Computer Graphics

Discussion 8

Garett Ridge garett@cs.ucla.edu,
Sam Amin samamin@ucla.edu,
Theresa Tong theresa.r.tong@gmail.com,
Quanjie Geng szmun.gengqj@gmail.com

Raytracer Project

- WebGL; Runs on your browser!
- We provide an obfuscated version showing the expected results
- The official hw3 spec is in the .html file of your download on Piazza

Test Cases

- Load a text file into your data structures
- Formatted like this:

NEAR <n>
LEFT <l>
RIGHT <r>
BOTTOM <b>
TOP <t>
RES <x> <y>
SPHERE <name> <pos x> <pos y> <pos z> <scl x> <scl y> <scl z> <r> <g> <b> <K_a> <K_d> <K_s> <K_r> <n>
LIGHT <name> <pos x> <pos y> <pos z> <I_r> <I_g> <I_b>
BACK <r> <g > <b>
AMBIENT <I_r> <I_g> <I_b>
OUTPUT <name>

Instructions

- You will be implementing a Ray Tracer.
- Your system need only handle the rendering of spheres
- Certain things ensure your output matches ours
  - Camera situated at the origin
  - Right handed, negative Z
  - Local illumination, reflections, refractions, and shadows must be implemented
- Exact match is less important than physical correctness of the lighting
- Extra step: Culling objects inside the near plane
- If it has all the features (specular, diffuse, shadow, etc) and visually looks like the test results you'll be OK

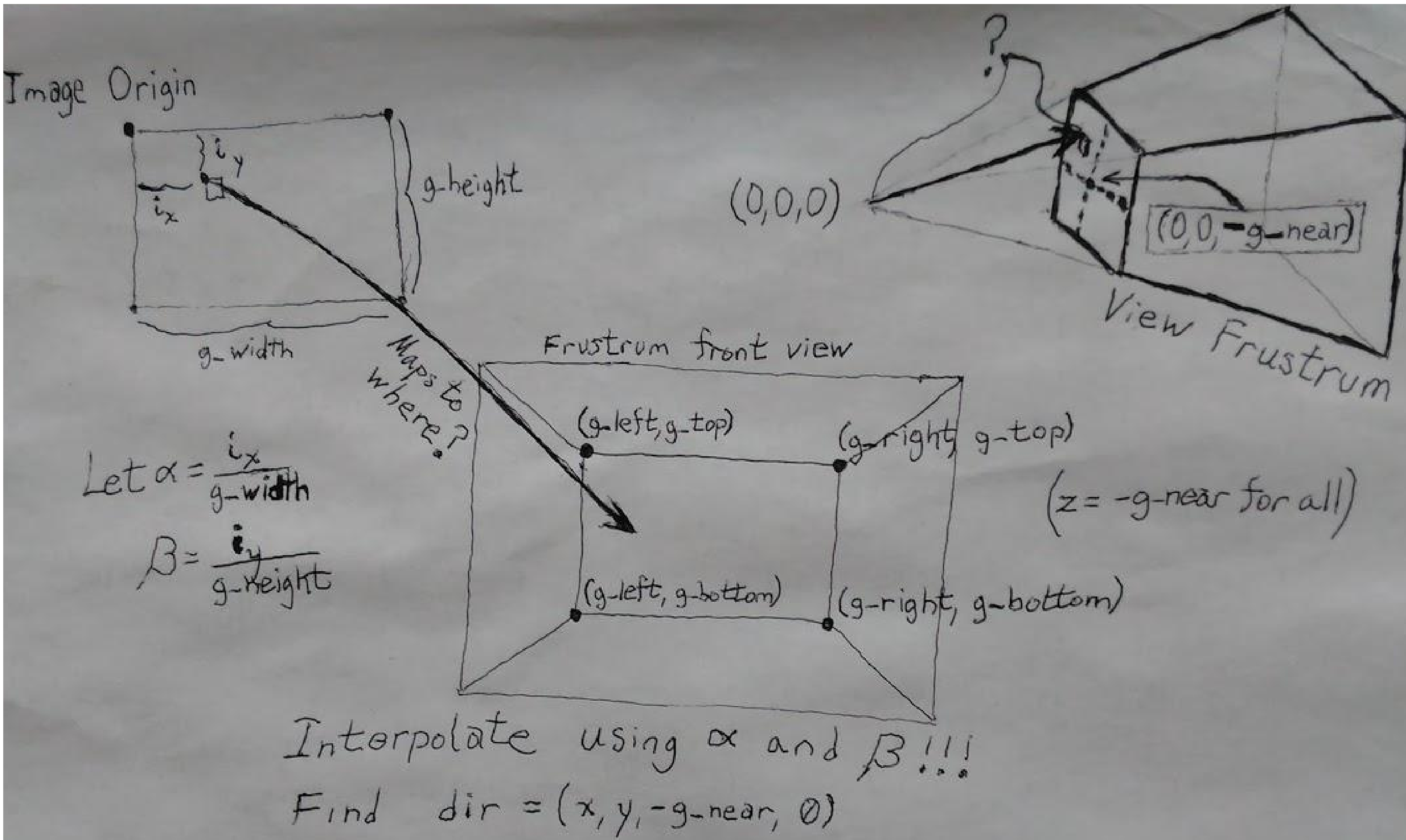
Phong-Blinn Shading model

- Phong reflection model applied to Project 2 in the shaders; can be applied again for hw3
- Describes ambient, diffuse, specular

- Plain Phong (no Phong-Blinn) is OK too -- just slightly different specular results from using $(R \text{ dot } V)^n$ versus the halfway vector $(H \text{ dot } N)^n$
- We can add in additional terms to Phong-Blinn since it's a raytracer – reflections and refractions

get_dir()

Note: Image Origin (0,0) should be shown as lower left, not upper. Variable names in your actual code are now like this.near, this.left, etc. in class Ray_Tracer.

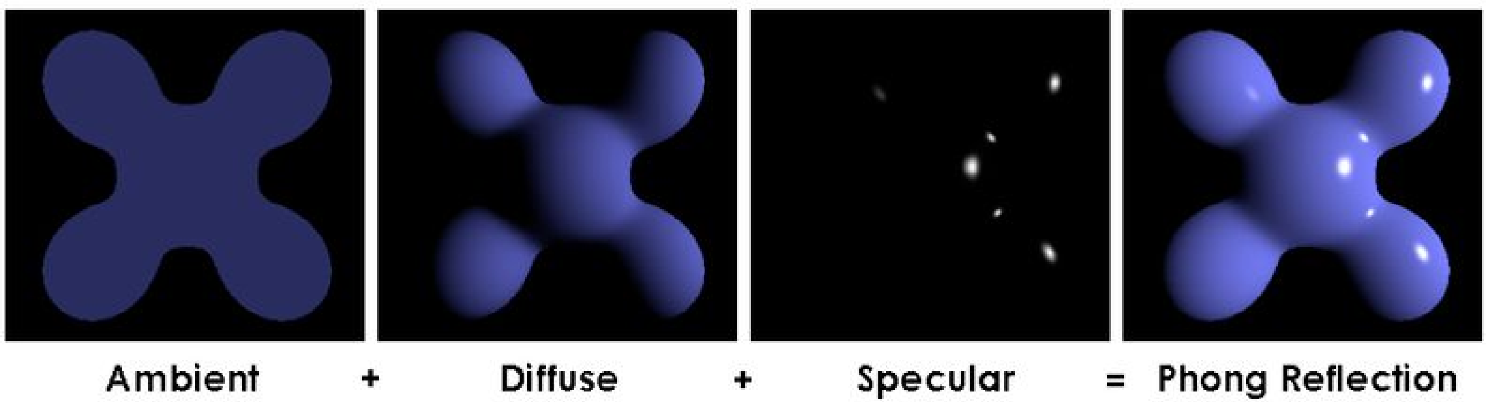


Grading Scheme

- No need to test for an exact image match, using diff or photoshop layers in subtraction mode
- If it has all the features (specular, diffuse, shadow, reflection, refraction) and visually looks like the test results you'll be OK

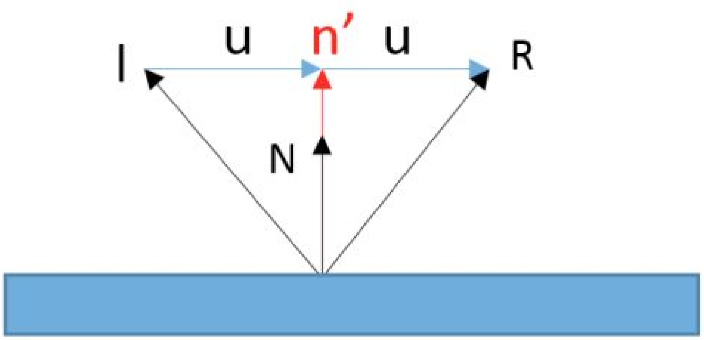
Phong-Blinn Model Review

Components of light



Light equation

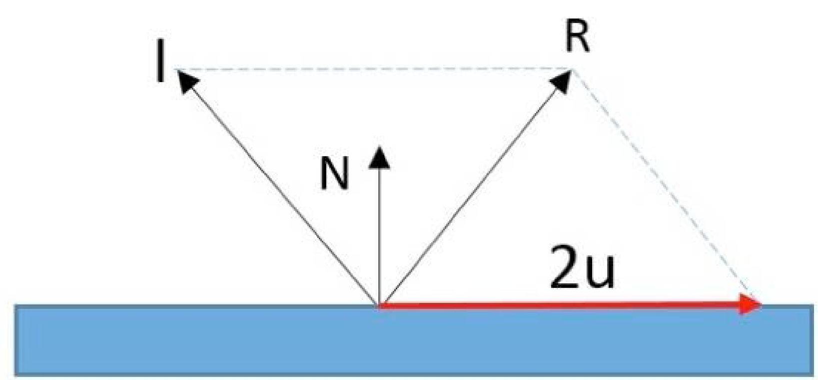
- Calculating R, the (non-physical, made-up) reflection of the point light source



The $\overrightarrow{n'}$ is the projection of $\vec{I}$ on $\vec{N}$

$$\overrightarrow{n'} = (\vec{N} \cdot \vec{I})\,\vec{N}, \text{ with } \|\vec{N}\|^2 = 1$$

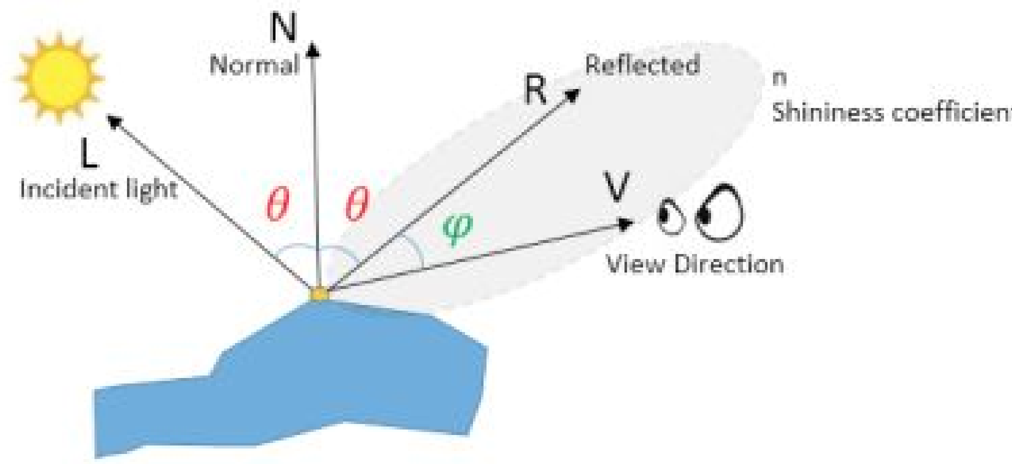
$$\vec{u} = \overrightarrow{n'} - \vec{I}$$



$$\vec{R} = \vec{I} + 2\vec{u} = \vec{I} + 2(\overrightarrow{n'} - \vec{I})$$

$$\vec{R} = 2(\vec{N} \cdot \vec{I})\,\vec{N} - \vec{I}$$

Light equation



I = emissive + ambient + diffuse + specular

emissive = k_e

ambient = $k_a * ambientColor$

diffuse = $k_d * lightColor * \cos(\theta)$

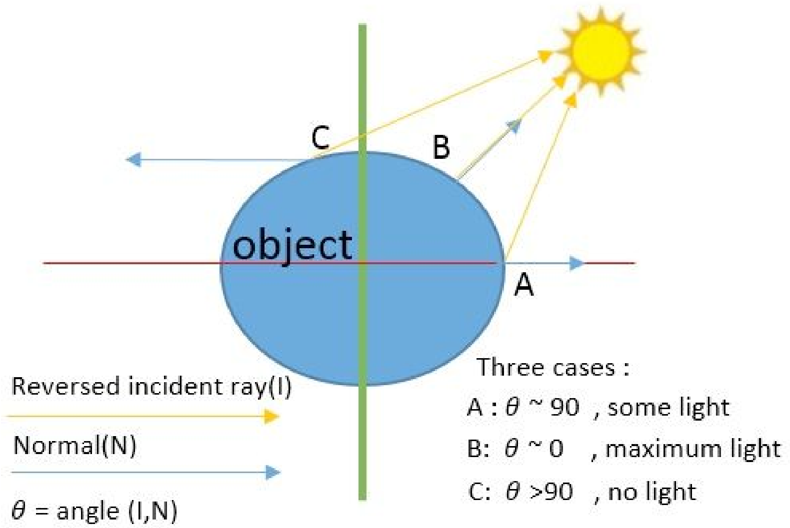
= $k_d * lightColor * \max(0, N \cdot L)$

specular = $k_s * lightColor * \cos(\varphi)^n$

= $k_s * lightColor * \max(0, R \cdot V)^n$

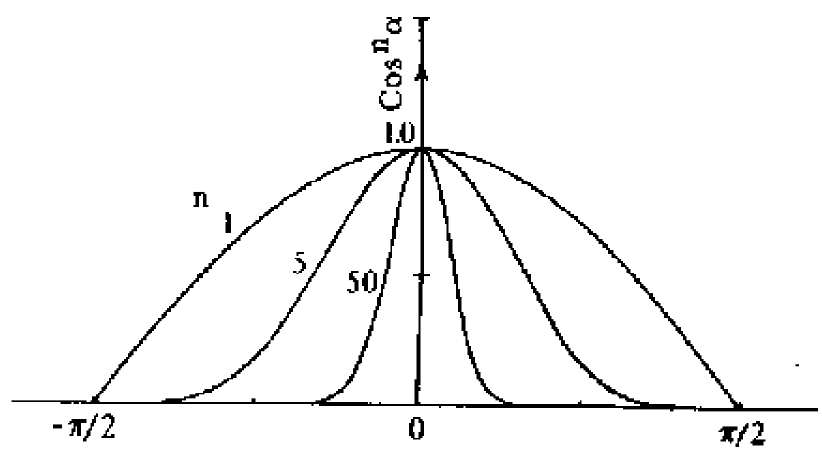
Lambert’s law

“The amount of reflected light is proportional with the cosine (dot product) of the angle between the normal and incident vector”



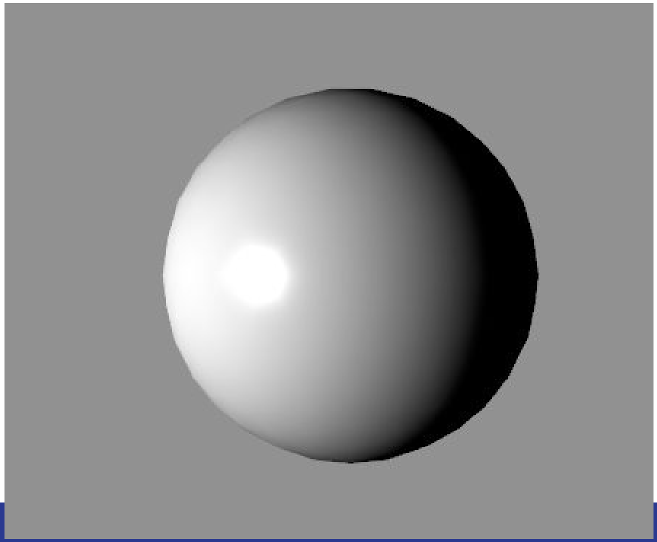
Specular term - Smoothness exponent effect

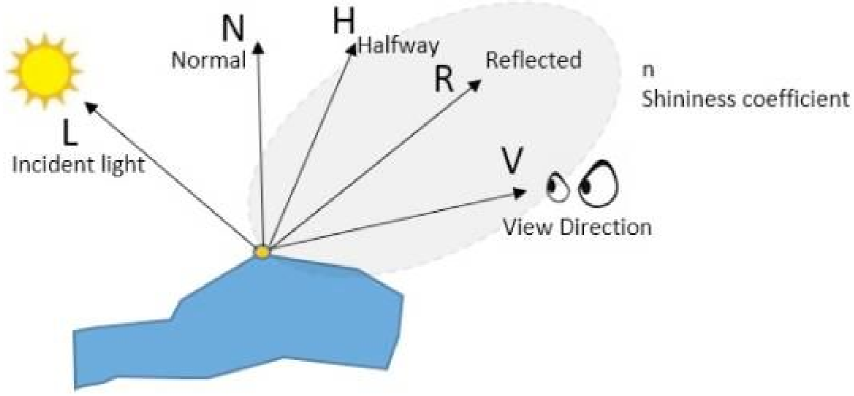
- Exponentiating a function that has values < 1 draws those values closer to zero
- Higher exponent = smaller region where point light's reflection is considered "aligned" with the viewer.
- Smaller shiny spot



Phong-Blinn

- Combine V and L, the two constants in the scene, into one vector
- Given H = halfway between V and L, Use (H dot N) instead of (R dot V)
- If directional light, you can compute H once per frame per light source and it's the same everywhere in the scene - no dependence on normal, just viewer and light
- Re-use it instead of re-calcuating in shader - shader only has to dot H with each N - cheap
- Also behaves better at glancing angles





$$H = \frac{L+V}{|L+V|}$$

End review

Phong Shading model

This is formula you'll use for Phong (modified to blend in reflections & refractions):

```
vec3 surface_color = k_a * sphere color + for each point light source (p) {
        this.lights[p].color * (
        k_d * (  N dot L,       positive only) * (the sphere's color) +
        k_s * ( (N dot H)^n, positive only) *  white       ) }
```

and then:

```
vec3 pixel_color = surface_color + (white - surface_color) *
        ( k_r * trace().slice(0,3) + k_refract * trace().slice(0,3) )
```

A few particular lecture slides are vital - you’ll consult the formulas on them a lot

From Lecture slides

Final Intersection

Inverse transformed ray

$$\mathbf{r}'(t) = \mathrm{M}^{-1} \begin{bmatrix} S_x \\ S_y \\ S_z \\ 1 \end{bmatrix} + t\mathrm{M}^{-1} \begin{bmatrix} c_x \\ c_y \\ c_z \\ 0 \end{bmatrix} = S' + t\mathbf{c}'$$

- Drop 1 and 0 to get $\mathbf{r}'(t)$ in 3D space

For each object

- Inverse transform ray, getting $S' + t\mathbf{c}'$
- Find t_h for intersection with the untransformed object
- Use t_h in the **untransformed ray** $S + t\mathbf{c}$ to find the point of intersection with the transformed object

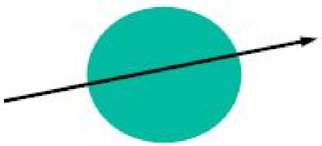
From Lecture slides

Ray-Object Intersections

Intersection of ray with unit sphere at origin:

$$\text{ray}(t) = S + t\mathbf{c}$$

$$\text{Sphere}(P) = |P| - 1 = 0$$



$$\text{Sphere}(\text{ray}(t)) = 0 \Rightarrow$$

$$|S + t\mathbf{c}| - 1 = 0 \Rightarrow$$

$$(S + t\mathbf{c}) \cdot (S + t\mathbf{c}) - 1 = 0 \Rightarrow$$

$$\boxed{|\mathbf{c}|^2 t^2 + 2(S \cdot t\mathbf{c}) + |S|^2 - 1 = 0}$$

This is a quadratic equation

Most useful lecture slides

Solving a Quadratic Equation

$$|\mathbf{c}|^2 t^2 + 2(S \cdot \mathbf{c})t + |S|^2 - 1 = 0$$
$$At^2 + 2Bt + C = 0$$

$$t_h = -\frac{B}{A} \pm \frac{\sqrt{B^2 - AC}}{A}$$
$$= -\frac{S \cdot \mathbf{c}}{|\mathbf{c}|^2} \pm \frac{\sqrt{(S \cdot \mathbf{c})^2 - |\mathbf{c}|^2 (|S|^2 - 1)}}{|\mathbf{c}|^2}$$

If $(B^2 - AC) = 0$ one solution

If $(B^2 - AC) < 0$ no solution

If $(B^2 - AC) > 0$ two solutions

Handling both intersections (determinant positive)

```
// Use the lesser of the two, unless that would be a degenerately near (re-)hit
if( hit_1 < minimum_dist || hit_1 > hit_2 ) hit_1 = hit_2;
if( hit_1 < minimum_dist || hit_1 >= existing_intersection.distance ) return;
// Make sure this is the closest intersection > minimum_dist so far before keeping it
```

Note: Depending on your quadratic equation code, hit1 may be necessarily < hit2

Most useful lecture slides

Summary: Raytracing

Recursive algorithm

function Main

for each pixel (c,r) on screen

determine ray $r_{c,r}$ from eye through pixel

color(c,r) = raytrace($r_{c,r}$)

end for

end

function raytrace(r)

find closest intersection P of ray r with objects

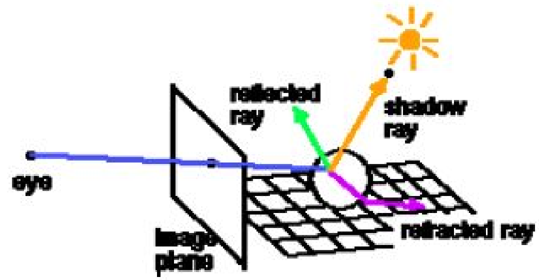
clocal = Sum(shadowRays(P,Light$_i$))

c_{rfl} = raytrace(r_{rfl})

c_{rfa} = raytrace(r_{rfa})

return c = clocal + k_{rfl}*c_{rfl} + k_{rfa}*c_{rfa}

end



Similar slide

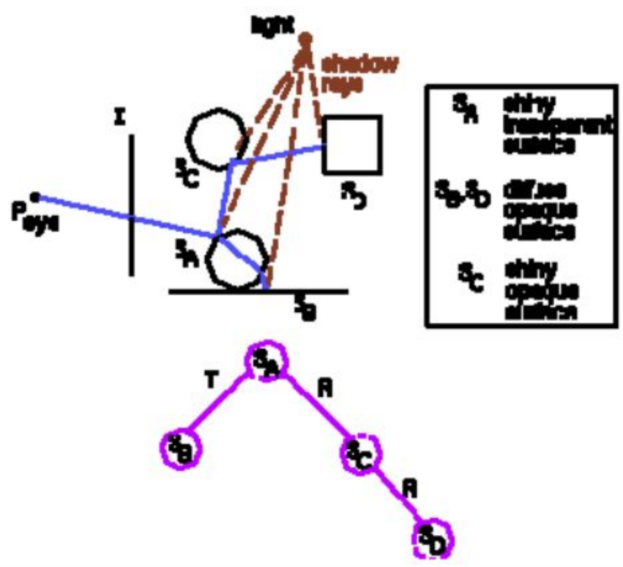
Backwards Raytracing Algorithm

For each pixel construct a ray: eye → pixel

raytrace(ray)

P = closest intersection
color_local = ShadowRay(light$_1$, P) + …
+ ShadowRay(light$_N$, P)
color_reflect = raytrace(reflected_ray)
color_refract = raytrace(refracted_ray)
color = color_local +
+ k$_{rfl}$* color_reflect
+ k$_{rfa}$* color_refract

return(color)



Rays

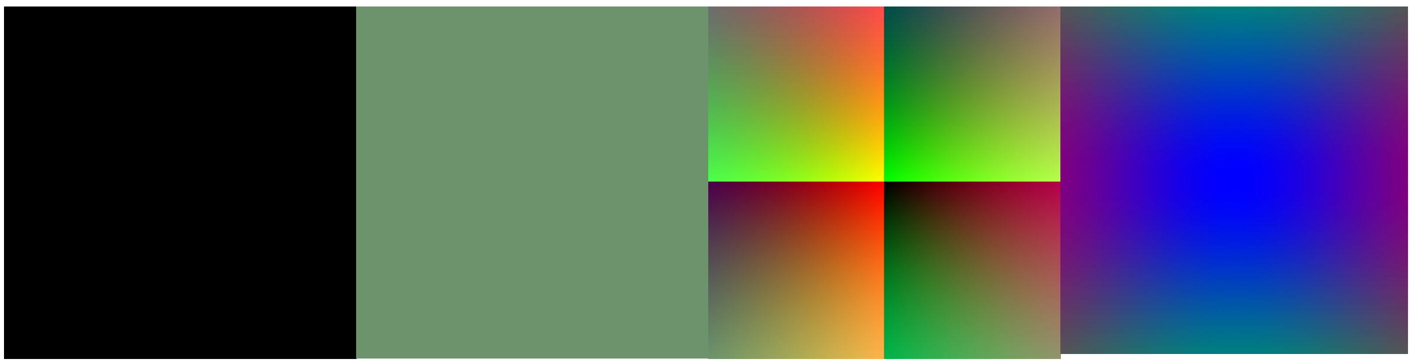
- Just a vector! Origin + direction
- However, to get from 3D world space to 2D screen pixel coordinates, need to do some math
- We're going backwards (start at the pixel, project onto world space, hit a sphere, find vectors to all light sources)

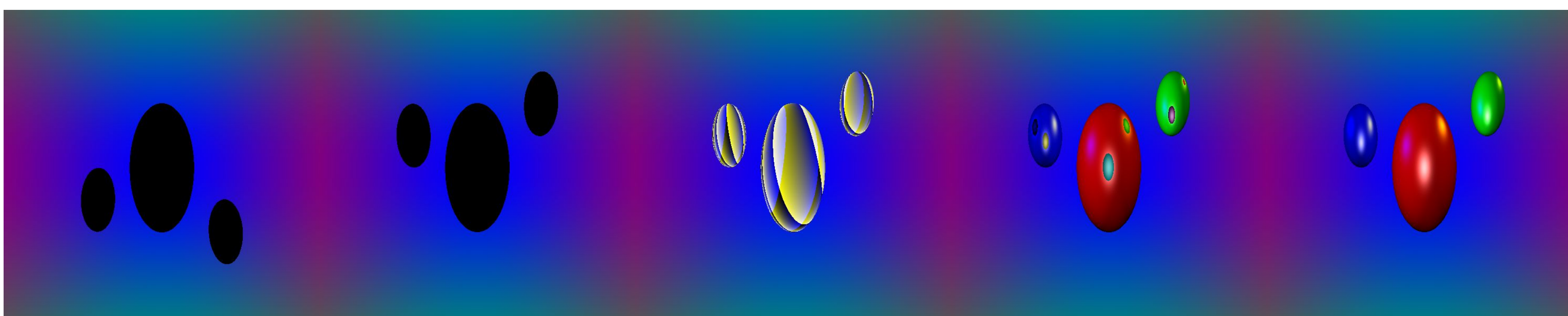
Order to do things:

- Fill in function Ray_Tracer::get_dir() to generate your ray directions
- Fill in class Ball’s constructor
- Fill in Ball::intersect()
- Fill in Ray_Tracer::trace_ray()

Debugging a ray tracer

- Set colors to intermediate values (dir, normals, L ,E) to help you picture your vectors, and verify them
- Examples of some iterations of the program:





Refraction angle

https://en.wikipedia.org/wiki/Snell's_law

Use the below formula, substituting your sphere's refraction index for (n1/n2):

$$\sin\theta_2 = \left(\frac{n_1}{n_2}\right)\sin\theta_1 = \left(\frac{n_1}{n_2}\right)\sqrt{1-(\cos\theta_1)^2}$$

$$\cos\theta_2 = \sqrt{1-(\sin\theta_2)^2} = \sqrt{1-\left(\frac{n_1}{n_2}\right)^2\left(1-(\cos\theta_1)^2\right)}$$

$$\mathbf{v}_{\mathrm{refract}} = \left(\frac{n_1}{n_2}\right)\mathbf{l} + \left(\frac{n_1}{n_2}\cos\theta_1 - \cos\theta_2\right)\mathbf{n}$$

The formula may appear simpler in terms of renamed simple values $r = n_1/n_2$ and $c = -\mathbf{n}\cdot\mathbf{l}$, avoiding any appearance of trig function names or angle names:

$$\mathbf{v}_{\mathrm{refract}} = r\mathbf{l} + \left(rc - \sqrt{1-r^2\left(1-c^2\right)}\right)\mathbf{n}$$

Performance

- Most expensive function: 4x4 matrix inverse(). Need it for colliding with a ray.
  - When to compute this?
    - Once per ray?
    - Once per sphere?

Handling larger scenes

- A data structure can help store all objects to accelerate collision lookup (including collisions with rays)
  - Voxels (volume pixels), implemented as hash table buckets
  - Shapes that occupy more than one cell can be added to several neighboring buckets (up to 8 as long as voxel size > 2 * object size)
  - Add more hash tables to allow a variety of shape sizes
- A quick and dirty demo made from roughly the same template as hw3:

https://www.youtube.com/watch?v=m0XqxCh1Gpc