



---

# Ray Casting



# Ray Casting Algorithm

---

For each pixel

1. Compute ray from eye through pixel
2. For each primitive
  - Test for ray-object intersection
3. Shade pixel using nearest primitive (or set to background color)



# Computing the Rays

---

**Choose eye point, view direction, up direction, fields of view (x and y)**

$p_t = \text{eye} + t * v$  ( $v$  typically normalized)

**Compute rays to two opposite corners**

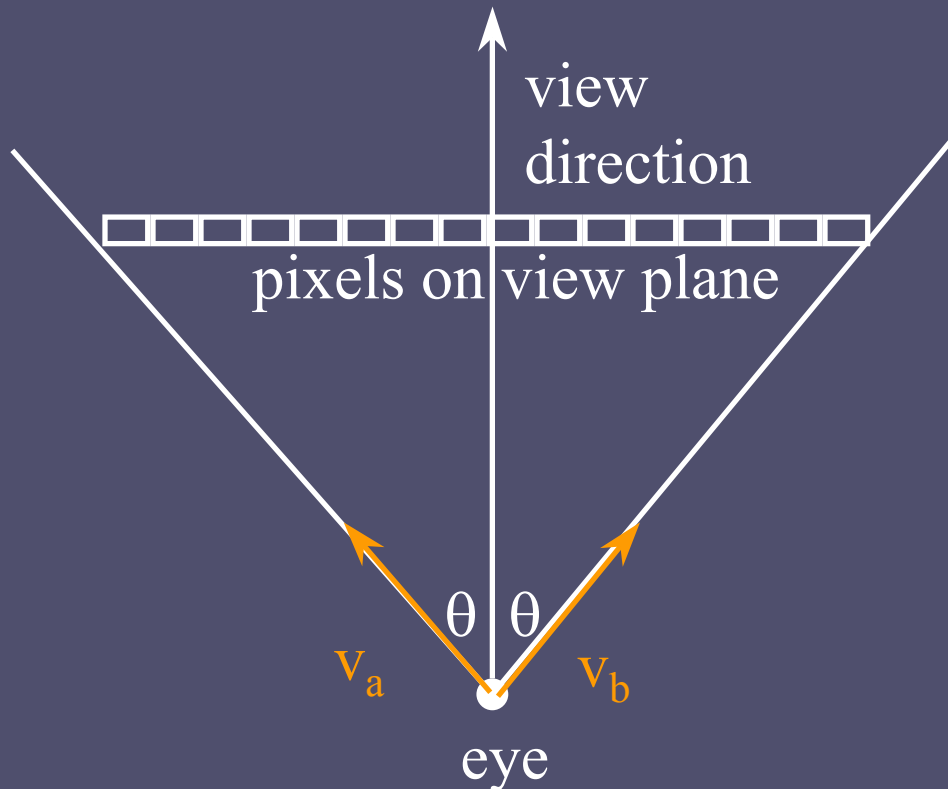
**Compute step sizes,  $\Delta x$  and  $\Delta y$  to go from pixel to pixel**

**To compute new ray: take step, then normalize**

---



# 2D ray calculation



Note: take equal-sized steps in viewing plane, not equal angles!

**view** is normalized  
view direction

$$\mathbf{right} = (\mathbf{view}_y, -\mathbf{view}_x)$$

$$\mathbf{v}_a = \mathbf{view} - \tan\theta * \mathbf{right}$$

$$\mathbf{v}_b = \mathbf{view} + \tan\theta * \mathbf{right}$$

$$\mathbf{step} = (\mathbf{v}_b - \mathbf{v}_a) / \text{num\_pixels}$$

$$\mathbf{v}_0 = \mathbf{v}_a + \mathbf{step} / 2$$

$$\mathbf{v}_i = \mathbf{v}_{i-1} + \mathbf{step}$$

In 3D, we have an additional step size and field-of-view angle as well as an up vector.



# Computing Intersections

---

Ray is in **parametric** form ( $t$  is parameter)

$$\mathbf{x} = \mathbf{p} + t \cdot \mathbf{v}$$

Represent primitive in **implicit** form:

$$f(\mathbf{x}, \mathbf{y}, \mathbf{z}) = 0$$

(any  $(\mathbf{x}, \mathbf{y}, \mathbf{z})$  on surface evaluates to zero)

Substitute  $(\mathbf{x}, \mathbf{y}, \mathbf{z})$  of ray into  $f(\mathbf{x}, \mathbf{y}, \mathbf{z})$  and solve for  $t$

- degree  $n$  implicit function will be degree  $n$  in  $t$
- quadric surfaces may be solved with quadratic equation -- pick real solution closest to eye



# Example Quadric Functions

---

**Sphere:**  $(x-a)^2 + (y-b)^2 + (z-c)^2 - r^2 = 0$

**Circular cylinder (parallel to z-axis):**

$$(x-a)^2 + (y-b)^2 - r^2 = 0$$

**Hyperbolic paraboloid:**

$$y^2/b^2 - x^2/a^2 - z = 0$$



# General Quadrics

General quadric has form:

$$Ax^2 + 2Bxy + 2Cxz + 2Dx + Ey^2 + 2Fyz + 2Gy + Hz^2 + 2Iz + J = 0$$

or...

$$x^t Q x = 0, \quad \text{where } x^t = [x \ y \ z \ 1] \text{ and}$$

$$Q = \begin{bmatrix} A & B & C & D \\ B & E & F & G \\ C & F & H & I \\ D & G & I & J \end{bmatrix}$$



# Quadric Intersections

---

**Quadric:**  $x^t Q x = 0$

**Ray:**  $x = p + tv$

**Substituting ray for  $x$ :**

$$(p + tv)^t Q (p + tv) = 0$$

$$p^t Q p + p^t Q tv + tv^t Q p + tv^t Q tv = 0$$

$$(v^t Q v)t^2 + (p^t Q v + v^t Q p)t + p^t Q p = 0$$

$$(v^t Q v)t^2 + (2v^t Q p)t + p^t Q p = 0$$

**( $Q$  is symmetric)**



# Common Ray-tracing Primitives

---

Sphere, ellipsoid

Cylinders

Plane, triangle

- $Ax + By + Cz + D = 0$

Torus

Bezier/Nurbs patches

- parametric, so use implicit form of ray

—intersection of two planes

---



# Local Illumination Shading

---

Compute normal at closest intersection

- $\nabla f = (\partial x, \partial y, \partial z)$  is normal vector field for implicit function,  $f$

For each light

- Use position and normal to compute light contribution
- Accumulate light contributions

Color pixel

- Clamp to avoid overflow
-



# Shadows

---

Only add contribution from a light if it is **visible** from the point (and vice versa)

- test for intersections along ray in L direction
- accumulate contribution if no occlusion

(illumination is no longer totally local)



# Truncating Primitives

---

Use another implicit function

- Test which **side** of the implicit function the intersection is on
- Keep intersection only if it is on the correct side

For example, truncate a cylinder using two plane equations (or perhaps a sphere)

- then cap using the two planes truncated by the cylinder



# Constructive Solid Geometry

---

Perform hierarchical set operations on primitives

Union:  $\cup$

Intersection:  $\cap$

Difference:  $-$



# CSG Operators

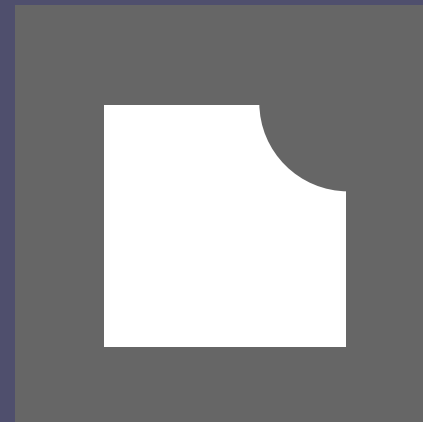
Square  $\cup$  Circle =



Square  $\cap$  Circle =

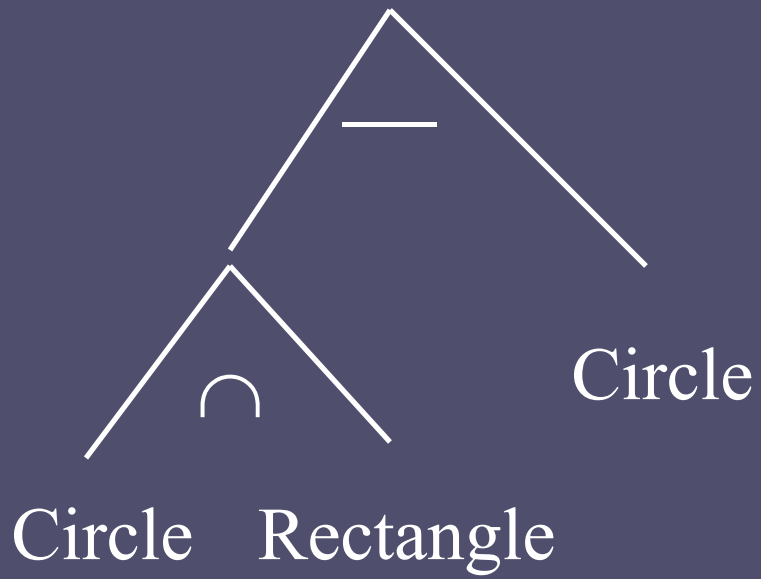


Square  $\text{---}$  Circle =





# CSG Hierarchy



=





# Ray Tracing CSG

---

Each “object” may be a primitive or a CSG hierarchy

Find **all** ray-primitive intersections for hierarchy

Use CSG operators to determine which **intervals** are solid or vacant

Use start of nearest solid interval as ray-object intersection

---



# CSG Tracing Algorithm

---

**Start at root of CSG Hierarchy**

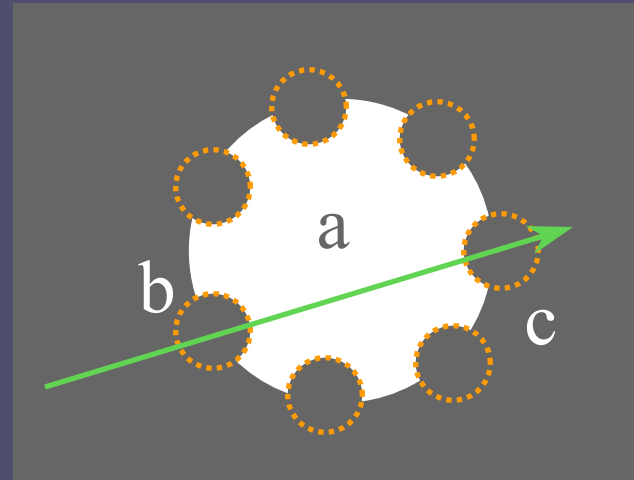
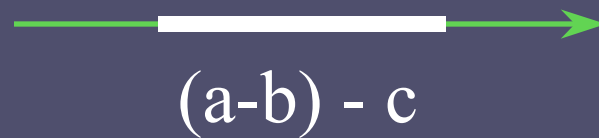
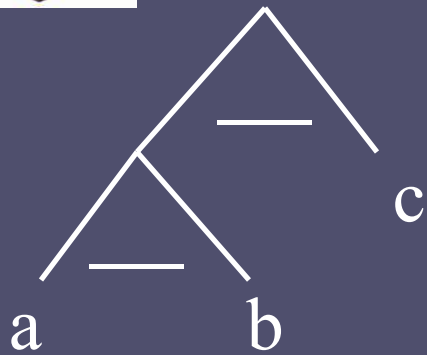
**Trace ray through left child - result is ordered list of intersections, forming solid and vacant intervals**

**Trace ray through right child**

**Merge lists of intersections/intervals by applying CSG operator of current node**



# CSG Example - golf ball





## Some CSG Details

---

**Each interval endpoint associated with intersection of ray with some surface**

**Normal computed from surface of intersection**

**Material parameters may come from either primitive**