CS 543 - Computer Graphics: Ray Tracing Detail, Part 1

by

Robert W. Lindeman

gogo@wpi.edu

(with help from Emmanuel Agu ;-)

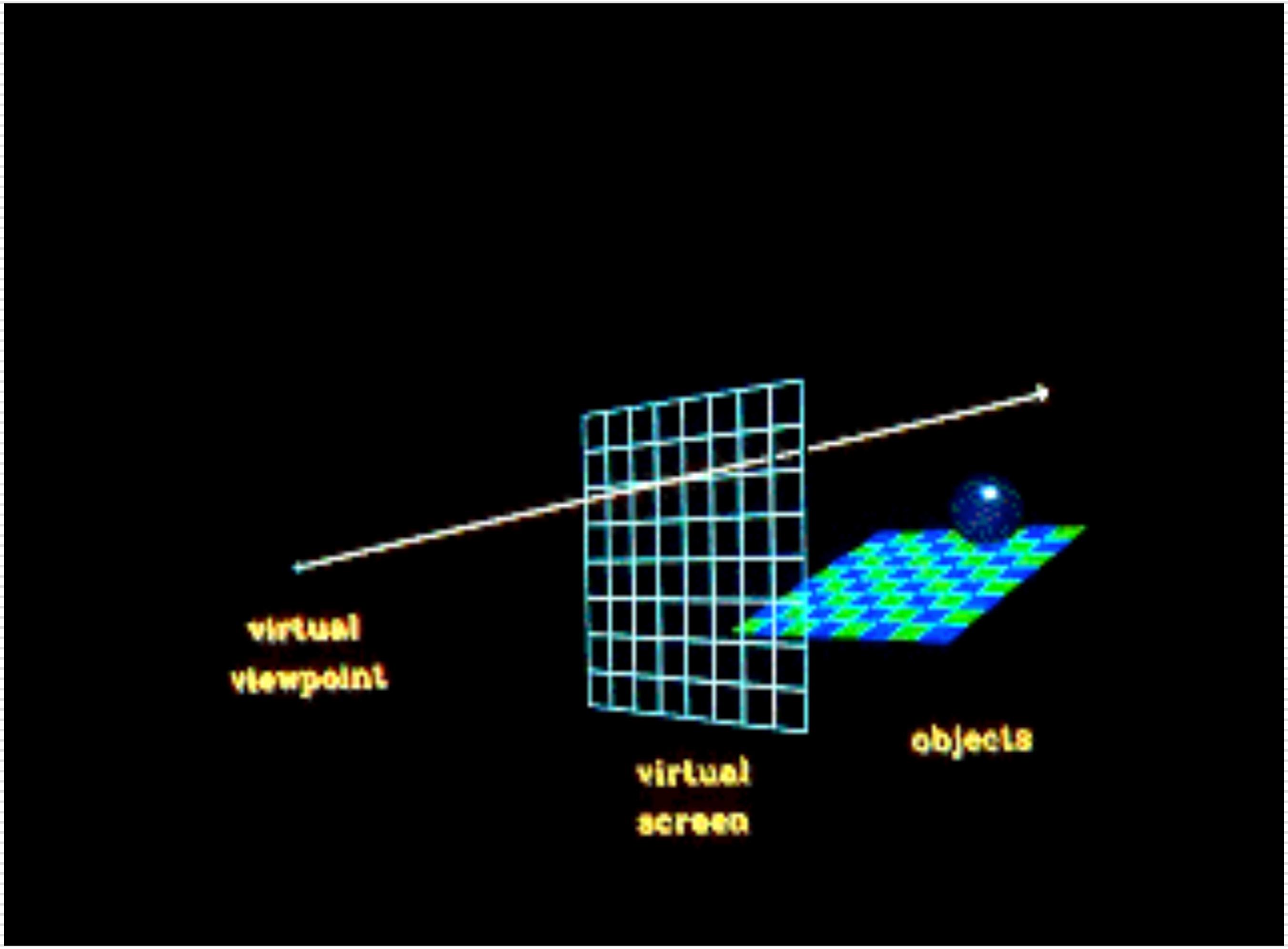
Ray Tracing

- Global illumination-based rendering method
- Simulates rays of light, natural lighting effects
- Because light path is traced, handles some effects that are tough for OpenGL:
  - Shadows
  - Multiple inter-reflections
  - Transparency
  - Refraction
  - Texture mapping
- Newer variations… e.g. photon mapping (caustics, participating media, smoke)
- **Note:** Ray Tracing can be whole semester graduate course!

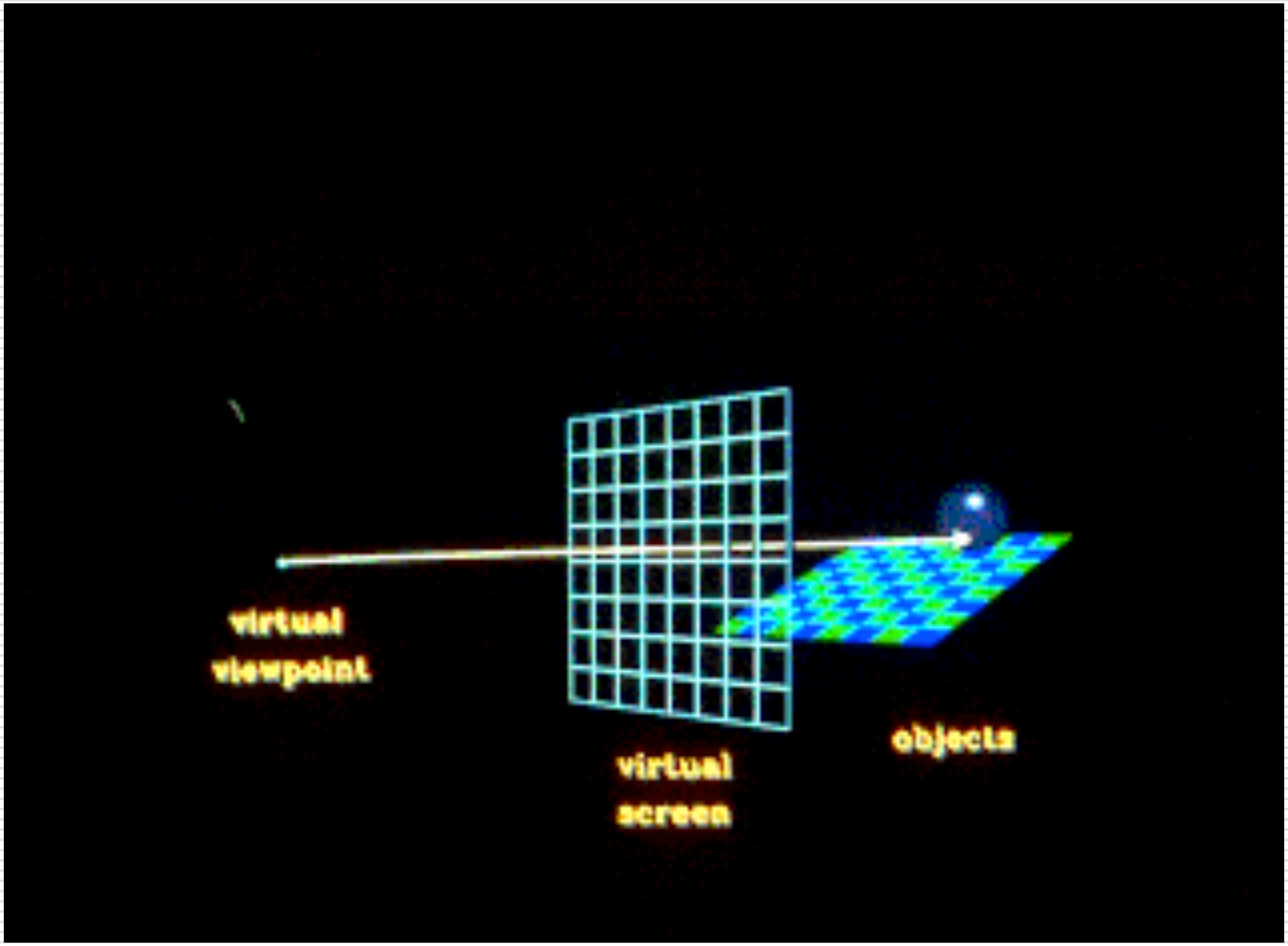
How Ray Tracing Works

- OpenGL is object space rendering
  - start from world objects, rasterize them
- Ray tracing is image space method
  - Start from pixel, what do you see through this pixel?
- Looks through each pixel (e.g., 640 x 480)
- Determines what eye sees through pixel
- Basic idea:
  - Trace light rays: eye -> pixel (image plane) -> scene
  - If a ray intersect any scene object in this direction
    - Yes? render pixel using object color
    - No? it uses the background color
- Automatically solves hidden surface removal problem

Case A: Ray misses all objects

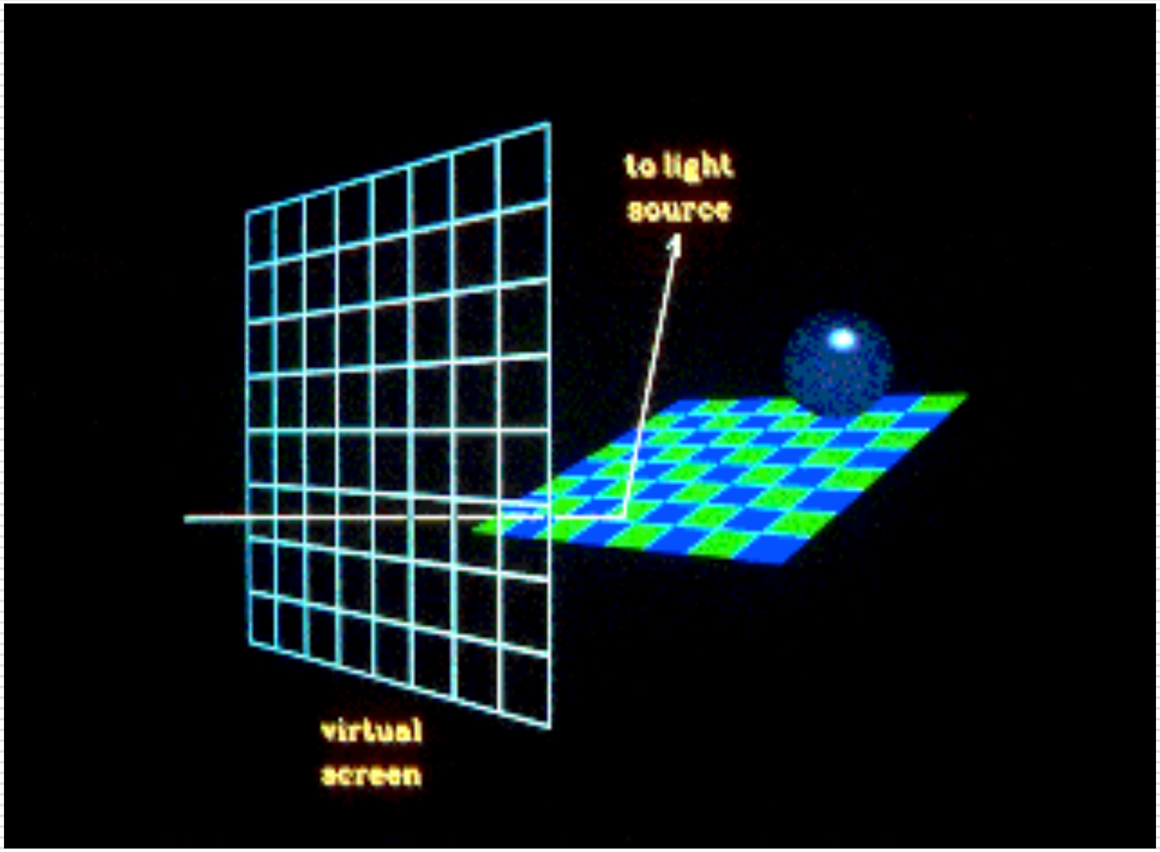


Case B: Ray hits an object



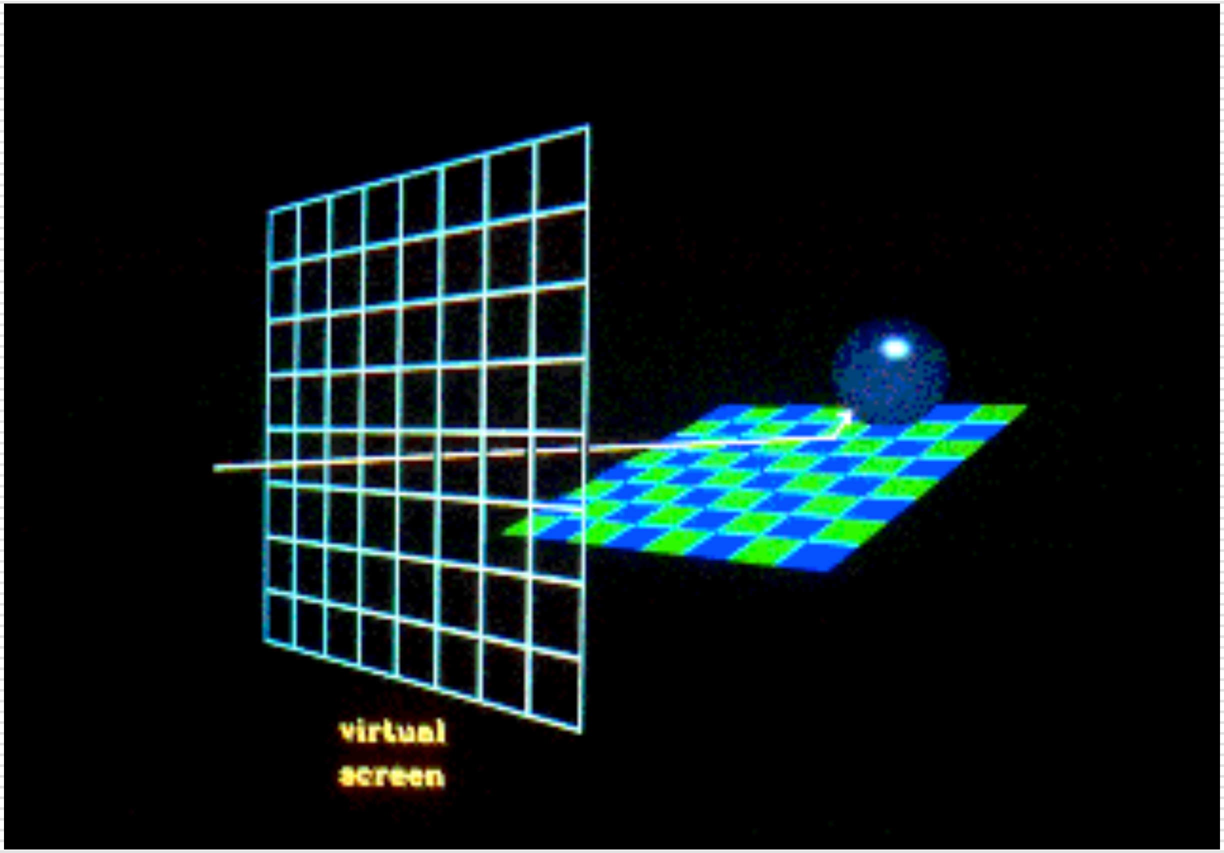
Case B: Ray hits an object

Ray hits object: Check if hit point is in shadow, build secondary ray (shadow ray) towards light sources.



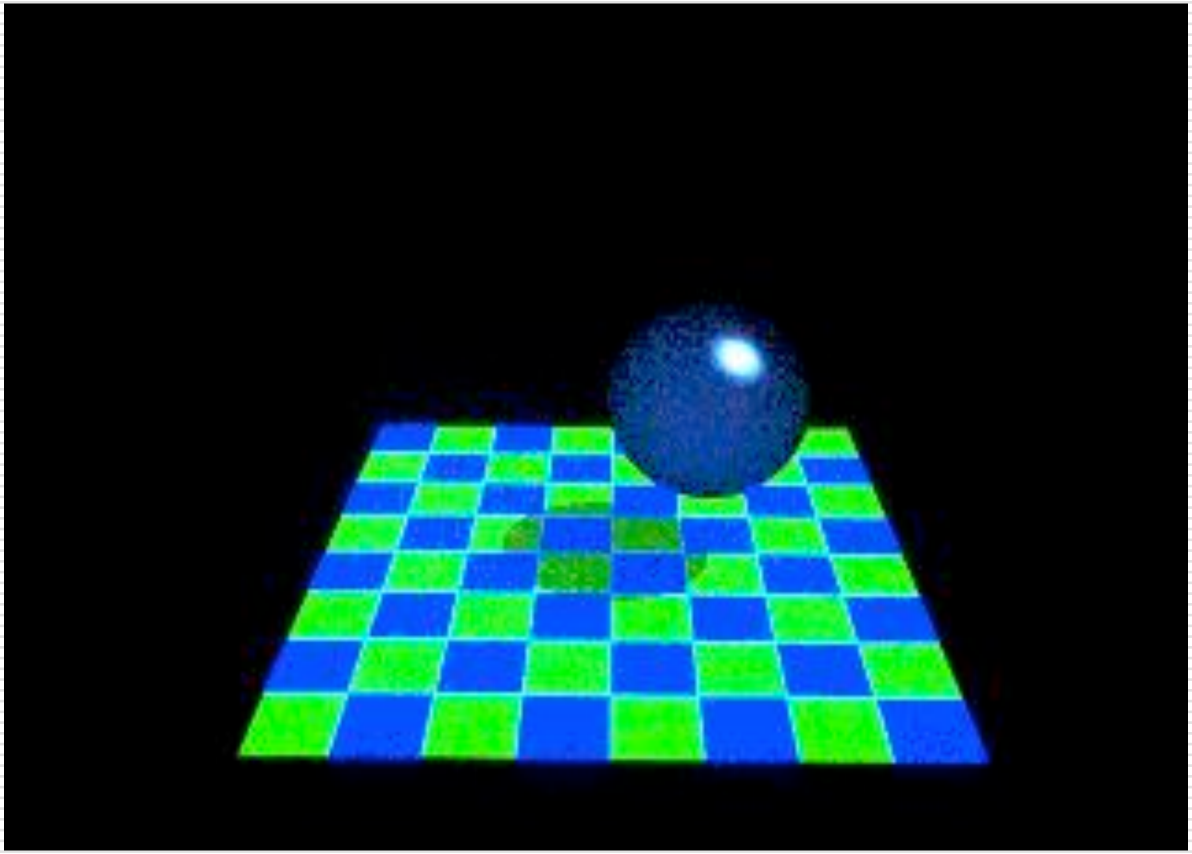
Case B: Ray hits an object

If shadow ray hits another object before light source: first intersection point is in shadow of the second object. Otherwise, collect light contributions



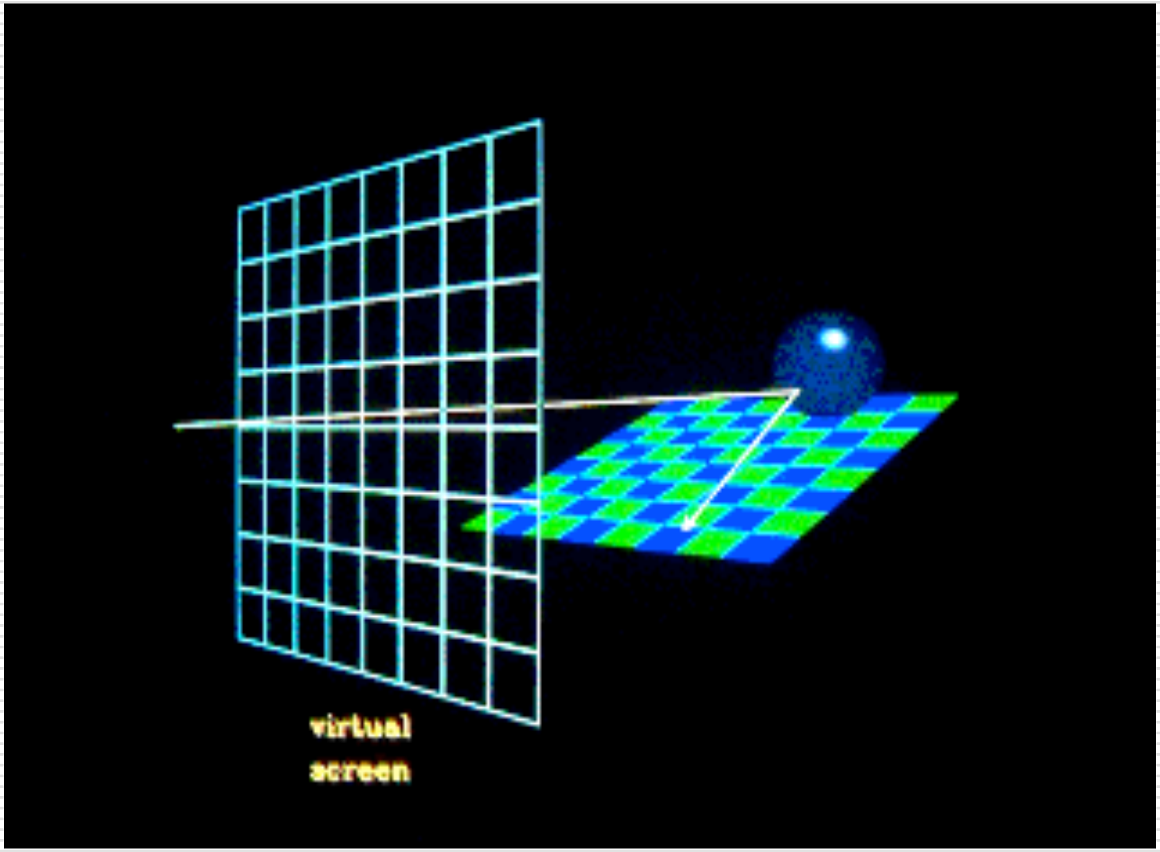
Case B: Ray hits an object

First Intersection point in the shadow of the second object is the shadow area.

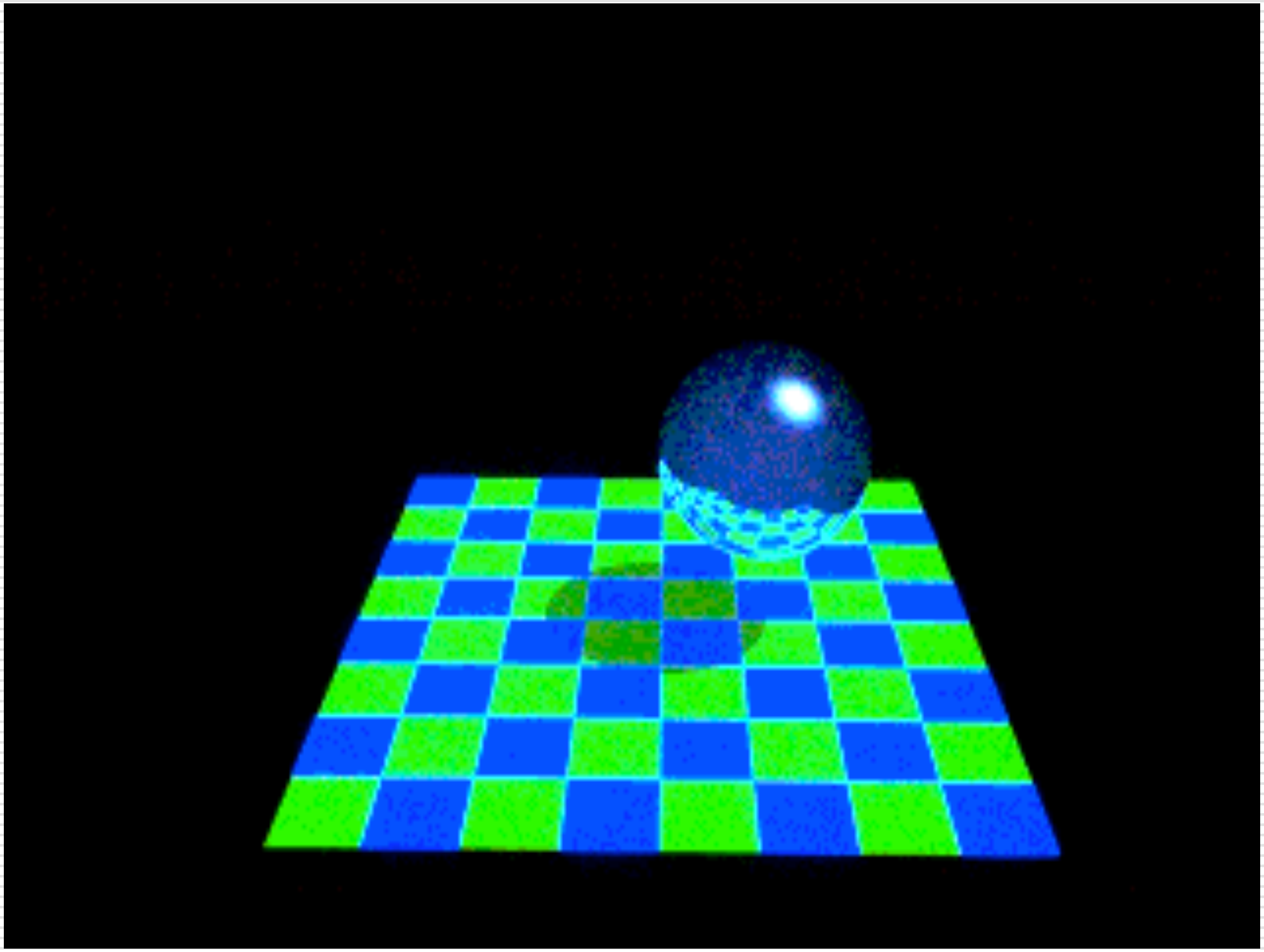


Reflected Ray

When a ray hits an object, a reflected ray is generated which is tested against all of the objects in the scene.

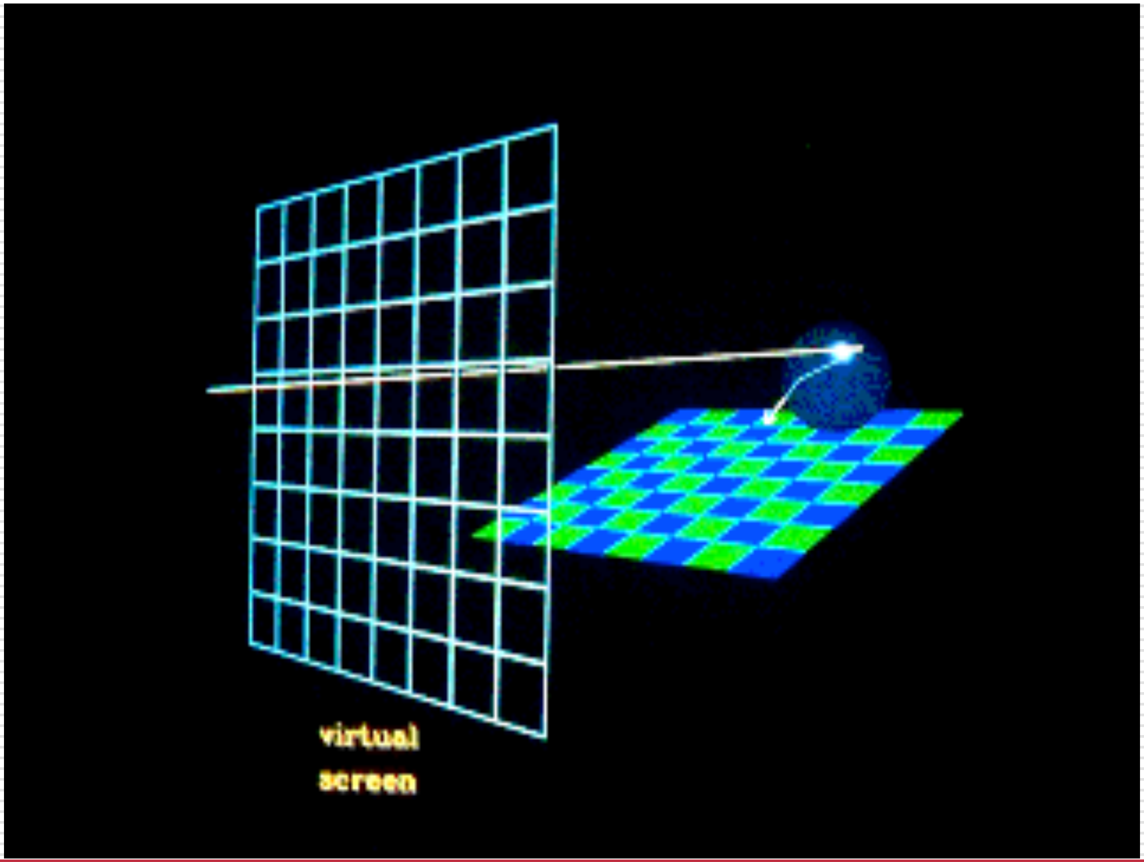


Reflection: Contribution from the reflected ray

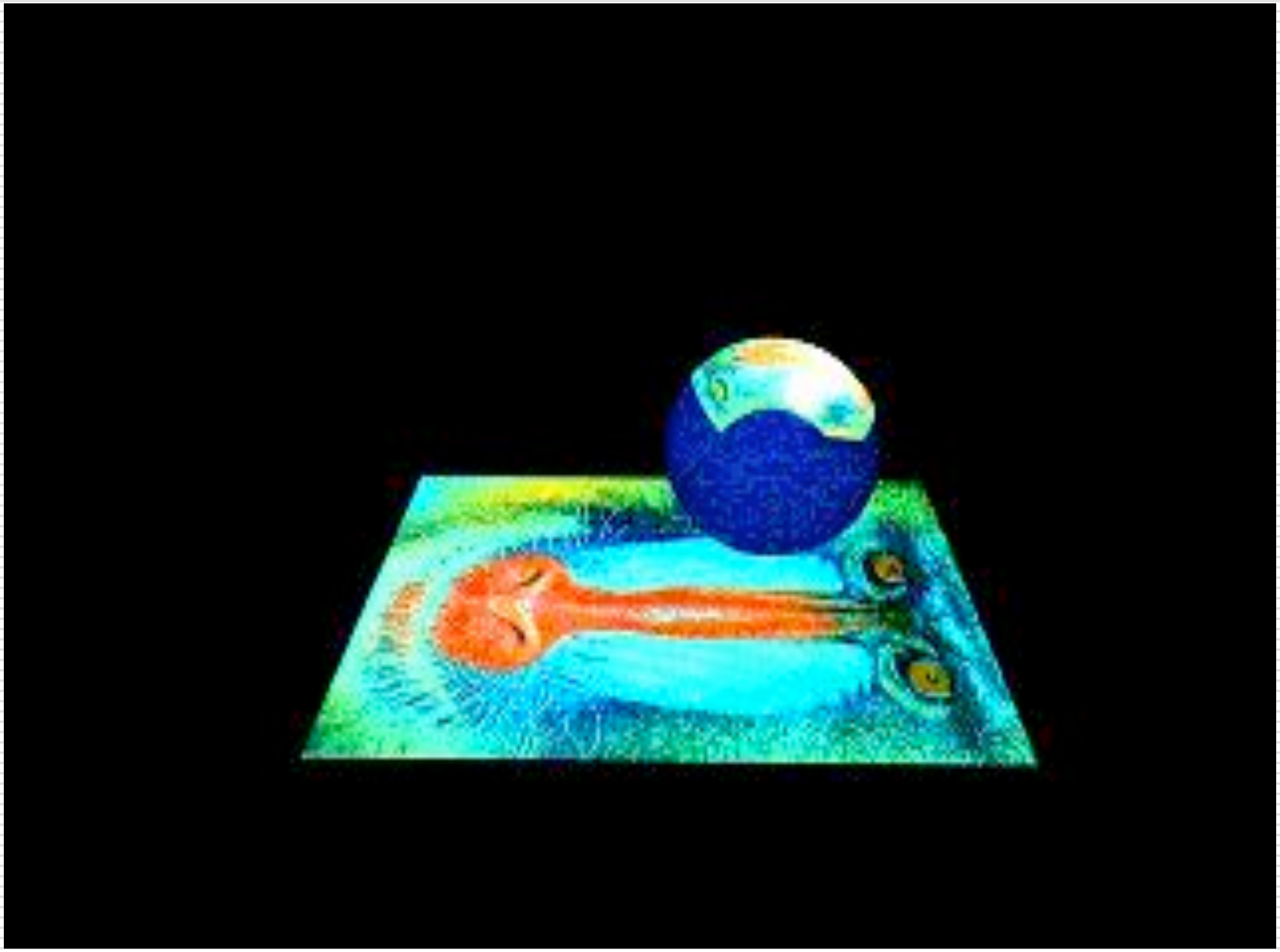


Transparency

If intersected object is transparent, transmitted ray is generated and tested against all the objects in the scene.

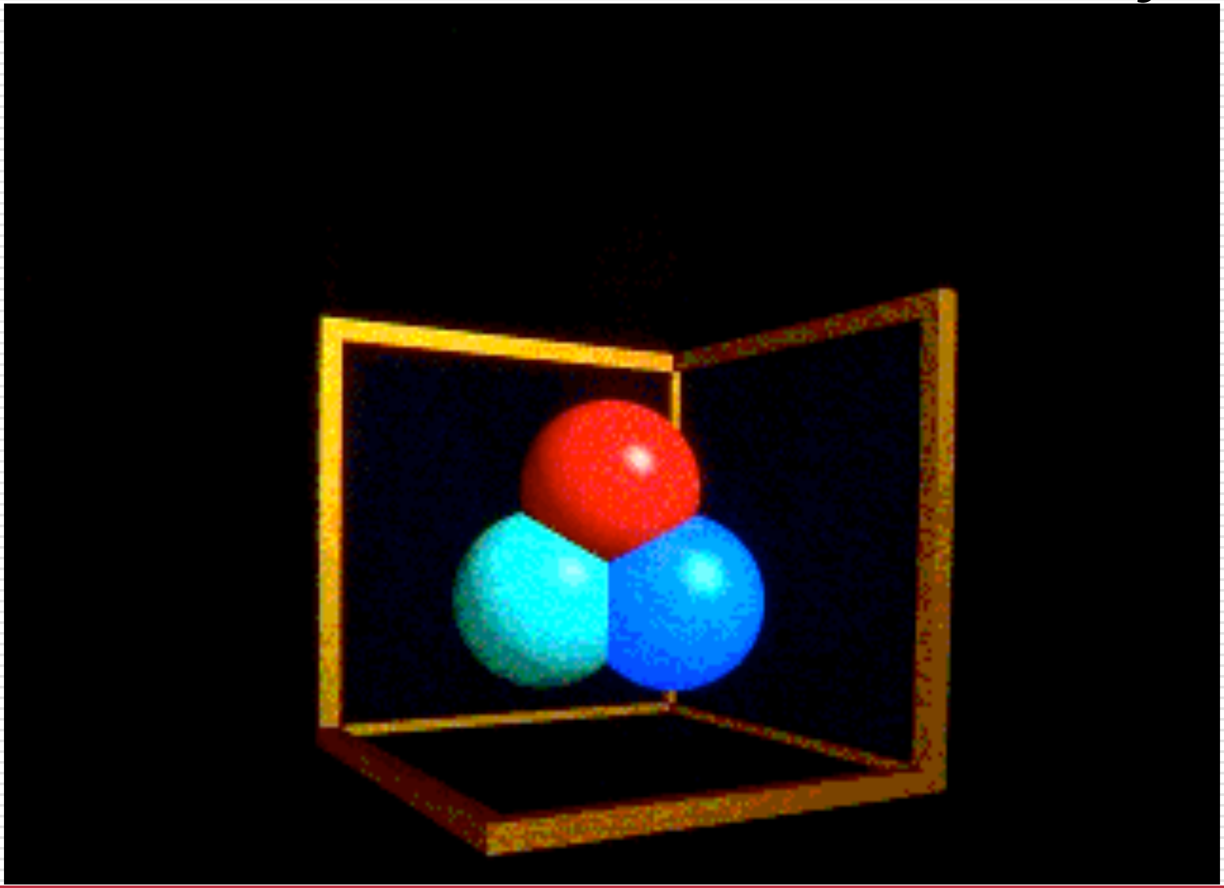


Transparency: Contribution from transmitted ray



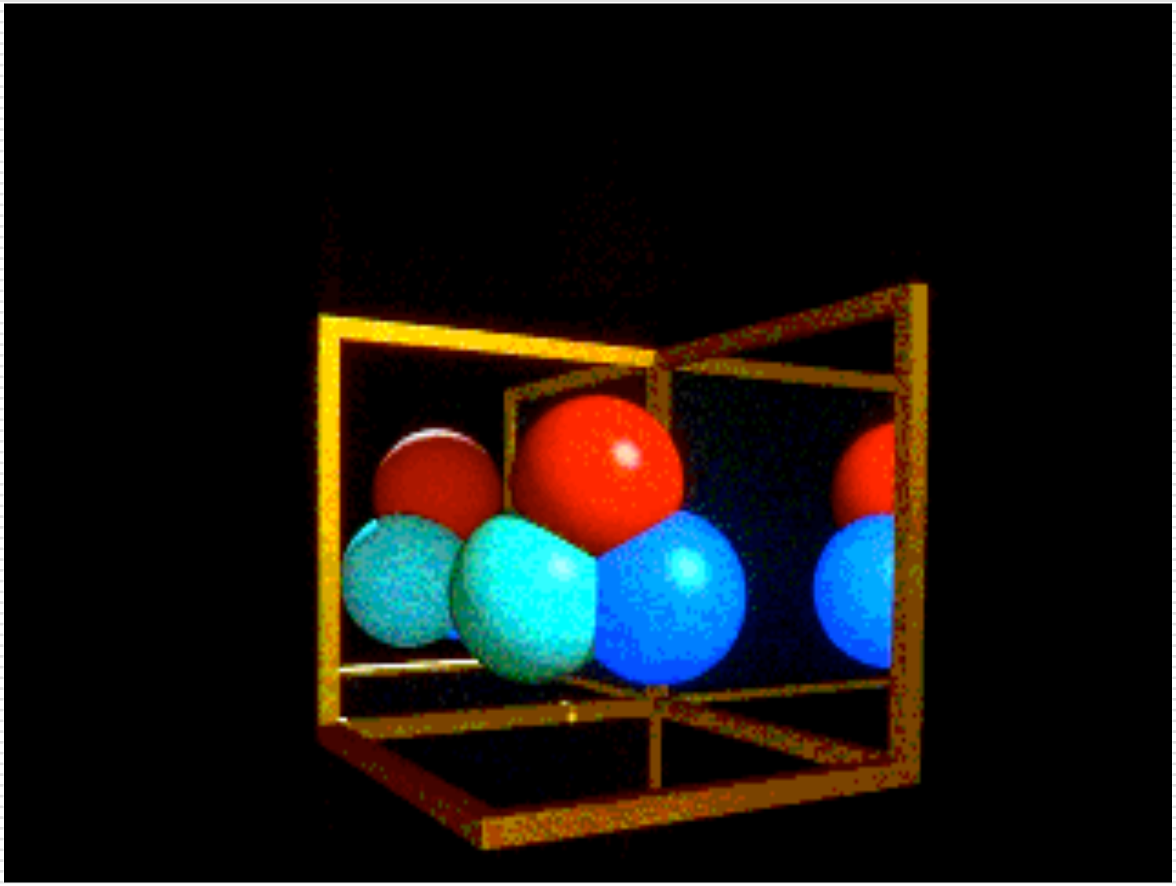
Reflected Ray: Recursion

Case A: *Scene with no reflection rays*



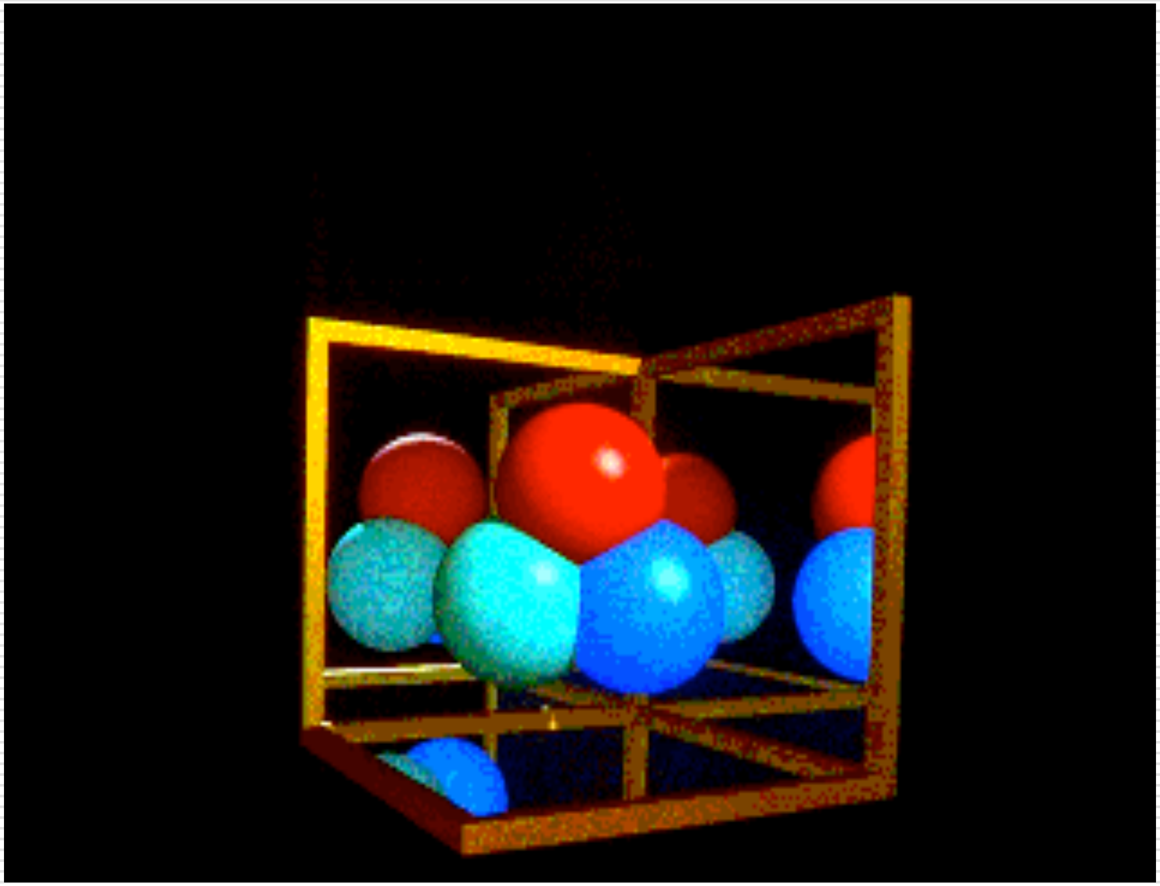
Reflected Ray: Recursion

Case B: *Scene with one level of reflection*



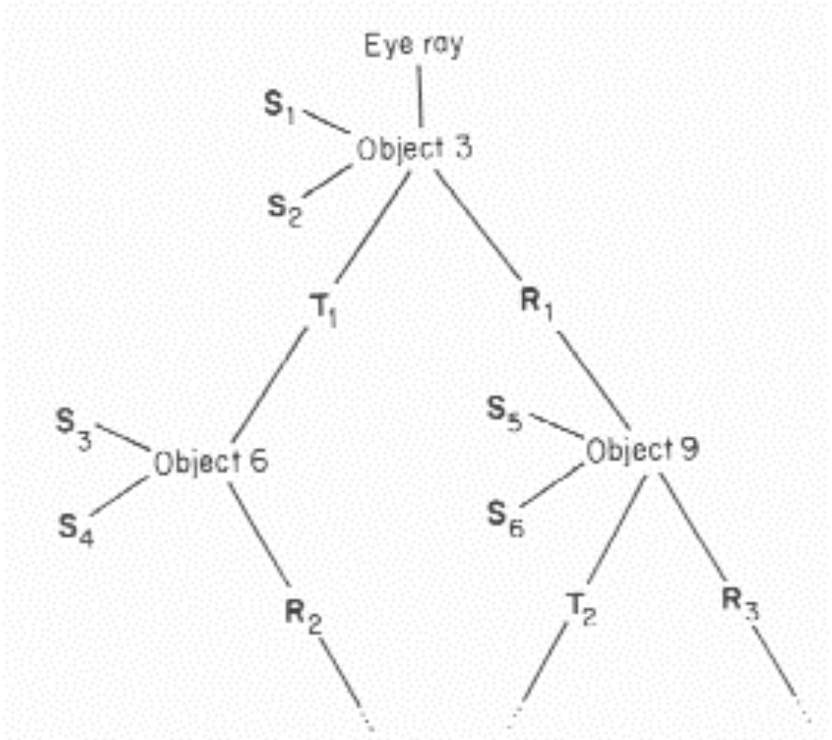
Reflected Ray: Recursion

Case C: *Scene with two levels of reflection*



Ray Tree

- Reflective and/or transmitted rays are continually generated until the ray leaves the scene without hitting any object, or a preset recursion level has been reached.



Ray-Object Intersections

- So, express ray as equation (origin is eye, pixel determines direction)
- Define a ray as:

  R0 = [x0, y0, z0] - origin of ray

  Rd = [xd, yd, zd] - direction of ray
- then define parametric equation of ray:

  R(t) = R0 + Rd * t with t > 0.0
- Express all objects (sphere, cube, etc.) mathematically
- Ray tracing idea:
  - Put ray mathematical equation into object equation
  - Determine if real solution exists.
  - Object with smallest hit time is object seen

Ray-Object Intersections

- Dependent on parametric equations of object
- Ray-Sphere Intersections
- Ray-Plane Intersections
- Ray-Polygon Intersections
- Ray-Box Intersections
- Ray-Quadric Intersections

  (cylinders, cones, ellipsoids, paraboloids)

Writing a RayTracer

- The first step is to create the model of the objects
- One should **NOT** hardcode objects into the program, but instead use an input file.
- This is called retained mode graphics
- We will use SDL
- Ray trace SDL files
- The output image/file will consist of three intensity values (Red, Green, and Blue) for each pixel.

Accelerating Ray Tracing

- Ray Tracing is very time-consuming because of intersection calculations
- Each intersection requires from a few (5-7) to many (15-20) floating point operations (fpos)
- Example: for a scene with 100 objects and computed with a spatial resolution of 512 x 512, assuming 10 fpos per object test there are about 250,000 X 100 X 10 = 250,000,000 fpos.
- Solutions:
  - Use faster machines
  - Use specialized hardware, especially parallel processors.
  - Note: ray tracing did not use 3D graphics cards (until now!)
  - Speed up computations by using more efficient algorithms
  - Reduce the number of ray - object computations

Reducing Ray-Object Intersections

- Adaptive Depth Control:
  - Stop generating reflected/transmitted rays when computed intensity becomes less than certain threshold.
- Bounding Volumes:
  - Enclose groups of objects in sets of hierarchical bounding volumes
  - First test for intersection with the bounding volume
  - Then only if there is an intersection, against the objects enclosed by the volume.
- First Hit Speed-Up:
  - Use modified Z-buffer algorithm to determine the first hit.

Writing a Ray Tracer

- Our approach:
  - Give arrangement of minimal ray tracer
  - Use that as template to explain process
- Minimal?
  - Yes! Basic framework
  - Just two object intersections
  - Minimal/no shading
- Paul Heckbert (CMU):
  - Ran ray tracing contest for years
  - Wrote ray tracer that fit on back of his business card

Pseudocode for Ray Tracer

- Basic idea

```
color Raytracer  {
  for( each pixel direction ) {
    determine first object in this pixel direction
    calculate color shade
    return shade color
  }
}
```

More Detailed Ray Tracer Pseudocode (fig 12.4)

```
Define the objects and light sources in the scene
Set up the camera
for(int r = 0; r < nRows; r++){
   for(int c = 0; c < nCols; c++){
        1. Build the rc-th ray
        2. Find all object intersections with rc-th ray
        3. Identify closest object intersection
        4. Compute the "hit point" where the ray hits the
             object, and normal vector at that point
        5. Find  color of light to eye along ray
        6. Set rc-th pixel to this color
   }
}
```

Define Objects and Light Sources in Scene

- Already know SDL, use it for input format
- Previously, in our program

```
Scene scn;
…
scn.read("your scene file.dat"); // reads scene file
scn.makeLightsOpenGL( ); // builds lighting data struct.
scn.drawSceneOpenGL( ); // draws scene using OpenGL
```

- Previously, OpenGL did most of the work, rendering
- Now, we replace drawSceneOpenGL with ray tracing code
- Minimally use OpenGL for setting pixel color

Set OpenGL up for 2D

- Ray tracing will do all the work
  - figure our pixel color
- Set OpenGL up for 2D drawing
- Just like project 2 (dino.dat)

```
// set up OpenGL for simple 2D drawing
glMatrixMode( GL_MODELVIEW );
glLoadIdentity( );
glMatrixMode( GL_PROJECTION );
glLoadIdentity( );
gluOrtho2D( 0, nCols, 0, nRows );
glDisable( GL_LIGHTING ); //we will handle lighting
…
do ray tracing
```

Ray Tracer Pseudocode

```
Define the objects and light sources in the scene
Set up the camera
for(int r = 0; r < nRows; r++){
   for(int c = 0; c < nCols; c++){
        1. Build the rc-th ray
        2. Find all object intersections with rc-th ray
        3. Identify closest object intersection
        4. Compute the "hit point" where the ray hits the
           object, and normal vector at that point
        5. Find  color of light to eye along ray
        6. Set rc-th pixel to this color
   }
}
```

Setting RC-th pixel to Calculated Color

- Can do as before. i.e. first set drawing color, then send vertex

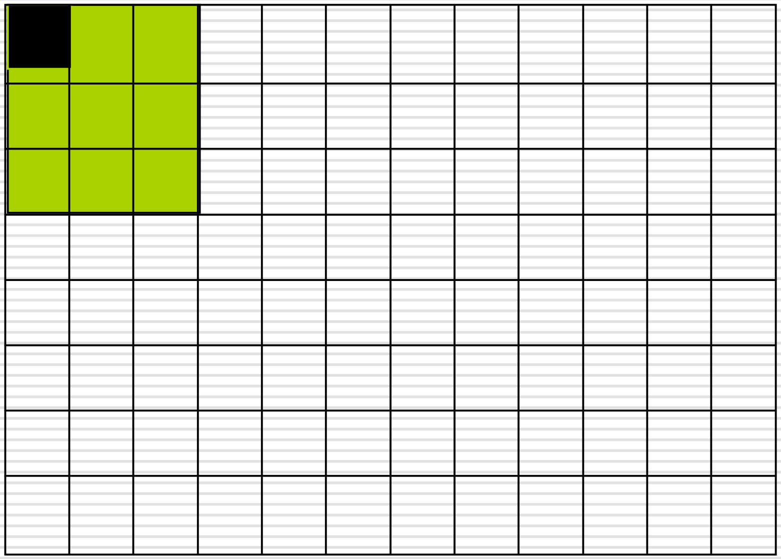
```
glColor3f( red, green, blue );  // set drawing color
glPointSize( 1.0 );             // set point size to 1

//…. .then send vertices
glBegin( GL_POINTS )
  glVertex2i( 100, 130 );
glEnd( );
```

- But ray tracing can take time.. minutes, days, weeks!! ☺?
- Use notion of blocksize to speedup ray tracing

Setting RC-th pixel to Calculated Color

- Break screen into blocks (fat pixels)
- Ray trace only top-left pixel of block
- 1 calculation, set entire block to calculated color
- E.g. blockSize = 3, ray trace, top-left pixel, set entire block to green



- Affects resolution of picture
- Initially use large block size to verify code, then set to 1

Modified Ray Tracer Pseudocode Using blockSize

```
Define the objects and light sources in the scene
Set up the camera
For(int r = 0; r < nRows; r+= blockSize){
  for(int c = 0; c < nCols; c+= blockSize){
      1. Build the rc-th ray
      2. Find all object intersections with rc-th ray
      3. Identify closest object intersection
      4. Compute the "hit point" where the ray hits the
          object, and normal vector at that point
      5. Find  color (clr) of light to eye along ray
         glColor3f(clr.red, clr.green, clr.blue);
         glRecti(c, r, c + blockSize, r + blockSize);
  }
}
```

Modified Ray Tracer Pseudocode Using blockSize

```
Define the objects and light sources in the scene
Set up the camera
For(int r = 0; r < nRows; r+= blockSize){
   for(int c = 0; c < nCols; c+= blockSize){
        1. Build the rc-th ray
        2. Find all object intersections with rc-th ray
        3. Identify closest object intersection
        4. Compute the "hit point" where the ray hits the
             object, and normal vector at that point
        5. Find  color (clr) of light to eye along ray
           glColor3f(clr.red, clr.green, clr.blue);
           glRecti(c, r, c + blockSize, r + blockSize);
   }
}
```

Build the RC-th Ray

- Parametric expression ray starting at eye and passing through pixel at row r, and column c

$$ray = origin + (direction)t$$

$$r(t) = eye + dir_{rc}t$$

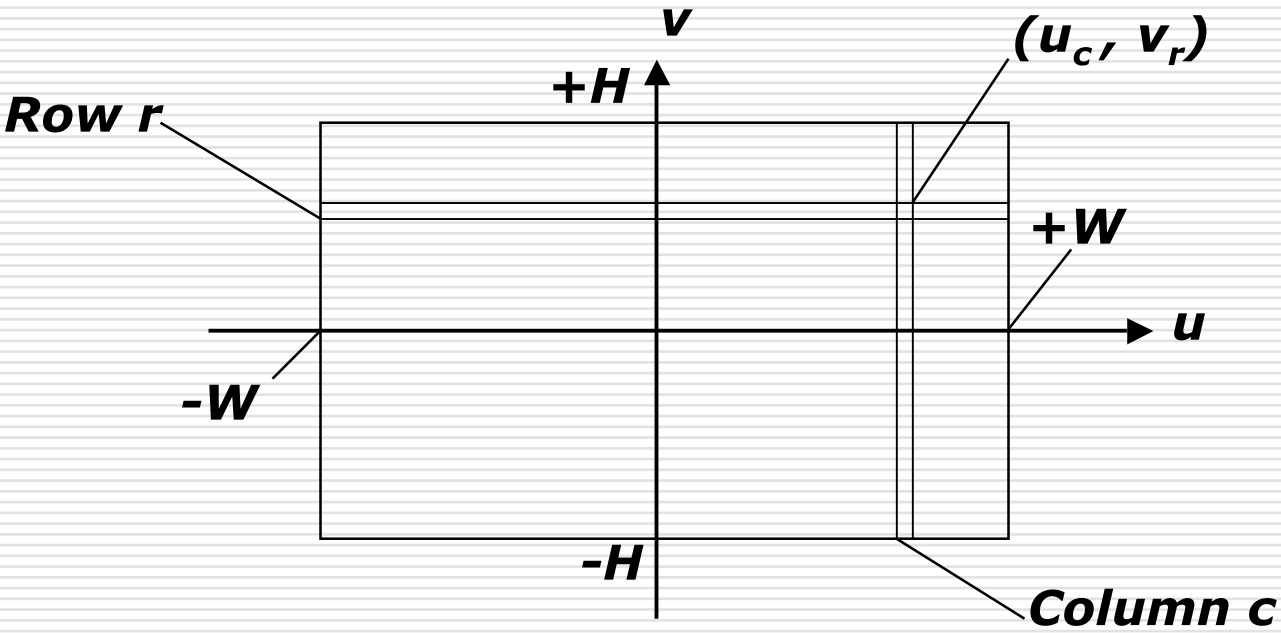
- But what exactly is this $dir_{rc}(t)$?
- Need to express ray direction in terms of variables r and c
- Now need to set up camera, and then express dir_{rc} in terms of camera r and c

Modified Ray Tracer Pseudocode Using blockSize

```
Define the objects and light sources in the scene
Set up the camera
for(int r = 0; r < nRows; r+= blockSize){
  for(int c = 0; c < nCols; c+= blockSize){
      1. Build the rc-th ray
      2. Find all object intersections with rc-th ray
      3. Identify closest object intersection
      4. Compute the "hit point" where the ray hits the
          object, and normal vector at that point
      5. Find  color (clr) of light to eye along ray
      glColor3f(clr.red, clr.green, clr.blue);
      glRecti(c, r, c + blockSize, r + blockSize);
  }
}
```

Set up Camera Geometry

- As before, camera has axes (**u, v, n**) and position **eye** with coordinates (eye.x, eye.y, eye.z)
- Camera extends from –W to +W in **u-direction**
- Camera extends from –H to +H in **v-direction**



Set up Camera Geometry

- Viewport transformation?
- Simplest transform: **viewport is pasted onto window at near plane**
  - Viewport (screen) width: 1 to nCols (or 0 to nCols –1)
  - Window width: -W to +W
- Can show that a given c maps to

$$u_c = -W + W \frac{2c}{nCols}$$

for c = 0, 1,......nCols - 1

Set up Camera Geometry

- Similarly
  - viewport (screen) height:
    - 1 to nRows ….(or 0 to nRows –1)
  - Window width: -H to +H
- Can show that a given r maps to

$$v_r = -H + H\frac{2r}{nRows}$$

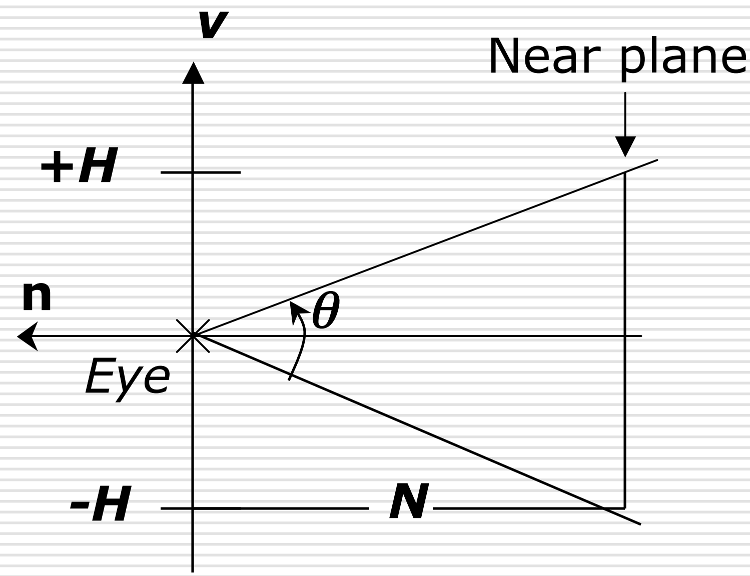
for r = 0, 1,…...nRows - 1

Set up Camera Geometry

- Near plane lies distance ***N*** along **n** axis
- Camera has aspect ratio ***aspect*** and view angle θ
- Such that

$$H = N\tan(\theta/2)$$
$$W = H \cdot aspect$$

- Thus pixel (r, c) location expressed in terms of **u, v** and **n**



$$eye - N\mathbf{n} + u_c\mathbf{u} + v_r\mathbf{v}$$

Set up Camera Geometry

- So, pixel location ..Near plane lies distance ***N*** along **n** axis

$$eye = -N\mathbf{n} + u_c\mathbf{u} + v_r\mathbf{v}$$

- Parametric form of ray starting at eye and going through pixel is then. **Note:** eye is at t =0, hits pixel at t = 1

$$r(t) = eye(1-t) + (eye - N\mathbf{n} + u_c\mathbf{u} + v_r\mathbf{v})t$$

- Manipulating expressions, if

$$r(t) = eye + \mathbf{dir}_{rc}t$$

$$\mathbf{dir}_{rc} = -N\mathbf{n} + W(\frac{2c}{nCols} - 1)\mathbf{u} + (\frac{2r}{nRows} - 1)\mathbf{v}$$

Set up Camera Geometry

- So, ray starts at t =0, hits pixel at t = 1
- Ray hits scene objects at time $t_{hit} > 1$
- If $t_{hit} < 0$, object is behind the eye
- For a given ray, if two objects have hit times t1 and t2, smaller hit time is closer to eye
- In fact, for all hit times along ray, smallest hit time is closest
- If we know hit time of an object, t_{hit}, we can solve for object's position (x, y, z) in space as

$$P_{hit} = eye + \mathbf{dir}_{rc} t_{hit}$$

- Do this separately for x, y and z
- Thus automatically, ray tracing solves Hidden surface removal problem

Where are we?

```
Define the objects and light sources in the scene
Set up the camera
for(int r = 0; r < nRows; r+= blockSize){
  for(int c = 0; c < nCols; c+= blockSize){
      1. Build the rc-th ray
      2. Find all object intersections with rc-th ray
      3. Identify closest object intersection
      4. Compute the "hit point" where the ray hits the
           object, and normal vector at that point
      5. Find  color (clr) of light to eye along ray
      glColor3f(clr.red, clr.green, clr.blue);
      glRecti(c, r, c + blockSize, r + blockSize);
  }
}
```

References

- Hill, chapter 12
- http://www.siggraph.org/education/materials/HyperGraph/raytrace/rtrace0.htm