Lecture 10: 7 October 1997

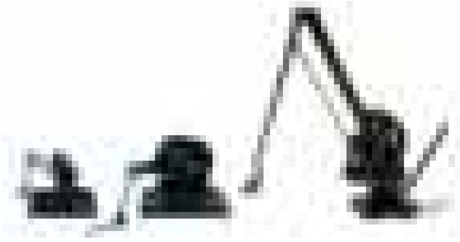
Today
- Asst 3B gallery – Inventor Scenes
- Ray Casting (Prelude to Ray Tracing)

Thursday
- Recursive Ray Tracing
- Asst 5 (Ray Casting/Tracing) out
  - This is the last assignment !

Next Week, Final Project Explorations
- Monday: Columbus Day Holiday
- Tuesday: Video Examples
- Thursday: Project Brainstorming
  - Start thinking of project ideas !

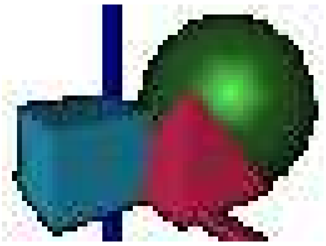
Special Event: Phantom Demo

Next Tuesday, 14 October, 7-9pm in 34-101



Scan Conversion and Visibility

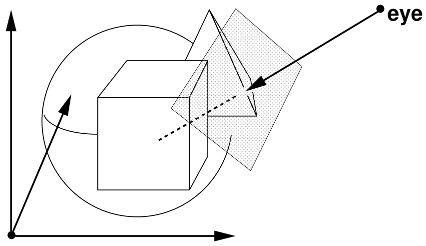
Hidden surfaces eliminated at each pixel
- scanline algorithms
- z buffering
- Shading computed many times at each pixel
- Each pixel stores only constant state



Limitations:
- Restricted to scan-convertable primitives
  - Causes faceting, shading artifacts
- Effective resolution h/w dependent (aliasing)
- No handling of shadows, reflection, transparency

Ray Casting for Visibility

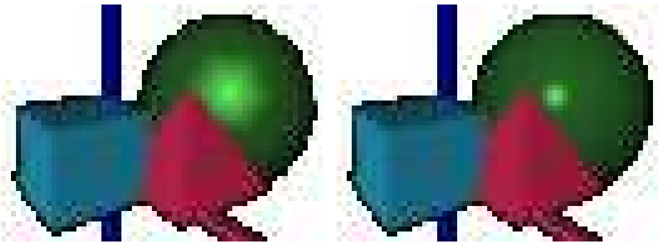
```
Object Cast ( Point R, Ray D ) {
  find min t>0 such that R + t D hits object
  if ( object hit )
    return object
  else
    return background object
} // Cast
```




```
FrameBuffer Render ( frustum, viewport ) {
  For each raster y
    For each pixel x
      E = ray from eye through pixel x, y
      obj = Cast ( eye, E );
      FB[x][y] = obj->color
  return FB
} // Render by Ray Casting
```

Ray Casting for Visibility

Advantages?
- Generality: can render anything with which a ray can be intersected !
- Smooth variation of normal, silhouettes

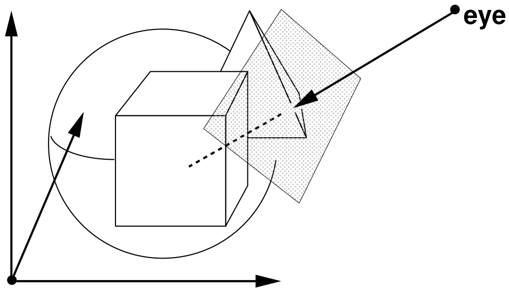


Disadvantages?
- Time complexity (N objects, R pixels)
- Wasted work at "background" pixels

Why aren't ray casters usually found in h/w?

Ray – Scene Intersection

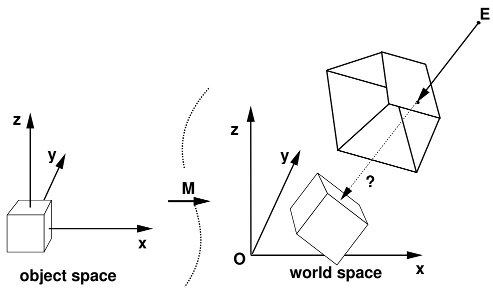
Fundamental operation in ray tracing:
- Given: a ray, and a scene of transformed primitives
- Determine:
  - Closest ray-primitive intersection **P**, if any
  - Primitive's surface normal **N** at point **P**



Typically do this by computing t for each intersection, and keeping track of t_{min}

Ray – Primitive Intersection

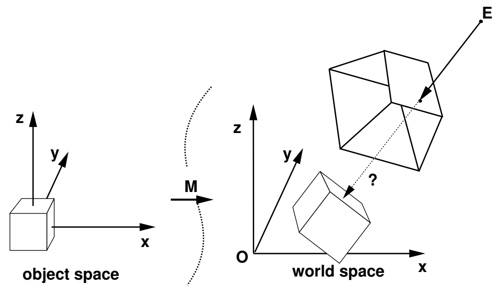
First concentrate on **individual** primitives
- Ray is typically generated in world space
- Primitives defined in object space, then transformed



What are our options?

Intersection in World Space

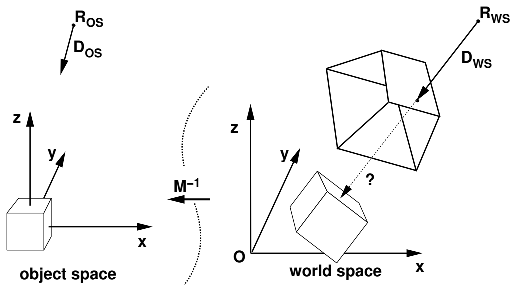
Option 1: transform *primitive* into *world* space
- Reexpress it there, then perform intersection



Polygon: not too hard (transform vertices, plane)
Polyhedra: collections of polygons
Quadrics: maybe; but what about scaling, shear, etc.?
Torii, spline surfaces, other primitives: eek!

Intersection in Object Space

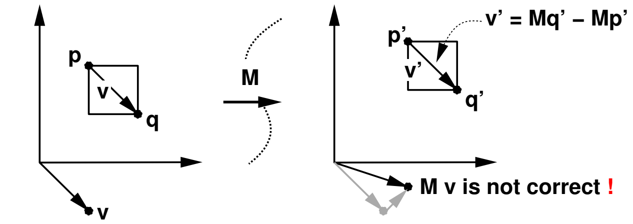
Option 2: transform *ray* into *object* space!
Intersections straightforward, even trivial there



How do we transform the ray's origin and direction?
Careful: must deal with isotropic, anisotropic scaling

Ray Transformation

Know how points transform: $\mathbf{p}' = \mathbf{M}\mathbf{p}$
Thus, transforming ray *origin* is easy



But how does the ray *direction* transform?
Clearly not as a point (see example)

Direction vector transformation

What's going on?
"Endpoints" of ray have equal translations
Equivalently: ray is a pure *direction*
So: transform both ends, subtract results

Equivalent to ignoring translation part of $\mathbf{M}$

$$\mathbf{M} = \begin{pmatrix} M_{11} & M_{12} & M_{13} & T_x \\ M_{21} & M_{22} & M_{23} & T_y \\ M_{31} & M_{32} & M_{33} & T_z \\ M_{41} & M_{42} & M_{43} & M_{44} \end{pmatrix}$$

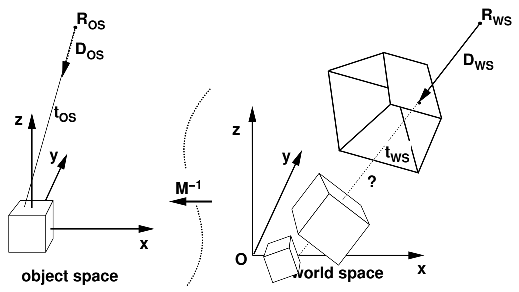
Instead, use

$$\mathbf{M}^* = \begin{pmatrix} M_{11} & M_{12} & M_{13} & \mathbf{0} \\ M_{21} & M_{22} & M_{23} & \mathbf{0} \\ M_{31} & M_{32} & M_{33} & \mathbf{0} \\ M_{41} & M_{42} & M_{43} & M_{44} \end{pmatrix}$$

Can think of direction (a, b, c) as homogeneous $(a, b, c, 0)$
[Inventor supplies this as M.multDirMatrix $(\mathbf{v}, \mathbf{v}')$]

Ray Scaling – Transforming t

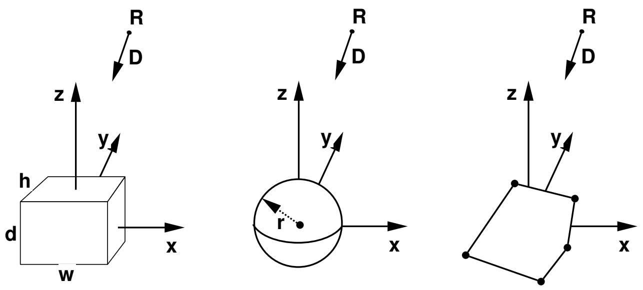
Problem: ray might be scaled *anisotropically* by $\mathbf{M}$
Ok; can still compute intersections in object space
(Unit length of $\mathbf{N}$ can come in handy...)
How to get t_{WS} from t_{OS}?



Two options:
Don't normalize $\mathbf{D_{OS}}$; compute t_{OS}
Then $t_{WS} = t_{OS}$!
Must handle unnormalized vectors, but xform easy
Normalize $\mathbf{D_{OS}}$; compute t_{OS}
Then $t_{WS} = t_{OS}/\|\mathbf{D}_{OS}\|$!
Need handle only normalized vecs, but xform harder

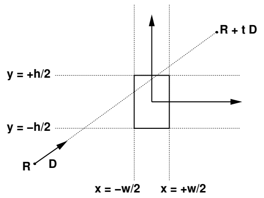
Primitive Intersection

Now we can work in object space, then
revert to world space to return values
Wish to compute closest intersection point,
and normal at that point (if any intersection)
Consider ray-primitive intersection, with primitives



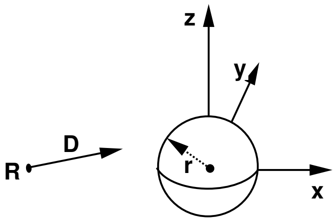
Axial parallelepiped; sphere of radius r; polygon

Ray-Parallelepiped Intersection




```
static Vector hats[3] = { xhat, yhat, zhat };
// ray is of form R + t D; find minimum t_intersection
float t1, t2, tmin = 0, tmax = HUGE; // from <math.h>
Vector extent = Vector ( width / 2, height / 2, depth / 2 );
// intersect ray with x, y, z ``slabs'' (k = 0, 1, 2)
for ( int k = 0; k < 3; k++ ) {
  if ( D[k] != 0. ) {
    t1 = (-extent[k] - R[k]) / D[k];    // plane x_k = -dx_k
    t2 = ( extent[k] - R[k]) / D[k];    // plane x_k = +dx_k
    tmin = fmax ( tmin, fmin (t1, t2)); // intersect [tmin..
    tmax = fmin ( tmax, fmax (t1, t2)); //  tmax],  [t1..t2]
    if ( tmax <= tmin ) return FALSE;   // no intersection
    else if ( tmin == t1 ) N = -hats[k]; // hit x_k = -dx_k
    else if ( tmin == t2 ) N =  hats[k]; // hit x_k = +dx_k
    } // if
  } // k
thit = tmin; // return parameter of closest intersection
return TRUE;
```

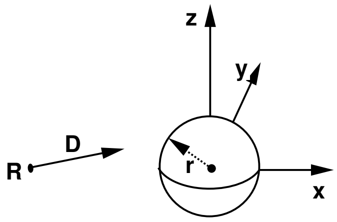

Ray-Sphere Intersection



Ray equation (explicit): $\mathbf{P}(t) = \mathbf{R} + t\mathbf{D}$, with $|\mathbf{D}| = 1$
Sphere equation (implicit): $\mathbf{P} \cdot \mathbf{P} = r^2$
Intersection means both are satisfied, so

$$
\begin{aligned}
0 &= \mathbf{P} \cdot \mathbf{P} - r^2 \\
&= (\mathbf{R} + t\mathbf{D}) \cdot (\mathbf{R} + t\mathbf{D}) - r^2 \\
&= \mathbf{R} \cdot \mathbf{R} + 2t\mathbf{D} \cdot \mathbf{R} + t^2\mathbf{D}^2 - r^2 \\
&= t^2 + 2t\mathbf{D} \cdot \mathbf{R} + \mathbf{R} \cdot \mathbf{R} - r^2
\end{aligned}
$$

Ray-Sphere Intersection



This is just a quadratic $at^2 + bt + c = 0$, where

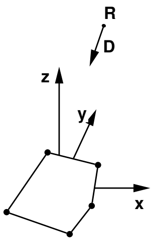
$$
\begin{aligned}
a &= 1 \\
b &= 2\mathbf{D} \cdot \mathbf{R} \\
c &= \mathbf{R} \cdot \mathbf{R} - r^2 \\
d &= \sqrt{b^2 - 4ac}
\end{aligned}
$$

with solutions

$$t_\pm = \frac{-b \pm d}{2a}$$

Three cases, depending on sign of $b^2 - 4ac$
Which root (t_+ or t_-) should you pick?

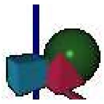
Ray-Polygon Intersection



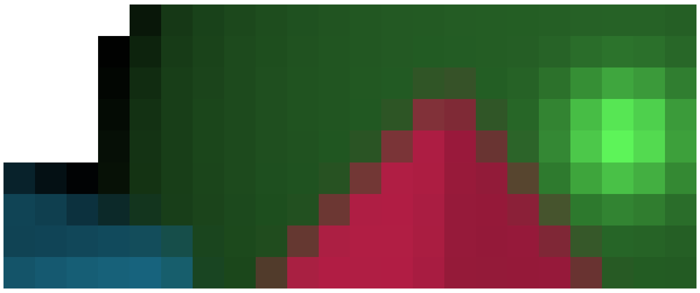
Options:
Project polygon to shadow on principal plane
  Solve 2D inclusion problem
Intersect ray with support plane
  Solve 3D inclusion problem
Tradeoffs?

Aliasing

So far, both rasterization and raycasting
  compute one color per pixel:



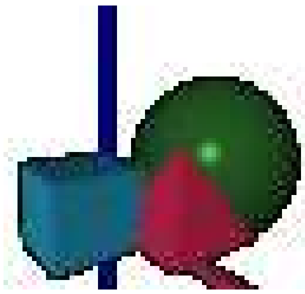
Look closely at:

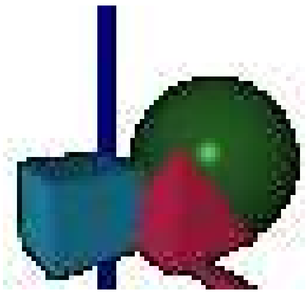


Notice “jaggies” at object boundaries
Can we do better with ray casting?

Super Sampling

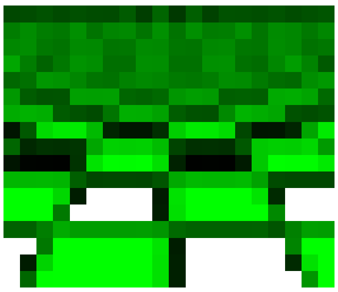
We can *average* multiple samples per pixel:

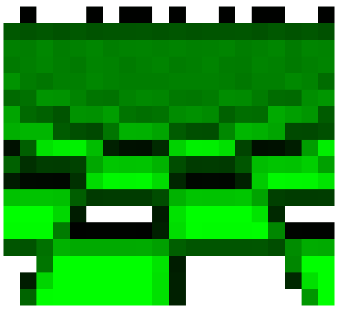




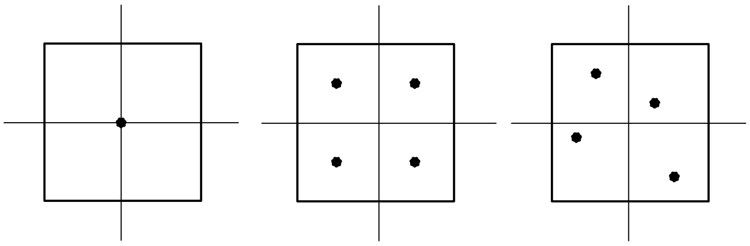
More Antialiasing

Important when rendering high-frequency textures:

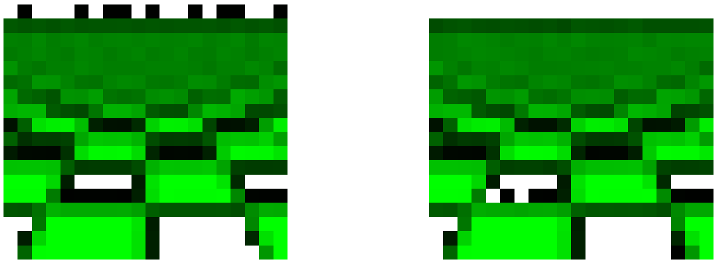




Placing Samples



Regular, Jittered Supersampling



Thursday

Recursive Ray Tracing