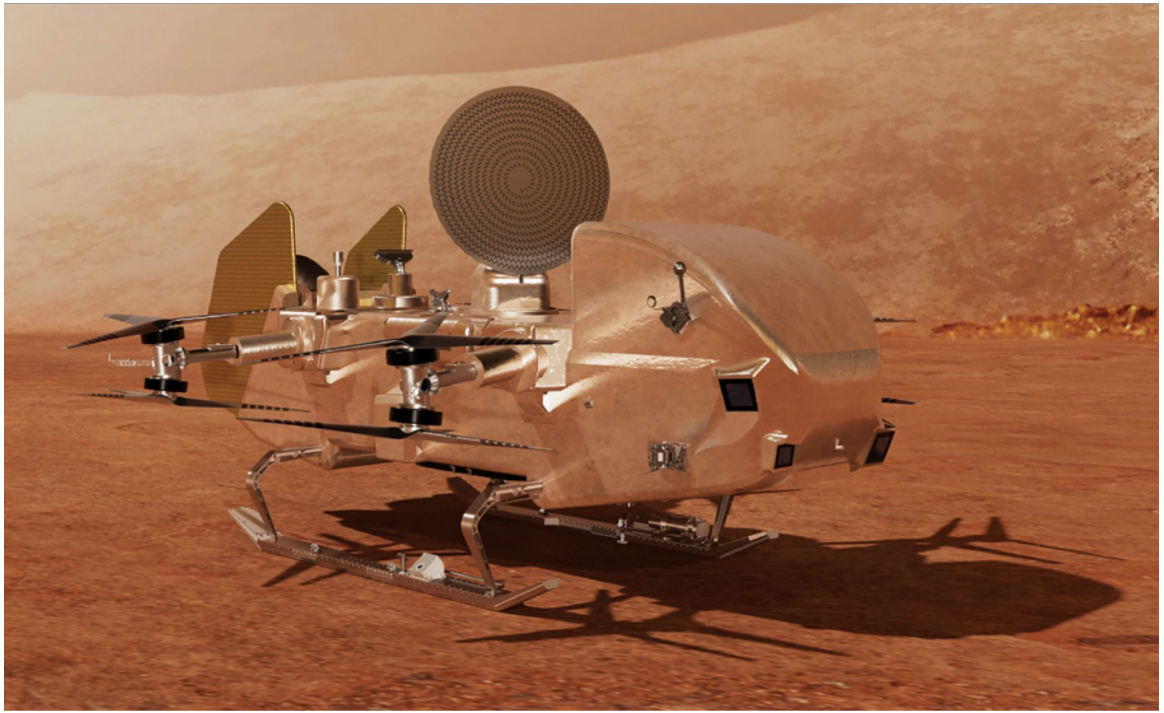
Raytracing (Part 1)

Slides by Sarah Jobalia and Kevin Li, with modifications by Kate Baker

Housekeeping

- **HW2 DUE TODAY**
  - See website for grading session hour change
- HW3 Released
- Project proposal released
- Ed private posts
- Additional OH on Tuesday

Something Cool



“An artist’s concept depicts Dragonfly soaring over the dunes of Saturn’s moon Titan (left) and preparing to sample and examine the surface of a landing site (right).”

Scientific American Magazine, July/August 2026, Volume 335, Issue 1

Last time…

- BRDF/BTDF/BSDF
- The lighting equation

- Today… how that sets us up well to learn about raytracing!

Motivation

- Today: ray-object intersections & shadows
- Next lecture: reflections, transmissions, and other recursive concepts

Rasterization (Scanline Rendering)

Ray Tracing

High-Level Idea: Rays and Shadows

- Shoot rays out from camera/eye
  - The ray then travels from the camera/eye, through the pixel on the screen, to the scene

High-Level Idea: Rays and Shadows

- Shoot rays out from camera/eye
  - The ray then travels from the camera/eye, through the pixel on the screen, to the scene
- We want to figure out where the ray **lands** on the scene
  - Then we use the lighting equation to find the color of this ray-scene intersection point

High-Level Idea: Rays and Shadows

- Shoot rays out from camera/eye
  - The ray then travels from the camera/eye, through the pixel on the screen, to the scene
- We want to figure out where the ray **lands** on the scene
  - Then we use the lighting equation to find the color of this ray-scene intersection point

- Shadow rays
  - From the intersection point, shoot a new ray toward the light source
  - If the ray hits something before reaching the light, we know the intersection point is in shadow (skip that light's contribution)

Lecture Outline

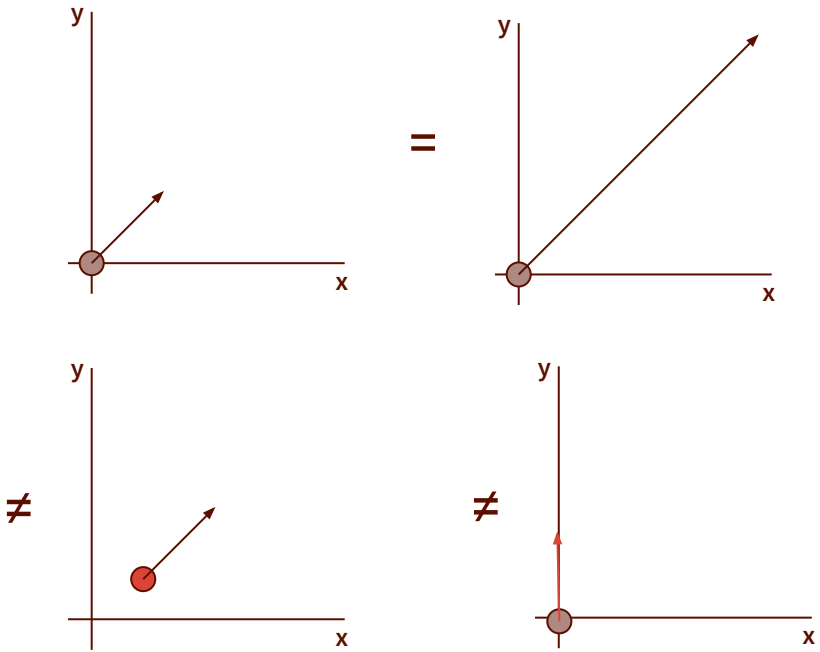
- Vector Math Review
- Constructing rays
- Ray-Triangle & Ray-Object Interaction
  - Ray-Plane Interaction
  - Project Triangle & Intersection to 2D
- Parallelization & Optimization
- Transferring Normals
- Shadows & Spurious Self-Occlusion

Lecture Outline

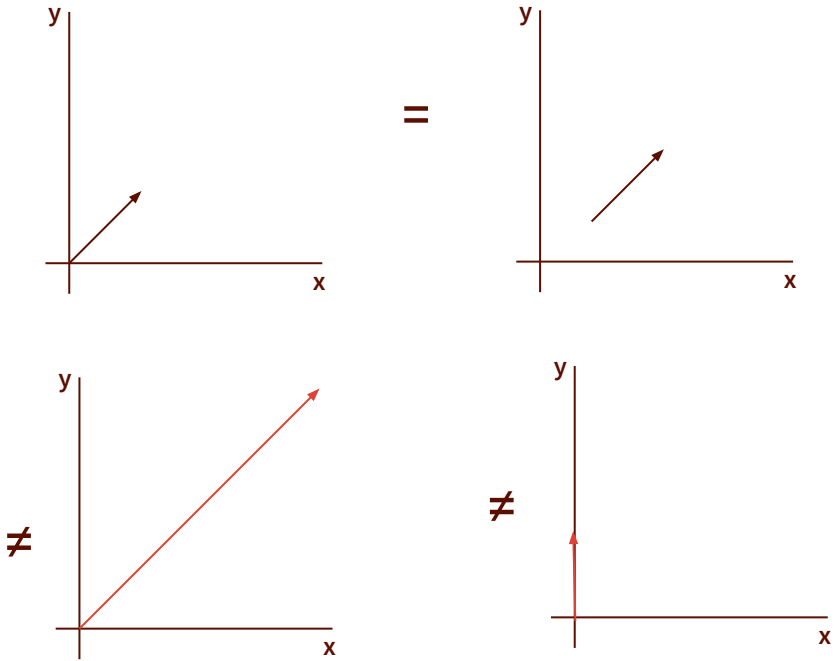
- Vector Math Review
- Constructing rays
- Ray-Triangle & Ray-Object Interaction
  - Ray-Plane Interaction
  - Project Triangle & Intersection to 2D
- Parallelization & Optimization
- Transferring Normals
- Shadows & Spurious Self-Occlusion

Vector Math Review

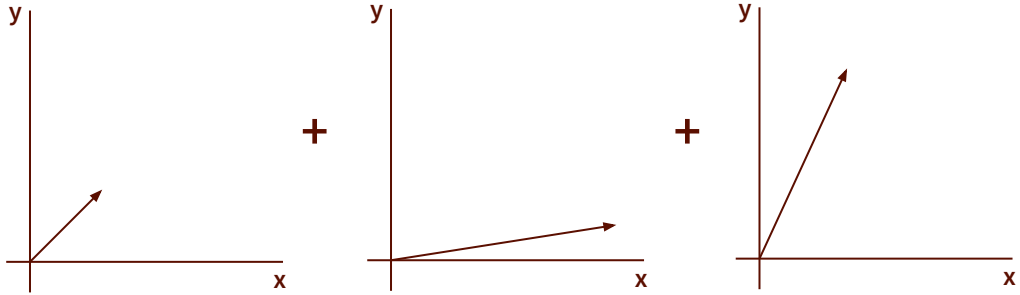
Ray = point + direction



Vector = direction + length

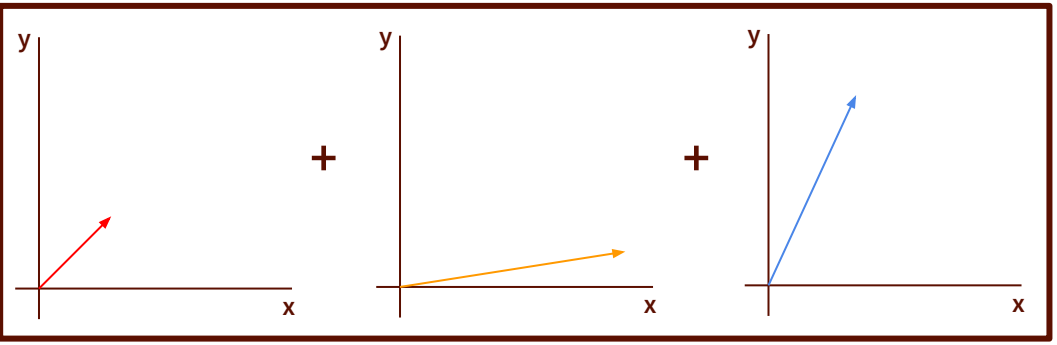


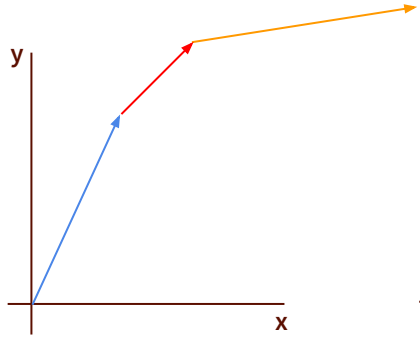
Vector Addition

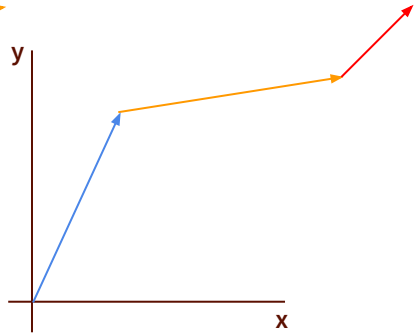


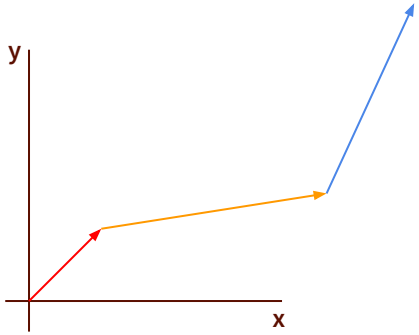
x

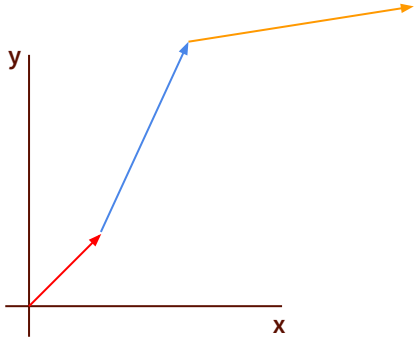
Vector Addition

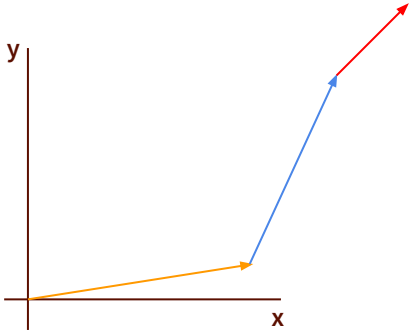


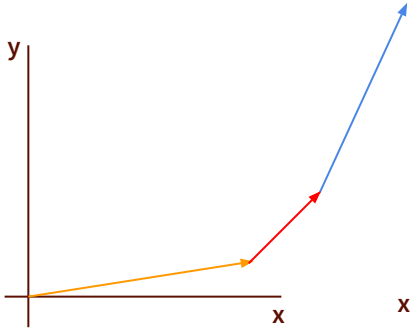




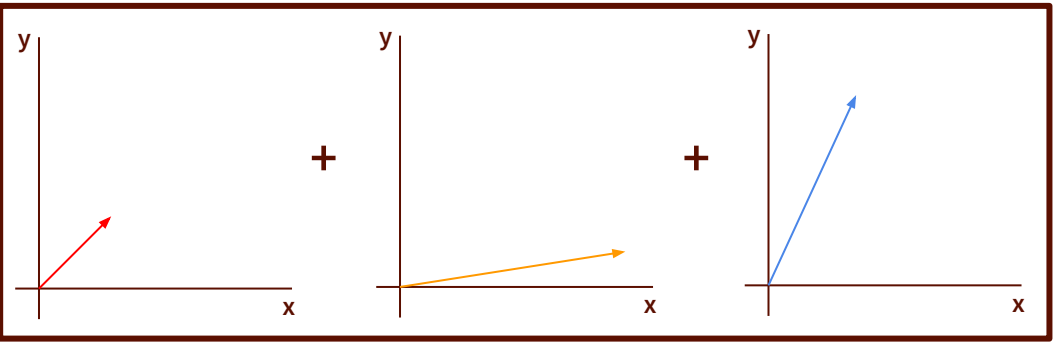


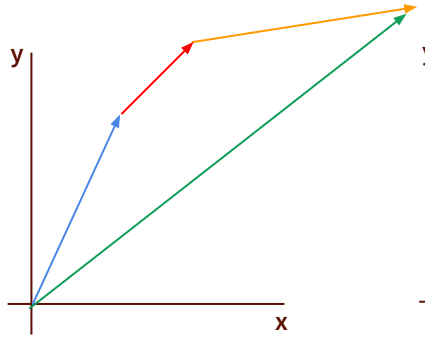


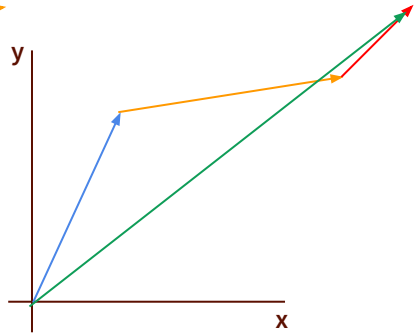


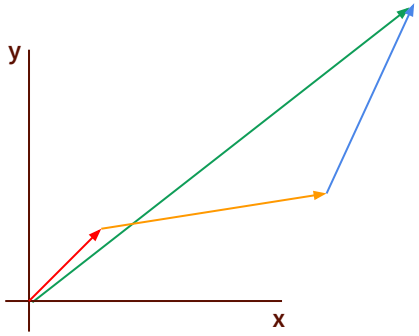


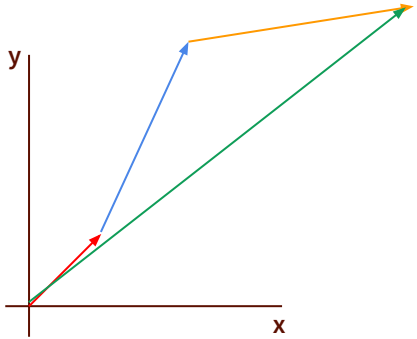
Vector Addition

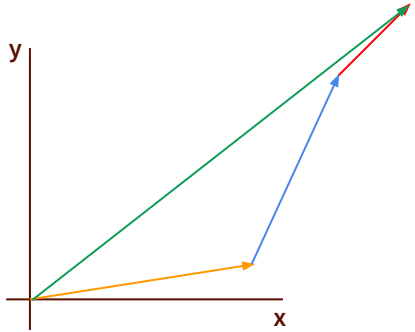


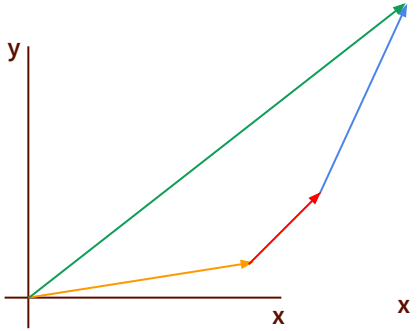




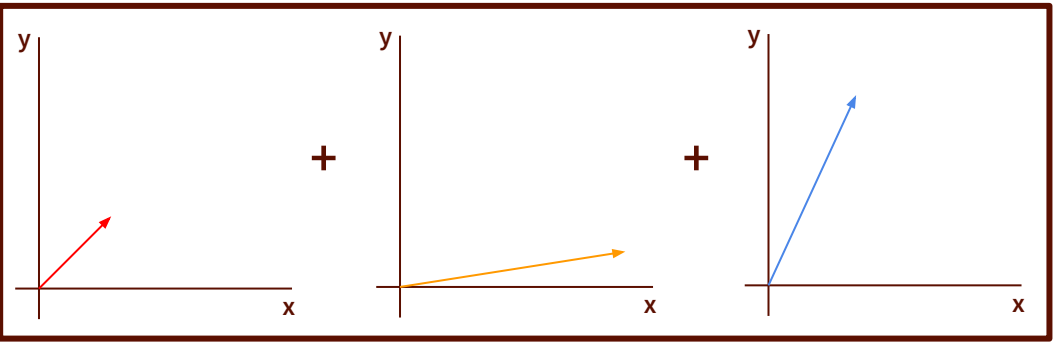


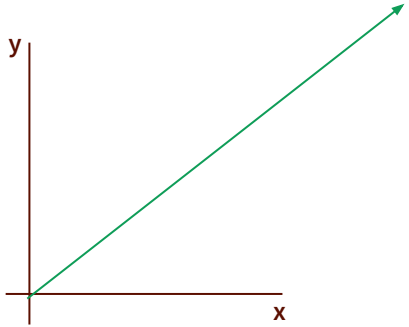


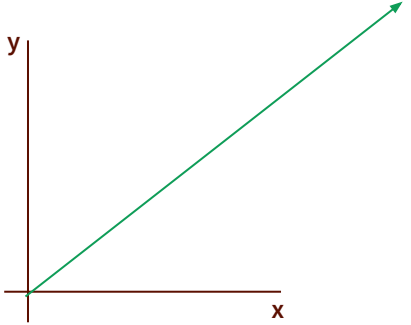


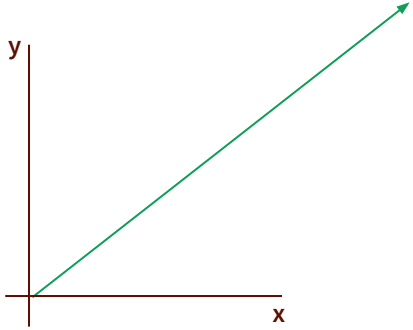


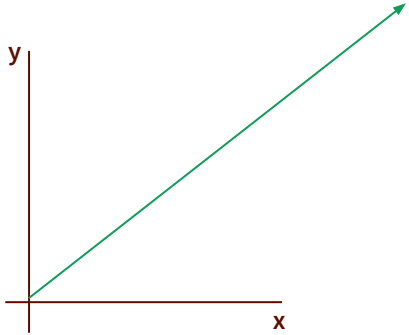
Vector Addition

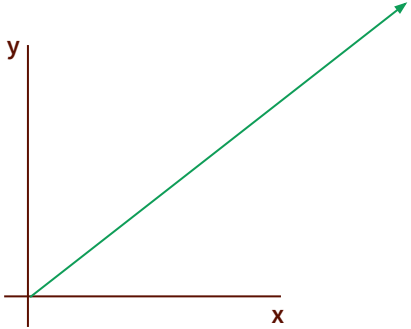


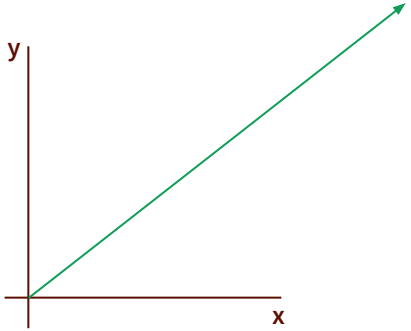




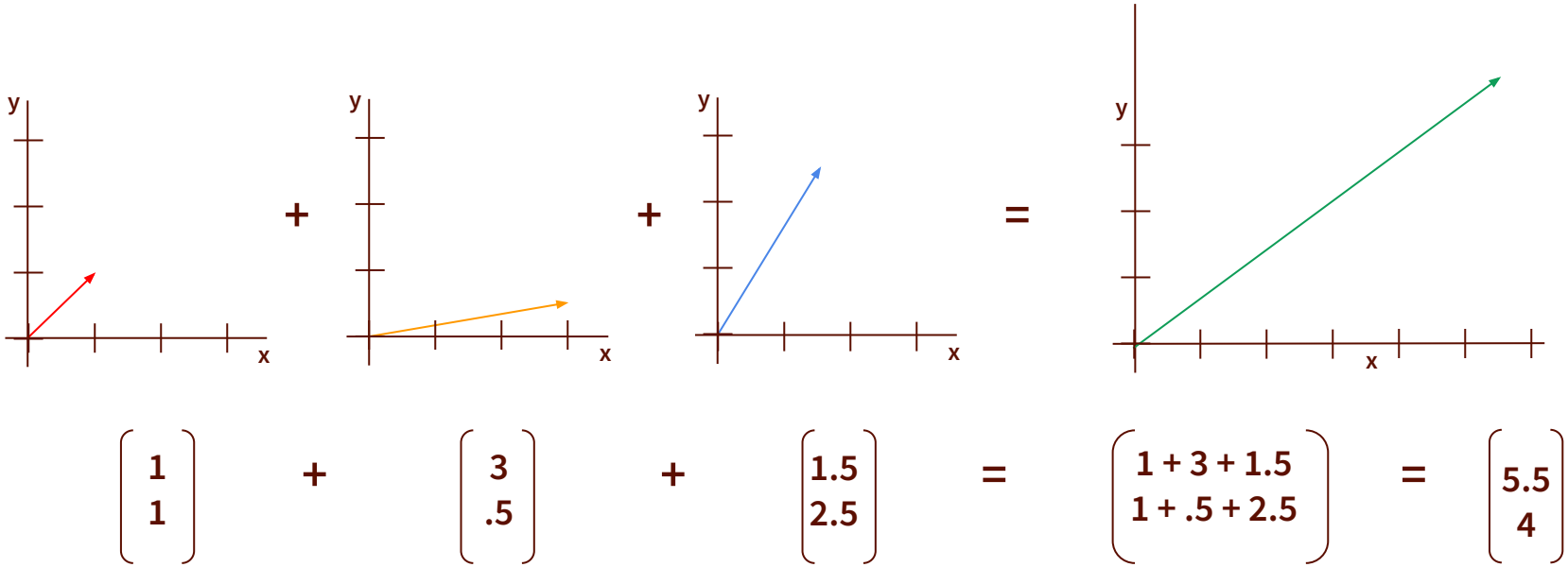




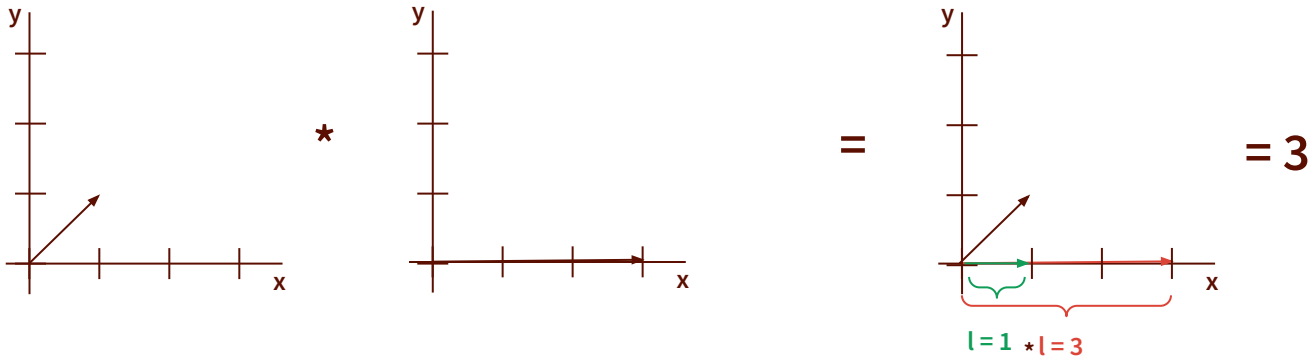




Vector Addition

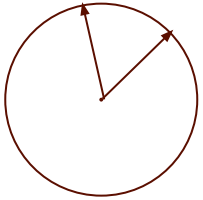


Dot Products

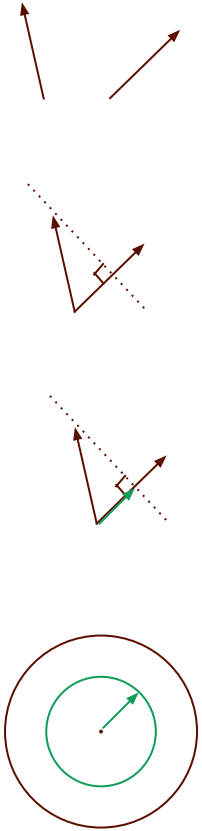


$$\begin{pmatrix} 1 \\ 1 \end{pmatrix} * \begin{pmatrix} 3 \\ 0 \end{pmatrix} = (1 * 3) + (1 * 0) = 3$$

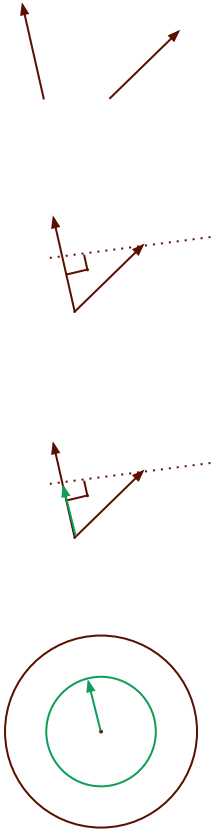
Dot Products



l = 1

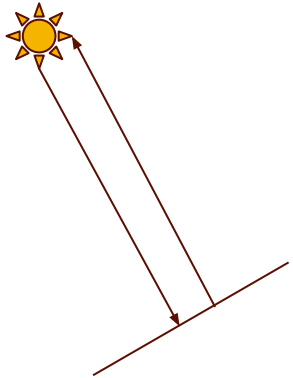


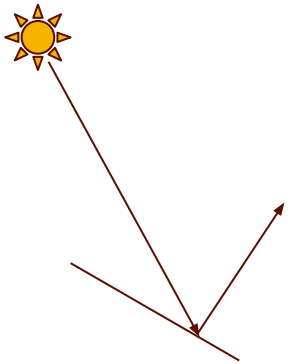
= .57

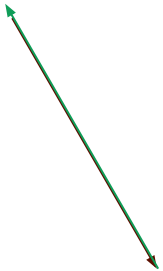


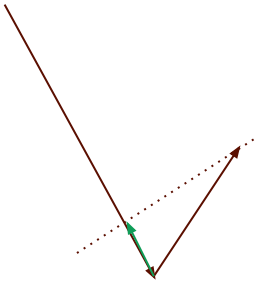
= .57

Dot Products

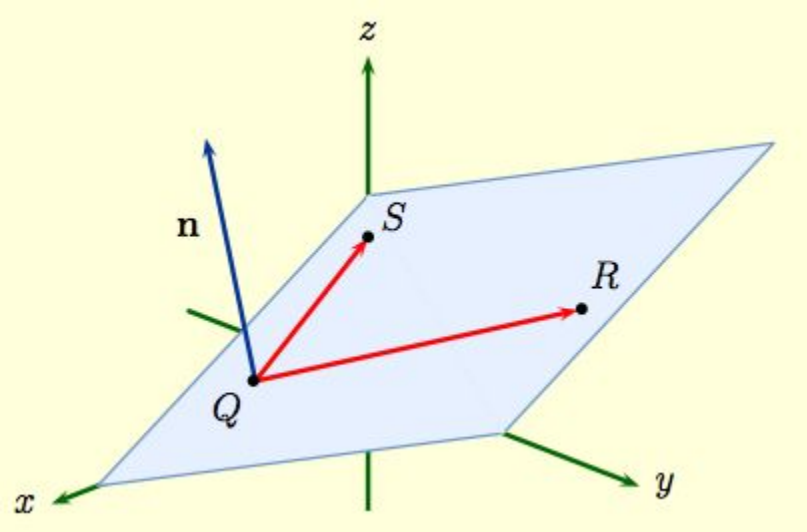








Cross Products



Lecture Outline

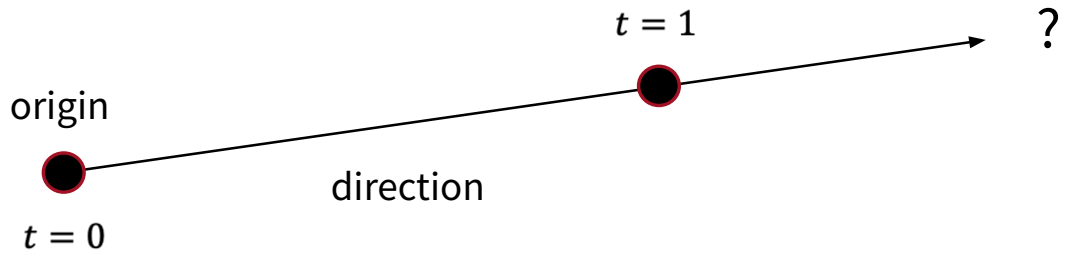
- Vector Math Review
- Constructing rays
- Ray-Triangle & Ray-Object Interaction
  - Ray-Plane Interaction
  - Project Triangle & Intersection to 2D
- Parallelization & Optimization
- Transferring Normals
- Shadows & Spurious Self-Occlusion

Constructing Rays

- In class today: **ray = equation** for derivation purposes

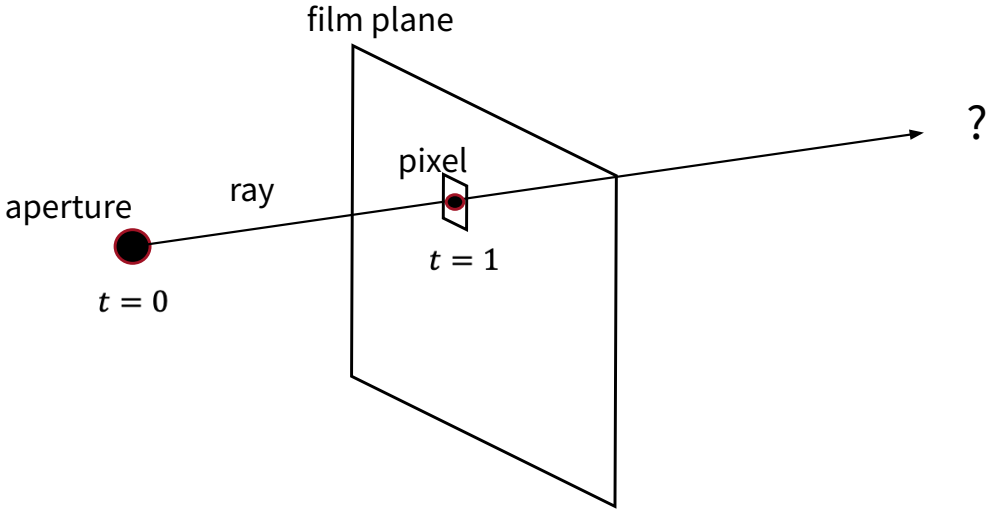
Constructing Rays

- In class today: **ray = equation** for derivation purposes
- In practice: **ray = 3 components** in coding
  - 1) an origin point
  - 2) a vector direction
  - 3) a parameter *t* that tells us how far along the direction we are



Constructing Rays

- In class today: **ray = equation** for derivation purposes
  - $R(t) = A + (P - A)t$



Constructing Rays

$$R(t) = A + (P - A)t$$

- In class today: **ray = equation** for derivation purposes
  - A ⇒ the aperture (camera position) as a 3D point
  - P ⇒ the pixel center as a 3D point
  - t ⇒ length of ray as a scalar. $t \in [0, \infty)$, but technically $t \in [1, t_{far}]$ to be inside the frustum

Constructing Rays

$$R(t) = A + (P - A)t$$

- In class today: **ray = equation** for derivation purposes
  - A ⇒ the aperture (camera position) as a 3D point
  - P ⇒ the pixel center as a 3D point
  - t ⇒ length of ray as a scalar. $t \in [0, \infty)$, but technically $t \in [1, t_{far}]$ to be inside the frustum
- We want to find the intersection of the ray with the smallest $t \in [1, t_{far}]$, which gives us a **3D point in space**.

Constructing Rays

$$R(t) = A + (P - A)t$$

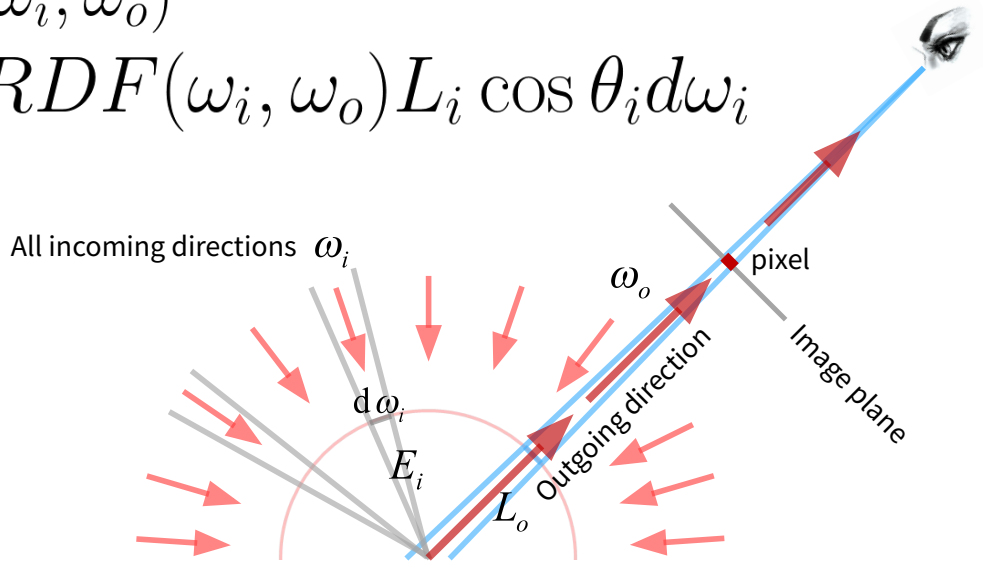
- In class today: **ray = equation** for derivation purposes
  - A ⇒ the aperture (camera position) as a 3D point
  - P ⇒ the pixel center as a 3D point
  - t ⇒ length of ray as a scalar. $t \in [0, \infty)$, but technically $t \in [1, t_{far}]$ to be inside the frustum
- We want to find the intersection of the ray with the smallest $t \in [1, t_{far}]$, which gives us a **3D point in space**.
- Then do our lighting calculations.

Recall: Lighting Equation

- We use the lighting equation to determine the color of a certain
- This requires understanding the rays involved!

$$L_o(\omega_o) = \sum_{i \in in} L_o(\omega_i, \omega_o)$$

$$L_o(\omega_o) = \int_{i \in in} BRDF(\omega_i, \omega_o) L_i \cos\theta_i d\omega_i$$



Lecture Outline

- Vector Math Review
- Constructing rays
- Ray-Triangle & Ray-Object Interaction
  - Ray-Plane Interaction
  - Project Triangle & Intersection to 2D
- Parallelization & Optimization
- Transferring Normals
- Shadows & Spurious Self-Occlusion

Ray-Triangle Intersection

- Recall: most of our object will be **triangle meshes**
  - We need to intersect our ray with a triangle
  - Triangles are **planar** (3 points are guaranteed to make a plane)

Ray-Triangle Intersection

- Recall: most of our object will be **triangle meshes**
  - We need to intersect our ray with a triangle
  - Triangles are **planar** (3 points are guaranteed to make a plane)

- One technique:
  - 1) Determine how a ray intersects a **plane** for an intersection point
  - 2) Then, check if this intersection point is **inside the triangle**

Ray-Triangle Intersection

- Recall: most of our object will be **triangle meshes**
  - We need to intersect our ray with a triangle
  - Triangles are **planar** (3 points are guaranteed to make a plane)

- One technique:
  - 1) Determine how a ray intersects a **plane** for an intersection point
  - 2) Then, check if this intersection point is **inside the triangle**
- Another approach:
  - Consider 3D ray-object intersection directly

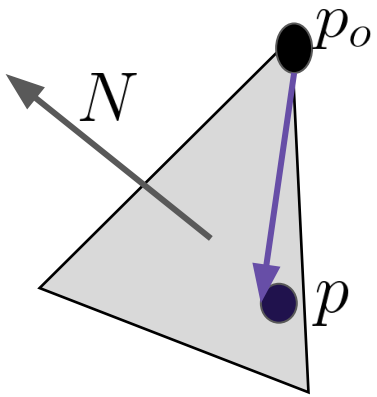
Ray-Triangle Intersection

- Recall: most of our object will be **triangle meshes**
  - We need to intersect our ray with a triangle
  - Triangles are **planar** (3 points are guaranteed to make a plane)

- One technique:
  - 1) Determine how a ray intersects a **plane** for an intersection point
  - 2) Then, check if this intersection point is **inside the triangle**
- Another approach:
  - Consider 3D ray-object intersection directly

Step 1: Ray-Plane Intersection

- A plane is defined by:
  - p_o ⇒ a point on the plane (can use any vertex of the triangle)
  - N ⇒ a normal vector to the plane (can use the triangle normal)
- A point p is on the plane if $(p - p_o) \cdot N = 0$



Step 1: Ray-Plane Intersection

- A plane is defined by:
  - p_o ⇒ a point on the plane (can use any vertex of the triangle)
  - N ⇒ a normal vector to the plane (can use the triangle normal)
- A point p is on the plane if $(p - p_o) \cdot N = 0$

- Take our ray $R(t) = A + (P - A)t$ and solve for *t* to find the smallest intersection: $(R(t) - p_o) \cdot N = 0$

Step 1: Ray-Plane Intersection

- A plane is defined by:
  - p_o ⇒ a point on the plane (can use any vertex of the triangle)
  - N ⇒ a normal vector to the plane (can use the triangle normal)
- A point p is on the plane if $(p - p_o) \cdot N = 0$

- Take our ray $R(t) = A + (P - A)t$ and solve for *t* to find the smallest intersection:

$$(R(t) - p_o) \cdot N = 0$$

$$(A - p_o) \cdot N + (P - A) \cdot Nt = 0$$

$$t = \frac{(p_o - A) \cdot N}{(P - A) \cdot N}$$

Step 1: Ray-Plane Intersection

- Our ray $R(t) = A + (P - A)t$ intersects the plane $(p - p_o) \cdot N = 0$ when:

$$t = \frac{(p_o - A) \cdot N}{(P - A) \cdot N}$$

- Remember to restrict to the frustum: $t \in [1, t_{far}]$

Step 1: Ray-Plane Intersection

- Our ray $R(t) = A + (P - A)t$ intersects the plane $(p - p_o) \cdot N = 0$ when:

$$t = \frac{(p_o - A) \cdot N}{(P - A) \cdot N}$$

- Remember to restrict to the frustum: $t \in [1, t_{far}]$

- N is a vector, so you can't cancel
  - The lengths cancel, though, so the normal doesn't have to be a unit vector

Step 1: Ray-Plane Intersection

- Our ray $R(t) = A + (P - A)t$ intersects the plane $(p - p_o) \cdot N = 0$ when:

$$t = \frac{(p_o - A) \cdot N}{(P - A) \cdot N}$$

- Remember to restrict to the frustum: $t \in [1, t_{far}]$

- N is a vector, so you can't cancel
  - The lengths cancel, though, so the normal doesn't have to be a unit vector
- Once we have $t \in [1, t_{far}]$, we plug it into $R(t)$for our intersection

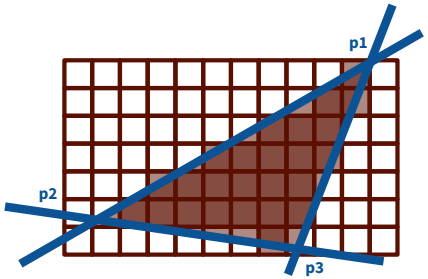
Ray-Triangle Intersection

- Recall: most of our object will be **triangle meshes**
  - We need to intersect our ray with a triangle
  - Triangles are **planar** (3 points are guaranteed to make a plane)

- One technique:
  - ~~1) Determine how a ray intersects a **plane** for an intersection point~~ (Done!)
  - 2) Then, check if this intersection point is **inside the triangle**
- Another approach:
  - Consider 3D ray-object intersection directly

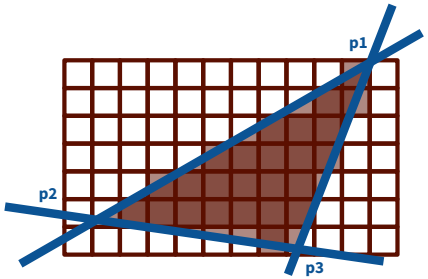
Step 2: Project Triangle & Intersection to 2D

- One approach:
  - Once we have the ray-plane intersection, project both the intersection point and triangle into **2D**
  - E.g. project onto the xy-plane by dropping the z-coordinates of both the intersection point & triangle vertices



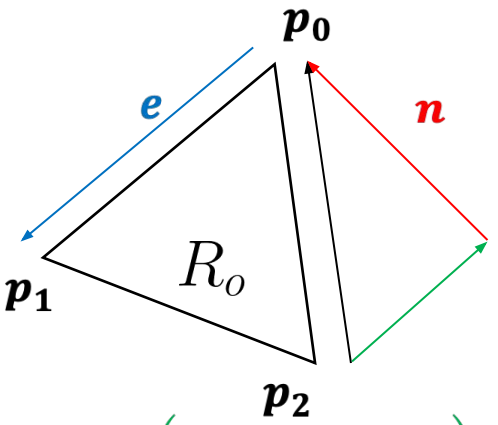
Step 2: Project Triangle & Intersection to 2D

- One approach:
  - Once we have the ray-plane intersection, project both the intersection point and triangle into **2D**
  - E.g. project onto the xy-plane by dropping the z-coordinates of both the intersection point & triangle vertices
  - More robustly: drop the coordinate that has the largest component in the triangle's normal vector
    - Then use techniques from 2D rasterization!



Alt. Step 2: 3D Point Inside 3D Triangle

- Check if the intersection point is in the triangle using 3D geometry

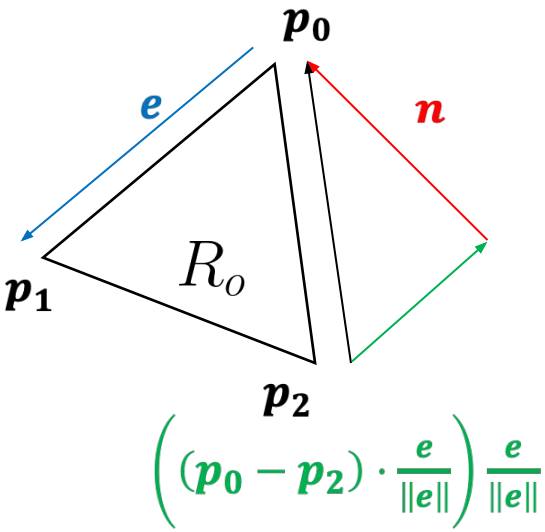


Alt. Step 2: 3D Point Inside 3D Triangle

- Check if the intersection point is in the triangle using 3D geometry

- Let R_o be our intersection point
- Take a directed edge *(e)* on our triangle:

$$e = p_1 - p_o$$



Alt. Step 2: 3D Point Inside 3D Triangle

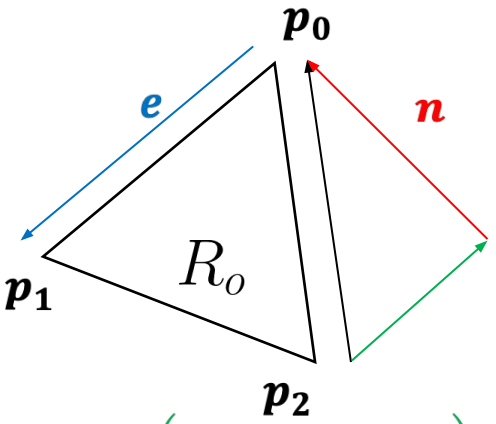
- Check if the intersection point is in the triangle using 3D geometry

- Let R_o be our intersection point
- Take a directed edge *(e)* on our triangle:

$$e = p_1 - p_o$$

- Compute a normal to edge as:

$$n = (p_o - p_2) - \left((p_o - p_2) \cdot \frac{e}{||e||} \right) \frac{e}{||e||}$$



Alt. Step 2: 3D Point Inside 3D Triangle

- Check if the intersection point is in the triangle using 3D geometry

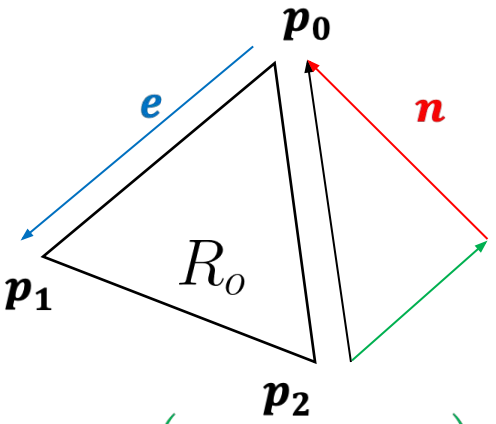
- Let R_o be our intersection point
- Take a directed edge *(e)* on our triangle:

$$e = p_1 - p_o$$

- Compute a normal to edge as:

$$n = (p_o - p_2) - \left((p_o - p_2) \cdot \frac{e}{||e||} \right) \frac{e}{||e||}$$

- R_o is interior to edge if $(R_o - p_o) \cdot n < 0$

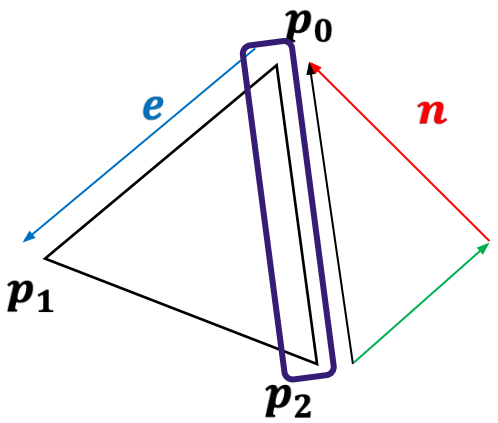


Alt. Step 2: 3D Point Inside 3D Triangle

- Check if the intersection point is in the triangle using 3D geometry

$$n = (p_o - p_2) - \left((p_o - p_2) \cdot \frac{e}{||e||} \right) \frac{e}{||e||}$$

Vector pointing from p2 to p0 (across triangle)



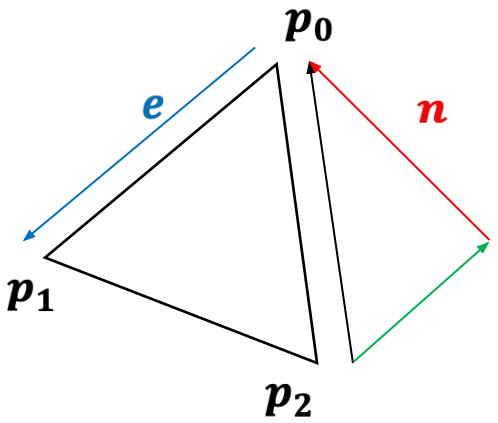
Alt. Step 2: 3D Point Inside 3D Triangle

- Check if the intersection point is in the triangle using 3D geometry

$$n = (p_o - p_2) - \left((p_o - p_2) \cdot \frac{e}{||e||} \right) \frac{e}{||e||}$$

Vector pointing from p2 to p0 (across triangle)

The component of vector (p0-p2) that is a "shadow" (projection) of e



Alt. Step 2: 3D Point Inside 3D Triangle

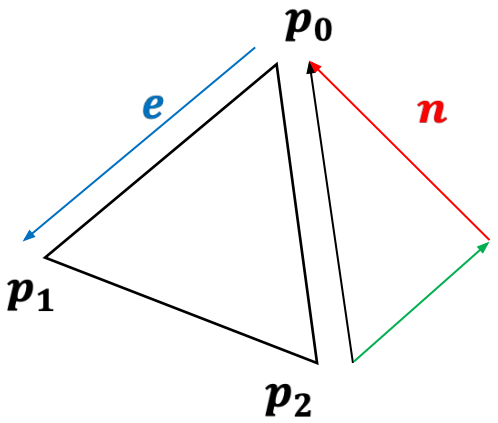
- Check if the intersection point is in the triangle using 3D geometry

$$n = (p_o - p_2) - \left((p_o - p_2) \cdot \frac{e}{||e||} \right) \frac{e}{||e||}$$

Vector pointing from p2 to p0 (across triangle)

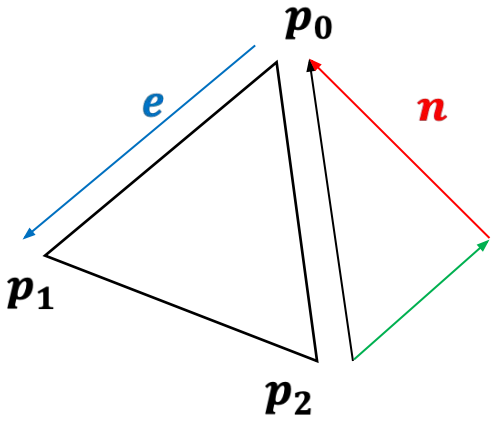
The component of vector (p0-p2) that is a "shadow" (projection) of e

Subtracting the projection removes the "along the edge" part, leaving only the perpendicular part

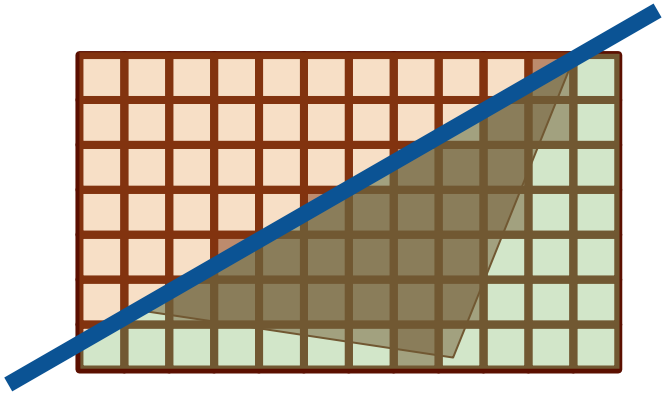


Alt. Step 2: 3D Point Inside 3D Triangle

- For ray-plane intersection: $(R(t) - p_o) \cdot N = 0$
- R_o is interior to ray if $(R_o - p_o) \cdot n < 0$
- If interior to all 3 rays, then interior to triangle!



$$n = (p_o - p_2) - \left((p_o - p_2) \cdot \frac{e}{||e||} \right) \frac{e}{||e||}$$



Each edge of the triangle splits space into halves. Take the intersection of 3 half-planes!

Ray-Triangle Intersection

- Recall: most of our object will be **triangle meshes**
  - We need to intersect our ray with a triangle
  - Triangles are **planar** (3 points are guaranteed to make a plane)

- One technique:
  - ~~1) Determine how a ray intersects a **plane** for an intersection point~~ (Done!)
  - ~~2) Then, check if this intersection point is **inside the triangle**~~
- Another approach:
  - Consider 3D ray-object intersection directly

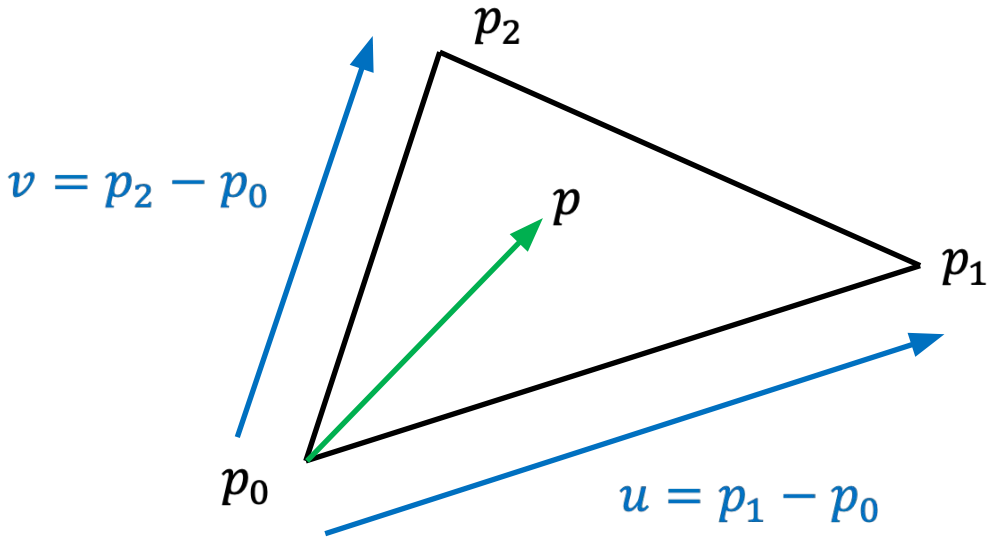
Triangle Basis Vectors

- Any point inside a triangle can be written as a sum of **one of the vertices + scalings of the edge vectors**

Triangle Basis Vectors

- Any point inside a triangle can be written as a sum of **one of the vertices + scalings of the edge vectors**

$$p = p_o + \beta_1 u + \beta_2 v \quad \text{with} \quad \beta_1, \beta_2 \in [0, 1], \beta_1 + \beta_2 \leq 1$$



Direct Ray-Triangle Intersection

- A point inside a triangle is given by: $p = p_o + \beta_1 u + \beta_2 v$

Direct Ray-Triangle Intersection

- A point inside a triangle is given by: $p = p_o + \beta_1 u + \beta_2 v$
- Substitute our ray: $R(t) = A + (P - A)t$

$$A + (P - A)t = p_o + \beta_1 u + \beta_2 v$$

Direct Ray-Triangle Intersection

- A point inside a triangle is given by: $p = p_o + \beta_1 u + \beta_2 v$
- Substitute our ray: $R(t) = A + (P - A)t$

$$A + (P - A)t = p_o + \beta_1 u + \beta_2 v$$

$$(u, v, A - P) \begin{pmatrix} \beta_1 \\ \beta_2 \\ t \end{pmatrix} = A - p_o$$

- Solve matrix equation for: $\beta_1, \beta_2 \in [0, 1], \beta_1 + \beta_2 \leq 1$
- And: $t \in [1, t_{far}]$

Ray-Object Intersections

- Reminder: scanline rendering needs triangle to rasterize
- Ray tracing generalizes well for non-triangular objects
  - … if there is a good geometric representation for our objects

Ray-Object Intersections

- Reminder: scanline rendering needs triangle to rasterize
- Ray tracing generalizes well for non-triangular objects
  - … if there is a good geometric representation for our objects

- Implicit surfaces can be represented as functions

- Example: **a sphere** (x^2 + y^2 + z^2 = r^2)

Ray-Sphere Intersections

- A point p on the sphere with center C and radius r satisfies:

$$|p - C| = r \quad \longrightarrow \quad (p - C) \cdot (p - C) = r^2$$

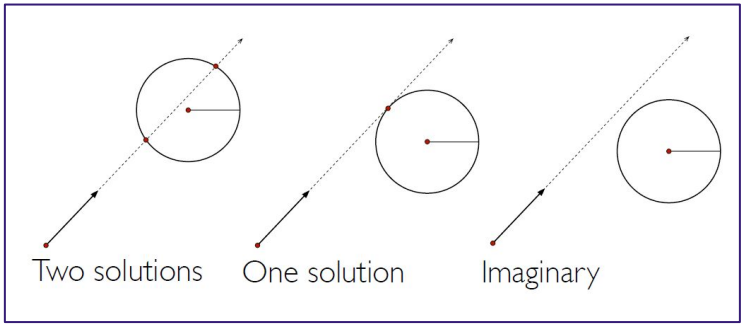
Ray-Sphere Intersections

- A point p on the sphere with center C and radius r satisfies:

$$|p - C| = r \quad \longrightarrow \quad (p - C) \cdot (p - C) = r^2$$

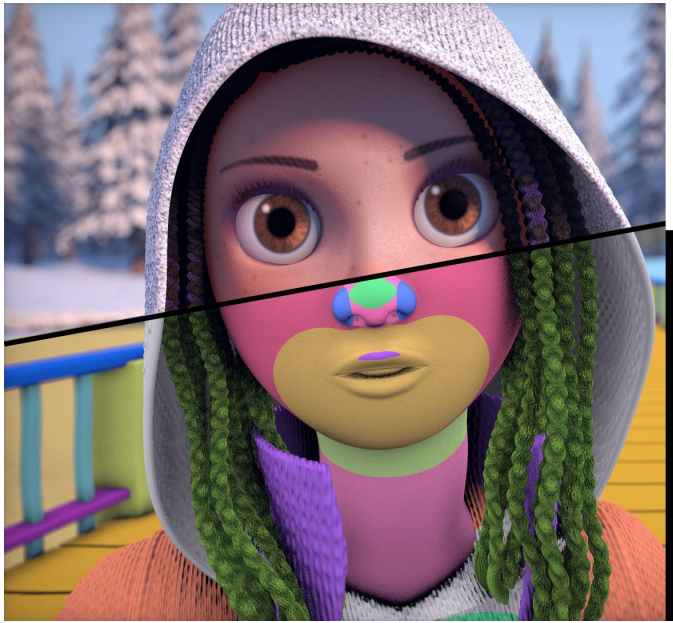
- Then we can substitute our ray ($R(t) = A + (P - A)t$) to make a quadratic equation:

$$(P - A) \cdot (P - A)t^2 + 2(P - A) \cdot (A - C)t + (A - C) \cdot (A - C) - r^2 = 0$$



Ray-Superquadric Intersections

- Whole professions dedicated to making shapes, figures, and objects from equations (rather than modeling)
- E.g. Iñigo Quilez (Shadertoy)



$$d = sdfEllipsoid(p, r)$$

$$\approx \left\| \frac{p}{r^2} \right\| \cdot \left(\left\| \frac{p}{r} \right\|^2 - \left\| \frac{p}{r} \right\| \right)$$

```
float sdEllipsoid( in vec3 p, in
{
    float k0 = length(p/r);
    float k1 = length(p/(r*r));
    return k0*(k0-1.0)/k1;
}

float sdBox( in vec3 p, in vec3 b
{
    vec3 d = abs(p) - b;
    return min( max(max(d.x,d.y),
}

float sdArc( in vec2 p, in vec2 s
{
    p.x = abs(p.x);
```

https://san-francisco.siggraph.org/2025/04/05/inigo-quilez-unlocking-creativity-with-signed-distance-fields/

Lecture Outline

- Vector Math Review
- Constructing rays
- Ray-Triangle & Ray-Object Interaction
  - Ray-Plane Interaction
  - Project Triangle & Intersection to 2D
- Parallelization & Optimization
- Transferring Normals
- Shadows & Spurious Self-Occlusion

Parallelization

- Historically, ray tracing was too slow for real time rendering
  - Optimization efforts focused on making real time scanline rendering

Parallelization

- Historically, ray tracing was too slow for real time rendering
  - Optimization efforts focused on making real time scanline rendering

- Now… we have GPUs, clusters, and parallel CPUs to speed it up
  - E.g. CUDA for GPU programming

Parallelization

- Historically, ray tracing was too slow for real time rendering
  - Optimization efforts focused on making real time scanline rendering

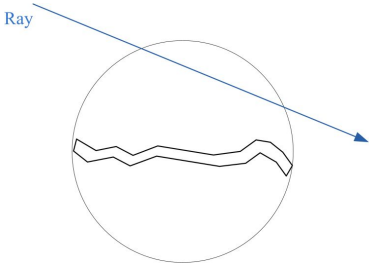
- Now… we have GPUs, clusters, and parallel CPUs to speed it up
  - E.g. CUDA for GPU programming
- Ray tracing is a per pixel operation, so it's inherently parallel
  - Each ray is independent of any other ray
  - Can assign nearby pixels/rays to the same core/processor
  - Put object data in shared memory

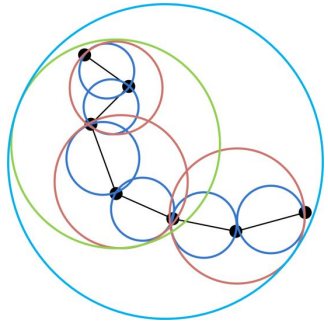
Code Acceleration

- Ray tracing: for each pixel, shoot a ray to see if it intersects a triangle
  - For every triangle, for every pixel: **nested for loop :(**

Code Acceleration

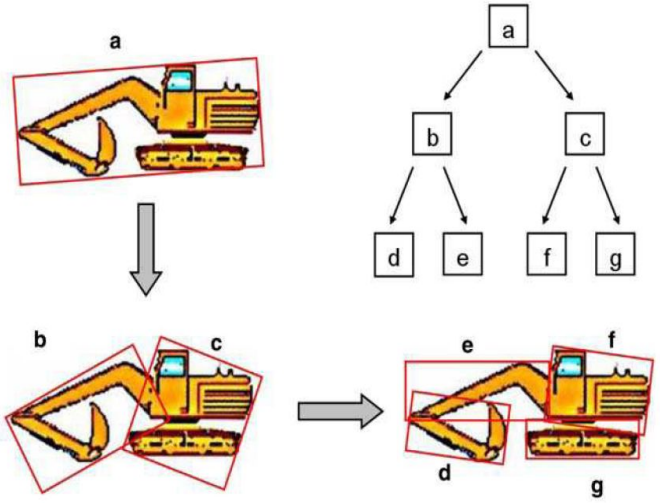
- Ray tracing: for each pixel, shoot a ray to see if it intersects a triangle
  - For every triangle, for every pixel: **nested for loop :(**
- Partial solution: surround objects in **bounding volumes**
  - First, see if the ray intersects the simpler bounding volume (instead of iterating over whole scene)
  - Then, worry about triangles in your object

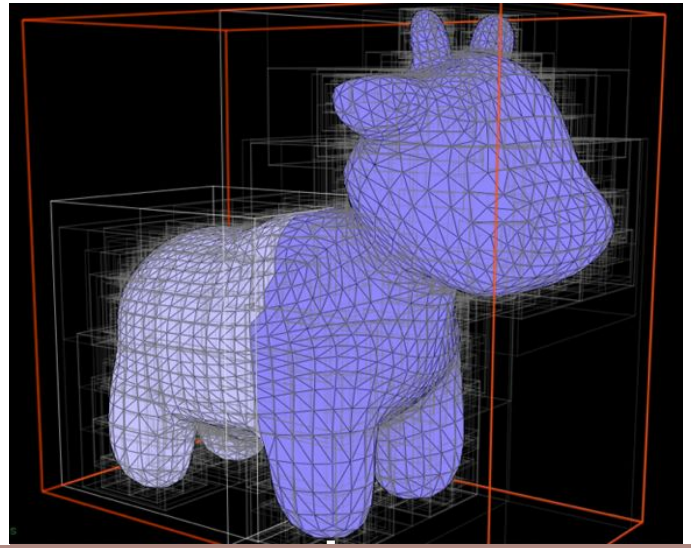




Bounding Volume Hierarchy (BVH)

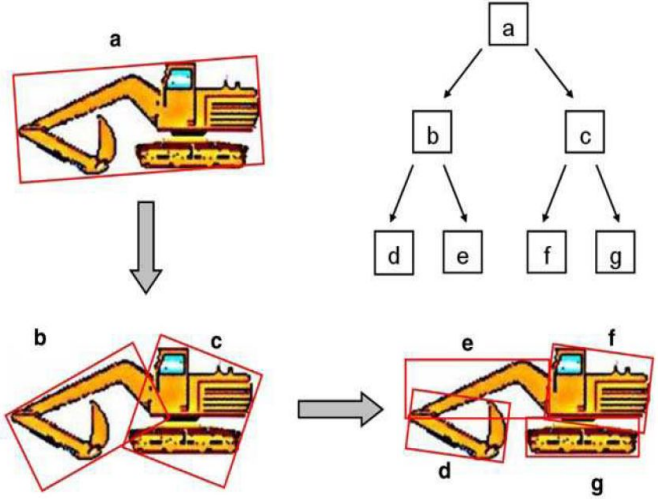
- Split bounding volumes into smaller bounding volumes
- Build a **bounding volume hierarchy** in **object space**
- O(n) triangle intersection operations speed up to…

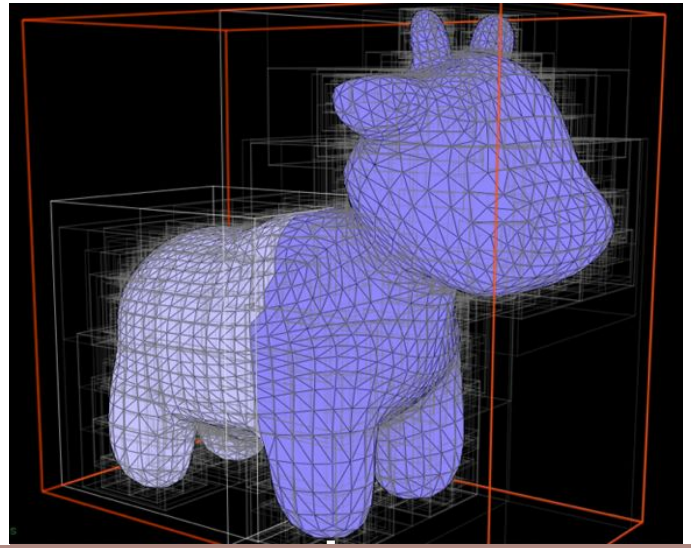




Bounding Volume Hierarchy (BVH)

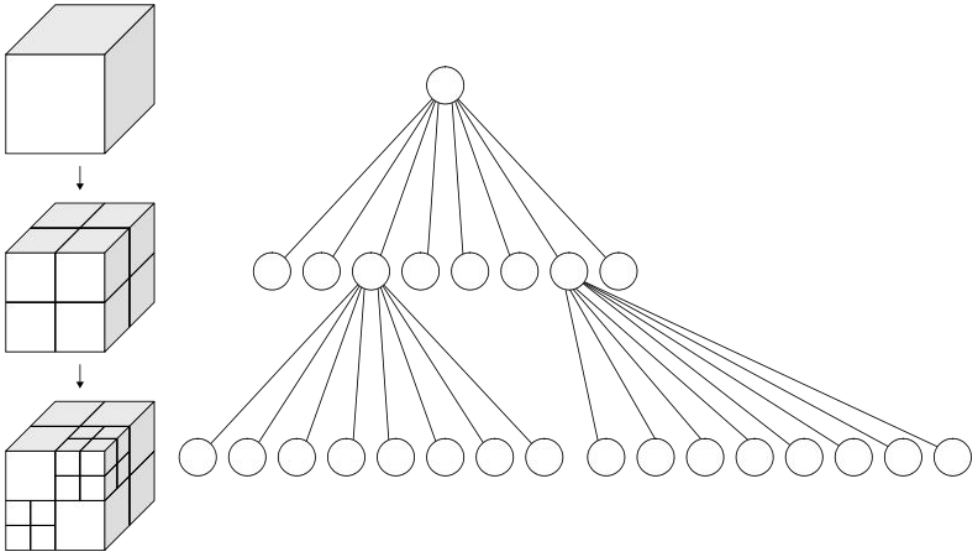
- Split bounding volumes into smaller bounding volumes
- Build a **bounding volume hierarchy** in **object space**
- O(n) triangle intersection operations speed up to… **O(log(n))**



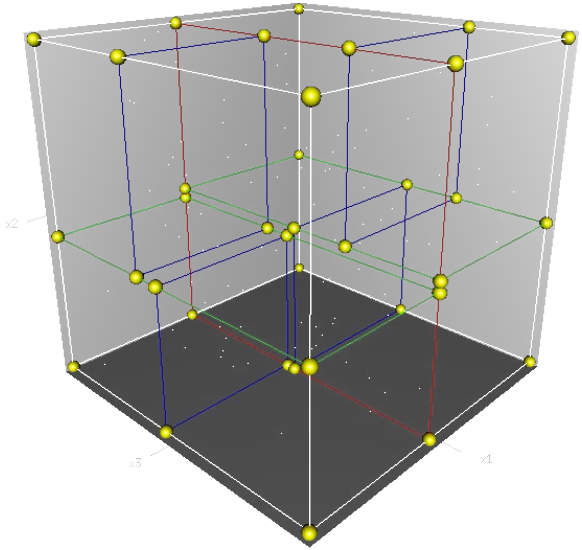


Bounding Volume Hierarchy (BVH)

- Can have more leaves in the tree



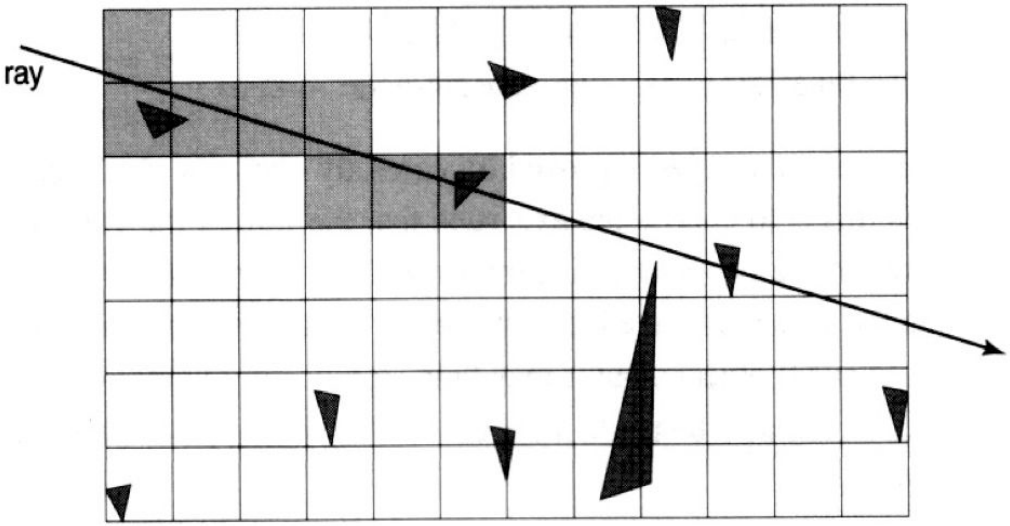
Octree: each volume split into 8 smaller volumes



K-D tree: each volume split into k smaller volumes

Uniform Partitions

- BVH can be expensive when there are many objects in the scene
- Solution: uniform special partitions (e.g. uniform grids)



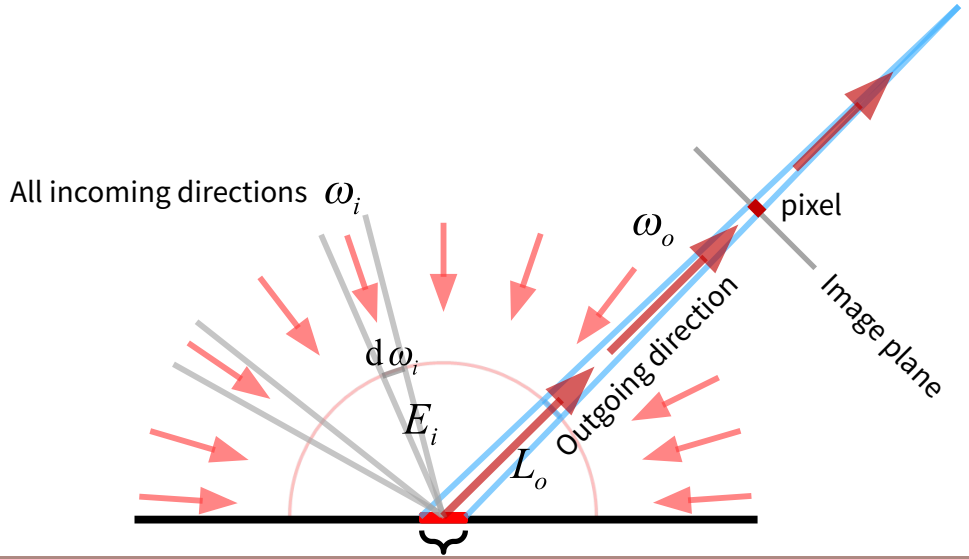
Lecture Outline

- Vector Math Review
- Constructing rays
- Ray-Triangle & Ray-Object Interaction
  - Ray-Plane Interaction
  - Project Triangle & Intersection to 2D
- Parallelization & Optimization
- Transferring Normals
- Shadows & Spurious Self-Occlusion

Recall: The Importance of the Normal

$$L_o(\omega_o) = \sum_{i \in in} L_o(\omega_i, \omega_o)$$

$$L_o(\omega_o) = \int_{i \in in} BRDF(\omega_i, \omega_o) L_i \cos\theta_i d\omega_i$$



Recall: The Importance of the Normal

$$L_o(\omega_o) = \sum_{i \in in} L_o(\omega_i, \omega_o)$$
$$L_o(\omega_o) = \int_{i \in in} BRDF(\omega_i, \omega_o) L_i \cos\theta_i d\omega_i$$

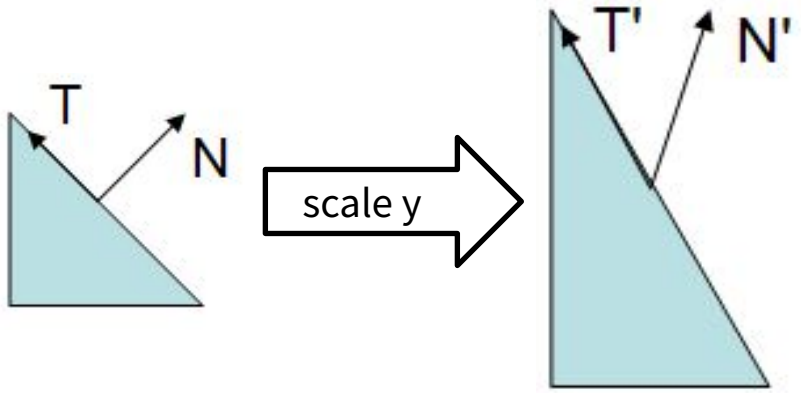
- $\theta \Rightarrow$ angle between incoming light direction and the **surface** normal

- We typically define normals in **object space**
- But lighting computations are done in **world space**

- We need a way to convert object space normals to world space normals to use our equation

Transforming Normals

- You can't directly apply a matrix transformation to normals
- If you scale *y*, the geometry stretches, but the normal points in the wrong direction (no longer perpendicular to the edge *u*)



- Goal: u · N = 0 (perpendicular)

Transforming Normals

- Goal: $u \cdot N = 0$ (perpendicular)
  - While applying transformation M to u
- What if we apply M^{-T} as our transformation for N?

Transforming Normals

- Goal: $u \cdot N = 0$ (perpendicular)
  - While applying transformation M to u
- What if we apply M^{-T} as our transformation for N?
- Let u be a triangle edge vector:

$$\begin{aligned} Mu \cdot M^{-T}\hat{N} &= (Mu)^T M^{-T}\hat{N} \\ &= u^T M^T M^{-T}\hat{N} = u^T\hat{N} = u \cdot \hat{N} = 0 \end{aligned}$$

  - It works!
- So the actual transform for N is: $\boxed{M^{-T}\hat{N}}$

Lecture Outline

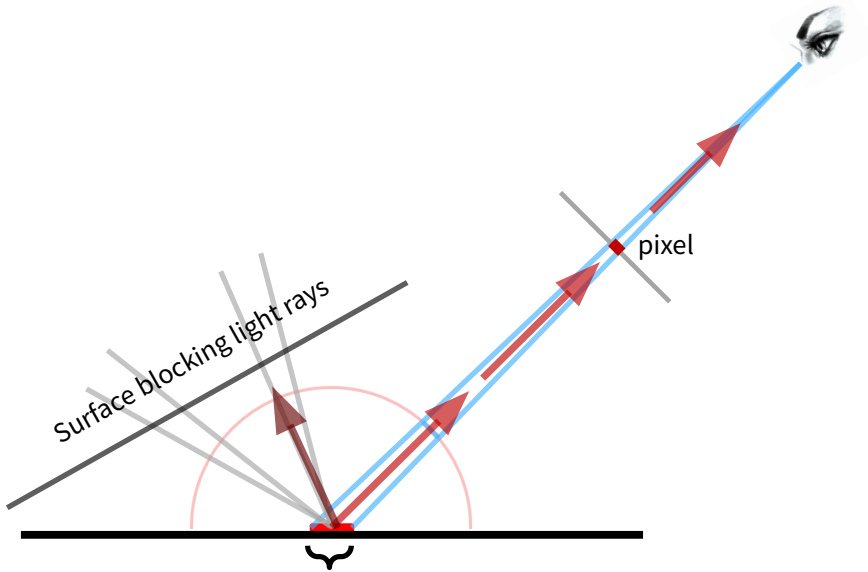
- Vector Math Review
- Constructing rays
- Ray-Triangle & Ray-Object Interaction
  - Ray-Plane Interaction
  - Project Triangle & Intersection to 2D
- Parallelization & Optimization
- Transferring Normals
- Shadows & Spurious Self-Occlusion

High-Level Idea: Shadows

- From the intersection point, shoot a new ray toward the light source
- If the ray hits something before reaching the light, we know the intersection point is in shadow (skip that light's contribution)
- Shadow ray: "should I add this particular direct light's contribution?"

Shadow Rays

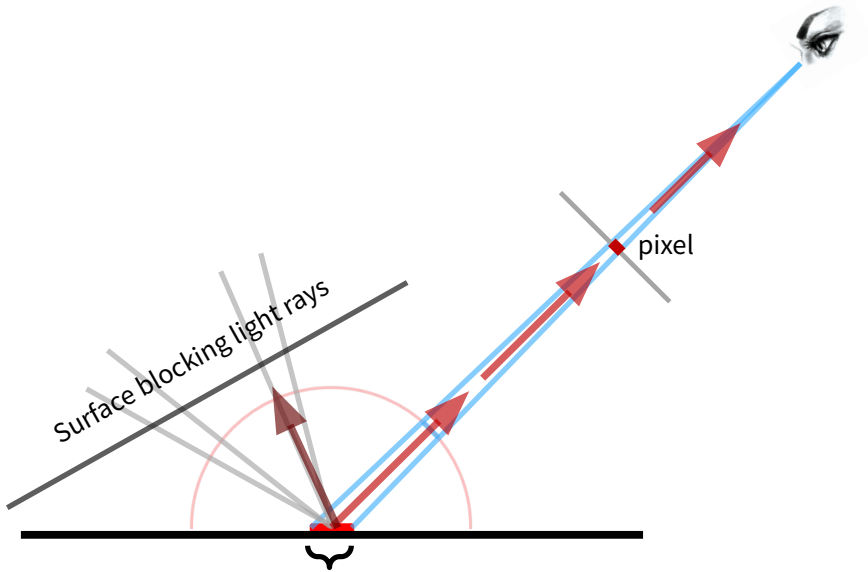
- Incoming light rays may get obscured by objects blocking the intersection
- For each light, cast a shadow ray in the direction of the light
  - Ray origin: intersection point
  - Direction: toward the light



Shadow Rays

- Incoming light rays may get obscured by objects blocking the intersection
- For each light, cast a shadow ray in the direction of the light
  - Ray origin: intersection point
  - Direction: toward the light

$$S(t) = R(t_{int}) - \hat{L}t$$



Shadow Rays

- For each light, cast a shadow ray in the direction of the light:

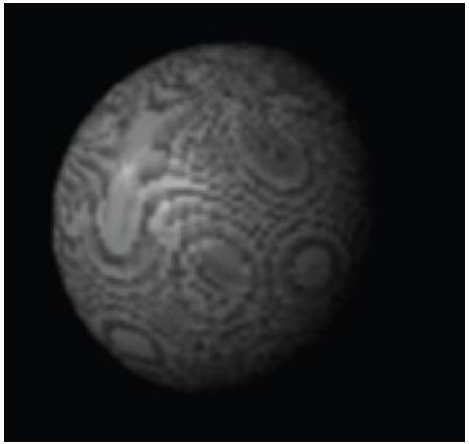
$$S(t) = R(t_{int}) - \hat{L}t$$
$$t \in (0, t_{light})$$

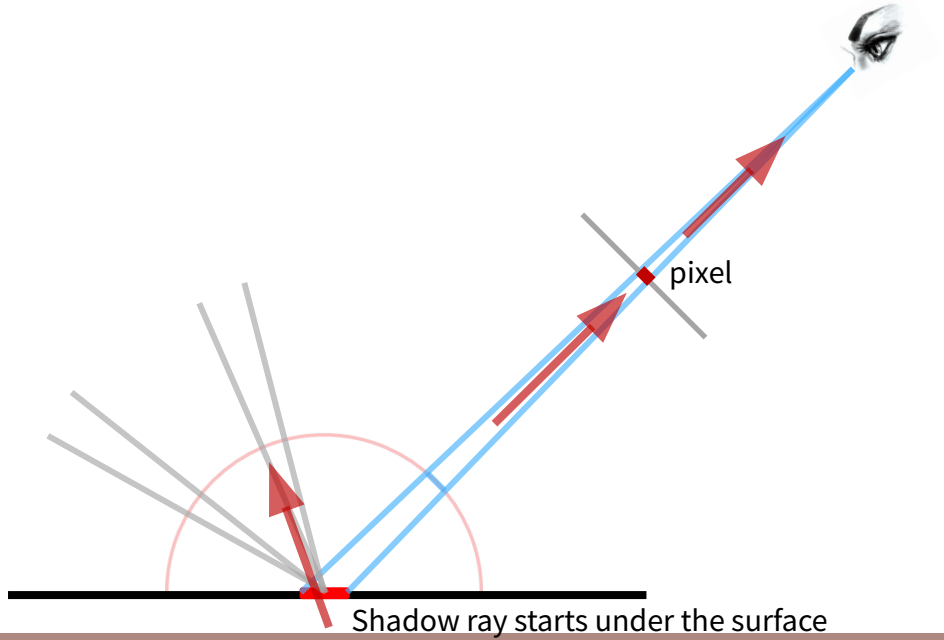
- Then we have the exact same ray tracing process we've been discussing
- If no ray-object intersection for $0 < t < t_{light}$, then do usual lighting
- Else, an object is blocking the light to the intersection, so there's **0 radiance coming from that blocked light**

Spurious Self-Occlusion

Spurious Self-Occlusion

- Shadow ray: “should I add this particular direct light’s contribution?”
- Due to numerical imprecision, the shadow ray might start underneath the surface





Spurious Self-Occlusion

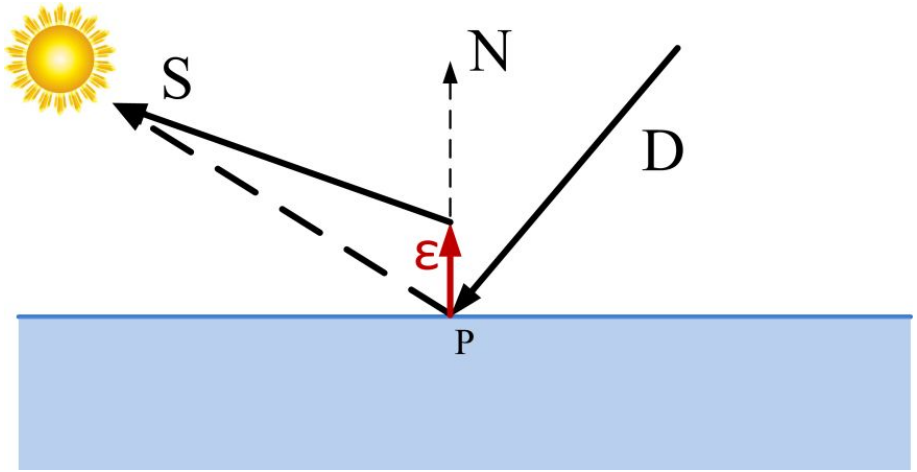
- Solution:

Spurious Self-Occlusion

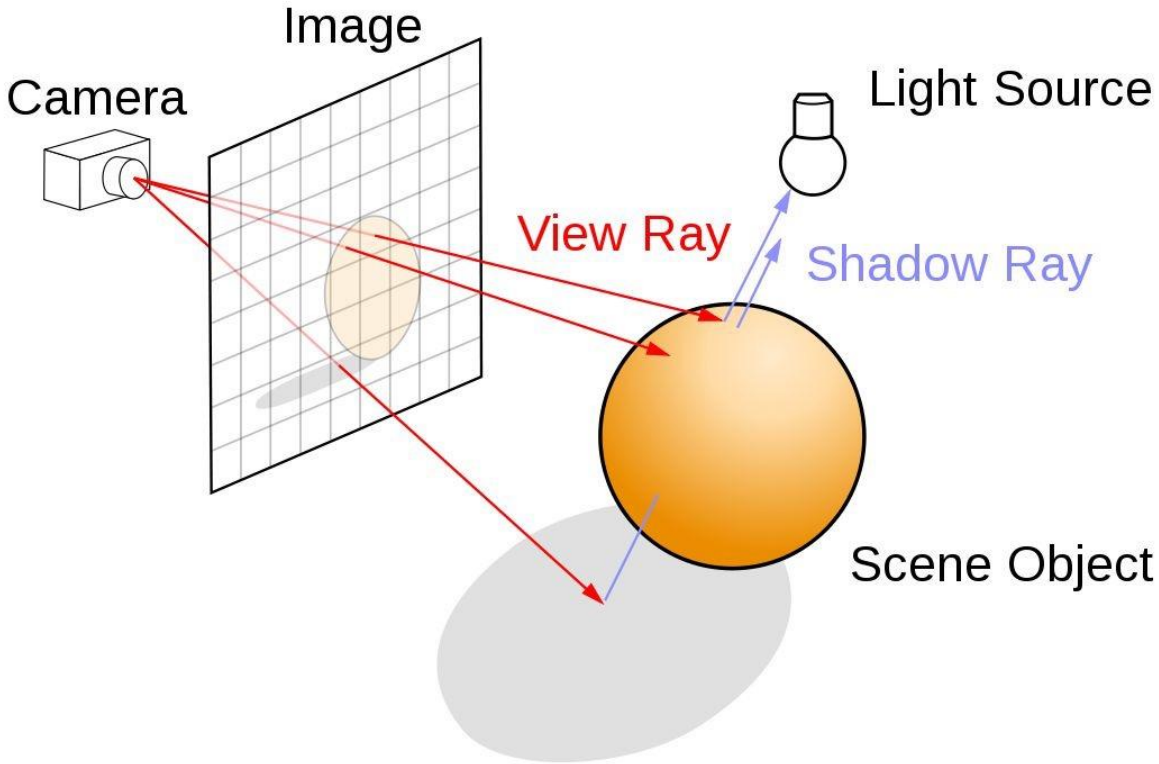
- Solution: perturb by a small **epsilon** in the **normal direction**

$$S(t) = (R(t_{int}) + \epsilon\hat{N}) - \hat{L}_{mod}t$$

- Light direction needs to be modified to start at: $(R(t_{int}) + \epsilon\hat{N})$



Ray tracing… more to come!



Additional Resources + Exploring!

- Iñigo Quilez: Unlocking Creativity with Signed Distance Fields
- NVIDIA Ray Tracing Guide
- Occlusion handling in video object tracking

- Fundamentals of Computer Graphics, Steve Marschner and Peter Shirley
  - Preview, Book