Johns Hopkins
Engineering for Professionals

605.767 Applied Computer Graphics

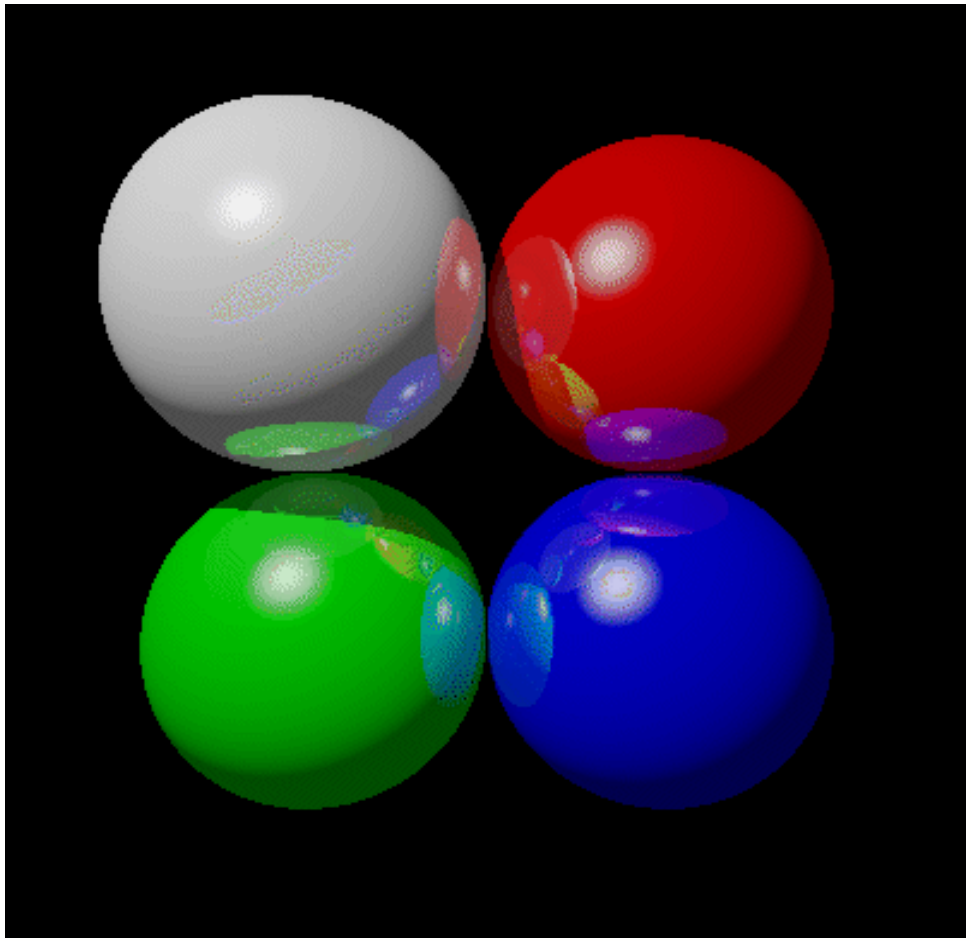
Brian Russin

Module 2B
Implementing a Ray Tracer

Ray Tracing: Intersections

- Primitive ray tracer checks intersection against every scene object
  - Looks for nearest intersection
  - Distance along the ray
  - Cannot simply stop search when an intersection has been found
    - Except for shadow rays
- Brute force strategy - recursive ray tracer spends most of its time testing for intersections
  - Each intersection spawns additional rays and subsequent intersect tests
  - Whitted estimated 95% in moderate complexity scenes
- For efficiency, need to address 2 problems:
  - Efficiently performing the intersection tests
    - Lecture 1
  - Strategy for guiding the order in which checks are performed
    - Limiting unnecessary intersection checks
    - Discuss some techniques at the end of this lecture

Ray-Tracing Spheres



Simple ray tracers often start with spheres placed in world coordinates. Simple intersection computation.

Intersecting Transformed Objects

- Wish to handles cases where an object is modeled in local coordinates and transformed to world coordinates
  - Translation, rotation, scaling (modeling transformation)
  - Transformation M maps an object to its image W
- Solution: solve for where the inverse transformed ray hits the original object
  - Rather than transform the object to world coordinates, transform the ray to the object's local coordinates
    - Less expensive!
  - Transform the ray origin by the inverse of the modeling transform
  - Transform the ray direction by the inverse of the modeling transform
    - Use a 0 as the homogenous factor – so the translate portion is ignored

$$r'(t) = M^{-1}\begin{bmatrix} R_x \\ R_y \\ R_z \\ 1 \end{bmatrix} + M^{-1}\begin{bmatrix} d_x \\ d_y \\ d_z \\ 0 \end{bmatrix}$$

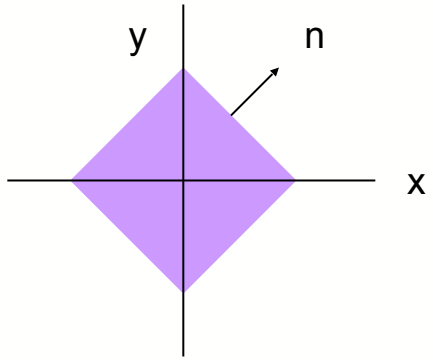
Where R is the ray origin and d is the ray direction

Handling Scaling

- Note that many ray intersection test methods assume a unit length direction vector for efficiency
  - For example: Ray3::Intersect(BoundingSphere)
- Any scaling within the modeling transformation **will impact the intersection distance (t)**
  - Need to “scale” t to counterbalance the change in distances the modeling transformation introduces
    - So we can recover the intersection point using the non-transformed ray

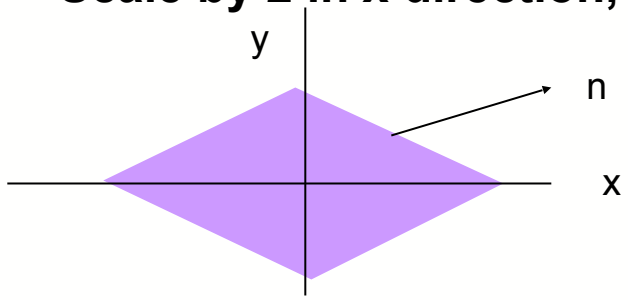
Transforming Normals

- Normal at the intersect point in local coordinates needs to be transformed to world coordinates
- Recall: normal vectors do not transform the same way as vertices
  - Non-uniform scaling in the modeling transform the normal vector will be distorted if transformed by the modeling transform



n = (0.707, 0.707, 0)

Scale by 2 in x direction, 1 in y



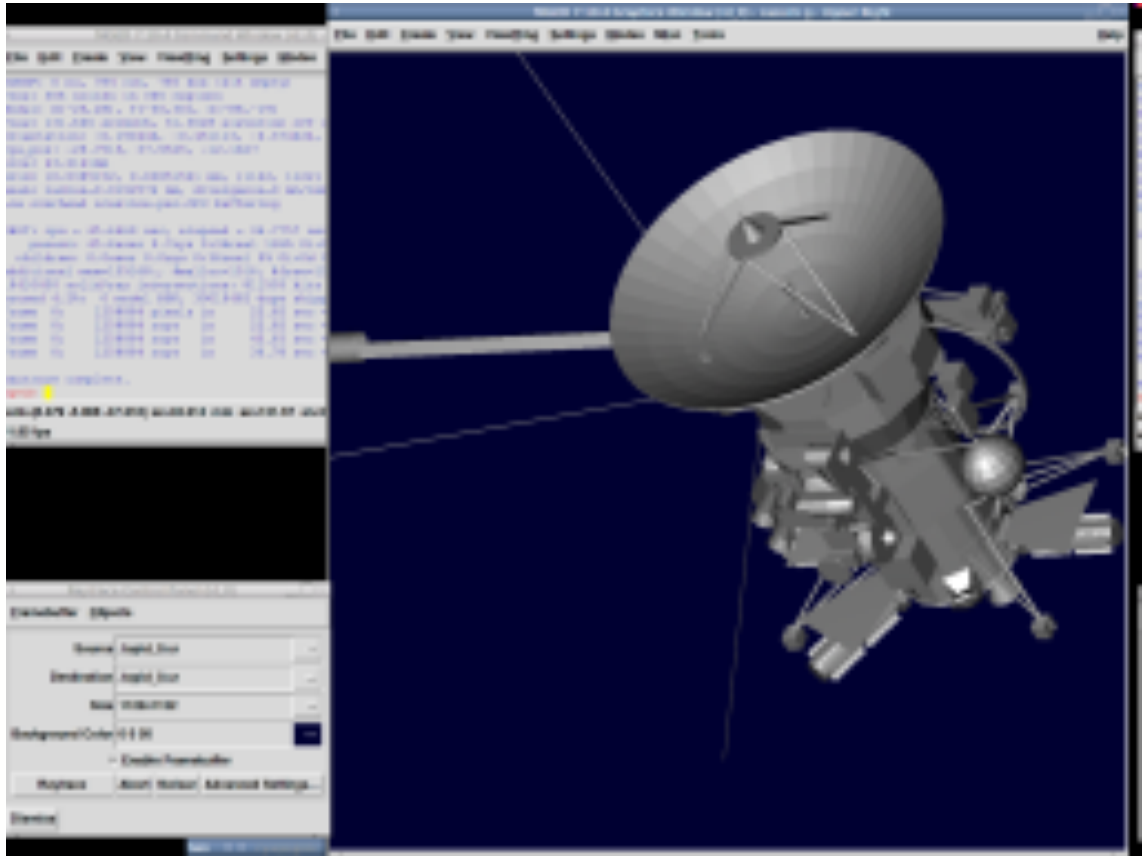
$$n = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.707 \\ 0.707 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1.414 \\ 0.707 \\ 0 \\ 1 \end{bmatrix}$$

Not Perpendicular to Surface!

Transforming Normals

- Think of normals as lying in a plane perpendicular to the surface
  - If p is a homogeneous plane (a,b,c,d) and v is a homogeneous vertex (x,y,z,w) on the plane, then pv = 0
    - ax+by+cz+dw = 0
  - Equivalent: $pM^{-1}Mv = 0$
    - If M is a nonsingular (has an inverse) vertex transformation matrix:
  - pM^{-1} is the **image of the plane** under the vertex transformation M
    - Plane must be transformed by premultiplication with the inverse of the modeling transform
- Alternatively, if v and n are perpendicular vectors
  - $n^Tv = 0$
    - Definition of perpendicular vectors
  - For any nonsingular transformation M: $n^TM^{-1}Mv = 0$
  - The transformed normal vector is: $(M^{-1})^Tn$
  - **Normal vectors are transformed by the inverse transpose of the matrix that transform vertices**

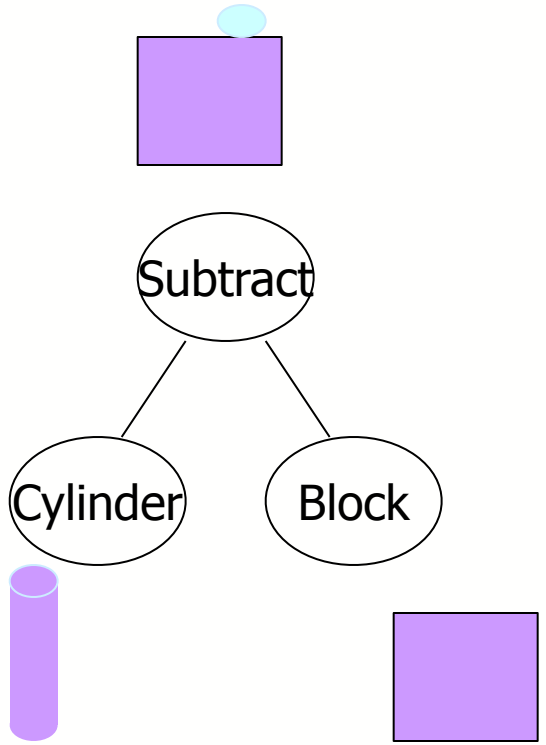
Constructive Solid Geometry



Cassini Probe
Using BRL-CAD – an Open Source Solid Modeling package
http://brlcad.org/

CSG Representation

- Object stored in a tree structure
  - Leaves contain simple, geometric primitives
  - Nodes store operators and transformations
  - Figure 2.13 in Watt provides another example
- Can produce geometrically complex objects
  - Figure 2.14 in Watt
- Simple editing
  - Increase diameter of a bolt hole: increase cylinder radius
    - Much more difficult for boundary representation

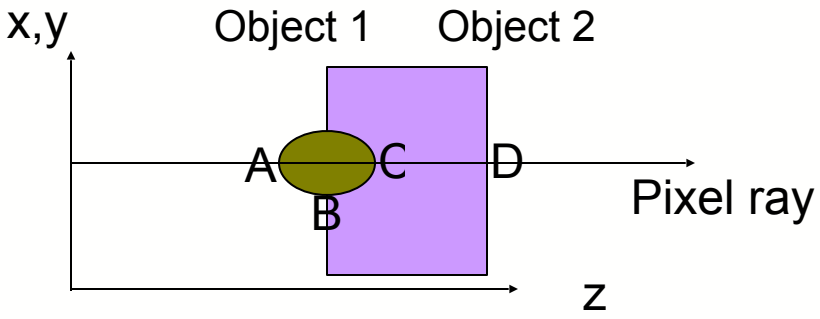


Rendering CSG Representations

- CSG trades rendering efficiency for interactivity
  - Easy to interact and change the object
  - Interactive design
  - Inefficient rendering
- CAD / CAM systems until recently were primarily wireframe
  - Special off-line solid rendering procedure
- Primary rendering strategies include:
  - CSG ray-tracing
    - http://web.cse.ohio-state.edu/~parent.1/classes/681/Lectures/19.RayTracingCSG.pdf
  - Conversion to space subdivision data type
    - Volume rendering
  - Conversion to polygon mesh
  - Rendering with a stencil buffer
    - Requires a polygon mesh surface representation of each primitive

CSG Ray-Tracing

- Watt: Section 4.3, F.S. Hill Section 12.13
- Expensive method, but reasonably simple
  - Cast a ray from each pixel in the view plane
    - Parallel projection simplifies since rays are parallel
  - For each ray - test intersection with each primitive
    - Sort intersections based on Z depth
  - Consider boolean operations between first 2 primitives along the ray to find the first intersect point
    - Determine limits for composite object based on boolean operation
  - Apply shading model based on simple reflection model



Operation	Surface Limits
Union	A,D
Intersection	B,C
Difference (O2-O1)	C,D

Tracing a Ray Through the View Plane

$$H = N\tan(\theta/2)$$

$$W = H * \text{aspect}$$

Camera extends from –W to +W in u and –H to +H in v
theta is the field of view angle in v direction
Aspect ratio is width over height of the framebuffer

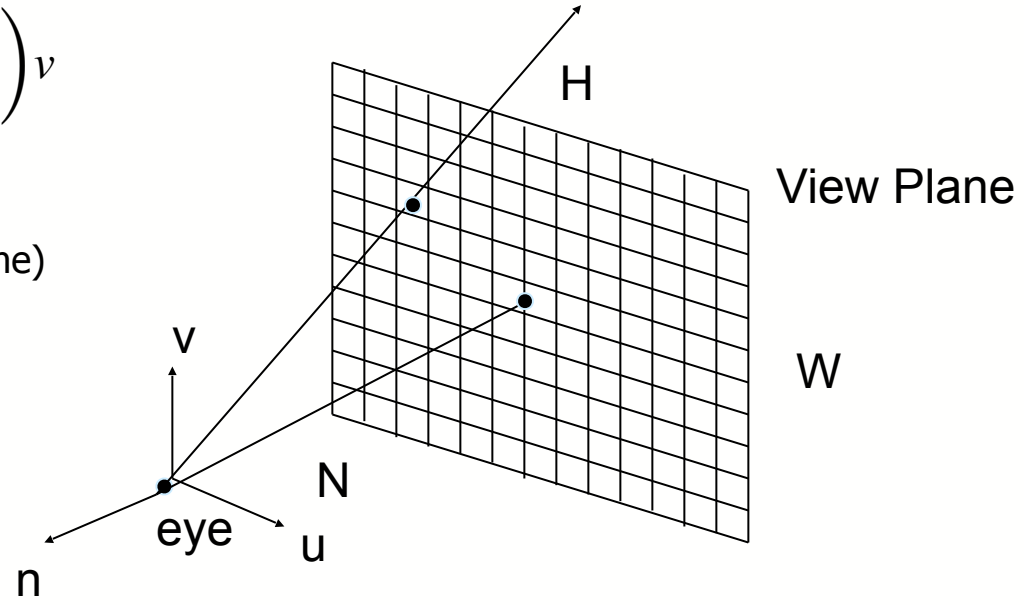
To find the ray through the pixel with row r and column c:

$$r(t) = \text{eye} + \text{ray}_{rc}t$$

$$\text{ray}_{rc} = -Nn + W\left(\frac{2c}{n\text{Cols}} - 1\right)u + H\left(\frac{2r}{n\text{Rows}} - 1\right)v$$

Eye is the eye/camera position in world coordinates
View axes are u,v,n
N is the near plane distance (distance from the eye to viewplane)
nRows, nCols are the image dimensions in pixels

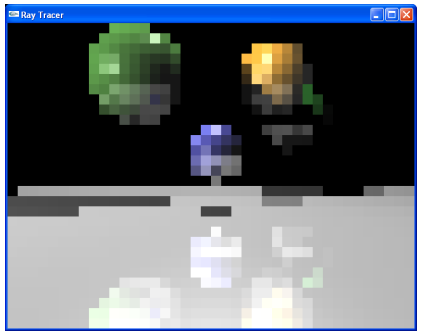
Extend CameraNode class from 605.667 to support construction of a ray through a pixel



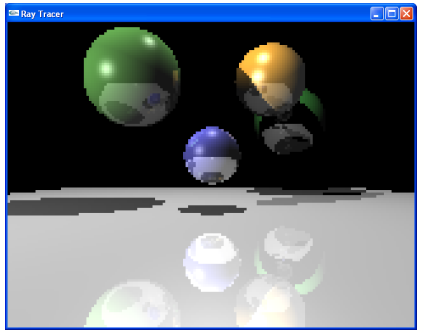
Drawing Pixel Blocks

- Ray casting/tracing can be time consuming
  - Would like to view scene prior to completion
  - Could render the scene with OpenGL
    - To verify camera and object positioning
- Sample lab renders “blocks” of pixels to help debug and test
  - Trace rays through one pixel, apply to a group of pixels
    - Builds a coarse approximation on first iteration
    - Successive iterations refine the image by reducing pixel block size
    - Can build a “memory framebuffer” to support pixel block drawing
  - Maintain a list of pixels that have been set
    - To avoid recalculating color at these pixels on successive iterations

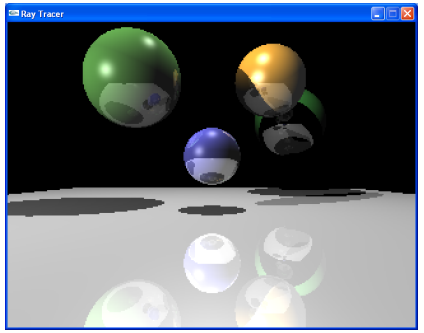
16



4



1



Pseudo Code for Pixel Block Drawing

```
void display()
{
   MyFrameBuffer->Clear();            //Clear the memory framebuffer
   for (int blockSize = 16; blockSize > 0; blockSize /= 2)
   {
      for (int x = 0; x < imageWidth; x += blockSize)
      {
         for (int y = 0; y < imageHeight; y += blockSize)
         {
            if (!MyFrameBuffer->Set(x, y)) // Check if framebuffer has been set for this pixel
            {
               // Construct a ray through the specified pixel and find the color
               // using the ray caster or recursive ray tracer
               MyFrameBuffer->Set(x, y, color, blockSize);
            }
         }
      }
      MyFrameBuffer->Render(); // Render the framebuffer
   }
}
```