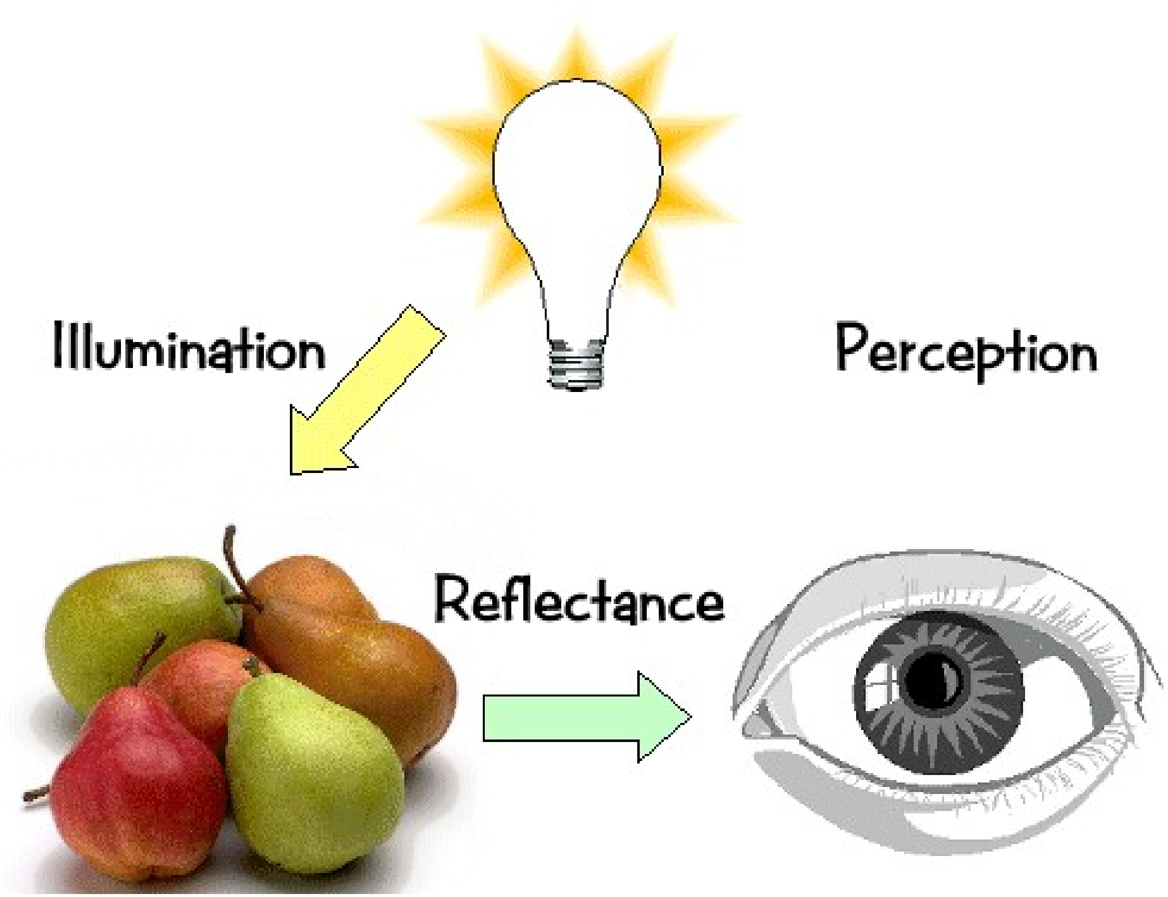
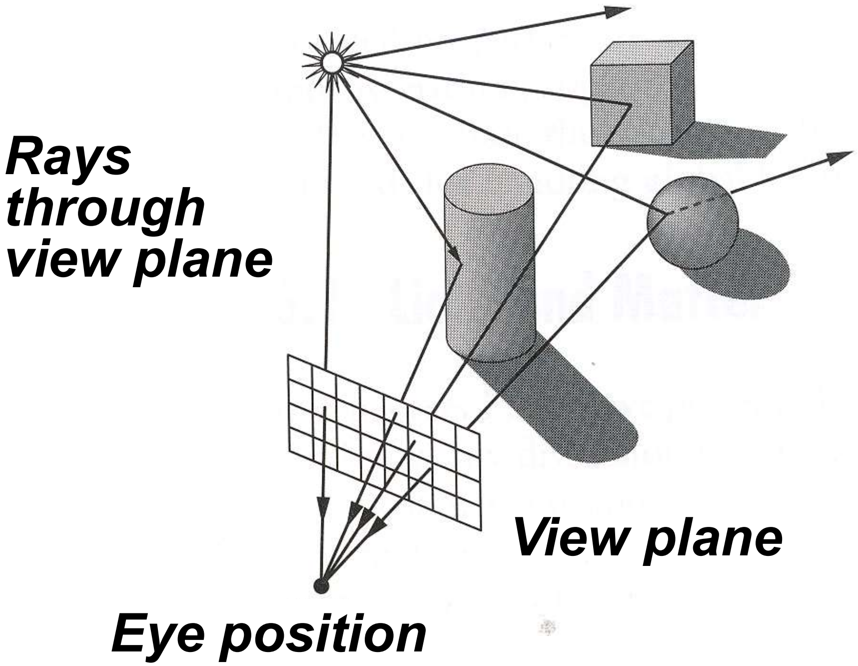

Illumination
Perception
Reflectance

- Ray Casting
- Ray-Surface Intersection Testing
- Barycentric Coordinates

3D Rendering

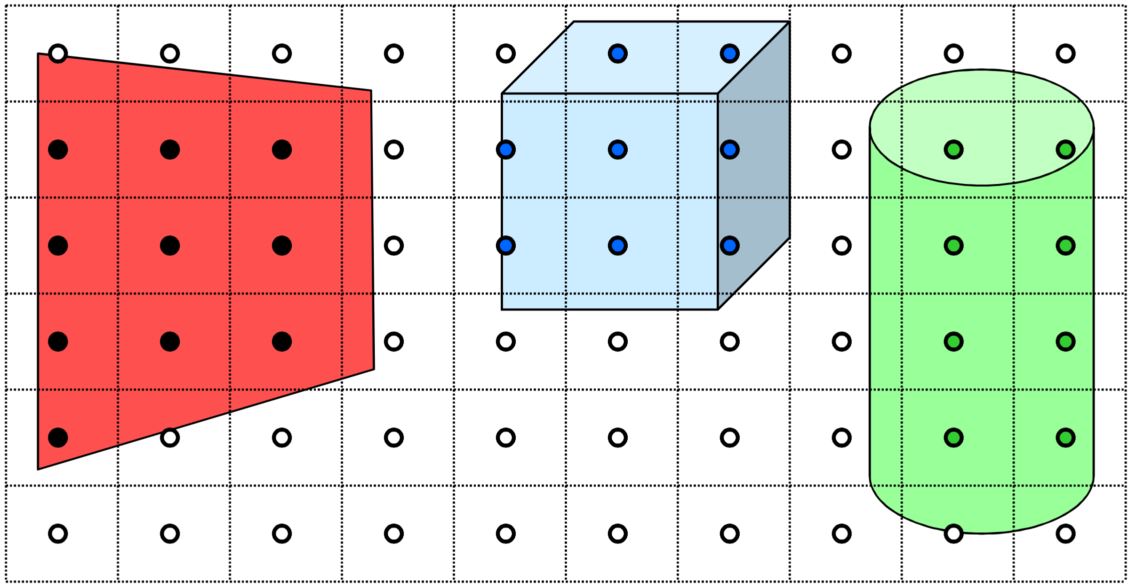
- The color of each pixel on the view plane depends on the radiance emanating from visible surfaces



Simplest method is ray casting

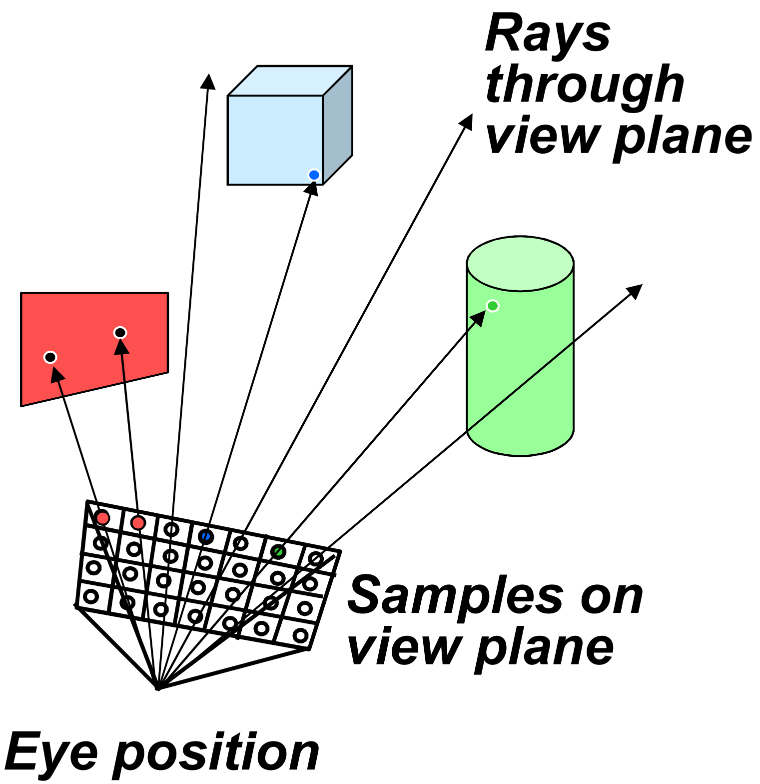
Ray Casting

- For each sample …
  - Construct ray from eye position through view plane
  - Find first surface intersected by ray through pixel
  - Compute color sample based on surface radiance



Ray Casting

- For each sample …
  - Construct ray from eye position through view plane
  - Find first surface intersected by ray through pixel
  - Compute color sample based on surface radiance



Ray Casting

- A very flexible visibility algorithm

  loop y

   loop x

      shoot ray from eye point through pixel (x, y) into scene

      intersect with all surfaces, find first one the ray hits

      shade that surface point to compute pixel (x, y)’ s color

A Simple Ray Caster Program

```
Raycast()                  // generate a picture
    for each pixel x,y
        color(pixel) = Trace(ray_through_pixel(x,y))

Trace(ray)                 // fire a ray, return RGB radiance
                                     // of light traveling backward along it
    object_point = Closest_intersection(ray)
    if object_point return Shade(object_point, ray)
    else return Background_Color

Closest_intersection(ray)
    for each surface in scene
            calc_intersection(ray, surface)
    return the closest point of intersection to viewer
    (also return other info about that point, e.g., surface normal,
       material properties, etc.)

Shade(point, ray)          // return radiance of light leaving
                                     // point in opposite of ray direction
    calculate surface normal vector
    use Phong illumination formula (or something similar)
    to calculate contributions of each light source
```

Ray Casting

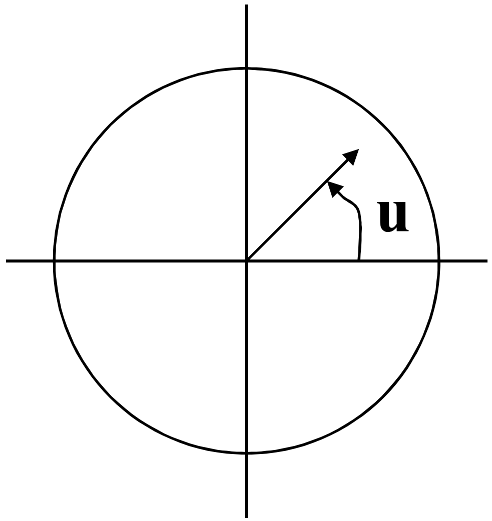
- This can be easily generalized to give recursive *ray tracing*, that will be discussed later
- calc_intersection (ray, surface) is the most important operation
  - compute not only coordinates, but also geometric or appearance attributes at the intersection point

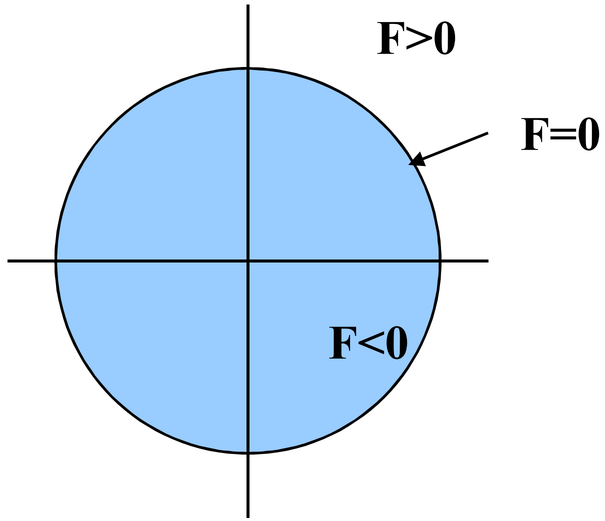
Ray-Surface Intersections

- How to represent a ray?
  - A ray is p+td: p is ray origin, d the direction
  - t=0 at origin of ray, t>0 in positive direction of ray
  - typically assume ||d||=1
  - p and d are typically computed in world space

Ray-Surface Intersections

- Surfaces can be represented by:
  - Implicit functions: $f(\mathrm{x}) = 0$
  - Parametric functions: $\mathrm{x} = g(u, v)$





Parametric

$$\mathbf{x(u) = r\cos(u)}$$
$$\mathbf{y(u) = r\sin(u)}$$

Implicit

$$\mathbf{F(x,y) = x^2 + y^2 - r^2}$$

Ray-Surface Intersections

- Compute Intersections:
  - Substitute ray equation for x
  - Find roots
  - Implicit: $f(\mathrm{p} + t\mathrm{d}) = 0$
    - » one equation in one unknown – univariate root finding
  - Parametric: $\mathrm{p} + t\mathrm{d} - g(u, v) = 0$
    - » three equations in three unknowns (t, u, v) – multivariate root finding
  - For univariate polynomials, use closed form solution otherwise use numerical root finder

The Devil's in the Details

- General case: non-linear root finding problem
- Ray casting is simplified using object-oriented techniques
  - Implement one intersection method for each type of surface primitive
  - Each surface handles its own intersection
- Some surfaces yield closed form solutions
  - quadrics: spheres, cylinders, cones, ellipsoids, etc…)
  - Polygons
  - tori, superquadrics, low-order spline surface patches

Ray-Sphere Intersection

- Ray-sphere intersection is an easy case
- A sphere's implicit function is: $x^2+y^2+z^2-r^2=0$ if sphere at origin
- The ray equation is: $x = p_x+td_x$
  $y = p_y+td_y$
  $z = p_z+td_z$
- Substitution gives: $(p_x+td_x)^2 + (p_y+td_y)^2 + (p_z+td_z)^2 - r^2 = 0$
- A quadratic equation in t.
- Solve the standard way: $A = d_x^2+d_y^2+d_z^2 = 1$ (unit vector)
  $B = 2(p_xd_x+p_yd_y+p_zd_z)$
  $At^2+Bt+C=0$
  $C = p_x^2+p_y^2+p_z^2 - r^2$
- Quadratic formula has two roots: $t=(-B\pm sqrt(B^2-4C))/2$
  - which correspond to the two intersection points
  - negative discriminant means ray misses sphere

Ray-Polygon Intersection

- Assuming we have a planar polygon
  - first, find intersection point of ray with plane
  - then check if that point is inside the polygon
- Latter step is a point-in-polygon test in 3-D:
  - inputs: a point x in 3-D and the vertices of a polygon in 3-D
  - output: INSIDE or OUTSIDE
  - problem can be reduced to point-in-polygon test in 2-D
- Point-in-polygon test in 2-D:
  - easiest for triangles
  - easy for convex n-gons
  - harder for concave polygons
  - most common approach: subdivide all polygons into triangles
  - for optimization tips, see article by Haines in the book *Graphics Gems IV*

Ray-Plane Intersection

- Ray: $x=p+td$
  - where p is ray origin, d is ray direction. we'll assume $||d||=1$ (this simplifies the algebra later)
  - $x=(x, y, z)$ is point on ray if $t>0$
- Plane: $(x-q)\bullet n=0$
  - where q is reference point on plane, n is plane normal. (some might assume $||n||=1$; we won't)
  - x is point on plane
  - if what you're given is vertices of a polygon
    - » compute n with cross product of two (non-parallel) edges
    - » use one of the vertices for q
  - rewrite plane equation as $x\bullet n+D=0$
    - » equivalent to the familiar formula $Ax+By+Cz+D=0$, where $(A, B, C)=n$, $D=-q\bullet n$
    - » fewer values to store

Ray-Plane Intersection

- Steps:
  - substitute ray formula into plane eqn, yielding 1 equation in 1 unknown (t).
  - solution: $t = -(\mathrm{p}\bullet\mathrm{n}+D)/(\mathrm{d}\bullet\mathrm{n})$
    - » note: if $\mathrm{d}\bullet\mathrm{n}=0$ then ray and plane are parallel - REJECT
    - » note: if $t<0$ then intersection with plane is behind ray origin - REJECT
  - compute t, plug it into ray equation to compute point x on plane

Projecting A Polygon from 3-D to 2-D

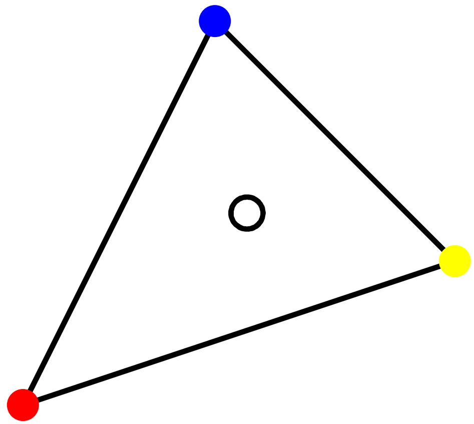
- Point-in-polygon testing is simpler and faster if we do it in 2-D
  - The simplest projections to compute are to the *xy*, *yz*, or *zx* planes
  - If the polygon has plane equation $Ax+By+Cz+D=0$, then
    - » $|A|$ is proportional to projection of polygon in *yz* plane
    - » $|B|$ is proportional to projection of polygon in *zx* plane
    - » $|C|$ is proportional to projection of polygon in *xy* plane
    - » Example: the plane $z=3$ has $(A, B, C, D)=(0, 0, 1, -3)$, so $|C|$ is the largest and *xy* projection is best. We should do point-in-polygon testing using *x* and *y* coords.
  - In other words, project into the plane for which the perpendicular component of the normal vector n is largest

Projecting A Polygon from 3-D to 2-D

- Optimization:
  - We should optimize the inner loop (ray-triangle intersection testing) as much as possible
  - We can determine which plane to project to, for each triangle, as a preprocess
- Point-in-polygon testing in 2-D is still an expensive operation
- Point-in-rectangle is a special case

Interpolated Shading for Ray Casting

- Suppose we know colors or normals at vertices
  - How do we compute the color/normal of a specified point inside?



- Color depends on distance to each vertex
  - How to do linear interpolation between 3 points?
  - Answer: *barycentric coordinates*
- Useful for ray-triangle intersection testing too!

Barycentric Coordinates in 1-D

- Linear interpolation between colors C_0 and C_1 by t

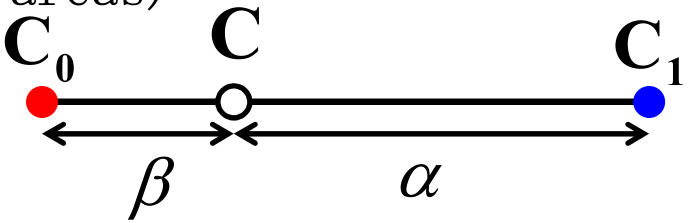
$$\mathbf{C} = (\mathbf{1} - t)\mathbf{C}_0 + t\mathbf{C}_1$$

- We can rewrite this as

$$\mathbf{C} = \alpha\mathbf{C}_0 + \beta\mathbf{C}_1 \qquad \textit{where } \alpha + \beta = 1$$

$$\mathbf{C} \text{ is between } \mathbf{C}_0 \text{ and } \mathbf{C}_1 \Leftrightarrow \alpha, \beta \in [0,1]$$

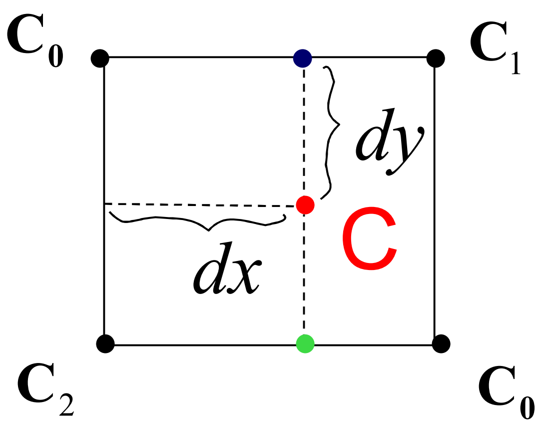
- Geometric intuition:
  - We are weighting each vertex by ratio of distances (or areas)



- α and β are called *barycentric* coordinates

Barycentric Coordinates in 2-D

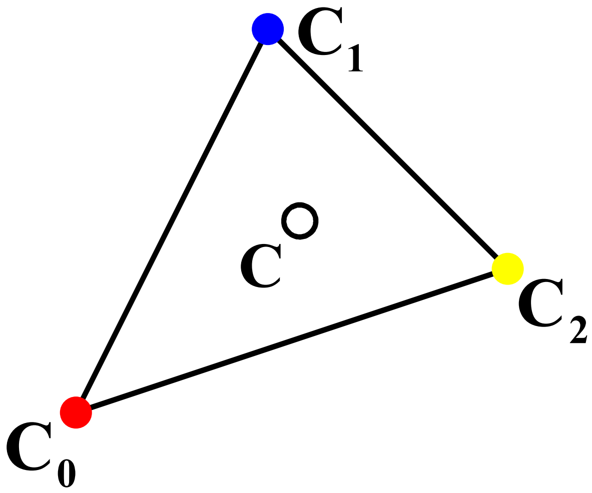
- Bilinear interpolation: 4 points instead of 2



$$\mathbf{C} = \underbrace{(1-dx)(1-dy)}_{\alpha}\mathbf{C}_0 + \underbrace{(dx(1-dy)}_{\beta}\mathbf{C}_1 + \underbrace{(1-dx)dy}_{\gamma}\mathbf{C}_2 + \underbrace{dxdy}_{\varphi}\mathbf{C}_3$$

Barycentric Coordinates in 2-D

- Now suppose we have 3 points instead of 2



- Define three barycentric coordinates: α, β, γ

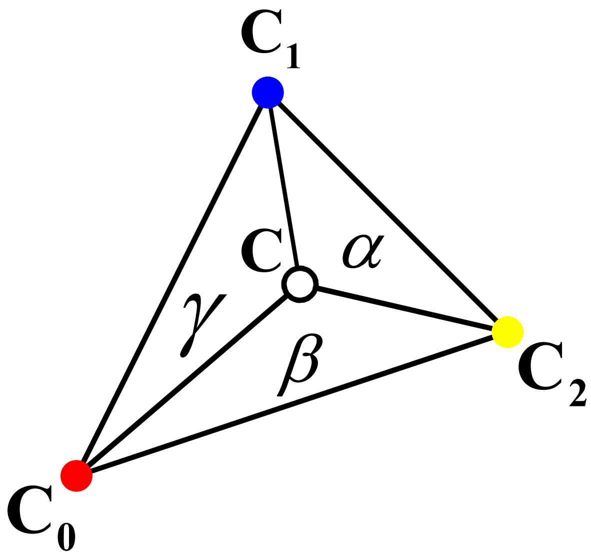
$$\mathbf{C} = \alpha\mathbf{C}_0 + \beta\mathbf{C}_1 + \gamma\mathbf{C}_2 \quad \text{where } \alpha + \beta + \gamma = 1$$

$$\mathbf{C} \text{ is inside } \mathbf{C}_0\mathbf{C}_1\mathbf{C}_2 \Leftrightarrow \alpha, \beta, \gamma \in [0,1]$$

- How to define α, β, and γ ?

Barycentric Coordinates for a Triangle

- Define barycentric coordinates to be ratios of triangle areas



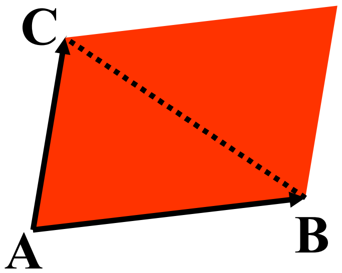
$$\alpha = \frac{Area(\mathbf{C}\mathbf{C}_1\mathbf{C}_2)}{Area(\mathbf{C}_0\mathbf{C}_1\mathbf{C}_2)}$$

$$\beta = \frac{Area(\mathbf{C}_0\mathbf{C}\mathbf{C}_2)}{Area(\mathbf{C}_0\mathbf{C}_1\mathbf{C}_2)}$$

$$\gamma = \frac{Area(\mathbf{C}_0\mathbf{C}_1\mathbf{C})}{Area(\mathbf{C}_0\mathbf{C}_1\mathbf{C}_2)} = 1 - \alpha - \beta$$

Computing Area of a Triangle

- in 3-D



  - $Area(\text{ABC})$ = parallelogram area / 2 = $\|(\text{B}-\text{A}) \times (\text{C}-\text{A})\|/2$
  - faster: project to xy, yz, or zx, use 2D formula

- in 2-D
  - $Area(xy\text{-}projection(\text{ABC})) = [(b_x-a_x)(c_y-a_y)-(c_x-a_x)(b_y-a_y)]/2$

    project A, B, C to xy plane, take z component of cross product
  - positive if ABC is CCW (counterclockwise)

Computing Area of a Triangle - Algebra

That short formula,

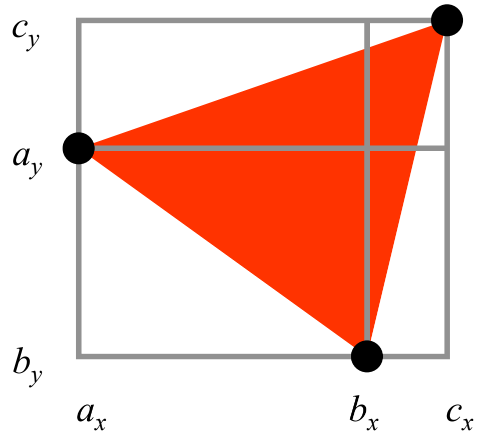
$$\mathit{Area}(\mathrm{ABC}) = [(b_x - a_x)(c_y - a_y) - (c_x - a_x)(b_y - a_y)]/2$$

Where did it come from?

$$\mathit{Area}(ABC) = \frac{1}{2}\begin{vmatrix} a_x & b_x & c_x \\ a_y & b_y & c_y \\ 1 & 1 & 1 \end{vmatrix}$$

$$= \left(\begin{vmatrix} b_x & c_x \\ b_y & c_y \end{vmatrix} - \begin{vmatrix} a_x & c_x \\ a_y & c_y \end{vmatrix} + \begin{vmatrix} a_x & b_x \\ a_y & b_y \end{vmatrix} \right) / 2$$

$$= (b_x c_y - c_x b_y + c_x a_y - a_x c_y + c_x a_y - a_x c_y)/2$$



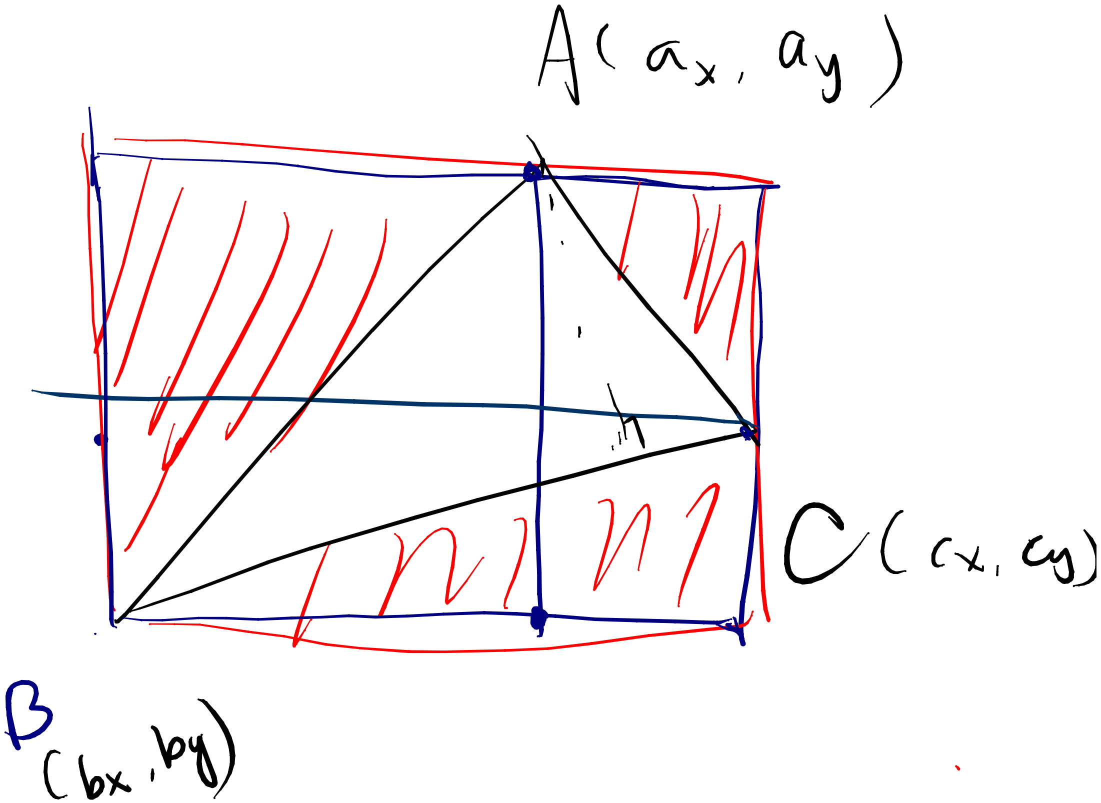
The short & long formulas above agree.

Short formula better because fewer multiplies. Speed is important!

Can we explain the formulas geometrically?

One Explanation

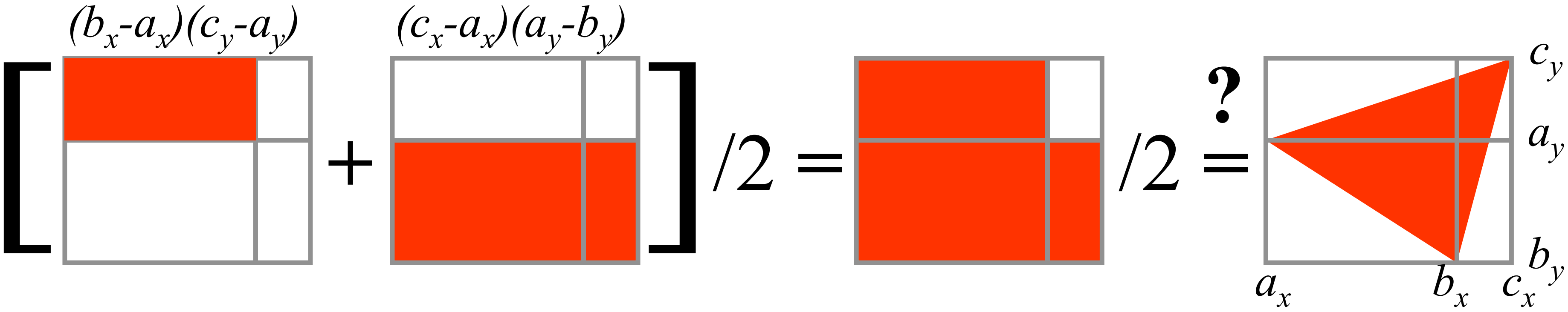
Area(ABC) = *area of the rectangle minus area of the red shaded triangles*

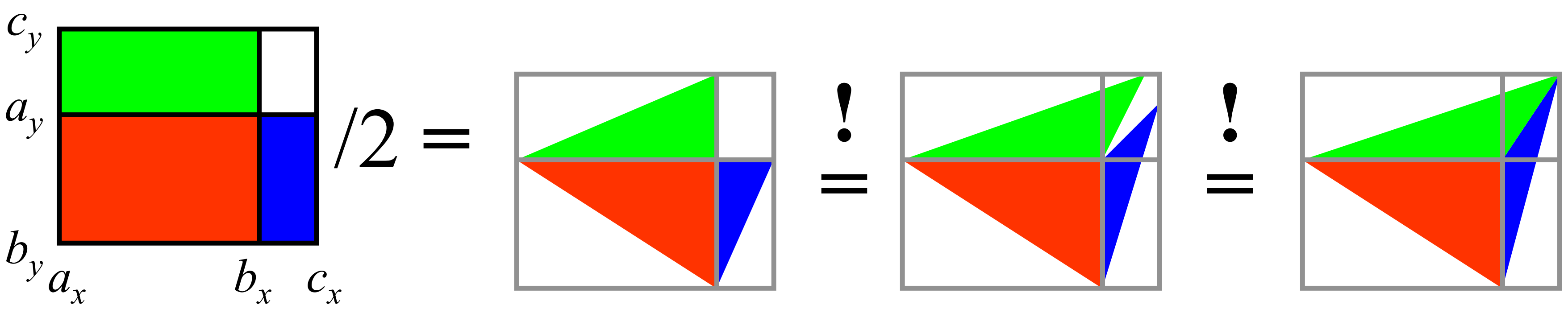


Another Explanation

$$Area(ABC) = [(b_x - a_x)(c_y - a_y) \;-\; (c_x - a_x)(b_y - a_y)]/2$$

is a sum of rectangle areas, divided by 2.



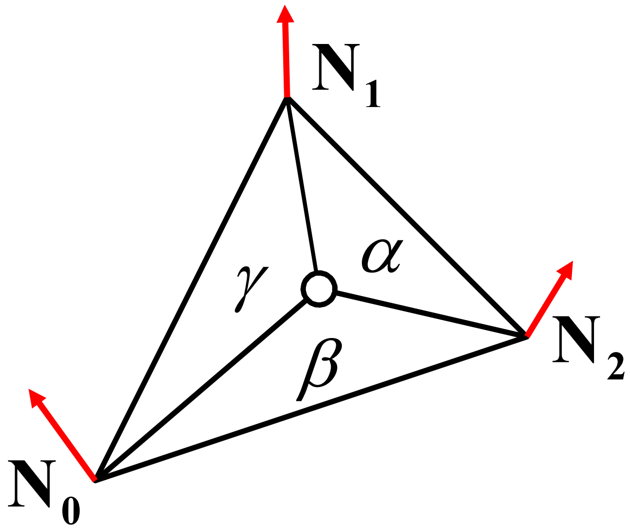


since triangle area = base*height/2

it works!

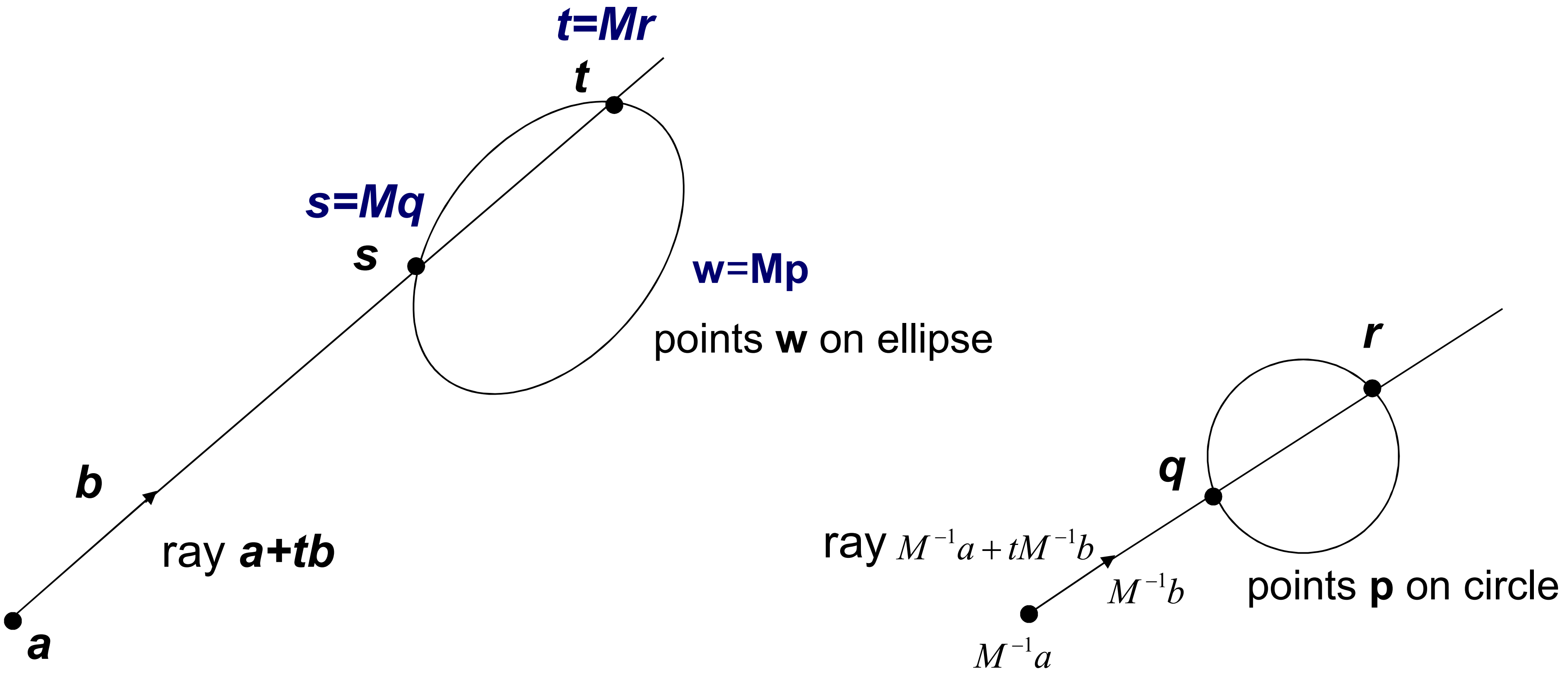
Uses for Barycentric Coordinates

- Point-in-triangle testing!
  - point is in triangle iff α, β, γ the same sign
  - note similarity to standard point-in-polygon methods that use tests of form $a_i x+b_i y+c_i<0$ for each edge i



- Can use barycentric coordinates to interpolate any quantity
  - color interpolation - Gouraud shading
  - normal interpolation - realizing Phong Shading
  - (s, t) texture coordinate interpolation - texture mapping

Instancing



t=Mr
t
s=Mq
s
w=Mp
points w on ellipse
b
ray a+tb
a
r
q
ray $M^{-1}a + tM^{-1}b$
$M^{-1}b$
points p on circle
$M^{-1}a$

Instancing

- The basic idea of instancing is that an object is distorted by a transformation matrix before the object is displayed. For example, in 2D an arbitrary ellipse is an instance of a circle because we can store a unit circle and the composite transformation matrix that transforms the circle to the ellipse. Thus the explicit construction of the ellipse is left as a future procedure operation at render time.
- With the concept of instancing, in ray tracing we can choose what space to do ray-object intersection in. If we have a ray a+tb (a: eye point; b: ray vector; t: parameter) we want to intersect with the transformed object, we can instead intersect an inverse-transformed ray with the untransformed object. That means, computing a ray and an ellipse intersection can be converted to a problem of computing ray-circle intersection instead.
- Pay attention to normal transformation for correct shading: if the normal at the intersection point of the base object is n, compute its correct normal in the transformed space.