

Computer Graphics (Spring 2008)

COMS 4160, Lectures 17, 18:

Nuts and bolts of Ray Tracing

Ravi Ramamoorthi

<http://www.cs.columbia.edu/~cs4160>

Acknowledgements: Thomas Funkhouser and Greg Humphreys

To Do

- Start early on raytracer assignment (assn 4)

Heckbert's Business Card Ray Tracer

```
typedef struct{double x,y,z;vec U,black,amb={.02,.02,.02};struct sphere{ vec cen,color;double rad,kd,ks,kt,kl,lr} *s; *best,sph=[0,.6,.5,1.,1.,1.,1.,.9,.05,2,.85,0,1,7,-1,.8,-.5,1.,.5,2,1.,7,.3,0.,.05,1,2,1,.8,-.5,1.,.8,.8,1.,.3,7,0,0.,1,2,3,-6,15,1.,.8,1.,7,0,0,0.,6,1.5,-3,-3,12.,.8,1.,1.,5,0,0,0.,5,1.5];yx:double u,b,tmin,sqrt(t),tan();double vdot(A,B);vec A,B;{return A.x*B.x+A.y*B.y+A.z*B.z;}vec vcomb(a,A,B)double a;vec A,B;{B.x+=a*A.x;B.y+=a*A.y;B.z+=a*A.z;return B;}vec vunit(A);vec A;{return vcomb(1./sqrt(vdot(A,A)),A,black);}struct sphere *intersect(P,D);vec P,D;{best=0;tmin=1e30;s=sph+5;while(s->sph)b=vdot(D,U=vcomb(-1.,P,)->cen);u=b*vdot(U,U)+s->rad*s->rad;u=u>0?sqrt(u):1e31;u=b-u-1e-7;b=u+b+u;tmin=u-1e-7;&&u<tmin?best=s,u:tmin;return best;}vec trace(level,P,D);vec P,D;{double d,eta,e;vec N,color;struct sphere *s; *i;if(!level--){return black;if(s=intersect(P,D));else return amb;color=amb;eta=s->rad*d;N=vunit(vcomb(-1.,P,vcomb(tmin,D,P),s->cen));if(d<0)N=vcomb(-1.,N,black);eta=1/eta;d=-d;l=sph+5;while(l-->sph){if(e=|kl*vdot(N,U=vunit(vcomb(-1.,P,)->cen))|>0&&intersect(P,U)=l)color=vcomb(e,l->color,color);U=s->color;color.x'=U.x;color.y'=U.y;color.z'=U.z;e=1-eta;eta*(1-d*d);return vcomb(s->kt,e*0?trace(level,P,vcomb(eta,D,vcomb(eta*d-sqrt(e),N,black)))&black,vcomb(s->ks,trace(level,P,vcomb(2*d,N,D)),vcomb(s->kd,color,vcomb(s->kl,U,black)))};main(){printf("%d\n",32,32);while(yx<32*32) Ux=yx%32/32/2,Uz=32/2-yx++/32,Uy=32/2-tan(25*114.5915590261),U=vcomb(255.,trace(3,black,vunit(U)),black)printf("%c.%f %c.%f\n",U);} *minray!/'
```

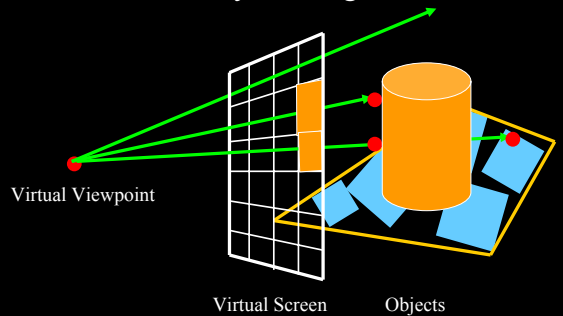
Outline

- Camera Ray Casting (choosing ray directions) [2.3]
- Ray-object intersections [2.4]
- Ray-tracing transformed objects [2.4]
- Lighting calculations [2.5]
- Recursive ray tracing [2.6]

Outline in Code

```
Image Raytrace (Camera cam, Scene scene, int width, int height)
{
    Image image = new Image (width, height) ;
    for (int i = 0 ; i < height ; i++)
        for (int j = 0 ; j < width ; j++) {
            Ray ray = RayThruPixel (cam, i, j) ;
            Intersection hit = Intersect (ray, scene) ;
            image[i][j] = FindColor (hit) ;
        }
    return image ;
}
```

Ray Casting



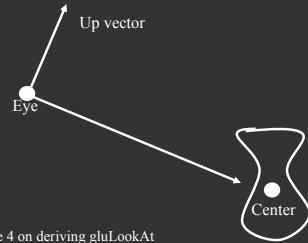
Ray-tracing is still the object's Patch shading from the light; (OpenGL)

Finding Ray Direction

- Goal is to find ray direction for given pixel i and j
- Many ways to approach problem
 - Objects in world coord, find dirn of each ray (we do this)
 - Camera in canonical frame, transform objects (OpenGL)
- Basic idea
 - Ray has origin (camera center) and direction
 - Find direction given camera params and i and j
- Camera params as in gluLookAt
 - Lookfrom[3], LookAt[3], up[3], fov

Similar to gluLookAt derivation

- gluLookAt(eyex, eyey, eyez, centerx, centery, centerz, upx, upy, upz)
- Camera at eye, looking at center, with up direction being up



From 4160 lecture 4 on deriving gluLookAt

Constructing a coordinate frame?

- We want to associate w with a , and v with b
- But a and b are neither orthogonal nor unit norm
 - And we also need to find u

$$w = \frac{a}{\|a\|}$$

$$u = \frac{b \times w}{\|b \times w\|}$$

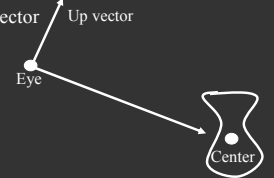
$$v = w \times u$$

Slide 20 from 4160 lecture 2

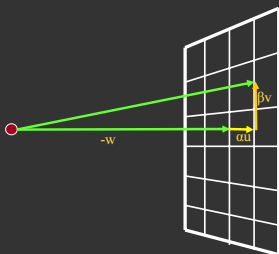
Camera coordinate frame

$$w = \frac{a}{\|a\|} \quad u = \frac{b \times w}{\|b \times w\|} \quad v = w \times u$$

- We want to position camera at origin, looking down $-Z$ dirn
- Hence, vector a is given by $\text{eye} - \text{center}$
- The vector b is simply the up vector



Canonical viewing geometry



$$\text{ray} = \text{eye} + \frac{\alpha u + \beta v - w}{\|\alpha u + \beta v - w\|}$$

$$\alpha = \tan\left(\frac{\text{fovx}}{2}\right) \times \left(\frac{j - (\text{width}/2)}{\text{width}/2}\right) \quad \beta = \tan\left(\frac{\text{fovy}}{2}\right) \times \left(\frac{(\text{height}/2) - i}{\text{height}/2}\right)$$

Outline

- Camera Ray Casting (choosing ray directions) [2.3]
- Ray-object intersections [2.4]
- Ray-tracing transformed objects [2.4]
- Lighting calculations [2.5]
- Recursive ray tracing [2.6]

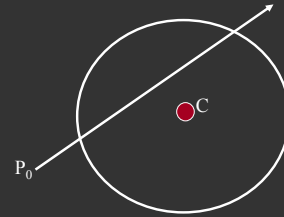
Outline in Code

```
Image Raytrace (Camera cam, Scene scene, int width, int height)
{
    Image image = new Image (width, height);
    for (int i = 0; i < height; i++)
        for (int j = 0; j < width; j++) {
            Ray ray = RayThruPixel (cam, i, j);
            Intersection hit = Intersect (ray, scene);
            image[i][j] = FindColor (hit);
        }
    return image;
}
```

Ray-Sphere Intersection

$$\text{ray} \equiv \vec{P} = \vec{P}_0 + \vec{P}_1 t$$

$$\text{sphere} \equiv (\vec{P} - \vec{C}) \cdot (\vec{P} - \vec{C}) - r^2 = 0$$



Ray-Sphere Intersection

$$\text{ray} \equiv \vec{P} = \vec{P}_0 + \vec{P}_1 t$$

$$\text{sphere} \equiv (\vec{P} - \vec{C}) \cdot (\vec{P} - \vec{C}) - r^2 = 0$$

Substitute

$$\text{ray} \equiv \vec{P} = \vec{P}_0 + \vec{P}_1 t$$

$$\text{sphere} \equiv (\vec{P}_0 + \vec{P}_1 t - \vec{C}) \cdot (\vec{P}_0 + \vec{P}_1 t - \vec{C}) - r^2 = 0$$

Simplify

$$t^2 (\vec{P}_1 \cdot \vec{P}_1) + 2t \vec{P}_1 \cdot (\vec{P}_0 - \vec{C}) + (\vec{P}_0 - \vec{C}) \cdot (\vec{P}_0 - \vec{C}) - r^2 = 0$$

Ray-Sphere Intersection

$$t^2 (\vec{P}_1 \cdot \vec{P}_1) + 2t \vec{P}_1 \cdot (\vec{P}_0 - \vec{C}) + (\vec{P}_0 - \vec{C}) \cdot (\vec{P}_0 - \vec{C}) - r^2 = 0$$

Solve quadratic equations for t

- 2 real positive roots: pick smaller root
- Both roots same: tangent to sphere
- One positive, one negative root: ray origin inside sphere (pick + root)
- Complex roots: no intersection (check discriminant of equation first)



Ray-Sphere Intersection

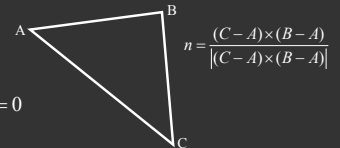
- Intersection point: $\text{ray} \equiv \vec{P} = \vec{P}_0 + \vec{P}_1 t$
- Normal (for sphere, this is same as coordinates in sphere frame of reference, useful other tasks)

$$\text{normal} = \frac{\vec{P} - \vec{C}}{|\vec{P} - \vec{C}|}$$

Ray-Triangle Intersection

- One approach: Ray-Plane intersection, then check if inside triangle
- Plane equation:

$$\text{plane} \equiv \vec{P} \cdot \vec{n} - \vec{A} \cdot \vec{n} = 0$$



Ray-Triangle Intersection

- One approach: Ray-Plane intersection, then check if inside triangle

- Plane equation:

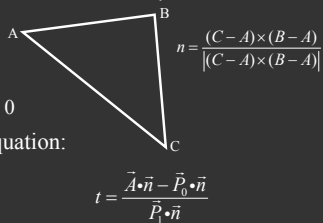
$$\text{plane} \equiv \vec{P} \cdot \vec{n} - \vec{A} \cdot \vec{n} = 0$$

- Combine with ray equation:

$$\text{ray} \equiv \vec{P} = \vec{P}_0 + \vec{P}_1 t$$

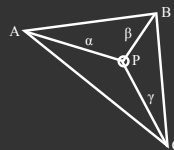
$$(\vec{P}_0 + \vec{P}_1 t) \cdot \vec{n} = \vec{A} \cdot \vec{n}$$

$$n = \frac{(C-A) \times (B-A)}{|(C-A) \times (B-A)|}$$

$$t = \frac{\vec{A} \cdot \vec{n} - \vec{P}_0 \cdot \vec{n}}{\vec{P}_1 \cdot \vec{n}}$$


Ray inside Triangle

- Once intersect with plane, still need to find if in triangle
- Many possibilities for triangles, general polygons (point in polygon tests)
- We find parametrically [barycentric coordinates]. Also useful for other applications (texture mapping)

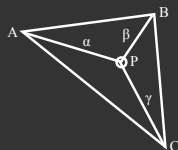


$$P = \alpha A + \beta B + \gamma C$$

$$\alpha \geq 0, \beta \geq 0, \gamma \geq 0$$

$$\alpha + \beta + \gamma = 1$$

Ray inside Triangle



$$P = \alpha A + \beta B + \gamma C$$

$$\alpha \geq 0, \beta \geq 0, \gamma \geq 0$$

$$\alpha + \beta + \gamma = 1$$

$$P - A = \beta(B - A) + \gamma(C - A)$$

$$0 \leq \beta \leq 1, 0 \leq \gamma \leq 1$$

$$\beta + \gamma \leq 1$$

Other primitives

- Much early work in ray tracing focused on ray-primitive intersection tests
- Cones, cylinders, ellipsoids
- Boxes (especially useful for bounding boxes)
- General planar polygons
- Many more
- Consult chapter in Glassner (handed out) for more details and possible extra credit

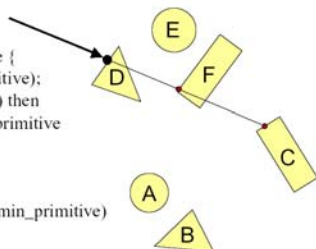
Ray Scene Intersection

Intersection FindIntersection(Ray ray, Scene scene)

```

{
  min_t = infinity
  min_primitive = NULL
  For each primitive in scene {
    t = Intersect(ray, primitive);
    if (t > 0 && t < min_t) then
      min_primitive = primitive
      min_t = t
  }
  return Intersect(min_t, min_primitive)
}

```



Outline

- Camera Ray Casting (choosing ray directions) [2.3]
- Ray-object intersections [2.4]
- Ray-tracing transformed objects [2.4]
- Lighting calculations [2.5]
- Recursive ray tracing [2.6]

Transformed Objects

- E.g. transform sphere into ellipsoid
- Could develop routine to trace ellipsoid (compute parameters after transformation)
- May be useful for triangles, since triangle after transformation is still a triangle in any case
- But can also use original optimized routines

Transformed Objects

- Consider a general 4x4 transform M
 - Will need to implement matrix stacks like in OpenGL
- Apply inverse transform M^{-1} to ray
 - Locations stored and transform in homogeneous coordinates
 - Vectors (ray directions) have homogeneous coordinate set to 0 [so there is no action because of translations]
- Do standard ray-surface intersection as modified
- Transform intersection back to actual coordinates
 - Intersection point p transforms as Mp
 - Distance to intersection if used may need recalculation
 - Normals n transform as $M^{-1}n$. Do all this before lighting

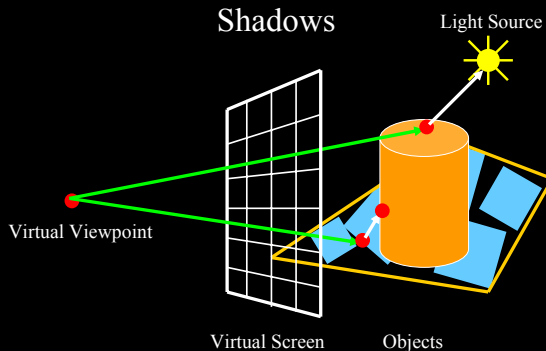
Outline

- Camera Ray Casting (choosing ray directions) [2.3]
- Ray-object intersections [2.4]
- Ray-tracing transformed objects [2.4]
- *Lighting calculations* [2.5]
- Recursive ray tracing [2.6]

Outline in Code

```
Image Raytrace (Camera cam, Scene scene, int width, int height)
{
    Image image = new Image (width, height) ;
    for (int i = 0 ; i < height ; i++)
        for (int j = 0 ; j < width ; j++) {
            Ray ray = RayThruPixel (cam, i, j) ;
            Intersection hit = Intersect (ray, scene) ;
            image[i][j] = FindColor (hit) ;
        }
    return image ;
}
```

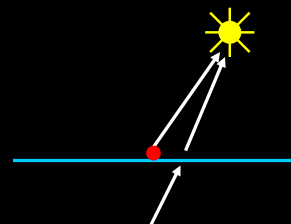
Shadows



Shadow ray to light is hit before object in shadow

Shadows: Numerical Issues

- Numerical inaccuracy may cause intersection to be below surface (effect exaggerated in figure)
- Causing surface to incorrectly shadow itself
- Move a little towards light before shooting shadow ray



Lighting Model

- Similar to OpenGL
- Lighting model parameters (global)
 - Ambient r g b (no per-light ambient as in OpenGL)
 - Attenuation const linear quadratic (like in OpenGL)

$$L = \frac{L_0}{const + lin * d + quad * d^2}$$

- Per light model parameters
 - Directional light (direction, RGB parameters)
 - Point light (location, RGB parameters)

Material Model

- Diffuse reflectance (r g b)
- Specular reflectance (r g b)
- Shininess s
- Emission (r g b)
- All as in OpenGL

Shading Model

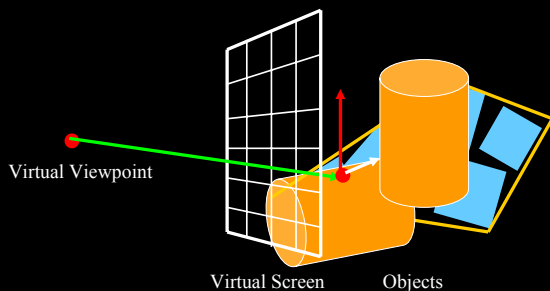
$$I = K_a + K_e + \sum_{i=1}^n V_i L_i (K_d \max(l_i \cdot n, 0) + K_s (\max(h_i \cdot n, 0))^s)$$

- Global ambient term, emission from material
- For each light, diffuse specular terms
- Note visibility/shadowing for each light (not in OpenGL)
- Evaluated per pixel per light (not per vertex)

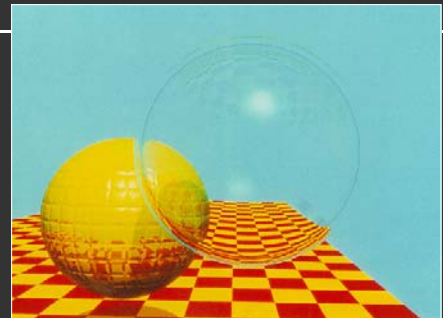
Outline

- Camera Ray Casting (choosing ray directions) [2.3]
- Ray-object intersections [2.4]
- Ray-tracing transformed objects [2.4]
- Lighting calculations [2.5]
- *Recursive ray tracing* [2.6]

Mirror Reflections/Refractions



Generate reflected ray in mirror direction,
Get reflections and refractions of objects



Turner Whitted 1980

Basic idea

For each pixel

- Trace Primary Eye Ray, find intersection
- Trace Secondary Shadow Ray(s) to all light(s)
 - Color = Visible ? Illumination Model : 0 ;
- Trace Reflected Ray
 - Color += reflectivity * Color of reflected ray

Recursive Shading Model

$$I = K_a + K_e + \sum_{i=1}^n I_i (K_d \max(l_i \cdot n, 0) + K_s (\max(l_i \cdot n, 0))^s) + K_t I_r + K_r I_r$$

- Highlighted terms are recursive specularities [mirror reflections] and transmission (latter is extra credit)
- Trace secondary rays for mirror reflections and refractions, include contribution in lighting model
- GetColor calls RayTrace recursively (the I values in equation above of secondary rays are obtained by recursive calls)

Problems with Recursion

- Reflection rays may be traced forever
- Generally, set maximum recursion depth
- Same for transmitted rays (take refraction into account)

Effects needed for Realism

- (Soft) Shadows
- Reflections (Mirrors and Glossy)
- Transparency (Water, Glass)
- Interreflections (Color Bleeding)
- Complex Illumination (Natural, Area Light)
- Realistic Materials (Velvet, Paints, Glass)

Discussed in this lecture so far

Not discussed but possible with distribution ray tracing

Hard (but not impossible) with ray tracing; radiosity methods

Some basic add ons

- Area light sources and soft shadows: break into grid of $n \times n$ point lights
 - Use jittering: Randomize direction of shadow ray within small box for given light source direction
 - Jittering also useful for antialiasing shadows when shooting primary rays
- More complex reflectance models
 - Simply update shading model
 - But at present, we can handle only mirror global illumination calculations

Acceleration

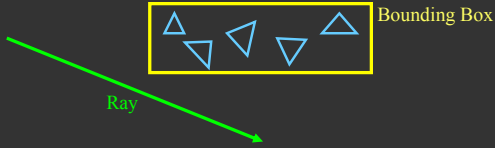
Testing each object for each ray is slow

- Fewer Rays
 - Adaptive sampling, depth control
- Generalized Rays
 - Beam tracing, cone tracing, pencil tracing etc.
- Faster Intersections
 - Optimized Ray-Object Intersections
 - Fewer Intersections

Acceleration Structures

Bounding boxes (possibly hierarchical)

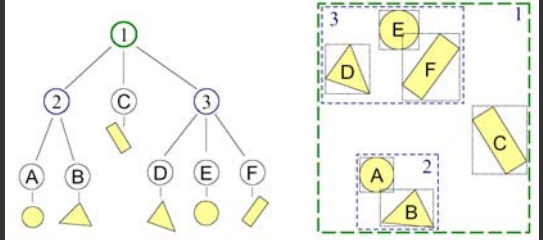
If no intersection bounding box, needn't check objects



Spatial Hierarchies (Oct-trees, kd trees, BSP trees)

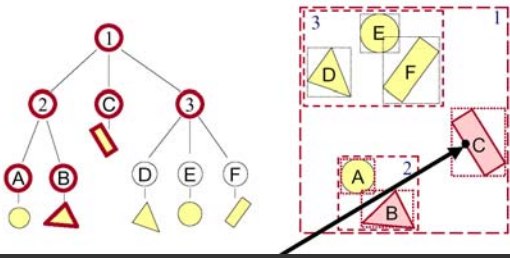
Bounding Volume Hierarchies 1

- Build hierarchy of bounding volumes
 - Bounding volume of interior node contains all children



Bounding Volume Hierarchies 2

- Use hierarchy to accelerate ray intersections
 - Intersect node contents only if hit bounding volume

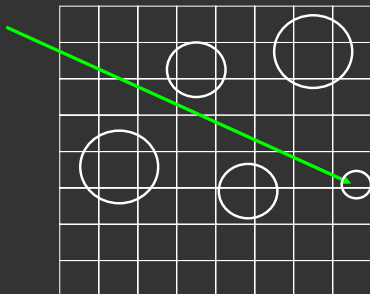


Bounding Volume Hierarchies 3

- Sort hits & detect early termination

```
FindIntersection(Ray ray, Node node)
{
    // Find intersections with child node bounding volumes
    ...
    // Sort intersections front to back
    ...
    // Process intersections (checking for early termination)
    min_t = infinity;
    for each intersected child i {
        if (min_t < bv_t[i]) break;
        shape_t = FindIntersection(ray, child);
        if (shape_t < min_t) { min_t = shape_t; }
    }
    return min_t;
}
```

Acceleration Structures: Grids

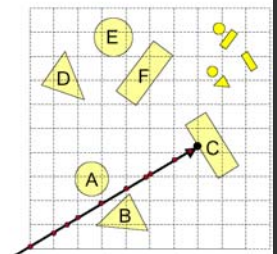


Uniform Grid: Problems

- Potential problem:
 - How choose suitable grid resolution?

Too little benefit
if grid is too coarse

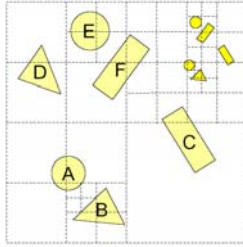
Too much cost
if grid is too fine



Octree

- Construct adaptive grid over scene
 - Recursively subdivide box-shaped cells into 8 octants
 - Index primitives by overlaps with cells

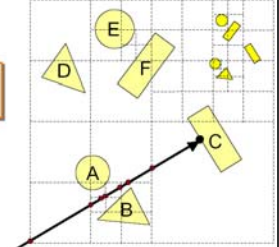
Generally fewer cells



Octree traversal

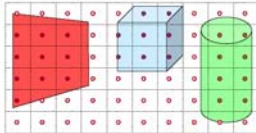
- Trace rays through neighbor cells
 - Fewer cells
 - More complex neighbor finding

Trade-off fewer cells for more expensive traversal



Other Accelerations

- Screen space coherence
 - Check last hit first
 - Beam tracing
 - Pencil tracing
 - Cone tracing
- Memory coherence
 - Large scenes
- Parallelism
 - Ray casting is "embarrassingly parallelizable"
- etc.



Interactive Raytracing

- Ray tracing historically slow
- Now viable alternative for complex scenes
 - Key is sublinear complexity with acceleration; need not process all triangles in scene
- Allows many effects hard in hardware
- OpenRT project real-time ray tracing (<http://www.openrt.de>)

Raytracing on Graphics Hardware

- Modern Programmable Hardware general streaming architecture
- Can map various elements of ray tracing
- Kernels like eye rays, intersect etc.
- In vertex or fragment programs
- Convergence between hardware, ray tracing

[Purcell et al. 2002, 2003]

<http://graphics.stanford.edu/papers/photongfx>

Ring - Stencil Routing



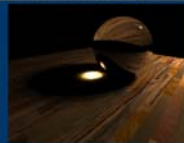
8s @ 512x384, 16K photons

Cornell Box - Bitonic Sort



64s @ 512x512, 45K photons

Glass Ball - Stencil Routing



11s @ 512x384, 5K photons

Cornell Box - Increased Search Radius

