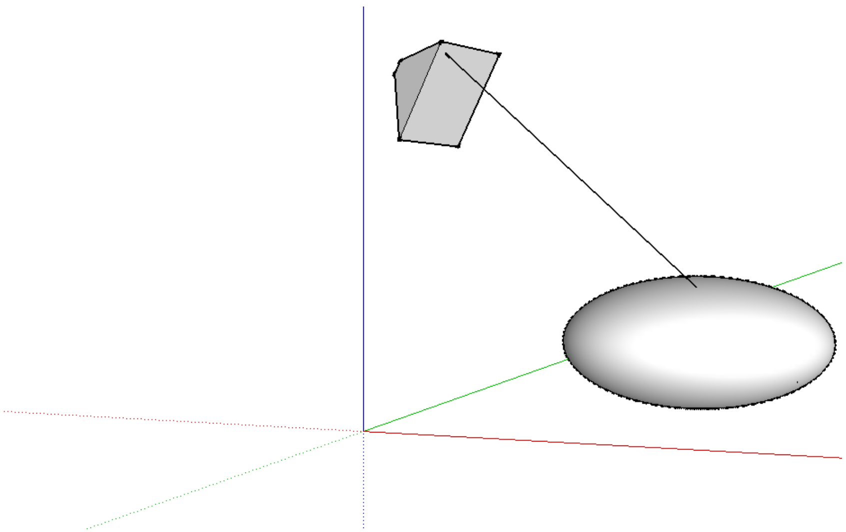
```
for (int i = 0; i < height; i++) {
    for (int j = 0; j < width; j++) {
        // Pixel in Camera Coordinates
        Point3D origin(0, 0, 0);
        Point3D imagePlane;
        imagePlane[0] = (-width/2 + i + 0.5)
        imagePlane[1] = (-height/2 + j + 0.5)
        imagePlane[2] = -1;

        //Ray Direction
        Vector3D direction = imagePlane - origin;

        //Convert to world-space
        direction = viewToWorld * direction;
        origin = viewToWorld * origin;
        Ray3D ray = Ray3D(origin, direction);

        pixelColour = cast_ray(ray)
    }
}
```

I Shot a Ray, Now What?

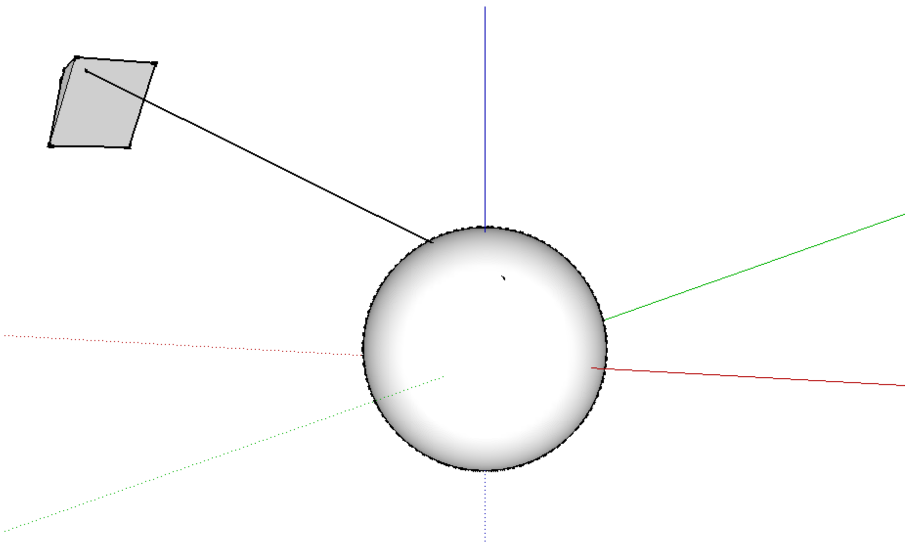
- Intersections!
- A ray shot into the scene can intersect with objects. An intersection is made up of the following elements:

```
struct Intersection {
    // 3D Location of intersection.
    Point3D point;
    // Normal vector at the intersection.
    Vector3D normal;
    // Material at the intersection.
    Material* mat;
    // The t value in r(t) = O + td along the ray where
    intersection happened
    double t_value;
    // Set to true when no intersection has occured.
    bool none;
};
```

Calculating Intersections



Intersection in World Coordinates



Intersection in Model Coordinates

- Difficult to do intersections with arbitrary objects in world space.
- Do intersections in Model Space (inverse of World to Model), and transform the result back to world space.

$$M\vec{r}(t) = M\left(\vec{0} + t\vec{d}\right) = M\vec{0} + tM\vec{d} \coloneqq \hat{O} + t\hat{d}$$ ← the ray in object coordinates

- Focus on unit primitives, such as unit planes, spheres, cylinders etc.

Ray – Plane Intersection

- Intersection of Ray with x-y Plane (z=0) centered at 0.
- Can find t of the ray at the intersection, if one exists.
- 2 Cases:
  - What if t<0?
    - Behind camera
  - What if $d_z = 0$?
    - Ray doesn't intersect plane (parallel to it)

Equation of ray:

$$\hat{r}(t) = \hat{O} + t\hat{d}$$

Equation of Plane

$$\vec{n} \cdot (\vec{p} - \vec{p}_0) = 0$$

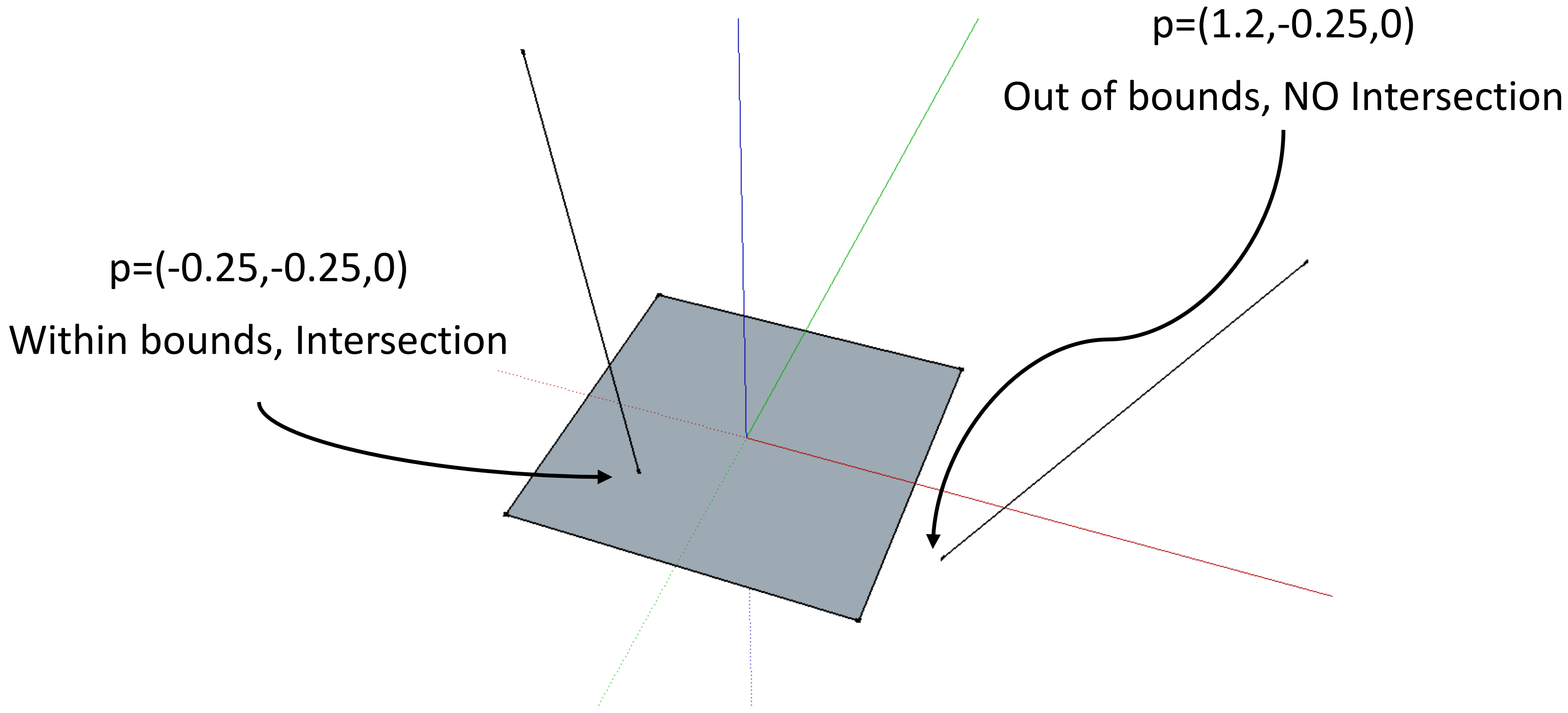
Notice: $\hat{p}_0$ is 0 and $n = [0,0,1,0]$

Intersection Point:

$$t = -\frac{O_z}{d_z}$$

Ray – Unit Plane Intersection

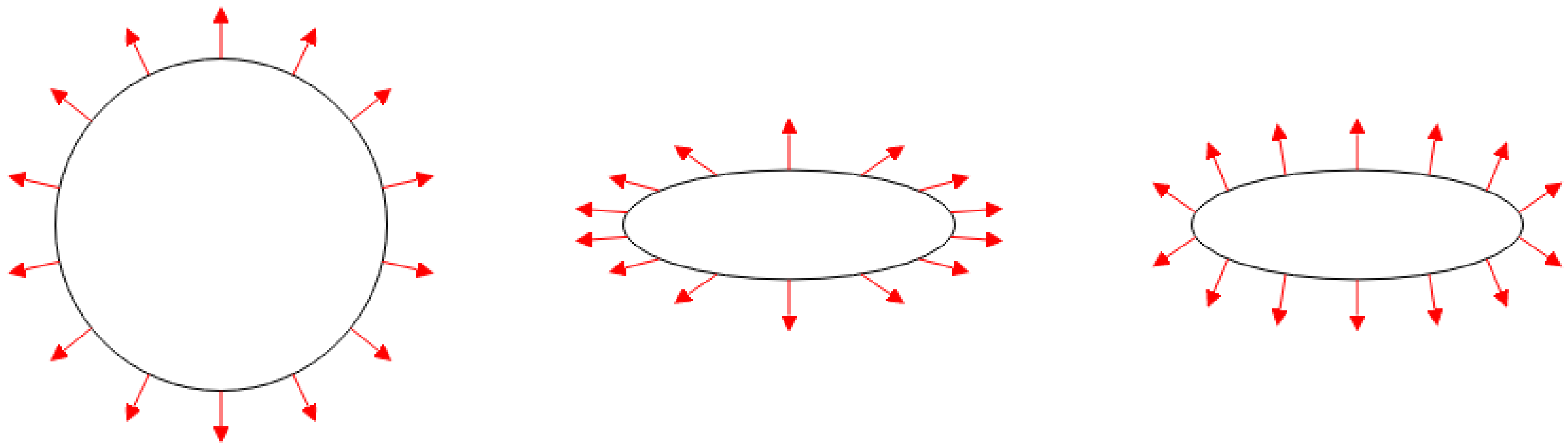
- The previous result will return an intersection along an infinite x-y plane.
- What if we want to bound it to a unit plane (i.e $x \in [-0.5, 0.5]$ $y \in [-0.5,0.5]$
- Simple. Find the 3D intersection point, and make sure its within the range. If not, no intersection occurred.



Ray – Unit Plane Intersection Pseudocode

```
bool intersect(ray, worldToModel,modelToWorld) {

     //Transform the ray (origin, direction) to object space
     origin = worldToModel * ray.origin;
     direction = worldToModel * ray.dir;

     double t = -origin[2] / direction[2];

     //invalid intersection
     if (t < 0 || direction[2] ==0){
          return false;
     }

     Point3D p = origin + t * direction;
     Vector3D normal = Vector3D(0,0,1);

     if (p[0] >= -0.5 && p[0] <= 0.5 && p[1] >= -0.5 && p[1] <= 0.5) {
          ray.intersection.t_value = t;
          ray.intersection.point = modelToWorld * p;
          ray.intersection.normal = TransformNormal(worldToModel, normal)
          return true;
     }

     return false;
}
```

Wait, How do you Transform Normals?



Unit Circle   Incorrect Normals   Correct Normals

- Can't just apply the same transformation to convert normals from model to world space

$$n^T \times t = n^T \times M_l^{-1} M_l \times t$$

$$n^T \times t = n^T \times M_l^{-1} M_l \times t = (M_l^{-1T} \times n)^T (M_l \times t)$$

$$n^T \times t = (M_l^{-1T} \times n)^T \times t'$$

$$n' = M_l^{-1T} \times n$$

$\times$ here is the dot-product, not cross-product.

- Therefore, to transform the normal correctly, we need to the "Inverse Transpose" of the model to world matrix.

Ray Shading

- Okay, so we know how to cast a ray, calculate intersections. Now we need to color our pixels!

$I(q)$	$=$	$L(n,v,l)$	$+$	$G(p)k_s$
Intensity at q	=	phong local illum.	+	global specular illum.

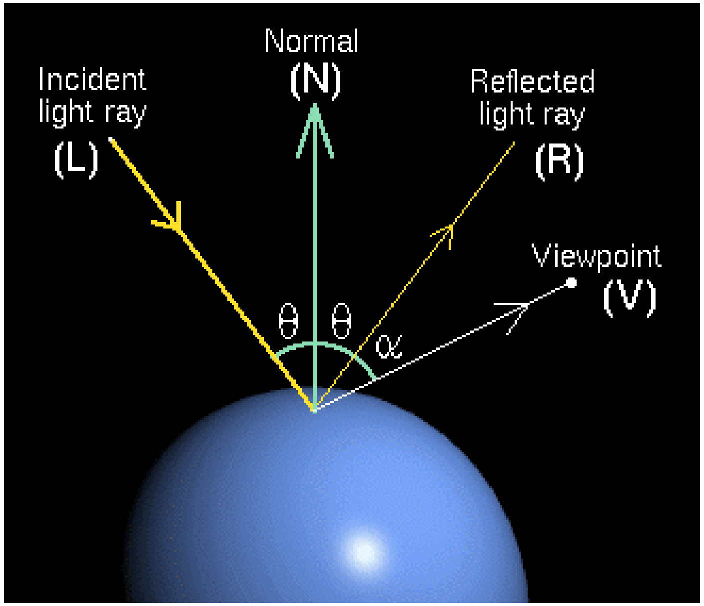
- We have the information we need to calculate Phong Illumination at the intersection point.

Phong Illumination Model

$$k_a I_a + k_d \max(0, \vec{N} \cdot \vec{L})\, I_d + k_s \max\left(0, (\vec{V} \cdot \vec{R})^{\alpha}\right) I_s$$

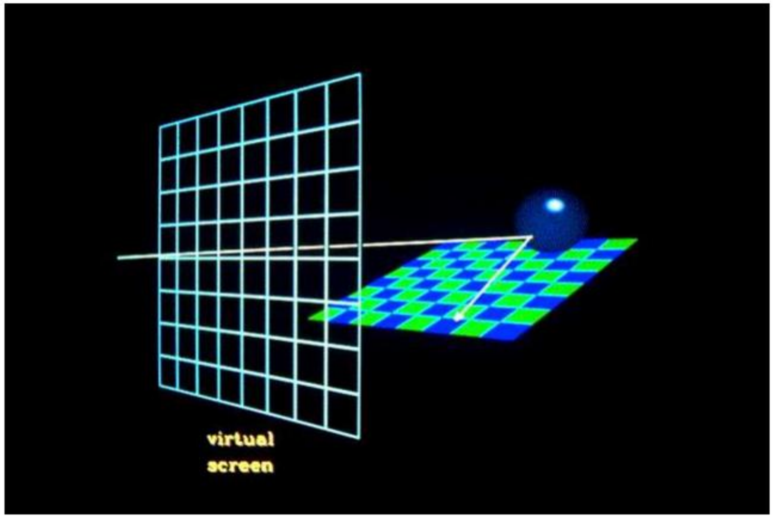
I_a, I_d, I_s are properties of the light
$\vec{R} = 2(\vec{L} \cdot \vec{N})\vec{N} - \vec{L}$

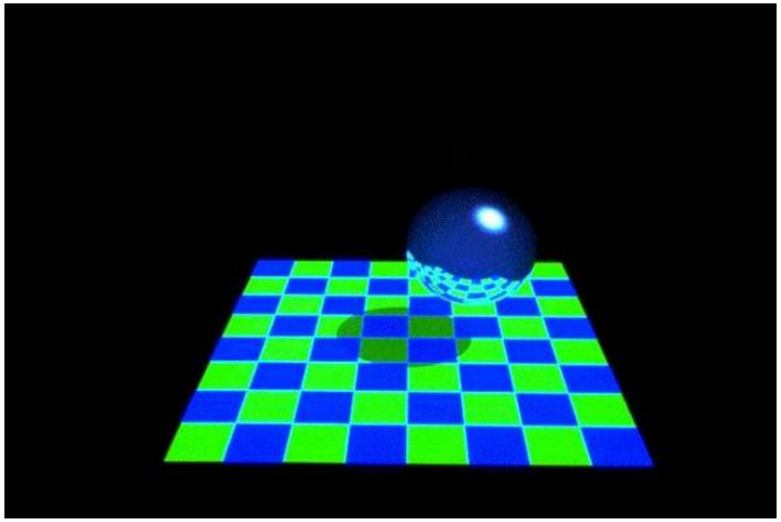
Note: Normalize these vectors before doing calculations
Note: The viewing direction here is the direction of the ray (specifically the –ve of the ray direction).



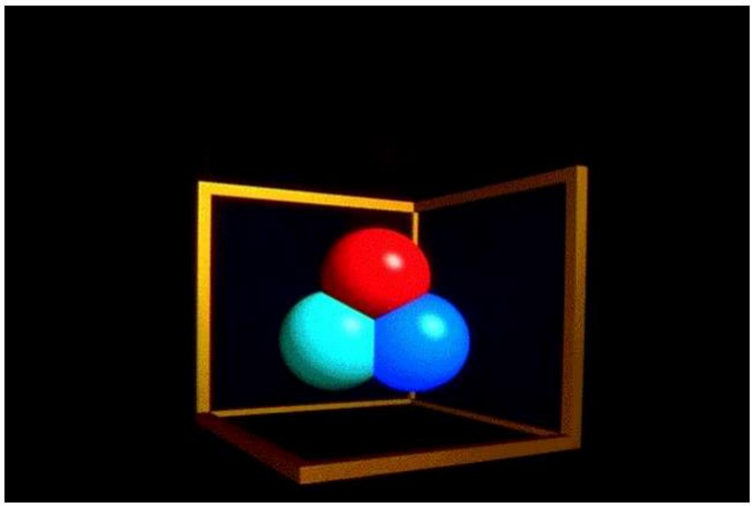
Global Specular Illumination

- Up until now, our ray tracer would have produced the same images as a rasterizing renderer (OpenGL)
- With ray tracing, we can easily extend it to include more global illumination properties (e.g Reflections). Just cast another ray from the intersection.

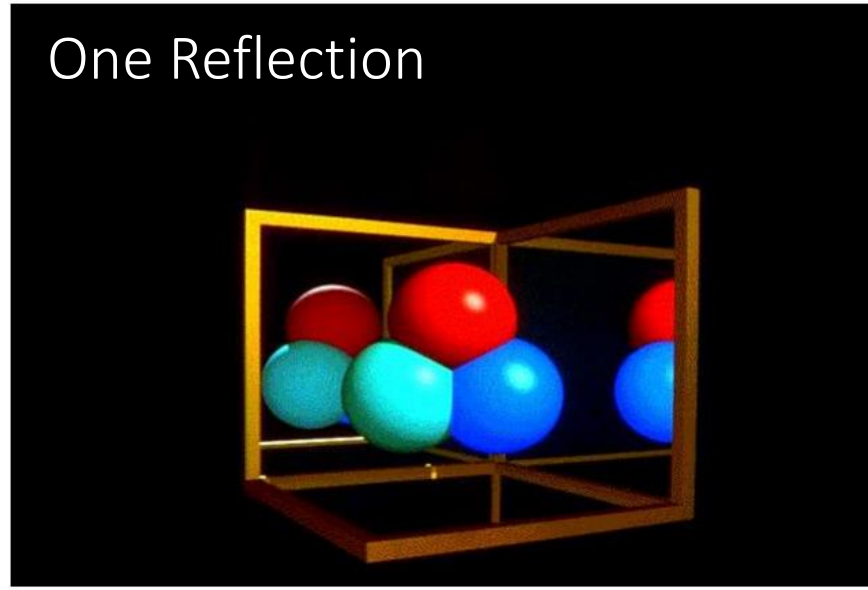


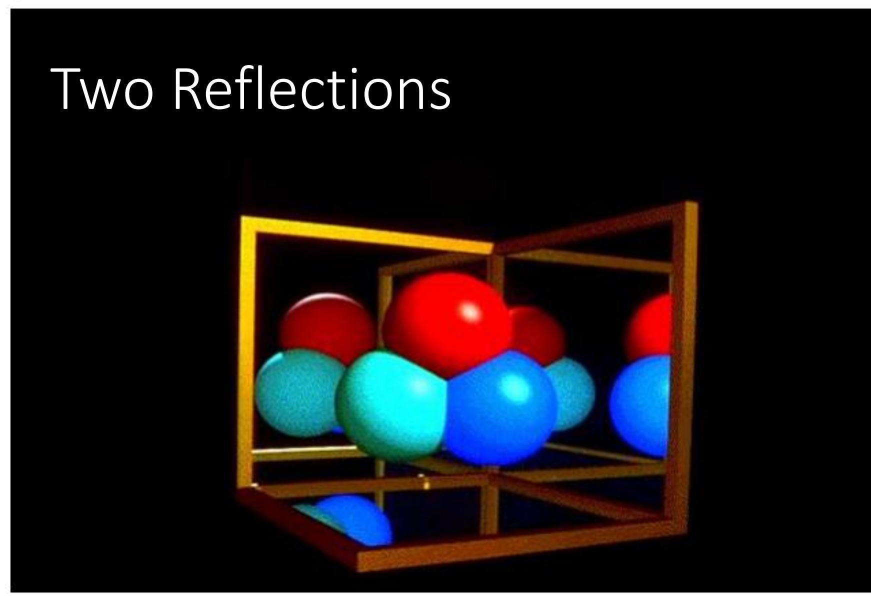


Reflections



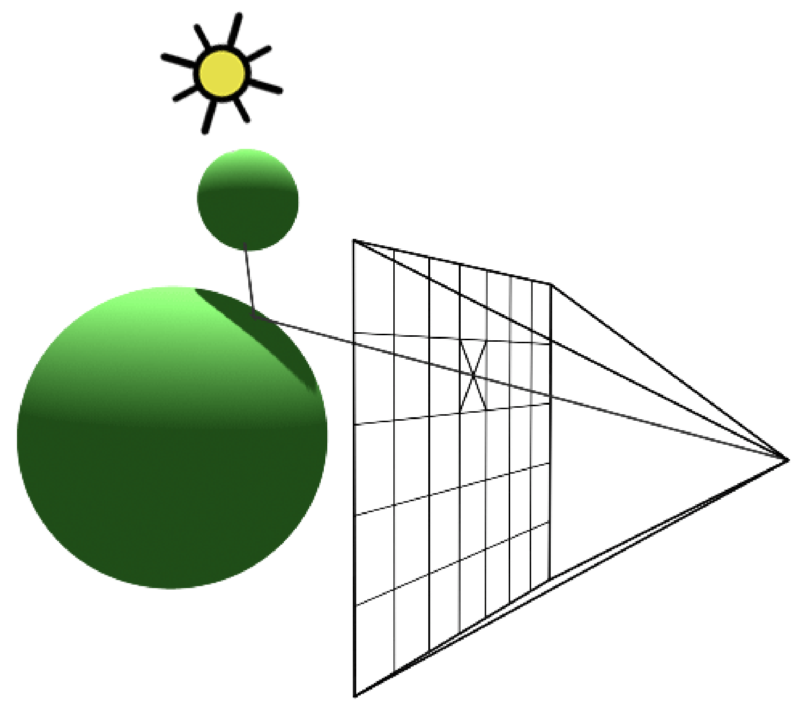
No Reflection





Shadows

- Even shadows are easy in ray tracers.
- How do we know if a light reaches a point?
- Use a ray!
- If a ray from our intersection point reaches the light before intersecting any other objects, then add the contribution of the light to the color
- Otherwise, the object is shadowed by an object



Pseudocode of Ray Shading

```
Color shadeRay(ray, depth) {
    findIntersections(ray);

    Colour color(0, 0, 0);
    if (!ray.intersection.none) {

        for (light in scene) {
            shadowRay = create ray from intersection to light;
            findIntersections(shadowRay);
            if (shadowRay.intersection.none) {
                color += phongIllumination(light, ray);
            }
        }

        if (depth > 0) {
            reflectRay = create a reflected ray at the intersection;
            color += shadeRay(reflectRay, --depth);
        }
    }
    return color;
}
```

Assignment 3/4

- Should have enough knowledge now to get a good start on A3.
- Tips:
  - Test render small resolutions first, cause raytracing takes a while!
  - Have a solid understanding of the math behind intersections, debugging them can be cumbersome at times
  - Next week, will go over more advanced ray tracing

Past A3 Submissions