Computer Graphics

CSE 167 [Win 17], Lectures 16, 17:

Nuts and bolts of Ray Tracing

Ravi Ramamoorthi

http://viscomp.ucsd.edu/classes/cse167/wi17

Acknowledgements: Thomas Funkhouser and Greg Humphreys

Heckbert's Business Card Ray Tracer

```
typedef struct{double x,y,z}vec;vec U,black,amb={.02,.02,.02};struct sphere{ vec cen,color;
double rad,kd,ks,kt,kl,ir}*s,*best,sph[]={0.,6.,.5,1.,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(),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,s->cen)),
u=b*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,*l;if(!level--)return black;if(s=intersect(P,D));else return amb;color=amb;eta=
s->ir;d= -vdot(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=l ->kl*vdot(N,U=vunit(vcomb(-1.,P,l->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 %d\n",32,32);while(yx<32*32) U.x=yx%32-32/2,U.z=32/2-
yx++/32,U.y=32/2/tan(25/114.5915590261),U=vcomb(255., trace(3,black,vunit(U)),black),printf
("%.0f %.0f %.0f\n",U);}/*minray!*/
```

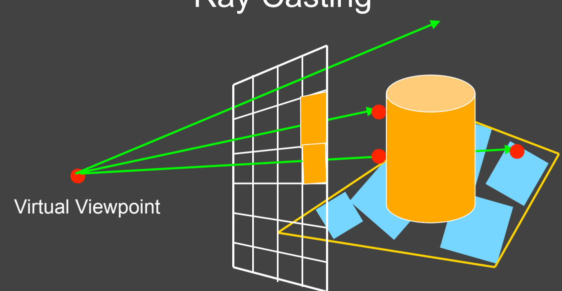
Outline

- *Camera Ray Casting (choose ray directions)*
- Ray-object intersections
- Ray-tracing transformed objects
- Lighting calculations
- Recursive ray tracing

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



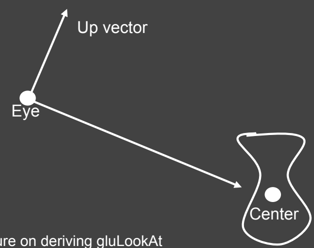
Multiple intersections: Use closest one (as does OpenGL)

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 earlier lecture 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**

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

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

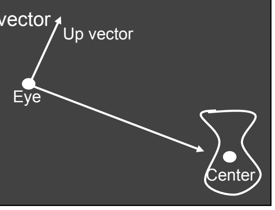
$$v = w \times u$$

From basic math lecture - Vectors: Orthonormal Basis Frames

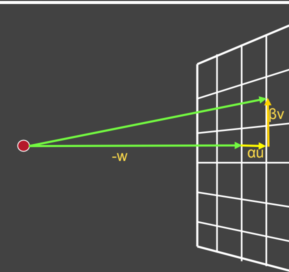
Camera coordinate frame

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

- We want to position camera at origin, looking down –Z dirn
- Hence, vector **a** is given by **eye** – **center**
- The vector **b** is simply the **up** vector



Canonical viewing geometry



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

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

Outline

- Camera Ray Casting (choosing ray directions)
- *Ray-object intersections*
- Ray-tracing transformed objects
- Lighting calculations
- Recursive ray tracing

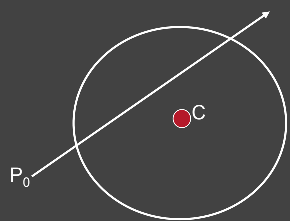
Outline in Code

```
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

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

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



Ray-Sphere Intersection

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

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

Substitute

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

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

Simplify

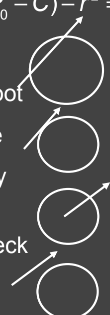
$$t^2(\vec{P}_1 \bullet \vec{P}_1) + 2t\,\vec{P}_1 \bullet (\vec{P}_0 - \vec{C}) + (\vec{P}_0 - \vec{C}) \bullet (\vec{P}_0 - \vec{C}) - r^2 = 0$$

Ray-Sphere Intersection

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

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

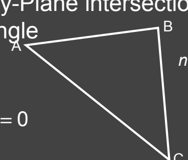
- Intersection point: $ray \equiv \vec{P} = \vec{P}_0 + \vec{P}_1 t$
- Normal (for sphere, this is same as coordinates in sphere frame of reference, useful other tasks)

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

Ray-Triangle Intersection

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

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



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

Ray-Triangle Intersection

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

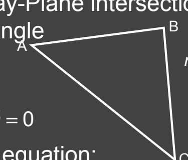
$$plane \equiv \vec{P} \bullet \vec{n} - \vec{A} \bullet \vec{n} = 0$$

- Combine with ray equation:

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

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

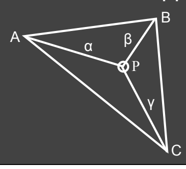
$$t = \frac{\vec{A} \bullet \vec{n} - \vec{P}_0 \bullet \vec{n}}{\vec{P}_1 \bullet \vec{n}}$$



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

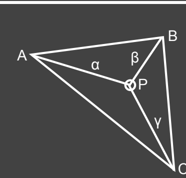
Ray inside Triangle

- 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



$$P = \alpha A + \beta B + \gamma C$$
$$\alpha \geq 0, \beta \geq 0, \gamma \geq 0$$
$$\alpha + \beta + \gamma = 1$$

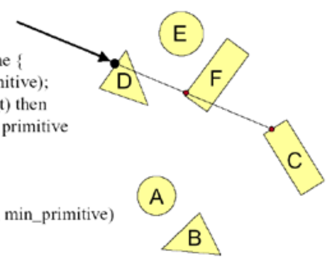
$$P - A = \beta(B - A) + \gamma(C - A)$$
$$0 \leq \beta \leq 1 \;,\; 0 \leq \gamma \leq 1$$
$$\beta + \gamma \leq 1$$

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 Intersection(min_t, min_primitive)
}
```



Outline

- Camera Ray Casting (choosing ray directions)
- Ray-object intersections
- *Ray-tracing transformed objects*
- Lighting calculations
- Recursive ray tracing

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

Ray-Tracing Transformed Objects

We have an optimized ray-sphere test
- But we want to ray trace an ellipsoid…

Solution: Ellipsoid transforms sphere
- Apply inverse transform to ray, use ray-sphere
- Allows for instancing (traffic jam of cars)
- Same idea for other primitives

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 $M^{-t}n$. Do all this before lighting

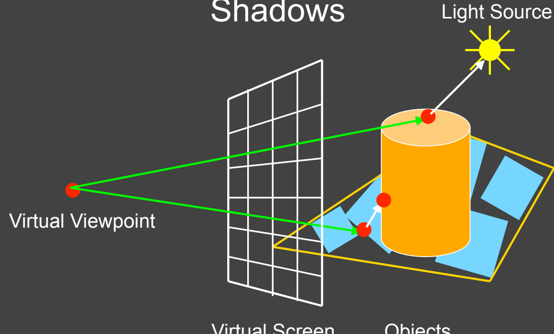
Outline

- Camera Ray Casting (choosing ray directions)
- Ray-object intersections
- Ray-tracing transformed objects
- *Lighting calculations*
- Recursive ray tracing

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

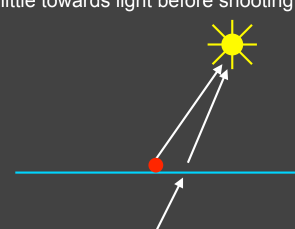


Shadow ray to light is blocked: object in shadow

Shadow ray to light is unblocked: object visible

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
  - Attenuation const linear quadratic

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

- Per light model parameters
  - Directional light (direction, RGB parameters)
  - Point light (location, RGB parameters)
  - Some differences from HW 2 syntax

Material Model

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

Shading Model

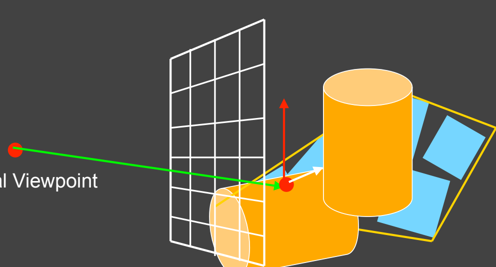
$$I = K_a + K_e + \sum_{i=1}^{n} V_i L_i (K_d \max(l_i \bullet n, 0) + K_s (\max(h_i \bullet n, 0))^s)$$

- 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)

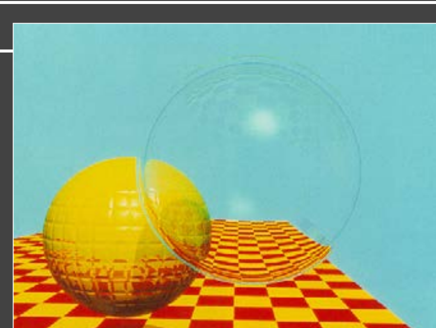
Outline

- Camera Ray Casting (choosing ray directions)
- Ray-object intersections
- Ray-tracing transformed objects
- Lighting calculations
- *Recursive ray tracing*

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

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

- 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 x 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

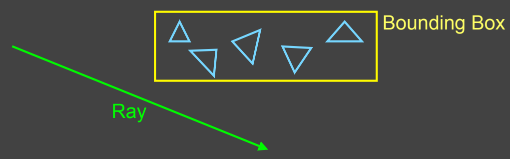
Testing each object for each ray is slow

- 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)

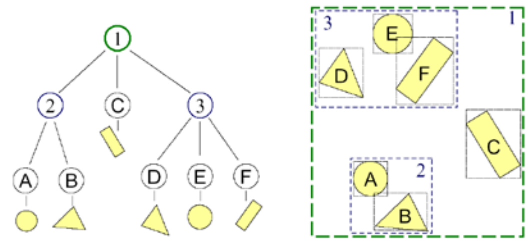
If no intersection bounding box, needn't check objects



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

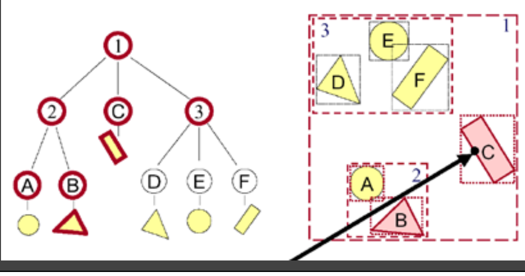
Bounding Volume Hierarchies 1

- 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

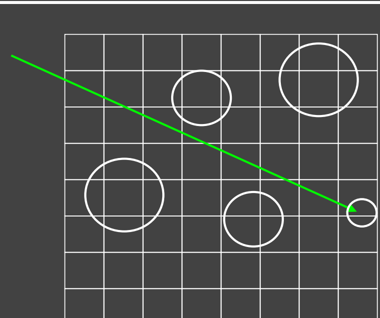


Bounding Volume Hierarchies 3

- 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

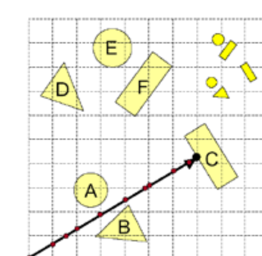


Uniform Grid: Problems

- Potential problem:
  - How choose suitable grid resolution?

Too little benefit if grid is too coarse

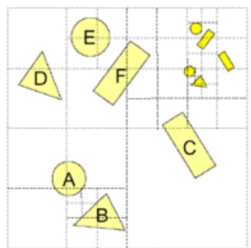
Too much cost if grid is too fine



Octree

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

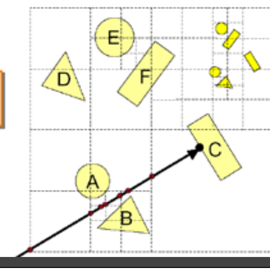
Generally fewer cells



Octree traversal

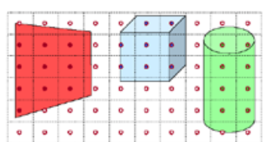
- Trace rays through neighbor cells
  - Fewer cells
  - More complex neighbor finding

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 “embarassingly parallelizable”
- etc.



Ray Tracing Acceleration Structures

- Bounding Volume Hierarchies (BVH)
- Uniform Spatial Subdivision (Grids)
- Binary Space Partitioning (BSP Trees)
  - Axis-aligned often for ray tracing: kd-trees
- Conceptually simple, implementation a bit tricky
  - Lecture relatively high level: Start early, go to section
  - Remember that acceleration a small part of grade

Math of 2D Bounding Box Test

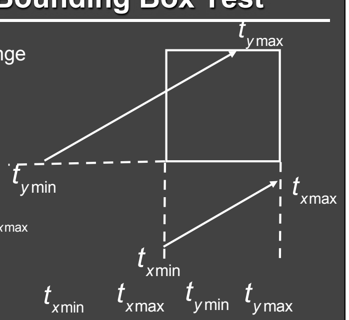
- Can you find a t in range

$$t > 0$$

$$t_{xmin} \leq t \leq t_{xmax}$$

$$t_{ymin} \leq t \leq t_{ymax}$$

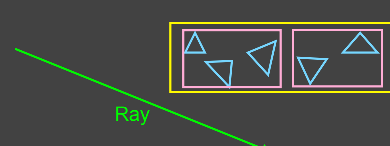
if $t_{xmin} > t_{ymax}$ *OR* $t_{ymin} > t_{xmax}$
  return false;
else
  return true;



No intersection if x and y ranges don’t overlap

Bounding Box Test

- Ray-Intersection is simple coordinate check
- Intricacies with test, see book
- Hierarchical Bounding Boxes



Hierarchical Bounding Box Test

- If ray hits root box
  - Intersect left subtree
  - Intersect right subtree
  - Merge intersections (find closest one)
- Standard hierarchical traversal
  - But caveat, since bounding boxes may overlap
- At leaf nodes, must intersect objects

Creating Bounding Volume Hierarchy

```
function bvh-node::create (object array A, int AXIS)
    N = A.length() ;
    if (N == 1) {left = A[0]; right = NULL; bbox = bound(A[0]);}
    else if (N == 2) {
        left = A[0] ; right = A[1] ;
        bbox = combine(bound(A[0]),bound(A[1])) ;
    else
        Find midpoint m of bounding box of A along AXIS
        Partition A into lists of size k and N-k around m
        left = new bvh-node (A[0…k],(AXIS+1) mod 3) ;
        right = new bvh-node(A[k+1…N-1],(AXIS+1) mod 3);
        bbox = combine (left -> bbox, right -> bbox) ;
```

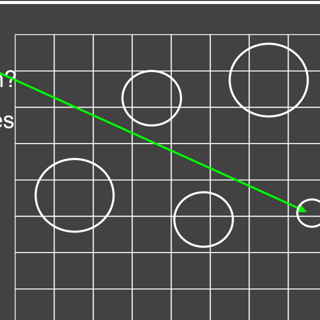
From page 285 of book

Uniform Spatial Subdivision

- Different idea: Divide space rather than objects
- In BVH, each object is in one of two sibling nodes
  - A point in space may be inside both nodes
- In spatial subdivision, each space point in one node
  - But object may lie in multiple spatial nodes
- Simplest is uniform grid (have seen this already)
- Challenge is keeping all objects within cell
- And in traversing the grid

Traversal of Grid High Level

- Next Intersect Pt?
- Irreg. samp. pattern?
- But regular in planes
- Fast algo. possible
- (more on board)

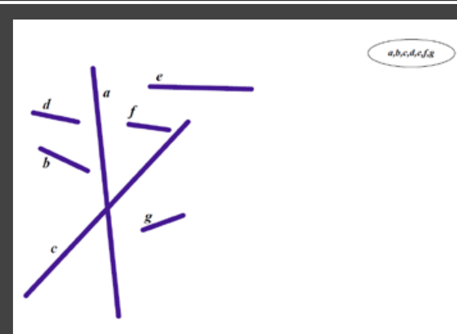


BSP Trees

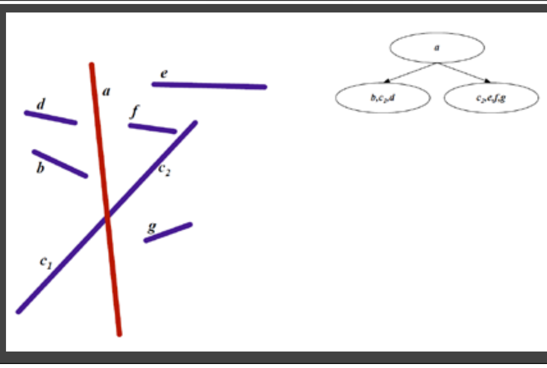
- Used for visibility and ray tracing
  - Book considers only axis-aligned splits for ray tracing
  - Sometimes called kd-tree for axis aligned
- Split space (binary space partition) along planes
- Fast queries and back-to-front (painter's) traversal
- Construction is conceptually simple
  - Select a plane as root of the sub-tree
  - Split into two children along this root
  - Random polygon for splitting plane (may need to split polygons that intersect it)

BSP slides courtesy Prof. O'Brien

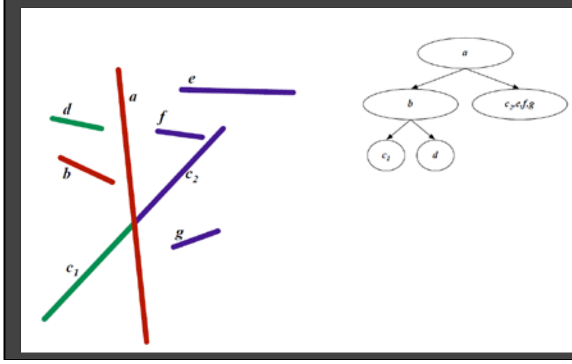
Initial State



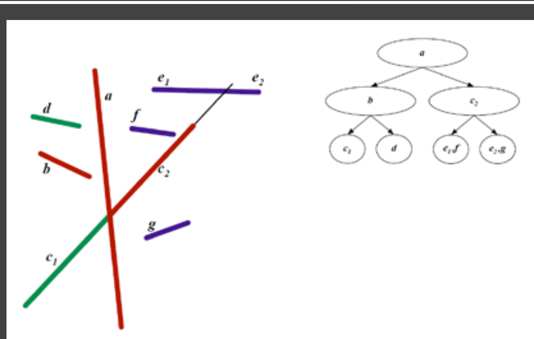
First Split



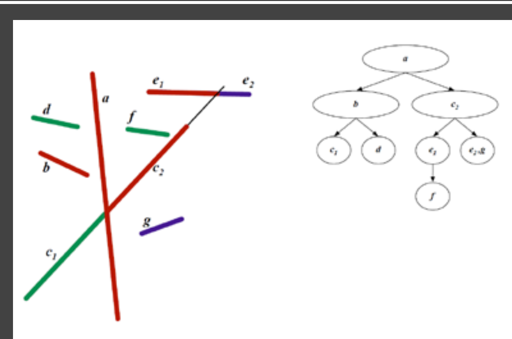
Second Split



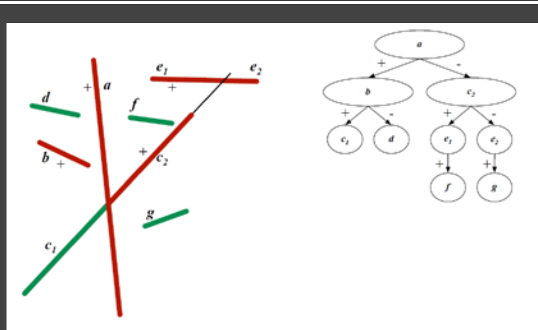
Third Split



Fourth Split



Final BSP Tree



BSP Trees Cont' d

- Continue splitting until leaf nodes
- Visibility traversal in order
  - Child one
  - Root
  - Child two
- Child one chosen based on viewpoint
  - Same side of sub-tree as viewpoint
- BSP tree built once, used for all viewpoints
  - More details in book

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
- Today graphics hardware (NVIDIA Optix)

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