

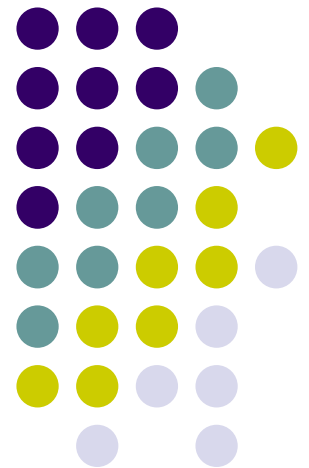
Computer Graphics (CS 543)

Lecture 10: Part 2

Ray Tracing (Part 1)

Prof Emmanuel Agu

*Computer Science Dept.
Worcester Polytechnic Institute (WPI)*





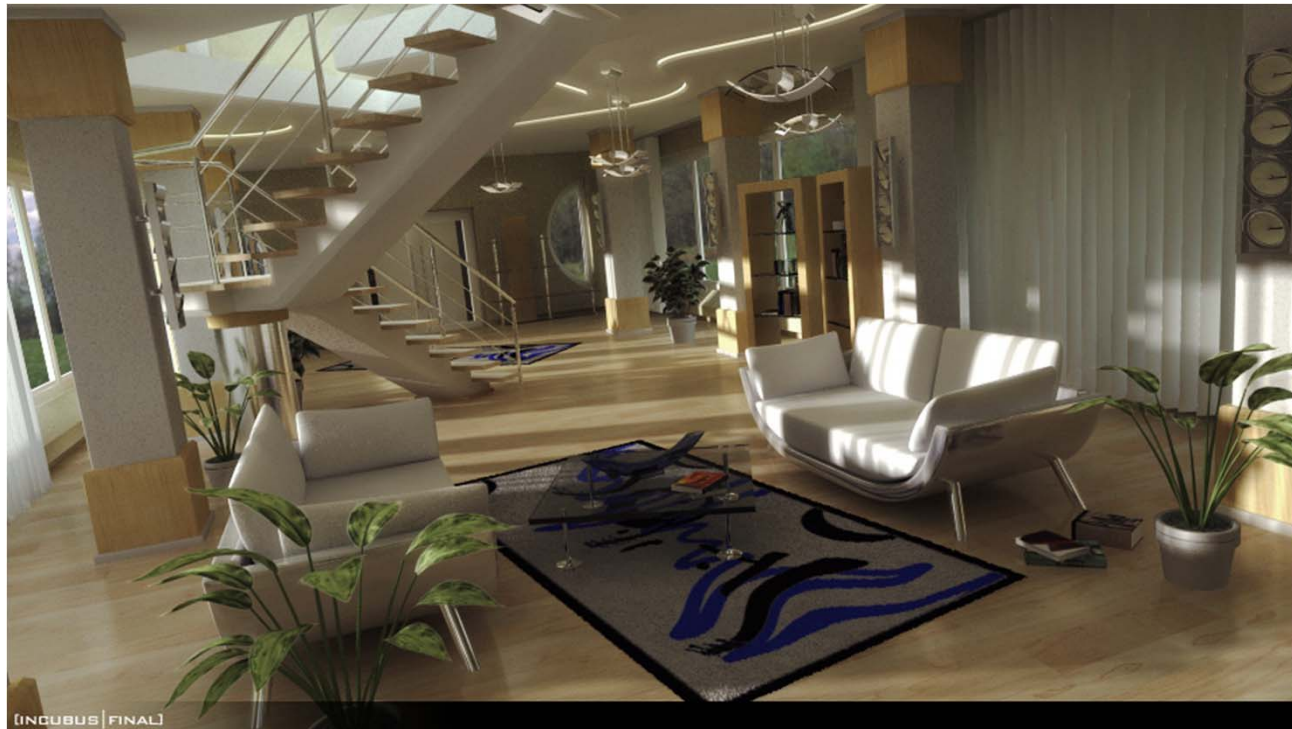
Raytracing

- Global illumination-based rendering method
- Simulates rays of light, natural lighting effects
- Because light path is traced, handles effects tough for OpenGL:
 - Shadows
 - Multiple inter-reflections
 - Transparency
 - Refraction
 - Texture mapping
- Newer variations... e.g. photon mapping (caustics, participating media, smoke)
- **Note:** raytracing can be semester graduate course
- Today: start with high-level description



Raytracing Uses

- Entertainment (movies, commercials)
- Games (pre-production)
- Simulation (e.g. military)
- Image: Internet Ray Tracing Contest Winner (April 2003)



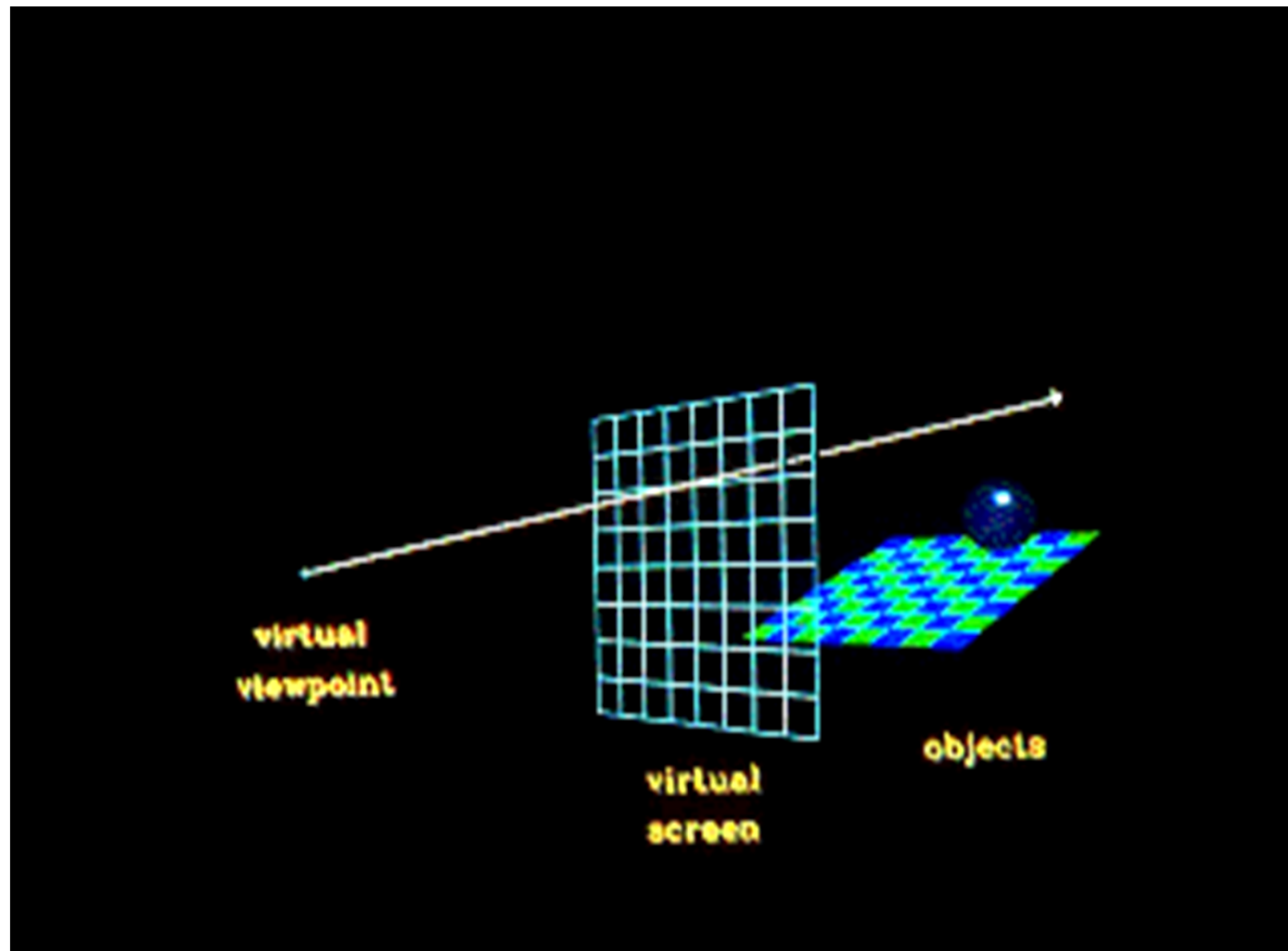


How Raytracing Works

- 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

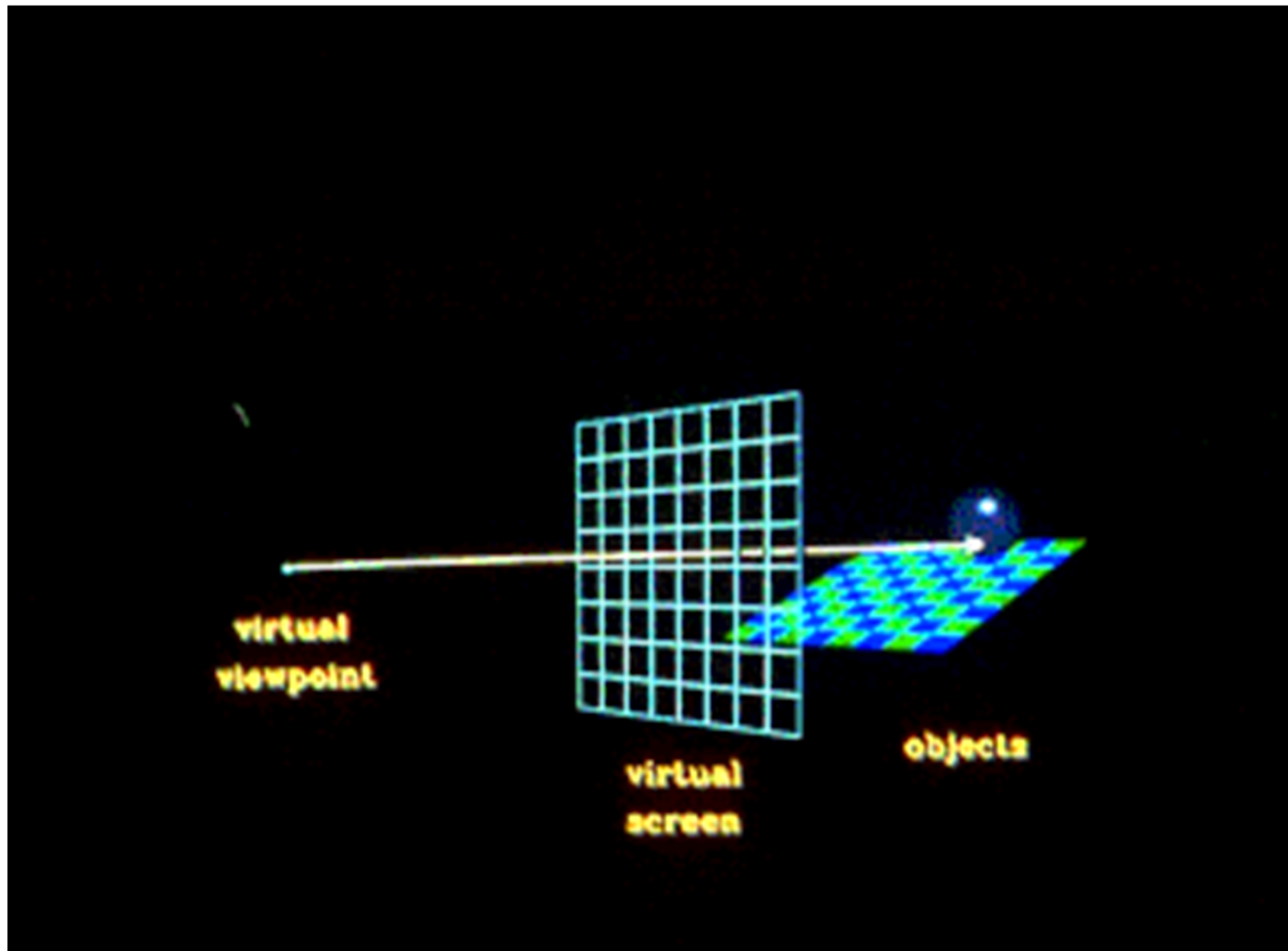


Case A: Ray misses all objects





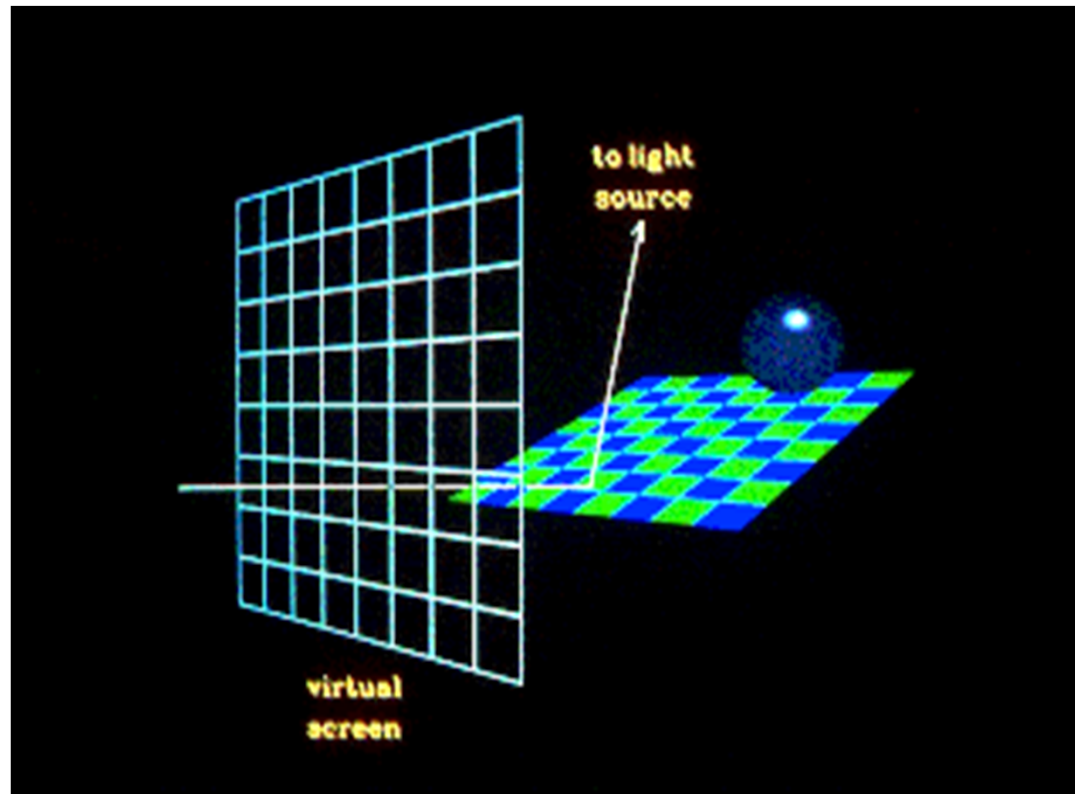
Case B: Ray hits an object





Case B: Ray hits an object

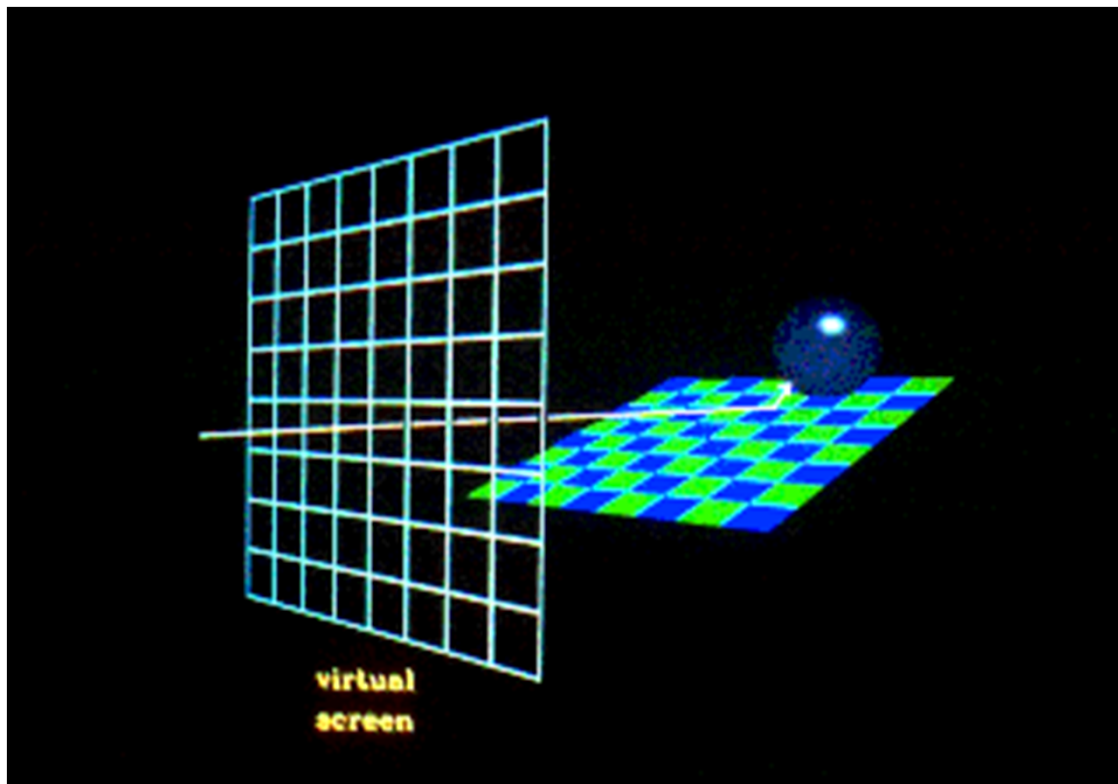
- **Ray hits object:** Check if hit point is in shadow, build secondary ray (shadow ray) towards light sources.





Case B: Ray hits an object

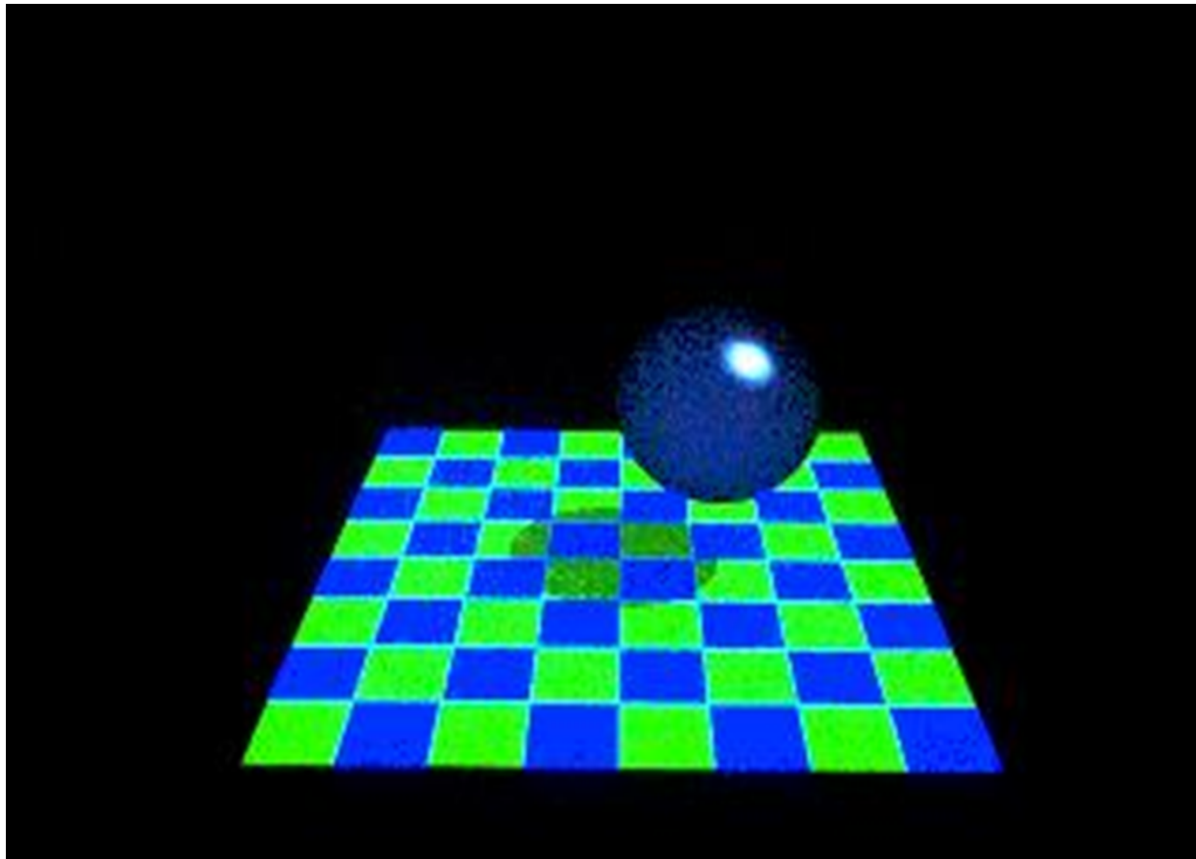
- If shadow ray hits another object before light source: first intersection point is in shadow of the second object. Otherwise, collect light contributions





Case B: Ray hits an object

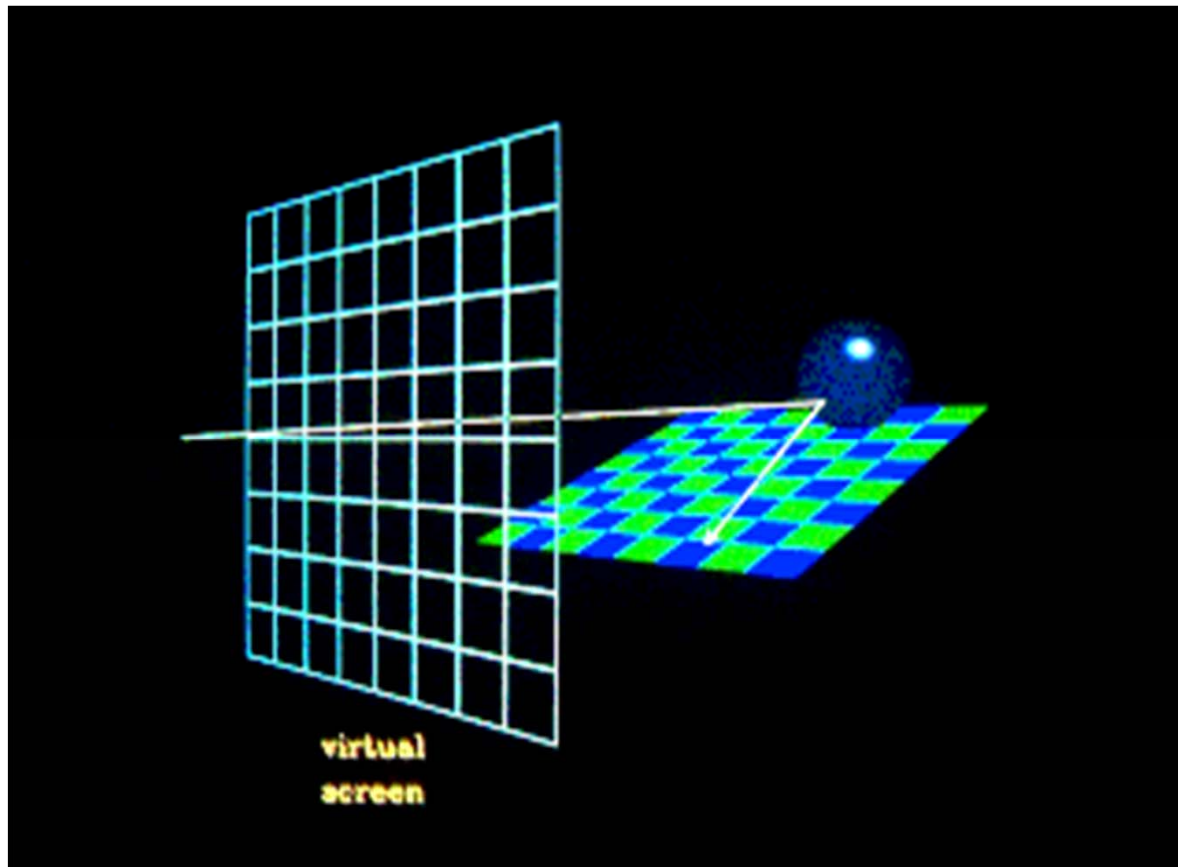
- First Intersection point in the shadow of the second object is the shadow area.



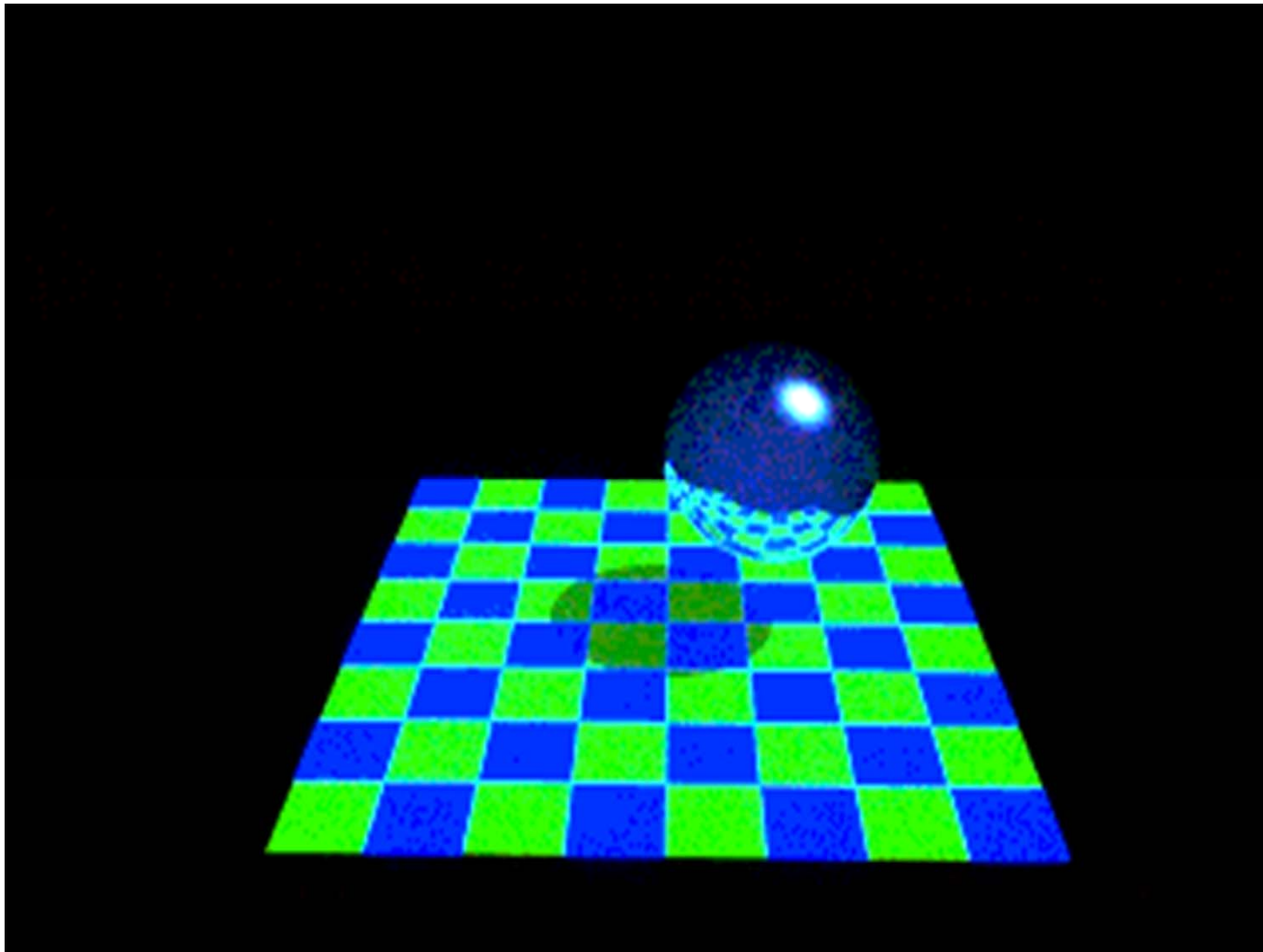


Reflected Ray

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



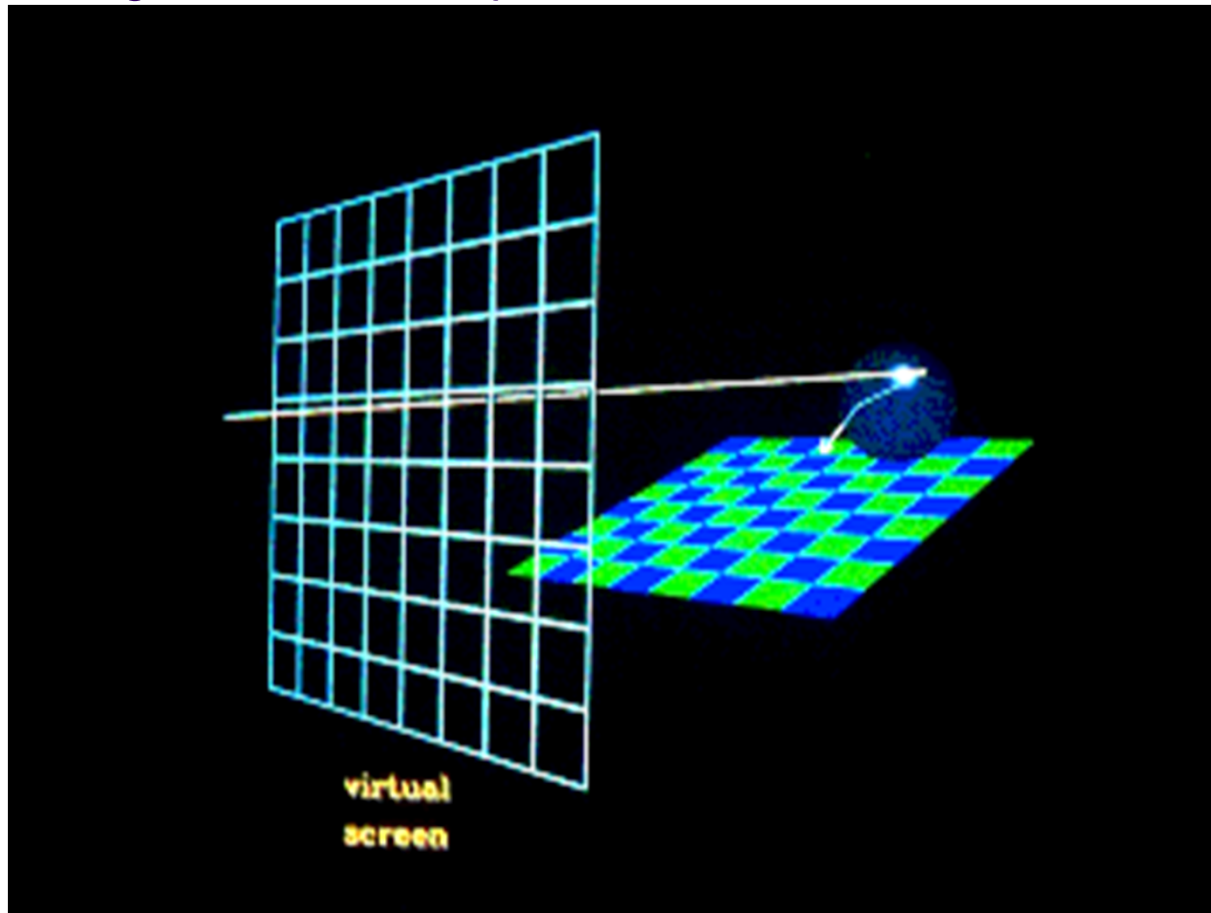
Reflection: Contribution from the reflected ray



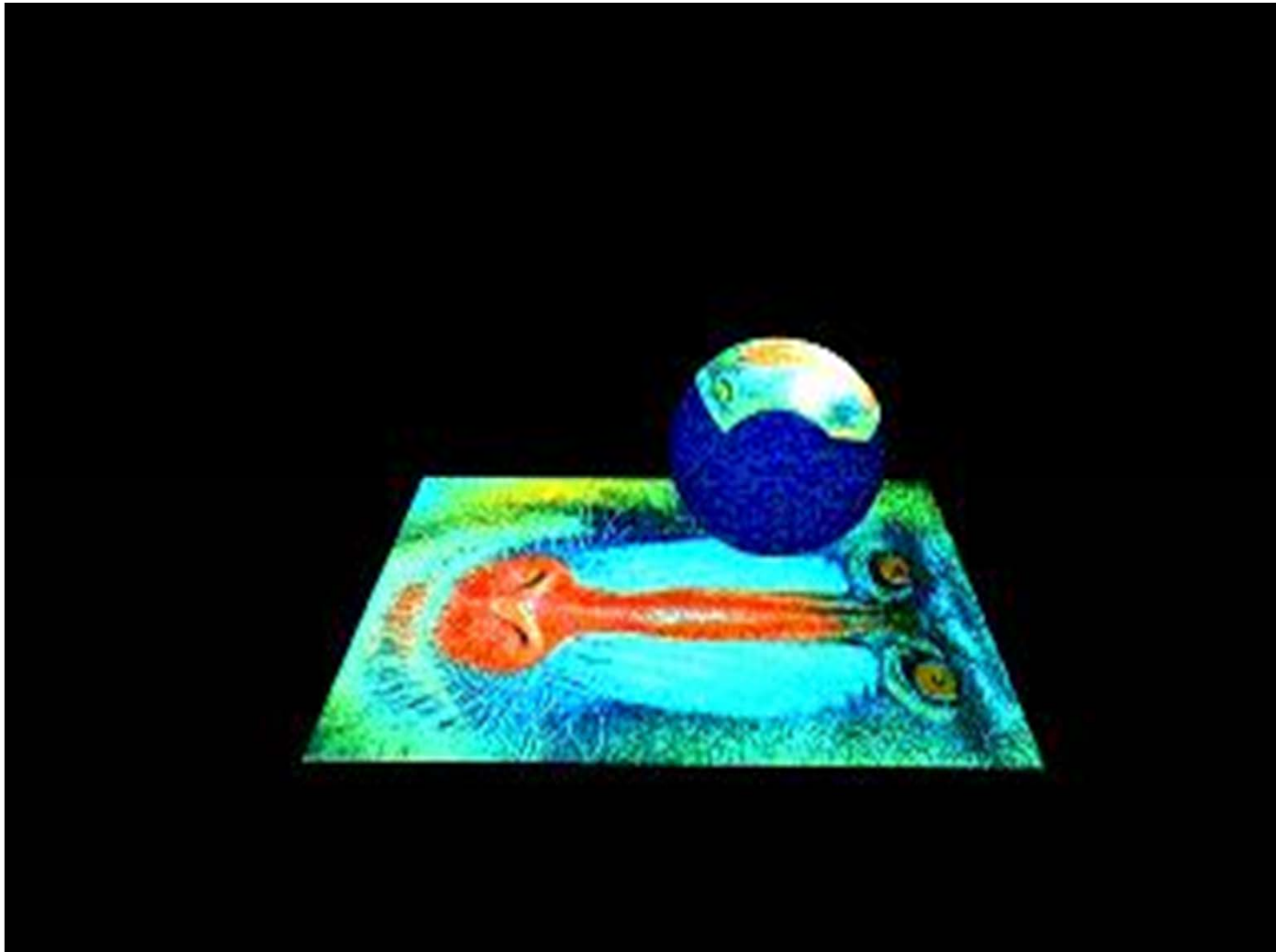


Transparency

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



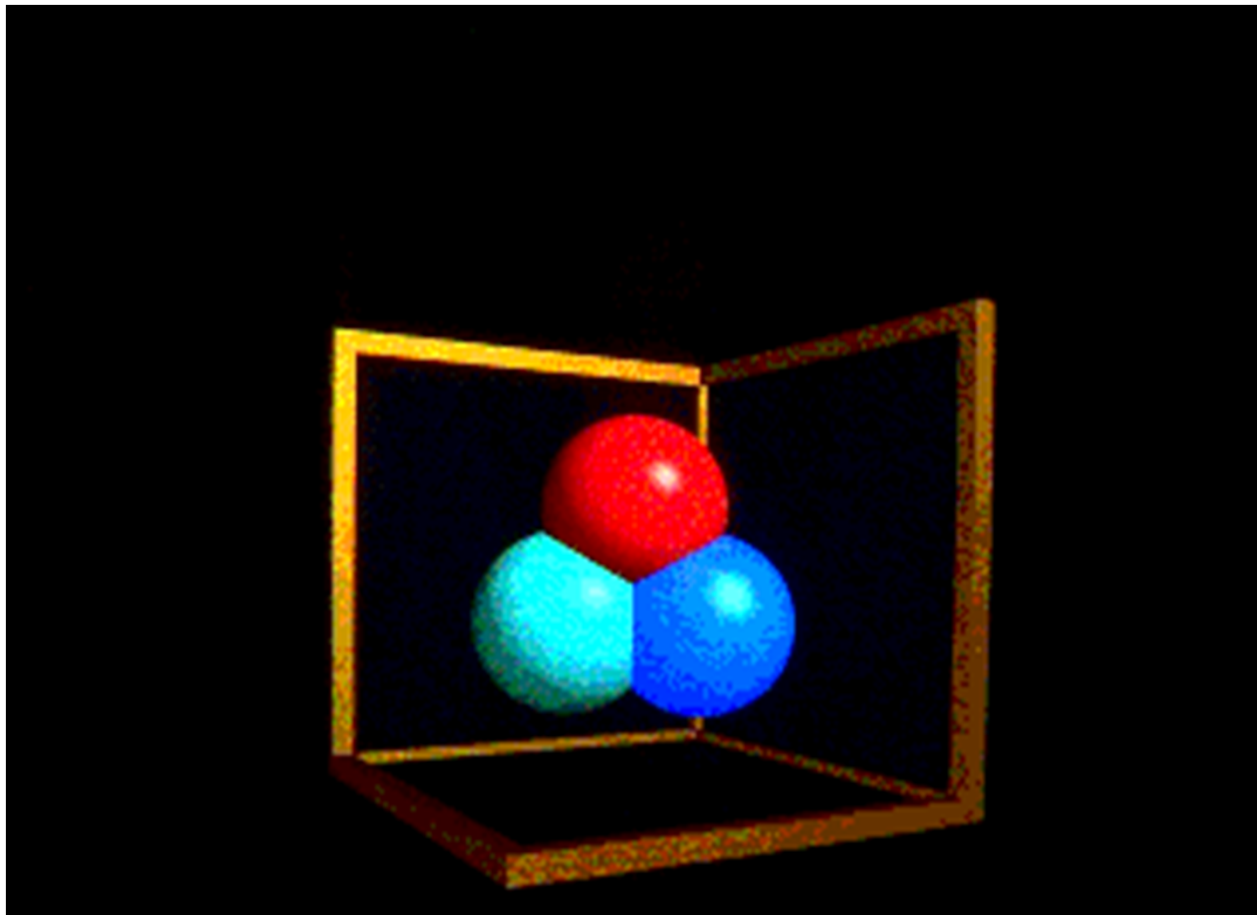
Transparency: Contribution from transmitted ray





Reflected Ray: Recursion

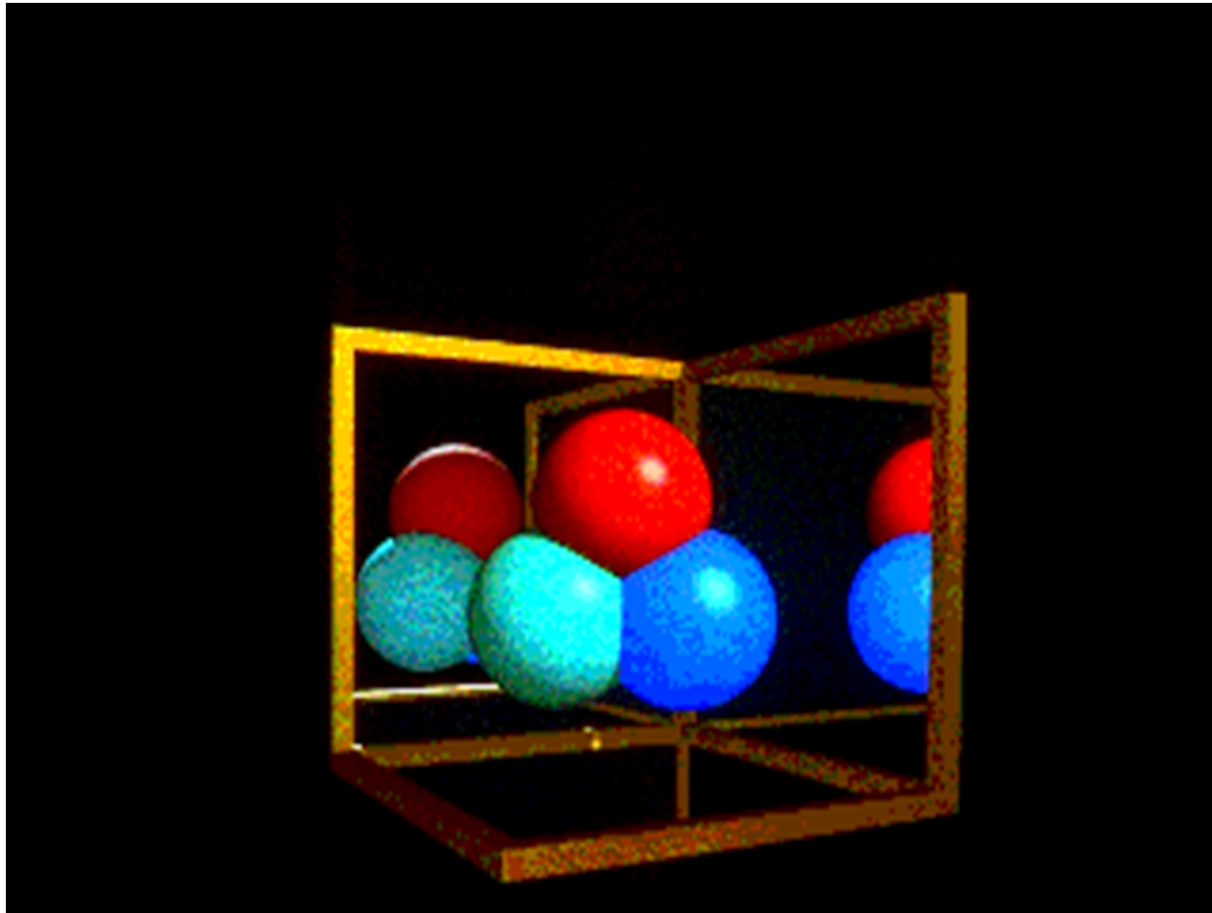
Reflected rays can generate other reflected rays that can generate other reflected rays, etc. *Case A: Scene with no reflection rays*





Reflected Ray: Recursion

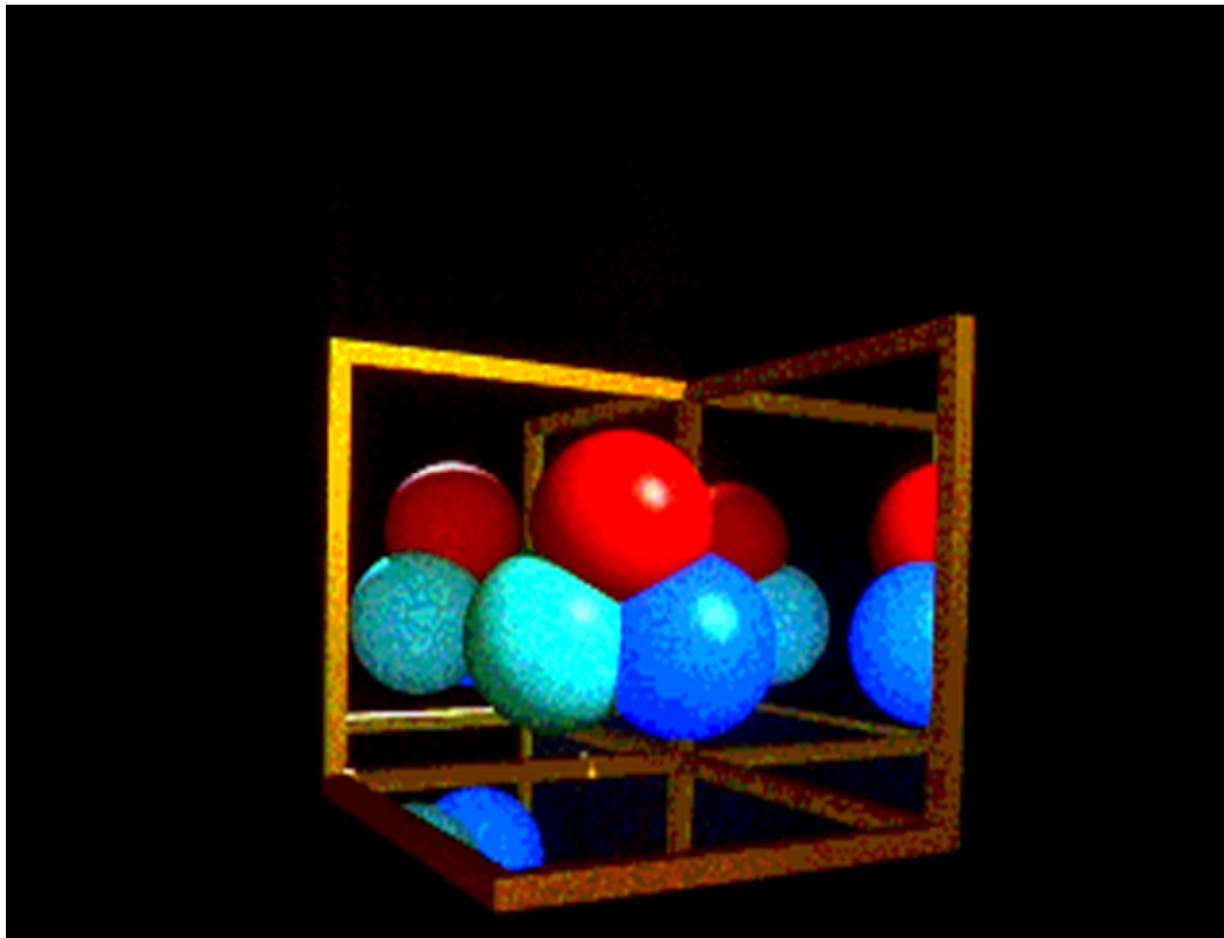
Case B: Scene with one layer of reflection





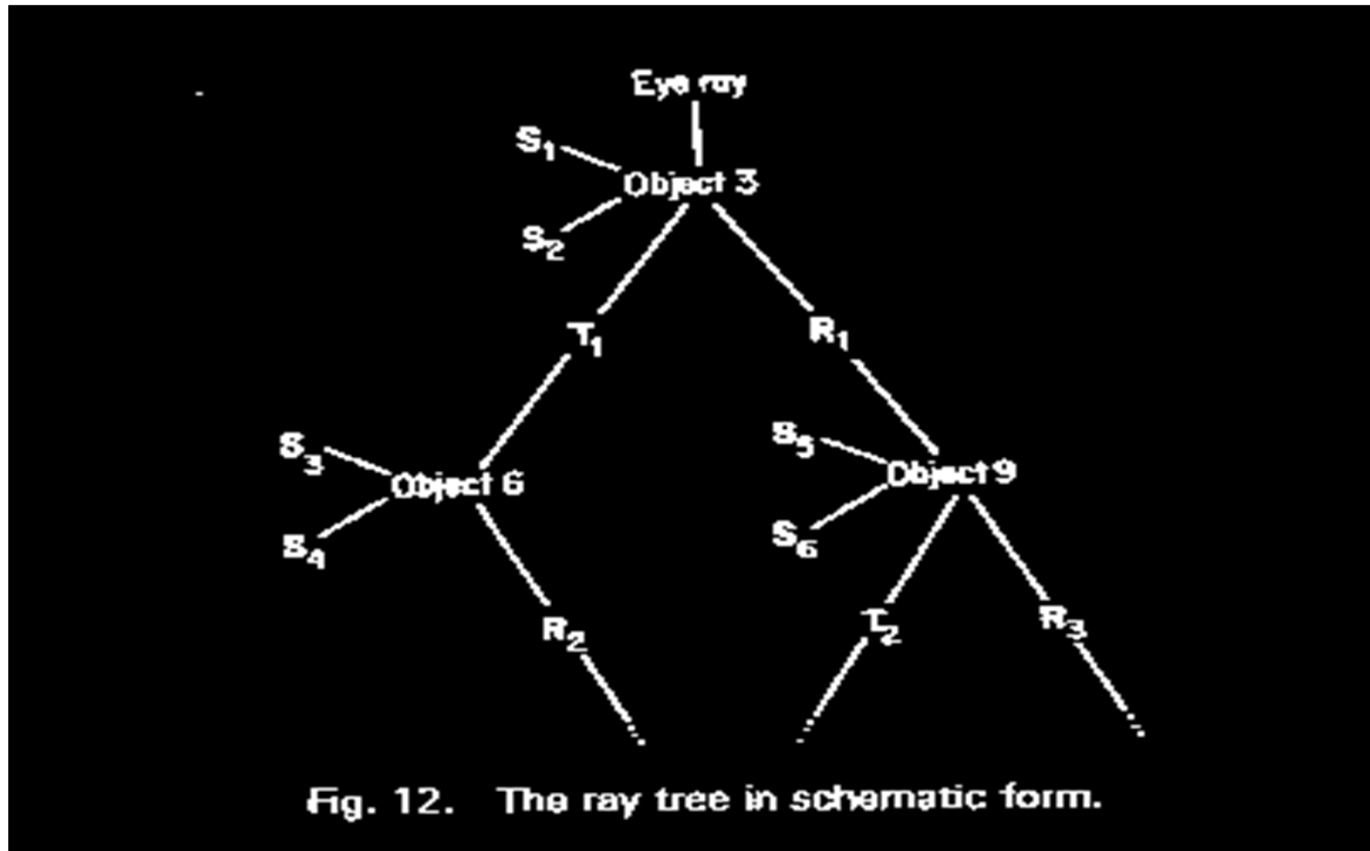
Reflected Ray: Recursion

Case C: Scene with two layers of reflection





Ray Tree



- Reflective and/or transmitted rays are continually generated until 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:
 $\mathbf{R0} = [x_0, y_0, z_0]$ - origin of ray
 $\mathbf{Rd} = [x_d, y_d, z_d]$ - direction of ray
- then define parametric equation of ray:
 $\mathbf{R}(t) = \mathbf{R0} + \mathbf{Rd} * t$ 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.
- But for this simple ray tracer we shall hardcode our scenes into .cpp file
- Just two shapes: sphere, mesh
- The output image/file will consist of three intensity values (Red, Green, and Blue) for each pixel.



Accelerating Ray Tracing

- Ray Tracing is time-consuming because of intersection calculations
- Each intersection requires from a few (5-7) to many (15-20) floating point (fp) operations
- Example: for a scene with 100 objects and computed with a spatial resolution of 512 x 512, assuming 10 fp operations per object test there are about $250,000 \times 100 \times 10 = 250,000,000$ fps.
- Solutions:
 - Use faster machines
 - Use specialized hardware, especially parallel processors.
 - Note: ray tracing does not use 3D graphics card (new drn)
 - 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

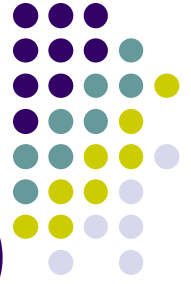
- Our approach:
 - Give arrangement of minimal ray tracer
 - Use that as template to explain process
- Minimal?
 - Yes! Basic framework
 - Just two object intersections
 - Minimal/no shading
- Paul Heckbert (CMU):
 - Ran ray tracing contest for years
 - Wrote ray tracer that fit on back of his business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



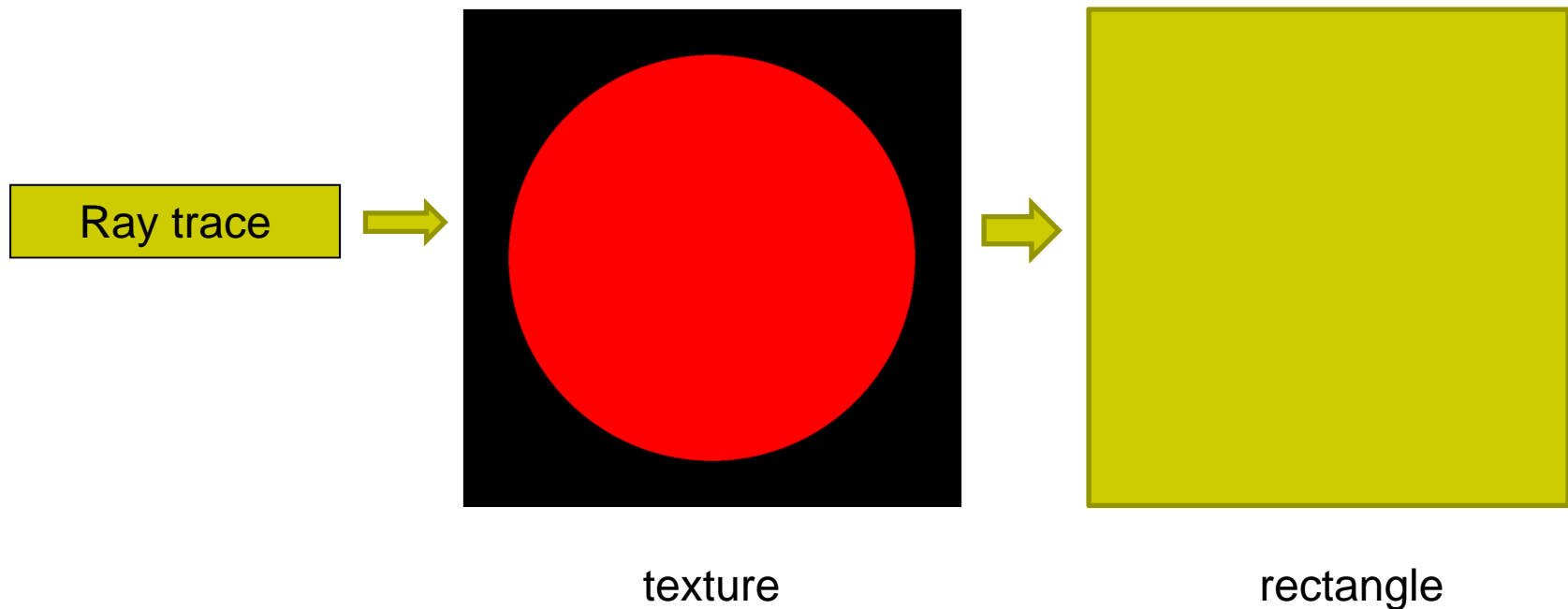
Scene Definition

- We shall deal with only one shape just to set up:
 - Sphere:
 - Dimension 1
 - No transformation (no translate, rotate, scale)
- Basically 1 unit sphere sitting at origin



Rendering?

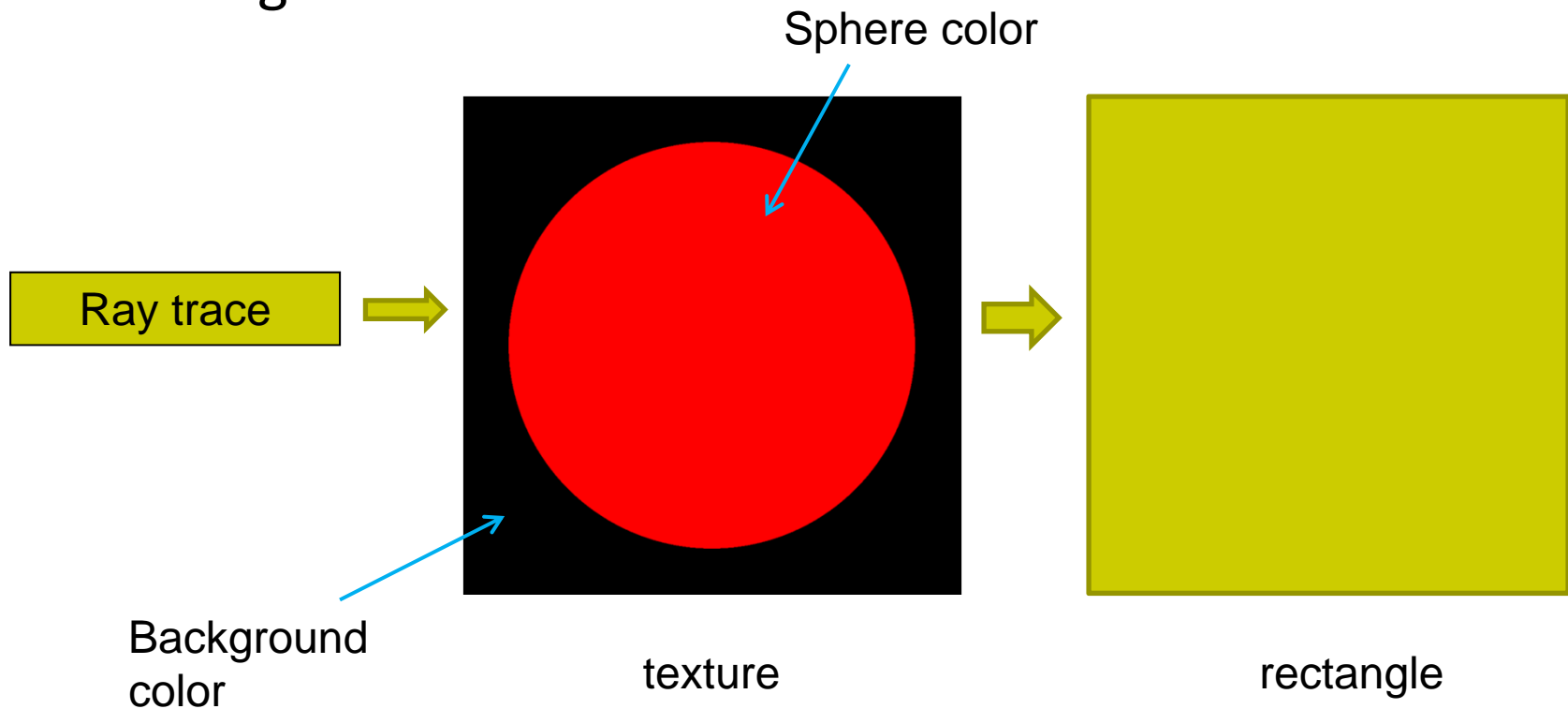
- Imagine big rectangle (2 triangles) facing camera
- Ray trace to texture image[N][N][3]
- Texture map image onto rectangle





Rendering?

- Declare two colors for texture
 - Sphere color
 - Background color



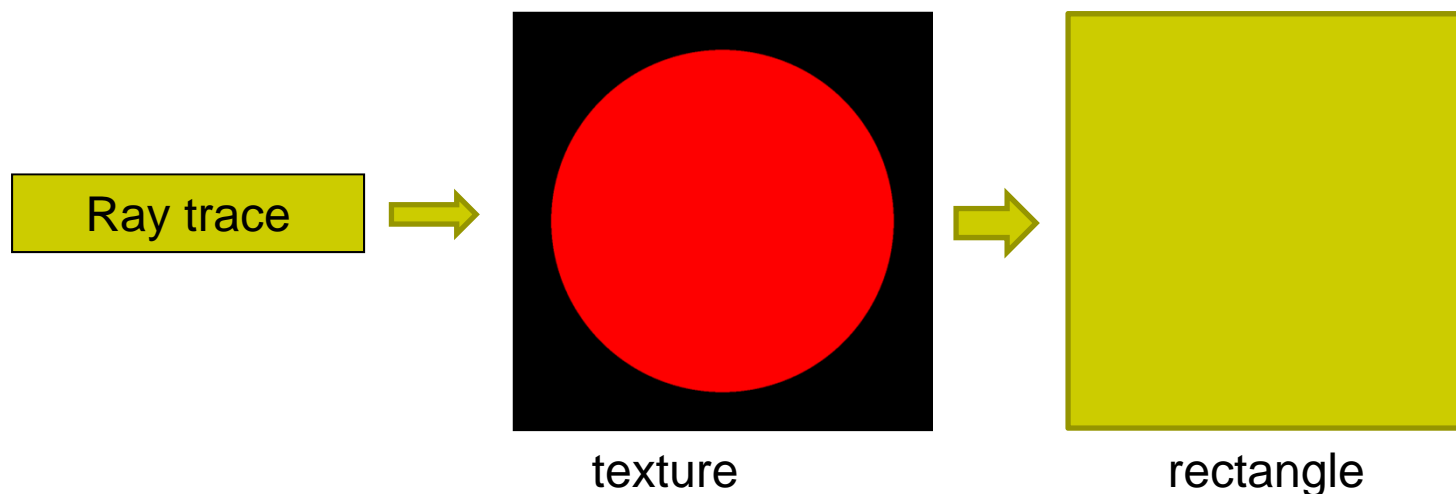


Map texture

- Declare polygon as two triangles

```
point4 points[6]
= {point4(0.0, 0.0, 0.0, 1.0, point4(0.0, 1.0, 0.0, 1.0),
    point4(1.0, 1.0, 0.0, 1.0, point4(1.0, 1.0, 0.0, 1.0),
    point4(1.0, 0.0, 0.0, 1.0, point4(0.0, 0.0, 0.0, 1.0))}

GLfloat tex_coord[6][2] = {{0.0, 0.0}, {0.0, 1.0}, {1.0, 1.0},
                           {1.0, 1.0}, {1.0, 0.0}, {0.0, 0.0}},
```

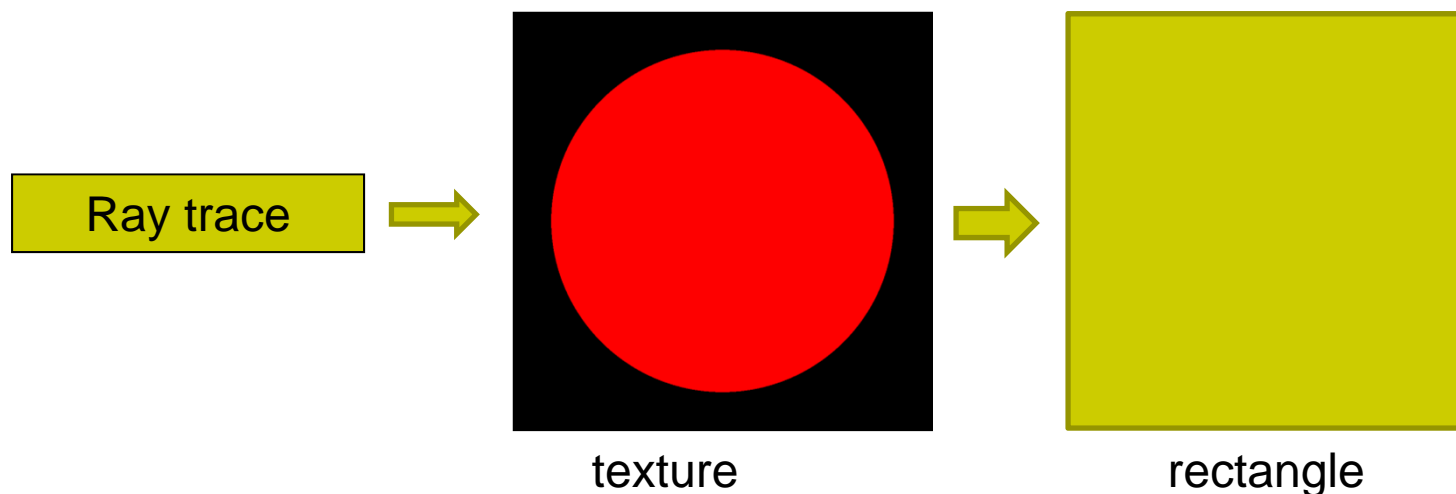




Rendering texture

- Generate image using ray tracing
- Move image, rectangle to GPU
- Texture map as usual

(See section 9.8.5 of text. Mandelbrot example 9.2 on book website)





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



- Simply write into appropriate location of texture

```
image[i][j][0] = r;
```

```
image[i][j][1] = g;
```

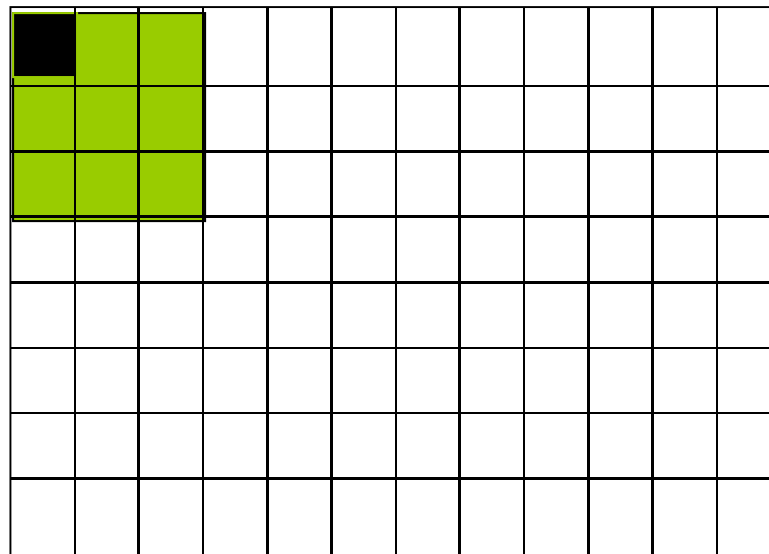
```
image[i][j][2] = b;
```

- Note: r, g, b is either color of sphere if pixel covers sphere, or background color
- 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 blocksize 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
           color_rect(r, g, b), r, c, blockSize);
    }
}
```

Define function that colors
in blockSize x blockSize with
Ray traced colors

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
           color_rect(r, g, b), r, c, 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

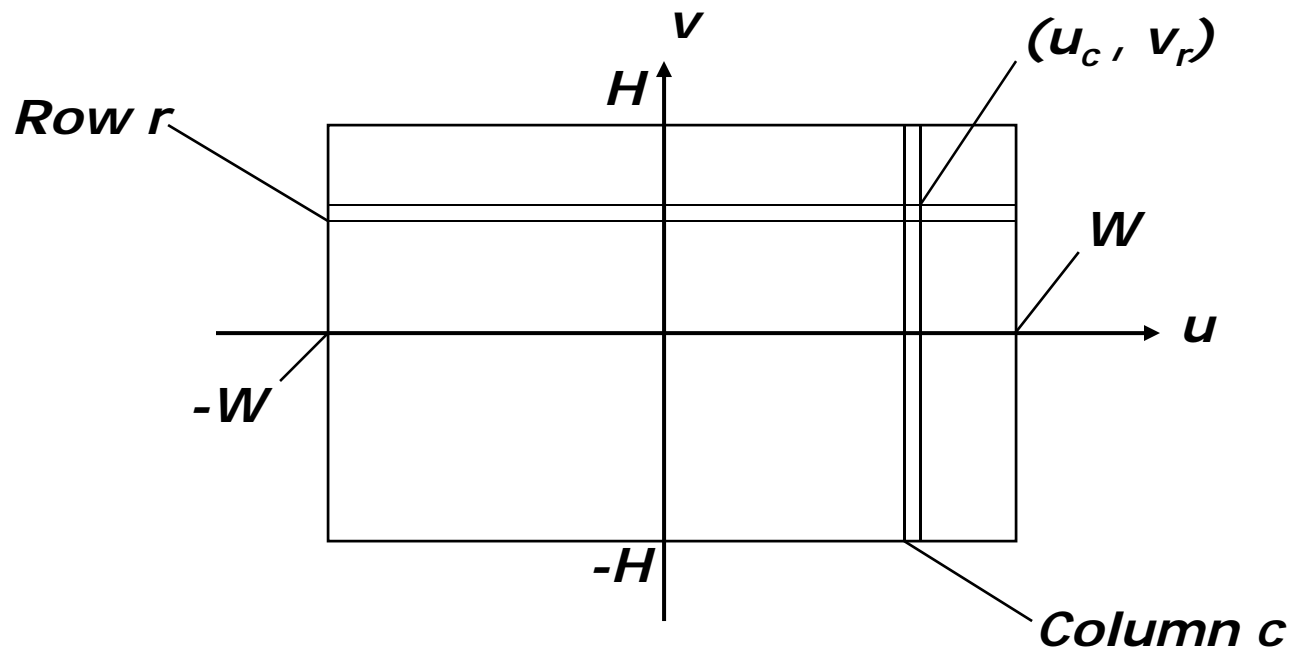
Set up the camera

```
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
        color_rect(r, g, b), r, c, blockSize);
    }
}
```



Set up Camera Geometry

- As before, camera has axes $(\mathbf{u}, \mathbf{v}, \mathbf{n})$ and position **eye** with coordinates $(\text{eye.x}, \text{eye.y}, \text{eye.z})$
- 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.** So,
 - 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 $c = 0, 1, \dots, nCols - 1$



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 $r = 0, 1, \dots, nRows - 1$



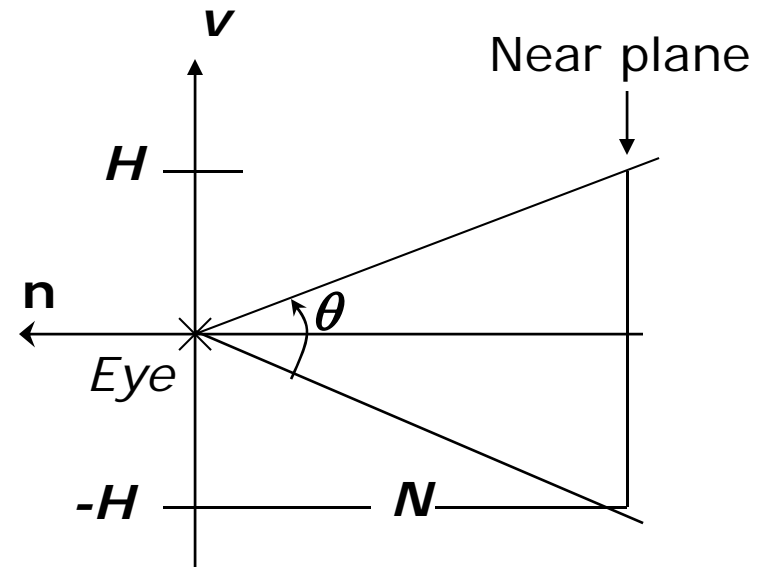
Set up Camera Geometry

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

$$H = N \tan(\theta / 2) \quad W = H \cdot \textit{aspect}$$

- Thus pixel (r, c) location expressed in terms of \mathbf{u} \mathbf{v} and \mathbf{n}

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





Set up Camera Geometry

- So, pixel location ..Near plane lies distance N along \mathbf{n} axis

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

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

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

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

- Manipulating expressions, if

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



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 t_1 and t_2 , 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

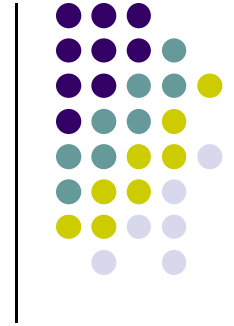


Where are we?

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
        color_rect(r, g, b), r, c, blockSize);
    }
}
```

References



- Hill and Kelley, Computer Graphics using OpenGL, 3rd edition, Chapter 12