Quick reminder how to transform the image plane into canonical representation.
Assume first \( n_x = n_y \) (#columns=#rows).

Witness points (first in 2D): If you prefer to align the cameras’ coordinate system with \( xyz \),

Use the matrix \( M = Rotation_{\vec{u},\vec{v},\vec{w}} \cdot Trans(-Eye) \cdot p \)

\[
Rotation_{\vec{u},\vec{v},\vec{w}} = \begin{bmatrix}
-\vec{u} & -\vec{v} & -\vec{w}
\end{bmatrix}
\]

https://www.geogebra.org/m/jdqhhwku

### Ray Tracing Algorithm

Ray tracing algorithm for each pixel {
  compute viewing ray
  intersect ray with scene
  compute illumination at intersection
  put result into image
}

\[\text{for each pixel} \{\]
\[\text{compute viewing ray}\]
\[\text{intersect ray with scene}\]
\[\text{compute illumination at intersection}\]
\[\text{store resulting color at pixel}\]
\[\}\]

### Intersection with Many Types of Shapes

- In a given scene, we also need to track which shape had the nearest hit point along the ray.

- This is easy to do by augmenting our interface to track a range of possible values for \( t \), \([t_{\text{min}}, t_{\text{max}}]\):

\[
\text{intersect(eye, dir, t_{\text{min}}, t_{\text{max}})};
\]

- After each intersection, we can then update the range
Intersection with Many Types of Shapes

for each pixel p in Image {
    let [eye, dir] = camera.compute_ray(p);
    let hit_surf = undefined; let hit_rec = undefined;
    let t_min = 0; let hit_t = Infinity;

    scene.surfaces.forEach(function(surf) {
        let intersect_rec = surf.intersect(eye, dir, t_min, hit_t);
        if (intersect_rec.hit) {
            hit_surf = surf;
            hit_t = intersect_rec.t;
            hit_rec = intersect_rec;
        }
    });

    // Compute a color c
    image.update(p, c);
}

Illumination

for each pixel {
    compute viewing ray
    intersect ray with scene
    compute illumination at intersection
    store resulting color at pixel
}

Our images so far

- With only eye-ray generation and scene intersection

for each pixel p in Image {
    let hit_surf = undefined;
    ...

    scene.surfaces.forEach(function(surf) {
        if (surf.intersect(eye, dir, ...)) {
            hit_surf = surf;
            ...
        }
    });

    c = hit_surf.ambient;
    Image.update(p, c);
}

Today: shading

From this (ambient shading) + Diffuse Shading + Specular Shading ⇒ this

https://en.wikipedia.org/wiki/Phong_shading
Shading

- Goal: Compute light reflected toward camera

- Inputs:
  - eye direction
  - light direction (for each of many lights)
  - surface normal
  - surface parameters (color, shininess, ...)

Light Sources

- There are many types of possible ways to model light, but for now we'll focus on point lights
- Point lights are defined by a position $p$ that irradiates equally in all directions
- Technically, illumination from real point sources falls off relative to distance squared, but we will ignore this for now.

Shading Models

Just to be sure:

- **Shading** ≠ **Shadows**
  - Shadows are casted by occluding sources of light.
  - Shading of a surface - changing of intensity of the reflected light due to surface properties and geometry, and its locations in 3D with respect to locations of viewer and light source.

We will cover Diffuse shading and Specular Shading. We will study a trick that is easy to program, and "looks" like physical diffuse shading.

Ambient coefficient ≠ Albedo coefficient

- Albedo coefficient - percentage of white light reflected by the object
- White light - might contains all visible frequencies, not only RGB.
- No attention to color.
**Ambient “shading” and Albedo**

- Ambient light - has no particular direction.
- Every material has 3 coefficients: \((k_d, k_r, k_g)\).
- \(k_d\) specifies the percentage of blue light that the surface reflects (obviously, as blue light).
- The location of viewer and the location of the light-source are irrelevant.
- If a sphere has Ambient coefficient \((k_d, k_r, k_g, k_b)\) = (0.1, 0.9, 0.9) it looks very dim in Red light, but bright in Blue or Green light.
- If illuminated by white light, then the sphere color is cyan.
- When describing a scene to (say) OpenGL, WebGL, processing.org etc, we could specify for every light source how much intensity it emits (in RGB).
- In reality, there is no ambient light.
- In OpenGL, we could specify 3 sets of coefficients (for ambient, for diffuse, and for specular). We can also specify the scene ambient RGB.
- E.g. specifying the ambient light in the scene as \([0.3, 0.1, 0.9]\), and a sphere with \([0, 0, 0.5]\), will be seen with \(RGB = (0, 0, 0.45)\)

---

**Lambertian (Diffuse) Shading**

- Simple model: amount of energy from a light source depends on the direction at which the light ray hits the surface.
- Results in shading that is view independent.

\[
L_d = k_d \max(0, \vec{n} \cdot \vec{I})
\]

\(L_d\) \hspace{1cm} \(k_d\) \hspace{1cm} \(\max(0, \vec{n} \cdot \vec{I})\)

**Intensity of light on \(f\) = #photons falling on \(1\text{in}^2\)**

The number of photons on \(e\) and on \(f\) is the same, but the area increases to \(1\text{in}^2/\cos\alpha\), so intensity now is \(I/f = I/1/\cos\alpha = I\cos\alpha\)

Let \(\vec{L}\) be a unit vector from \(f\) toward the light source, and let \(\vec{n}\) be the normal to the door.

\[
\cos\alpha = \vec{L} \cdot \vec{n}
\]

The intensity of light reflected from \(f\) is intensity of light hitting \(f\) times \(k_d\)

Conclusion: To create diffuse shading, render \(f\) with \(RGB = k_d \ I \ \vec{L} \cdot \vec{n}\)
Lambertian Shading

- $k_d$ is a property of the surface itself (3 constants - one per each color channel)
- Produces matte appearance of varying intensities

The moon paradox

- why don’t we see this gradual shading when looking at the moon?

Toward Specular Shading: Perfect Mirror

- Many real surfaces show some degree of shininess that produce specular reflections
- These effects move as the viewpoint changes (as oppose to diffuse and ambient shading)
- Idea: produce reflection when $v$ and $l$ are symmetrically positioned across the surface normal

Mirrors - perfect reflections

- Before talking about specular reflection, let's see how to render a scene that contains mirror.
- Ray tracing: For each pixel on the image plane, trace a ray $d$ from the eye via this pixel, till hits an object. If this object is a mirror, we need to continue this ray in the deflected direction $r$.
- How could find $r$?
- Claim: $r = d - 2(d \cdot n)n$, $n$ is a unit vector orthogonal to the mirror.
- Proof
  - Assume wlog that $n=(0,1)$ (vertical upward).
  - Look at the components: $d=(d.x,d.y)$, $r=(r.x,r.y)$
  - $r$ and $d$ have the same $x$-value, but opposite $y$-value:
    - $r.x = d.x$ and
    - $r.y = -d.y = r.y + (-2r.y) = r.y - 2(n \cdot r)$
  - $(d \cdot n)n = (0, r.y)$. 
**Application: mirror sphere**

- A ray \( d \) that hits the sphere \( B \). We find the intersection point \( P \), find the normal to \( B \) at \( P \), \[ n = \frac{P - c}{|P - c|} \]
- and bounces in the direction \( r = d - 2(n \cdot d)\).d

**Blinn-Phong (Specular) Shading**

- Many real surfaces show some degree of shininess that produce specular reflections.

- These effects move as the viewpoint changes (as oppose to diffuse and ambient shading).

- Idea: produce reflection when \( \vec{v} \) and \( \vec{l} \) are symmetrically positioned across the surface normal.

\[ \text{Reflected \_ Value} = \cos^n(\alpha) = (h \cdot n)^m. \]

**Blinn-Phong (Specular) Shading**

- For any two unit vectors \( \vec{v}, \vec{l} \), the vector \( \vec{v} + \vec{l} \) is a bisector of the angle between these vectors.
- Normalize \( \vec{v} + \vec{l} \)
  \[ h = (\vec{v} + \vec{l}) / \|\vec{v} + \vec{l}\| \]
- In a perfect mirror, the 100% of the reflection occurs at the surface point where \( h \) is the normal \( n \)
- Diffuse reflection. Reflect large value for points where \( h \) is "almost" \( n \)
- Phong heuristic:
  \[ L_s = k_s I_{\max}(0, (n \cdot h)^p) \]

**Blinn-Phong Decomposed**

- Ambient + Diffuse + Specular = Phong Reflection

**Note:** shadows are additional effort, not a specular effect

https://en.wikipedia.org/wiki/Phong_shading
Blinn-Phong Shading

- Increasing $p$ narrows the lobe
- This is kind of a hack, but it does look good

Putting it all together

- Usually include ambient, diffuse, and specular in one model
  
  $L = L_a + L_d + L_s$

  
  $L = k_d I_a + k_d I \max(0, n \cdot l) + k_s I \max(0, n \cdot h)^p$

- And, the final result accumulates for all lights in the scene
  
  $L = k_d I_a + \sum_i (k_d I_i \max(0, n \cdot l) + k_s I_i \max(0, n \cdot h)^p)$

- Be careful of overflowing! You may need to clamp colors, especially if there are many lights.

Simple Ray Tracer

```javascript
function ray_cast(eye, dir, near, far) {
  let hit_surf = undefined; let hit_rec = undefined;
  let t_min = 0; let hit_t = Infinity;
  let color = background; //default background color

  scene.surfaces.forEach(function(surf) {
    let intersect_rec = surf.hit(eye, dir, t_min, hit_t);
    if (intersect_rec.hit) {
      hit_surf = surf;
      hit_t = intersect_rec.t;
      hit_rec = intersect_rec;
    }
  });
  if (hit_surf !== undefined) {
    color = hit_surf.kA * Ia;
    scene.lights.forEach(function(light) {
      //compute l, h
      color = color + hit_surf.kD*max(0,n·l) + hit_surf.kS*max(0,n·h)^p;
    });
  }
  return color;
}
```

for each pixel p in Image {
  let [eye, dir] = camera.compute_ray(p);
  let c = ray_cast(eye, dir, 0, Infinity);
  image.update(p, c);
}

Refraction and Snell Law

- When light passes from one medium to another, (say air → glass or glass → air, its direction might change.

- This happens when the speed of light in the two mediums are different

Credit: Wikipedia
Following the wavefront

For the wavefronts to stay connected at the boundary the wave must change direction.

Refractive and Snell’s Law

- When ray of light traverses from one medium (e.g., from air to water) it might bend. This is called refraction.

The transmissive material:

$$\eta_T = \text{material index}$$

$$\eta_i = 1$$

$$\eta_f = 1$$

$$\eta = \text{material index}$$

$$n_{water} = 1.3$$

Examples of refractions

- Camera lens
- Fiber optics

- Total Internal Reflection
  - No transmission

Refraction and Snell’s Law

- Governs the angle at which a refracted ray bends when traversing from air to glass, water etc.

- Computation based on refraction index (confusingly denoted $n_T$) of the mediums. The mediums here are air and glass.

- Typical air has refraction indexed

  $$n_{air} = 1$$

  $$n_{glass} = 1.5$$

  $$n_{water} = 1.3$$

  $$n_{fiber optics} = 1.46$$

- Snell law: $n_i \sin \theta = n \sin \phi$

$$\phi = \arcsin \left( \frac{1}{1.3} \sin \theta \right)$$
Snell’s Law and vector calculus

- Working with cosine’s are easier because we can use dot products
- Can derive the vector for the refraction direction $\mathbf{t}$ as

$$\mathbf{t} = \frac{n(d + n \cos \theta)}{n_t} - n \cos \phi$$

$$= \frac{n(d - n(d \cdot n))}{n_t} - n \sqrt{1 - \frac{n^2(1 - (d \cdot n)^2)}{n_t^2}}$$

Careful:
don’t confuse $\mathbf{n}$ (a normal vector) with $1.3n_t$ for water) and with $n (=1$ for air)

Total Internal Reflection

Recursive Ray Tracing

- Idea: after finding the closest hit, cast a ray to each light source to determine if it is visible
- Be careful not to intersect with the object itself. Two solutions:
  - Only check for hits against all other surfaces
  - Start shadow rays a tiny distance away from the hit point by adjusting $t_{\text{min}}$
Recursive Ray Tracer

Color ray_cast(Ray ray, SurfaceList scene, float near, float far) {
    //initialize color; compute hit_surf, hit_position;
    ...
    if (hit_surf is valid) {
        color = hit_surf.kA * Ia;
        scene.lights.forEach(
            function (light)
                //compute \vec{l}_i, \vec{h}_i
                //check for shadow rays to decide if the light illuminates
                if (ray from hit_position in direction of \vec{l}_i does not hit scene) {
                    color += hit_surf.kD*\vec{I}_i * max(0, \vec{n} \cdot \vec{l}_i) + hit_surf.kS*\vec{I}_i * max(0, \vec{n} \cdot \vec{h}_i);
                }
        );
        //compute reflect direction
        \vec{r}_i
        //call ray_cast() recursively for mirror reflections
        color += hit_surf.kM*ray_cast(hit_position, \vec{r}_i, scene, epsilon, +inf);
    }
    return color;
}

Ray Casting vs Ray Tracing

- Ray casting: tracing rays from eyes only
- Ray tracing: tracing secondary rays

Secondary rays are used for testing shadows, doing reflections, refractions, etc.

Shadows

- Surface should only be illuminated if nothing blocks the light from hitting the surface
- This can be easily checked by intersecting a new ray with the scene!

(hard) Shadows

- Idea: after finding the closest hit, cast a ray to each light source to determine if it is visible
- Be careful not to intersect with the object itself. Two solutions:
  - Only check for hits against all other surfaces
  - Start shadow rays a tiny distance away from the hit point by adjusting $t_{min}$
Distribution Ray Tracing

Reality Check: Do These Pictures Look Real?

What’s Wrong?

- No surface is a perfect mirror because no surface is perfectly smooth

What have we modeled?

- Ideal specular (mirror)
- Glossy specular
- Lambertian
Most Surfaces have Microgeometry

Ideal Reflection/Refraction

Non-Ideal Reflection/Refraction

- Can approximate the microgeometry

Non-Ideal glossy reflection

Non-Ideal refraction

Glossy (as opposed to mirror) reflection

Glossy (as opposed to perfect) refraction
Approach: Distribution Glossy Reflection by Randomly Sampling Rays

Glossy Reflection: Compute Many Rays per Bounce and Average

Ideal Reflection: One Ray Per Bounce

Other Uses of Distribution Ray Tracing
Approach: Distribution Glossy Reflection by Randomly Sampling Rays

Ideal Reflection: One Ray Per Bounce

Glossy Reflection: Compute Many Rays per Bounce and Average

Variation in this distribution is controlled by the glossiness of the surface

Other Uses of Distribution Ray Tracing
Problem: Hard Shadows

- One shadow ray per intersection per point light source

Distributed Ray Tracing
Robert L. Cook
Thomas Porter
Loren Carpenter
Computer Division
Lucasfilm Ltd.

Abstract

Ray tracing is one of the most elegant techniques in computer graphics. Many phenomena that are difficult or impossible with other techniques are simple with ray tracing, including shadows, reflections, and refraction of light. Ray directions, however, have been determined precisely, and this has limited the capabilities of ray tracing. By distributing the directions of the rays according to the analytic function they sample, ray tracing can incorporate fuzzy phenomena. This provides correct and easy solutions to some previously unsolved or partially solved problems, including motion blur, depth of field, point sources, transparency, and fuzzy reflections. Motion blur and depth of field calculations can be integrated with the scene surface calculations, avoiding the problems found in previous methods.

Hard Shadows

Soft Shadows

http://erich.realtimerendering.com/shadow_comparison.html
What Causes Soft Shadows

Lights aren’t all point sources

Distribution Soft Shadows

Randomly sample light rays

Computing Soft Shadows

- Model light sources as spanning an area
- Sample random positions on area light source and average rays

one shadow ray (to random location)

lots of shadow rays
Problem: Aliasing
Drawing a black line on a white board

Some pixels need to be rendered as gray, with gray level:

\[
\text{Area of black region in pixel} \quad \text{Area of pixel}
\]

- Problem: Hard to calculate how much of the pixel is covered
- Solution: Random sample points in the pixel.
- Calculate what is the percentage of the point of each color

Distribution Antialiasing w/ Regular Sampling

Multiple rays per pixel

Problem: Aliasing

Antialiasing w/ Supersampling

- Cast multiple rays per pixel, average result
Distribution Antialiasing

Multiple rays per pixel

Distribution Antialiasing w/ Regular Sampling

Multiple rays per pixel

Distribution Antialiasing w/ Random Sampling

Remove Moiré patterns

Random Sampling Could Miss Regions Without Enough Sampling
Stratified (Jittered) Sampling

One ray per box

Problem: Focus
Real Lenses Have Depth of Field

Depth of Field

- Multiple rays per pixel, sample lens aperture

Justin Legakis
Distribution Depth of Field

Problem: Exposure Time
Real Sensors Take Time to Acquire

Randomly sample eye positions

Motion Blur

- Sample objects temporally over a time interval