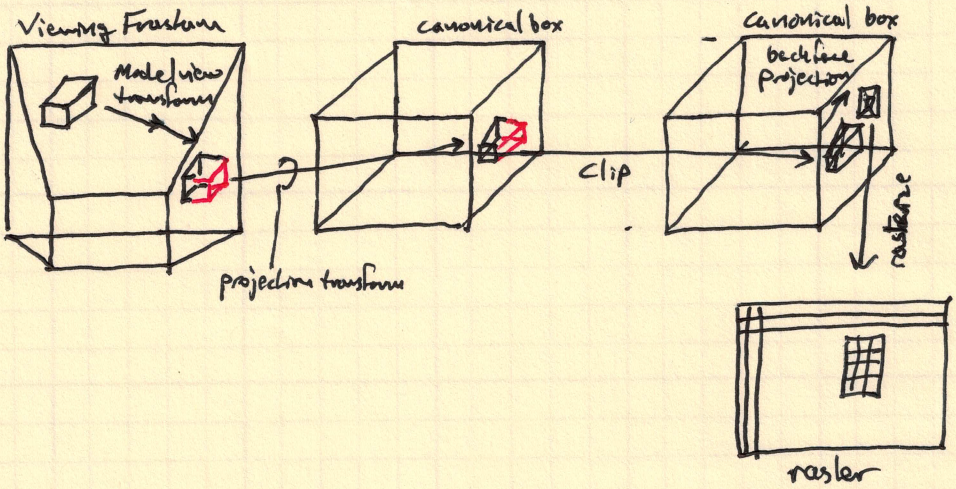
Outline

- Basic rendering pipeline
- Introduction to Ray tracing
- The Ray Tracing Algorithm
- Antialiasing in Ray tracing
- Intersection Calculations
- Soft Shadows
- Efficiency Considerations

Pipeline Operation

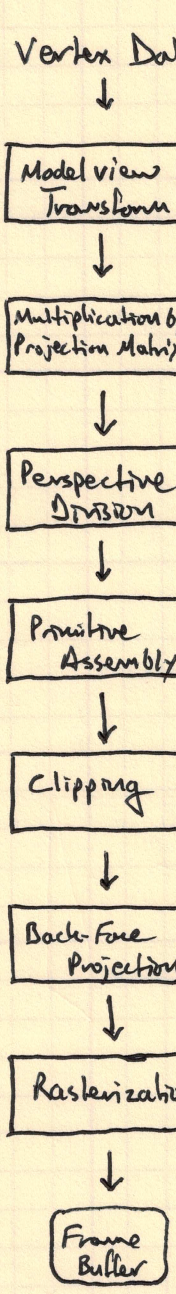
- In these notes we will limit the discussion to fixed-function pipelines (ie, those prior to shader-programmable pipelines)
- Essentially the programmer does the following
  - Specifies the scene
    - Lights
    - Objects
    - Materials/Textures
  - Then does nothing else besides letting the render take control
- This is essentially OpenGL as we have been using it
- Additionally we can consider another approach → Ray Tracing

- Synthetic-Camera Pipeline
  - The basic process of rendering as we know it



Synthetic-Camera Pipeline

- This then leads us to a minimal synthetic camera rendering pipeline



$[x\ y\ z\ 1]^T$

$[x^M\ y^M\ z^M\ 1]^T$ — Model View Transform

$[x^{PM}\ y^{PM}\ z^{PM}\ w^{PM}]^T$ — Projection Transform

$\left[\frac{x^{PM}}{w^{PM}}\ \frac{y^{PM}}{w^{PM}}\ \frac{z^{PM}}{w^{PM}}\right]^T$ — perspective division w/rules to handle zero w-values

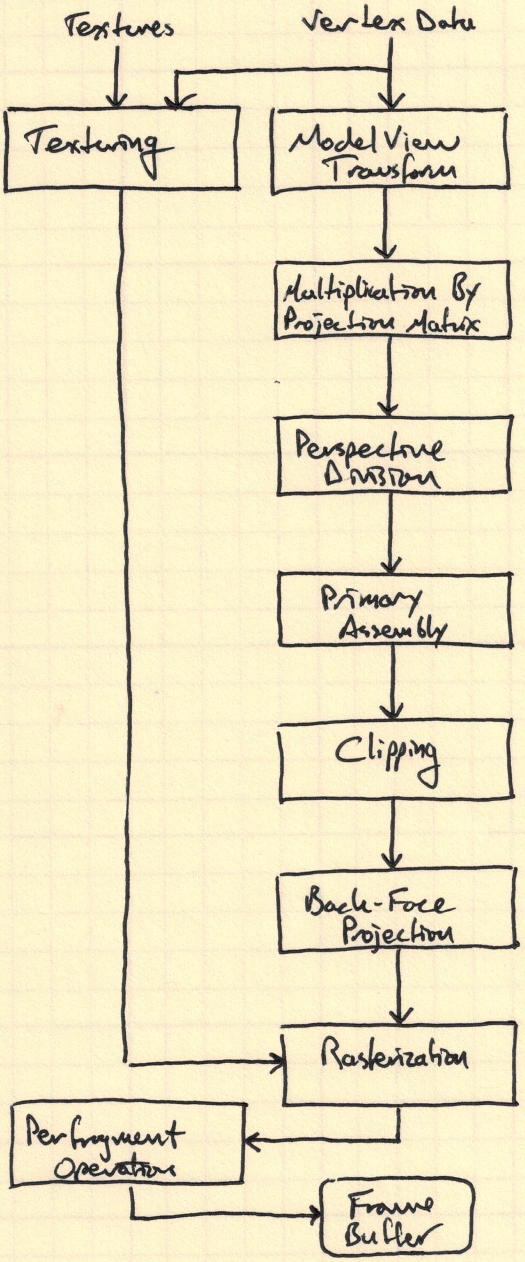
Primitive assembly — Defining points, lines and triangles

$\left[\frac{x^{PM}}{w^{PM}}\ \frac{y^{PM}}{w^{PM}}\ \frac{z^{PM}}{w^{PM}}\right]^T$ — Clipping to the canonical box

$\left[\frac{x^{PM}}{w^{PM}}\ \frac{y^{PM}}{w^{PM}}\right]^T$ — Projection to the back of the canonical box (z-values possibly returned for depth testing)

$[i\ j]^T$ — Rasterization w/perspective correction

OpenGL Pipeline (First Gen)



Additions

- Texturing ≡ where vertex data and a set of controlling parameters are used to combine textures into the raster.
- Per-Fragment Ops ≡
  - fragment ≡ a raster pixel with color data and a z-value
  - Consists of 4 tests in order:
    - Scissor test
    - Alpha test (Discarded as of 3.1)
    - Stencil test
    - Depth test

    followed by
    - Blending

Ray Tracing

Ray Tracing vs. Radiosity

- There are two major approaches to rendering a lit scene, including the light relationships between different surfaces:
  - Ray Tracing: Tends to produce images that are rather flashy
  - Radiosity: Tends to produce softer, muted images
- There are also hybrid approaches which attempt to combine the best of both ray tracing and radiosity

Overview of Ray Tracing

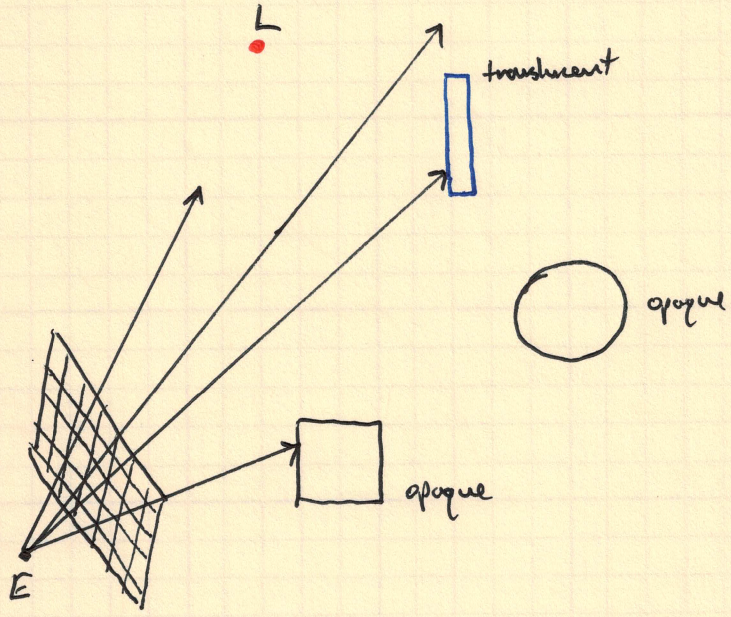
- Turner Whitted first introduced ray tracing back in 1980
- Although ray tracing has significantly progressed since, the basic recursive algorithm proposed by Whitted still underlies all ray tracers
- Current research in ray tracing emphasizes real time ray tracing, hybrid approaches with the rasterization approach, and the development of computer architectures to facilitate ray tracing
- The goal is to simulate the cumulative effect of every ray of light that leaves a light source which
  - bounces of objects getting absorbed
  - reflected
  - refracted

  which finally, possibly as several secondary rays, makes it to the virtual screen into
- The first problem: There are an infinite number of rays coming out of a light source, so tracing them all down isn't possible
  - As a result, in standard ray tracing we analyze the rays in the other direction. i.e., we start at the viewers position and run a ray backwards through each pixel into the scene. This is called Forward Ray Tracing
  - The approach were rays are tracked in the correct direction from the light source back to the viewer is called Backward Ray Tracing

The Basic Ray Tracing Algorithm

```
void topLevelRoutineCallsTheRayTracer() {

    positionEye = position of the eye in world space
    for (each pixel P of the virtual screen)
    {
        d = unit vector from positionEye toward the center
            point of P
        color of P = rayTracer(positionEye, d);
    }

    Color rayTracer(Point p, Direction d) {

        if (ray from p along d does not intersect the surface of any object)
            return backgroundColor;
        else {
            q = first point of intersection with an object's surface
            computedColor = color computed using Phong's lighting model;
            return computed color;
        }
    }
```

- This is the local non-recursive version which simply stops upon the first intersection and does not take into account shadows or other effects.
  - Note this will render the same thing as the basic implementation of OpenGL's rendering pipeline.

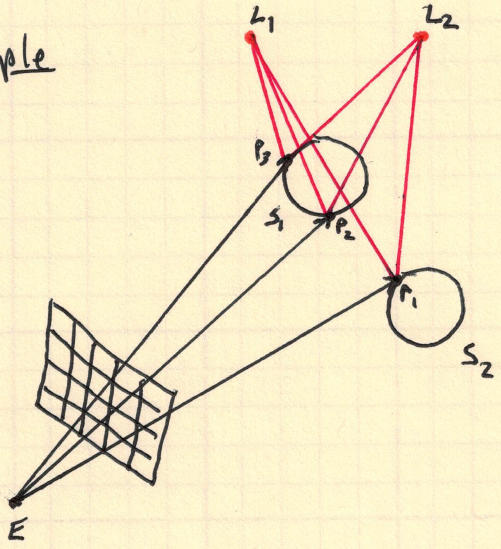


L - Light Source

E - Viewer Eye Position

Going Global: Shadows

- If a ray through a pixel intersects a surface and then sends a "feeler" ray towards each light source and these intersect an object's surface prior to reaching the light, then the original point of intersection is in the shadow of the object struck by the feeler ray.
- Such points are not directly illuminated by that light source, but (according to Phong's model) are illuminated ambient and any reflected specular and diffuse light.
- Example



Here P_1 is directly illuminated by L_2, but is in the shadow cast by L_1 on S_1.

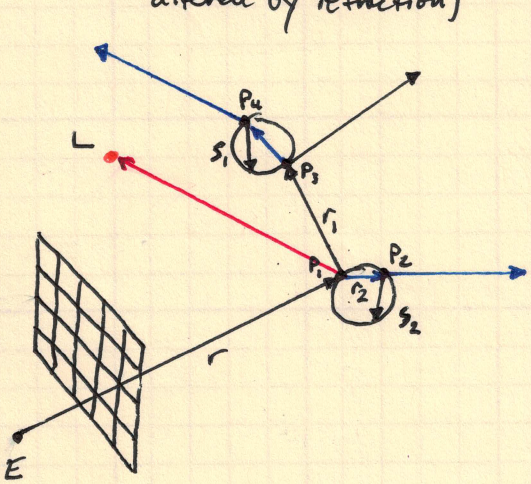
This is because the feeler ray from P_1 to L_1 intersects S_1

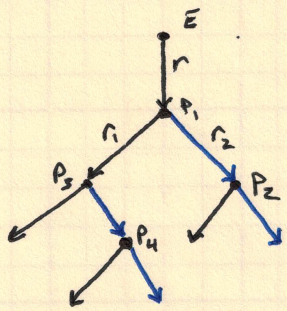
- Ray Tracing With Shadows (non-recursive)

```
void topLevelRoutineCallsTheRayTracer(); // see prior version

Color rayTracer(Point p, Direction d) {
   if (ray from p along d does not intersect any object) return backgroundColor
   else {
      q = first point of intersection
      computedColor = (0,0,0) // black
      for each (light source L) {
         if (feeler ray from q towards L does not intersect any object before reaching L)
            computedColor += color computed at q, due to light from L, using Phong's model

         else computedColor += ambient component of color computed at q, due to light
                               from L, using phong's model.
      }
   return computedColor
   }
}
```

More Global: Reflection and Transmission

- Rays from a light source can bounce from object to object, or pass through them, several times before reaching the viewer.
- This gives rise to phenomena such as reflection and translucence.
- To handle this we must follow the ray after it strikes an object.
- The physics of light suggest that when struck by a ray part of the light is reflected and part is transmitted through it, depending on the characteristics of the material, as well as the color of the light.
  - For example, an opaque object transmits almost zero light and reflects the remainder according to its surface finish and color, while a translucent one transmits most.
- Thus, we shall enhance our ray tracer in the following way
  - each ray striking an object will spawn two additional rays
    - ~~A reflected ray~~
    - A **reflected ray** in the direction of perfect reflection
    - A **transmitted ray** passing through the surface (possibly with its direction altered by refraction)





transmitted rays
reflected rays
feeler rays

- The color computed at a point now has the following three components:

$$\text{computedColor} = \text{color}_{\text{local}} + \text{coef}_{\text{refl}}\,\text{color}_{\text{refl}} + \text{coef}_{\text{trans}}\,\text{color}_{\text{trans}}$$

where: $\text{coef}_{\text{refl}}$ is the material-dependent multiplicative factor for reflection
$\text{coef}_{\text{trans}}$ is the material-dependent multiplicative factor for transmission
$\text{color}_{\text{refl}}$ and $\text{color}_{\text{trans}}$ are the recursively returned color values.
$\text{color}_{\text{local}}$ is the Phong defined local color

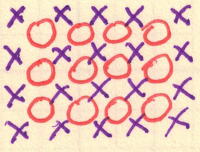
The modified algorithm

```
void topLevelRoutineCallsTheRayTracer() {

    positionEye = position of eye in world space
    for (each pixel P of the virtual screen) {
        d = unit vector from positionEye toward the center point of P
        color of P = rayTracer(positionEye, d, maxDepth)
    }
}

Color rayTracer(Point p, Direction d, int depth) {
    if (ray from p along d does not intersect the surface of any object) return background
    else {
        q = first point of intersection
        computedColor = (0,0,0) // black

        for (each light source L) { // Local Component
            // Object not shadowed
            if (feeler ray from q toward L does not intersect the surface of any object
                before reaching L)
                computedColor += color computed at q, due to light from L, using Phong's model

            // Object shadowed
            else computedColor += ambient component of color computed at q, due to light from
                                  L, using Phong's model
        }

        // Global Component
        if (depth > 0) {
            d1 = unit vector from q in direction of perfect reflection;
            d2 = unit vector from q in direction of transmission

            // Reflected component added in recursively.
            computedColor += coefRefl * rayTracer(q, d1, depth-1)

            // Transmitted component added in recursively
            computedColor += coefTran * rayTracer(q, d2, depth-1)
        }
        return computedColor;
    }
}
```

- As we can see in this final version of the basic ray tracer we use the depth of recursion as our means by which we stop the process.
- Another (and potentially more useful) approach is to keep track of the accumulated percentages and to stop when they reach some minimum threshold
  - Typical values used are: 0.01 to 0.05

Anti-Aliasing and Ray Tracing

- Standard ray tracing tends to give rise to some aliasing effects, which is a fancy way of saying that it can give rise to jaggies in the image.
- The problem is that if one ray hits an object, and an adjacent ray just misses the object, then the difference in intensity can be significant, and if you continue this down a smooth edge, say in a diagonal direction, you can get jaggies.
- To avoid this, one approach is to go to, say, 2k x 2k resolution, and then average groups of four values to get the pixel values
- However, this increases the computational cost by a factor of 4
- A better approach, which is almost free, is to run rays through the gaps between pixels, and then use the average of the 4 pixels to compute the intensities of each pixel
- In this way there will even be an averaging effect where pixels very close to an object will get the smoothing effect of both inside and outside the object.
- On a typical 1920 x 1080 screen, this will mean that we'll have to compute the effect of an extra row and column and have to compute 1921 x 1081 rays which is a computational increase of less than 0.2%.
  - E.g., if we have a ridiculously small 3 x 4 screen shown with pixel locations as circles below, then we will run rays through the 4 x 5 locations shown with X's and each pixel will get the average of the intensities of its four surrounding values



- For a screen with $n \times m$ resolution, this will mean an increase of $n+m+1$ ray value computations, and since this is so tiny relative to $n \times m$ for large n and large m, every ray tracing program should do this.
- Another approach is to use adaptive sampling, where extra rays are used when it is detected that rays are close to the edge of an object, or in other high noise areas.
- The simplest approach is to detect when the colors of two adjacent rays differ by more than some threshold, and then whenever this occurs to generate a new ray between the two current rays. This will continue recursively to a user specified depth.

Efficiency Considerations

- It is usually claimed that about 95% of the time in every ray tracer is spent computing intersections between rays and objects in the scene
- As people come up with more efficient ways to find the intersections, other people then use these new techniques to render more complex scenes, and the 95% rule stays in effect.
- As a result, improving the speed of these intersection requests is the place where all efficiency considerations should be concentrated, at least until they are as efficient as possible
- One approach is to simplify the scene. The original ray tracing images were usually based on some objects that were spheres (typically glass to make reflections and refractions look spectacular) and flat surfaces.
- This was because computing intersections with planes and spheres is much easier than computing intersections with more complex objects.
- Assuming that we are rendering more interesting scenes, we need to concentrate on how to find the closest object intersection
- One solution is to surround all objects with simpler objects (e.g., bounding boxes or spheres), and then when we determine whether a ray intersects an object, we first check for an intersection with the bounding object.
- If that doesn't intersect, then move on, but if it does intersect then move on, but if it does intersect, then do a more precise intersection calculation
- We can also order the objects in a scene by depth from a viewer and check for intersections with closer objects first.
- An additional speedup is to use object coherence to improve efficiency

- If say, we want to see whether a ray intersects one of three objects, all of which are about the same distance from the viewer, and the previous (adjacent pixel) ray hit one of them, then it makes sense to check intersection with the object first.

- Also we can use the intersection of the previous ray with that object as a starting point for an iterative search (e.g., using a two-variable Newton's method) for the intersection of this ray with the object, which will often mean that the intersection is found very quickly, even on very complex objects.

- The most common way to find the closest intersected object is to use some kind of space partitioning representations of the scene, e.g., octtrees or BSP trees. They are used in an identical way as discussed for hidden surface removal

Computing Intersections w/ Spheres, Planes and Boxes

- Parametrically, a ray from (x_1, y_1, z_1) to (x_2, y_2, z_2) has three equations:

$$\begin{aligned} x &= x_1 + (x_2 - x_1)t = x_1 + it \\ y &= y_1 + (y_2 - y_1)t = y_1 + jt \\ z &= z_1 + (z_2 - z_1)t = z_1 + kt \end{aligned} \qquad 0 \le t \le 1 \text{ or } 0 \le t$$

Sometimes a ray will be limited in length, when the condition $0 \le t \le 1$ will be used, or we might just be looking for a first intersection, in which case we have a starting point and a direction vector and will use $0 \le t$

- Ray/Sphere Intersections

  - A sphere at center (l, m, n) with radius R has the familiar equation:

$$(x-l)^2 + (y-m)^2 + (z-n)^2 = R^2$$

  - Substituting the values for x, y, and z gives a quadratic equation for t

$$at^2 + bt + c = 0$$

Where

$$\begin{aligned} a &= i^2 + j^2 + k^2 \\ b &= 2(i(x_1 - l) + j(y_1 - m) + k(z_1 - n)) \\ c &= l^2 + m^2 + n^2 + x_1^2 + y_1^2 + z_1^2 - 2(lx_1 + my_1 + nz_1) - R^2 \end{aligned}$$

  - The discriminant $d = b^2 - 4ac$ of this equation tells us what we need for the intersection decision

    - If $d < 0$, then the ray does not intersect the sphere
    - If $d = 0$, then the ray is tangent to the sphere
    - If $d > 0$, then the ray does a double intersection with the sphere

- If we have an intersection then the roots of the quadratic give the t values for the intersections, where the smaller t is the closest to the source of the ray.
- Substituting the t values back into the original equations gives the intersection points.
- If we have found an intersection at (x_i, y_i, z_i), then the unit normal at that point is:

$$\left(\frac{x_i - l}{r}, \frac{y_i - m}{r}, \frac{z_i - n}{r}\right)$$

- Although the quadratic equation approach is common, there are rules called Haines' rules which in general perform better to determine whether or not a ray intersects a sphere, and if so where it intersects. The following is a somewhat simplified version from the original paper with the full system: "Essential Ray Tracing Algorithms", E. Haines, 1989

1. Determine whether or not the ray's origin is outside the sphere
2. If the ray starts outside and points away from the sphere, it cannot intersect
3. Find the closest approach of the ray to the sphere's center. If this is inside or on the surface then the ray intersects (or is tangent to the sphere)
4. Use the quadratic formula to determine any intersections
5. Calculate surface normals

- Step 1 can be computed by putting (x_1, y_1, z_1) into the spherical equation.
- Step 2 is a dot product
- Step 3 is a bit more complicated. The square of the distance from any point on the ray with parameter t to the center is

$$(x_1 + it - l)^2 + (y_1 + jt - m)^2 + (z_i + kt - n)^2$$

Clearly the distance to the center is minimized when this is minimized, so differentiating with respect to t and setting the result to $\emptyset$ gives:

$$2i(x_1 + it - l) + 2j(y_1 + jt - m) + 2k(z_1 + kt - n) = 0$$

Canceling out the 2's and separating out the t terms gives:

$$t = \frac{i(l - x_1) + j(m - y_1) + k(n - z_1)}{i^2 + j^2 + k^2}$$

• Ray/Polygon Intersections

- Intersecting a ray with a polygon is fairly direct. If the polygonal equation is $ax+by+cz+d=0$ then the intersection of the ray with the plane can be found by substitution of the parametric ray equations giving:

$$a(x_1+it)+b(y_1+jt)+c(z_1+kt)+d=0$$

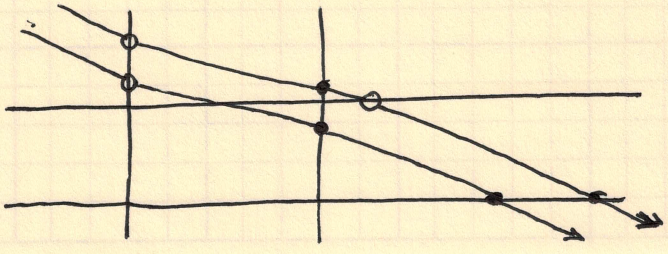
and so

$$t=-\frac{ax_1+by_1+cz_1+d}{ai+bj+ck}$$

- Then we first reject anything where $t<0$ or $t>1$.
- If t is in range then we need to determine whether the intersection lies inside or outside the polygon.
  - This can be determined by summing all angles formed by taking lines from the intersection point to the vertices, or through a number of other specialized approaches
- Obviously there is a problem with the t equation if $ai+bj+ck=0$, and so this will be when there is no well defined intersection.
  - This is a condition that tells us that the ray is parallel to the plane containing the polygon
  - So either there are no intersections or there are an infinite number, when the ray lies in the plane

• Ray/Box Intersections

- The approach here is more easily visualized in 2D, and then extended to 3D. We'll be assuming that we have a 3D box aligned along the axis system used for the ray coordinates. This is the usual ~~situation~~ situation when using bounding boxes.
- In 2D we use the picture below, where we have a bounding rectangle instead of the box, and I'm showing two rays one of which misses the rectangle and one of which intersects it.



- The open circles show the first intersections ~~with~~ with each pair of parallel lines that define the rectangle and the filled circles show the second intersections.
- In the top line the largest (in terms of t) first intersection is greater than the smallest second intersection, which means that the ray misses the rectangle.
- In the second case both of the first intersections are smaller than both of the second intersections, and so the ray intersects at the largest of the first intersections and then later at the smallest of the second intersections. Note that here we are basically treating the ray as an infinite line, so we still need to ensure that the t value is in the legal range.
- In 3D this is the same.
  - If the largest of the three first planar intersections is less than the smallest of the three second planar values, then there is an intersection at these two parametric values, and otherwise there isn't.
  - If the box is defined by the two extent points (x_l, y_l, z_l) and (x_r, y_r, z_r), then the intersection with the plane $x = x_l$ is given by:

$$x_1 + it = x_l, \text{ and so } t = \frac{x_l - x_1}{i}$$

  - The remaining 5 intersections are equally simple

- Intersections w/ Other Objects
  - There has been a lot of work done on how to compute intersections with a number of different types of objects.
  - Quadrics other than the sphere are probably the most common, and in particular bounding ellipsoids have some advantages over bounding spheres.
  - NURBS patch intersections have been looked at by, amongst others, Whitted and Kajiya. Kajiya also had a paper where he discussed intersections with fractal surfaces and surfaces of revolution
  - Usually, however, if a shape becomes too complex then programmers must resort to bounding objects or space-subdivision approaches to eliminate most objects from consideration, and then have to resort to a two-variable Newton's Method to iterate to the intersection.
  - To avoid the Newton's as much as possible, sometimes a hierarchy of bounding objects will be used with, for example, first of all a bounding box around an object, and then, when needed bounding boxes around components of the object like its arms, legs, trunk, and head.

- Soft Shadows

- The easiest approach (but unfortunately computationally brutal) is to represent the light source as a number of points on the surface of a sphere, and throw shadow feelers to each of these points.
- If this approach is taken then a vast majority of our rays at each ray/object intersection will be shadow rays, and so shadow computations will dominate the cost of the ray tracing algorithm
- Obviously this approach is impractical,
- An alternative is to fake shadows.
  - Use the point light source to get hard shadows, and then near the edge of the shadow put in a decreasing shadow effect up to the edge. This is easy to do, and works pretty well.
- Computing shadow feeler intersections can be somewhat easier than the standard ray intersection process. With the regular process we want to find the closest object that is intersected, and then find the point on that object that is intersected.
- With a shadow feeler we don't care about the point on the intersected object, or even if it is the closest object. Once we've found that the feeler intersects any object that is closer than the light source we can stop looking for closer objects intersections and say that the point is in shadow.
- A complication is what happens when a feeler intersects a semi-transparent surface (like a glass sphere or a wave on a water surface. There are a number of difficulties that occur.
  - The hardest is a light effect called a caustic, which is a strange effect that you'll see at the edge of a swimming pool.
  - Solutions to this effect are not easy and form a major sub-literature of ray tracing.
  - Other problems include having to refract the feeler through the object, and to have to calculate the feelers at different wavelengths both because of the different refraction angles and also because the object will presumably absorb differently based on the wavelengths

° Efficiency Considerations

- Ray tracing is a computationally expensive process, and so efficiency considerations are obviously important

- Such considerations include:
  - ° Adaptive Depth Control
  - ° Bounding Object Techniques
  - ° Data structures used
  - ° Parrallel Processing

- Bounding volumes are most commonly spheres and boxes (aligned or non-aligned) Any scene could have a mixture of different types, depending on the shapes of the individual objects.

  - ° Weghorst defines the void area of a particular bounding object for a scene object to be the area differences between the bounding object and the scene object when projected perpendicular to the ray.

  - ° He then compares the total cost for different bounding volumes based on false hits, costs of doing the scene object intersection calculations, and the costs of doing different bounding object intersections.

- Another use of bounding volumes is to have hierarchical bounding volumes.

- Kay and Kajiya gave a list of requirements for any hierarchical scene. Some were obvious including that you should only cluster objects that are near to each other, that the volume of each bounding object should be minimal, and that the time spent constructing the hierarchy should be less than the time saved during ray tracing.

  - ° Two less obvious rules are:

    - The sum of the volumes of all bounding objects should be minimal
    - The most important part of the tree to get right is near the root of the tree, so spend construction time there.

- Creating an automatic system for constructing an optimal (or near-optimal) hierarchy which meets the above conditions is a good idea for a PhD Thesis

° Space Subdivision

- Most obvious techniques: kD-Trees, Oct-Trees, BSP Trees
- Basically we know where in space the ray is, and that at most one object can share this point, so check it for an intersection. If it hits you are done otherwise jump to the next piece of space along the path of the ray. There are fast traversal algorithms for this.
- A similar system is the Spatially Enumerated Auxilliary Data Structures (SEADS) where space is divided into equally sized voxels which can be traversed using Bresenham's line algorithm

- For Next Time

- Additional Notes