# Ray Casting

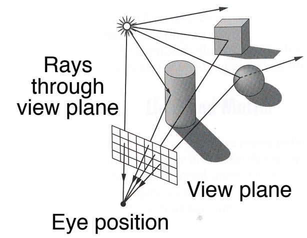
Adam Finkelstein & Tim Weyrich
Princeton University
COS 426, Spring 2008



## 3D Rendering

- The color of each pixel on the view plane depends on the radiance emanating from visible surfaces

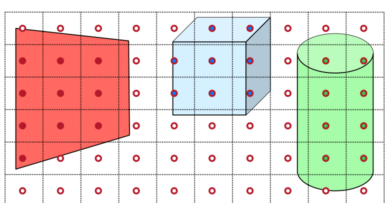
Simplest method is ray casting






## Ray Casting

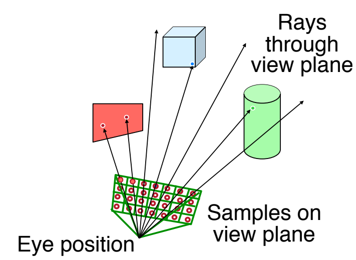
- For each sample …
  - Construct ray from eye position through view plane
  - Find first surface intersected by ray through pixel
  - Compute color sample based on surface radiance





## Ray Casting

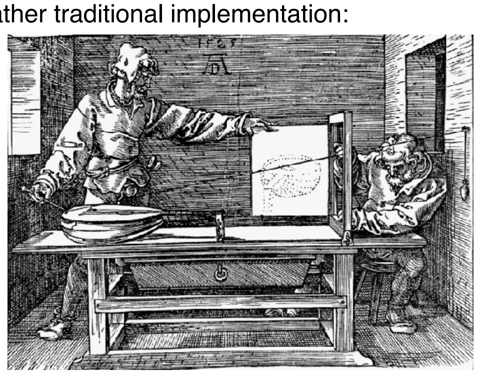
- For each sample …
  - Construct ray from eye position through view plane
  - Find first surface intersected by ray through pixel
  - Compute color sample based on surface radiance






## Ray Casting

- Rather traditional implementation:



A. Dürer, 16th century



## Ray Casting

- Simple implementation in C++:

```
Image RayCast(Scene scene, int width, int height)
{
    Image image = new Image(width, height);
    for (int i = 0; i < width; i++) {
        for (int j = 0; j < height; j++) {
            Ray ray = ConstructRayThroughPixel(scene.camera, i, j);
            Intersection hit = FindIntersection(ray, scene);
            image[i][j] = GetColor(scene, ray, hit);
        }
    }
    return image;
}
```

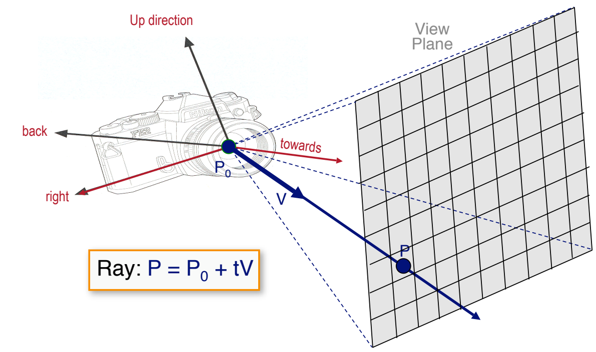
## Ray Casting

- Simple implementation:

```
Image RayCast(Scene scene, int width, int height)
{
    Image image = new Image(width, height);
    for (int i = 0; i < width; i++) {
        for (int j = 0; j < height; j++) {
            Ray ray = ConstructRayThroughPixel(scene.camera, i, j);
            Intersection hit = FindIntersection(ray, scene);
            image[i][j] = GetColor(scene, ray, hit);
        }
    }
    return image;
}
```



## Constructing Ray Through a Pixel




Ray: $P = P_0 + tV$

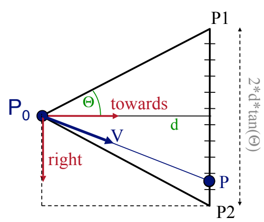


## Constructing Ray Through a Pixel

- 2D Example

Θ = frustum half-angle
d = distance to view plane

right = towards x up




$P1 = P_0 + d*towards - d*\tan(\Theta)*right$
$P2 = P_0 + d*towards + d*\tan(\Theta)*right$

$P = P1 + ((i + 0.5) / width) * (P2 - P1)$
$V = (P - P_0) / ||P - P_0||$

Ray: $P = P_0 + tV$



## Ray Casting

- Simple implementation:

```
Image RayCast(Scene scene, int width, int height)
{
    Image image = new Image(width, height);
    for (int i = 0; i < width; i++) {
        for (int j = 0; j < height; j++) {
            Ray ray = ConstructRayThroughPixel(scene.camera, i, j);
            Intersection hit = FindIntersection(ray, scene);
            image[i][j] = GetColor(scene, ray, hit);
        }
    }
    return image;
}
```



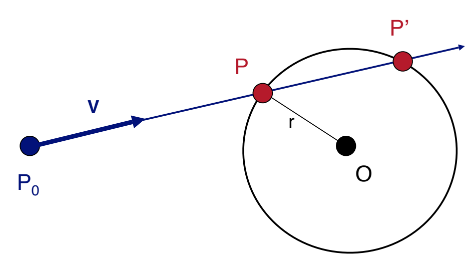
## Ray-Scene Intersection

- Intersections with geometric primitives
  - Sphere
  - Triangle
  - Groups of primitives (scene)
- Acceleration techniques
  - Bounding volume hierarchies
  - Spatial partitions
    - » Uniform grids
    - » Octrees
    - » BSP trees



## Ray-Sphere Intersection

Ray: $P = P_0 + tV$
Sphere: $|P - O|^2 - r^2 = 0$



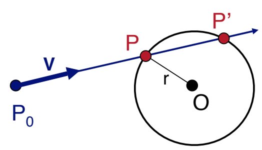
## Ray-Sphere Intersection I

Ray: $P = P_0 + tV$
Sphere: $|P - O|^2 - r^2 = 0$

Algebraic Method

Substituting for P, we get:
$|\mathbf{P_0 + tV} - O|^2 - r^2 = 0$

Solve quadratic equation:
$at^2 + bt + c = 0$
where:
$a = 1$
$b = 2 V \cdot (P_0 - O)$
$c = |P_0 - C|^2 - r^2 = 0$
$P = P_0 + tV$






## Ray-Sphere Intersection II

Ray: $P = P_0 + tV$
Sphere: $|P - O|^2 - r^2 = 0$

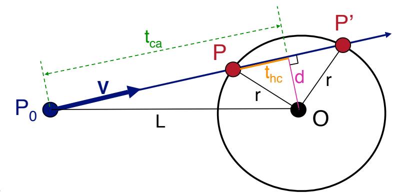
Geometric Method

$L = O - P_0$

$t_{ca} = L \cdot V$
if ($t_{ca} < 0$) return 0

$d^2 = L \cdot L - t_{ca}^2$
if ($d^2 > r^2$) return 0

$t_{hc} = \mathrm{sqrt}(r^2 - d^2)$
$t = t_{ca} - t_{hc}$ and $t_{ca} + t_{hc}$
$P = P_0 + tV$

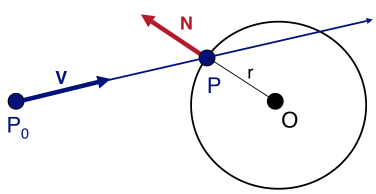





## Ray-Sphere Intersection

- Need normal vector at intersection for lighting calculations

$N = (P - O) / ||P - O||$





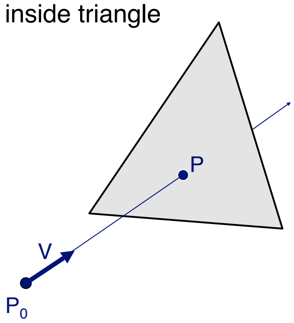

## Ray-Scene Intersection

- Intersections with geometric primitives
  - Sphere
  - » **Triangle**
  - Groups of primitives (scene)
- Acceleration techniques
  - Bounding volume hierarchies
  - Spatial partitions
    - » Uniform grids
    - » Octrees
    - » BSP trees



## Ray-Triangle Intersection

- First, intersect ray with plane
- Then, check if point is inside triangle





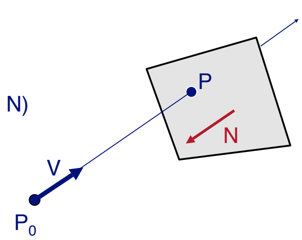

## Ray-Plane Intersection

Ray: $P = P_0 + tV$
Plane: $P \cdot N + d = 0$

Algebraic Method

Substituting for P, we get:
$(\mathbf{P_0 + tV}) \cdot N + d = 0$

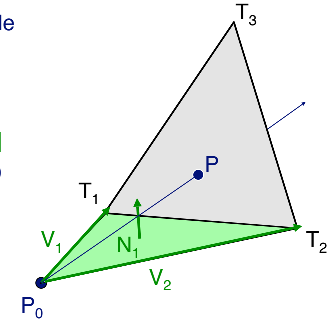
Solution:
$t = -(P_0 \cdot N + d) / (V \cdot N)$
$P = P_0 + tV$



## Ray-Triangle Intersection I

- Check if point is inside triangle algebraically

For each side of triangle
  $V_1 = T_1 - P_0$
  $V_2 = T_2 - P_0$
  $N_1 = V_2 \times V_1$
  [opt.: Normalize $N_1$]
  if $((P - P_0) \cdot N_1 < 0)$
    return FALSE;
end





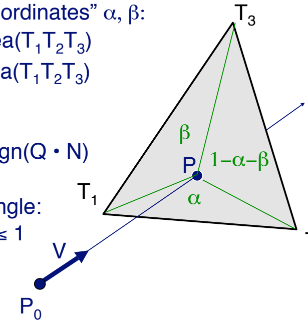

## Ray-Triangle Intersection II

- Check if point is inside triangle parametrically

Compute "barycentric coordinates" $\alpha$, $\beta$:
  $\alpha = \text{Area}(T_1T_2P) / \text{Area}(T_1T_2T_3)$
  $\beta = \text{Area}(T_1PT_3) / \text{Area}(T_1T_2T_3)$

$Q = (T2\text{-}T1) \times (T3\text{-}T1)$
$\text{Area}(T_1T_2T_3) = \|½ Q\| \text{ sign}(Q \cdot N)$

Check if point inside triangle:
  $0 \leq \alpha \leq 1$ and $0 \leq \beta \leq 1$
  and $\alpha + \beta \leq 1$






## Other Ray-Primitive Intersections

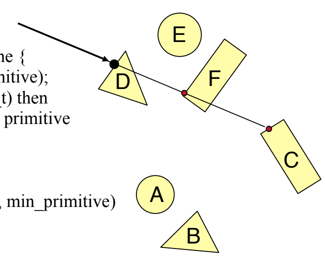
- Cone, cylinder, ellipsoid:
  - Similar to sphere
- Box
  - Intersect 3 front-facing faces, return closest
- Convex polygon
  - Same as triangle (check point-in-polygon algebraically)
- Concave polygon
  - Same plane intersection
  - More complex point-in-polygon test



## Ray-Scene Intersection

- Find intersection with front-most primitive in group

```
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)
}
```






## Ray-Scene Intersection

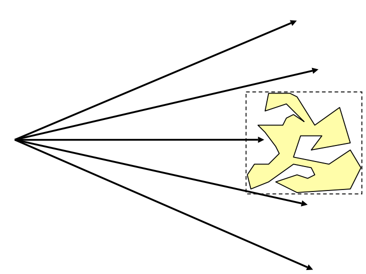
- Intersections with geometric primitives
  - Sphere
  - Triangle
  - Groups of primitives (scene)

» Acceleration techniques
  - Bounding volume hierarchies
  - Spatial partitions
    - » Uniform grids
    - » Octrees
    - » BSP trees



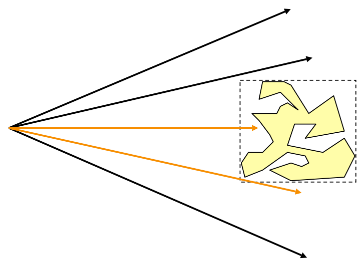
## Bounding Volumes

- Check for intersection with simple shape first



## Bounding Volumes

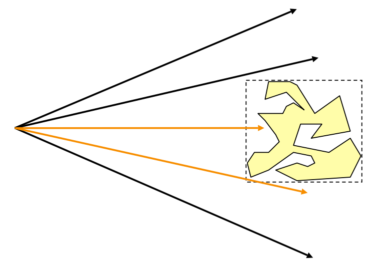
- Check for intersection with simple shape first





## Bounding Volumes

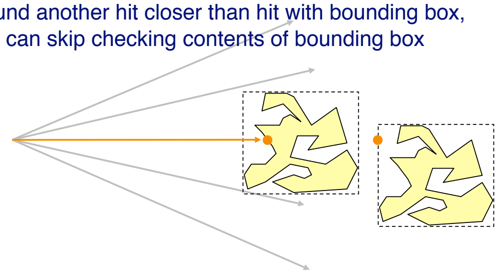
- Check for intersection with simple shape first
  - If ray doesn't intersect bounding volume, then it doesn't intersect its contents





## Bounding Volumes

- Check for intersection with simple shape first
  - If ray doesn't intersect bounding volume, then it doesn't intersect its contents
  - If found another hit closer than hit with bounding box, then can skip checking contents of bounding box





## Bounding Volumes

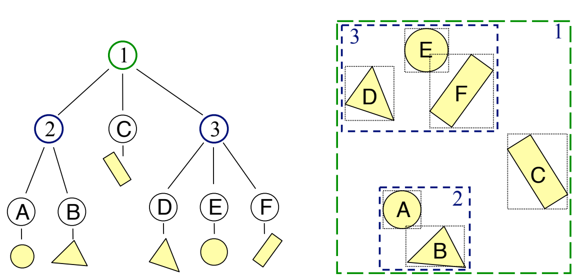
- Sort hits & detect early termination

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



## Bounding Volume Hierarchies I

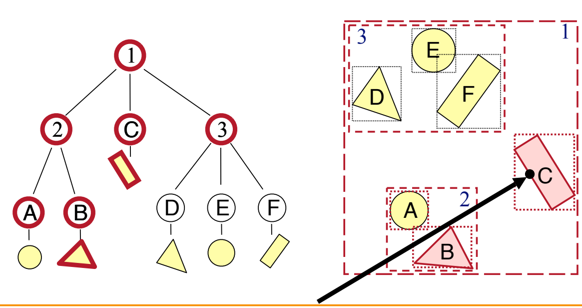
- Build hierarchy of bounding volumes
  - Bounding volume of interior node contains all children






## Bounding Volume Hierarchies

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



## Bounding Volume Hierarchies III

- Traverse scene nodes recursively

```
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;
}
```



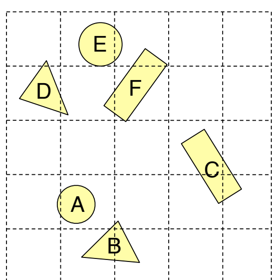
## Ray-Scene Intersection

- Intersections with geometric primitives
  - Sphere
  - Triangle
  - Groups of primitives (scene)
- » Acceleration techniques
  - Bounding volume hierarchies
  - Spatial partitions
    - » Uniform grids
    - » Octrees
    - » BSP trees



## Uniform Grid

- Construct uniform grid over scene
  - Index primitives according to overlaps with grid cells

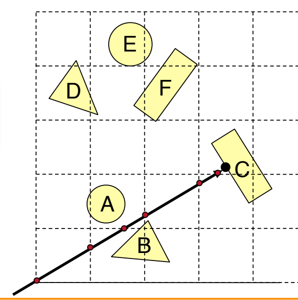





## Uniform Grid

- Trace rays through grid cells
  - Fast
  - Incremental

Only check primitives in intersected grid cells



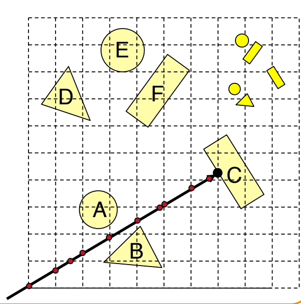



## Uniform Grid

- Potential problem:
  - How choose suitable grid resolution?

Too little benefit if grid is too coarse

Too much cost if grid is too fine





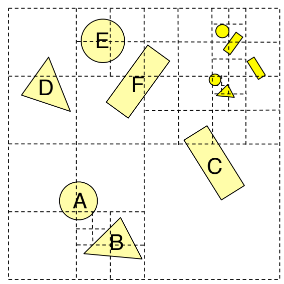

## Ray-Scene Intersection

- Intersections with geometric primitives
  - Sphere
  - Triangle
  - Groups of primitives (scene)
- » Acceleration techniques
  - Bounding volume hierarchies
  - Spatial partitions
    - » Uniform grids
    - » Octrees
    - » BSP trees

## Octree

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

Generally fewer cells

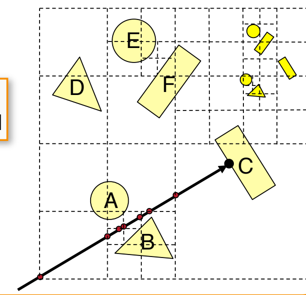





## Octree

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

Trade-off fewer cells for more expensive traversal






## Ray-Scene Intersection

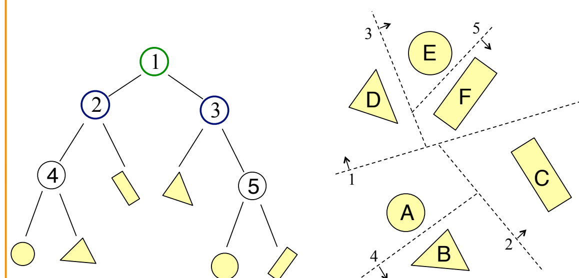
- Intersections with geometric primitives
  - Sphere
  - Triangle
  - Groups of primitives (scene)

» Acceleration techniques
  - Bounding volume hierarchies
  - Spatial partitions
    - » Uniform grids
    - » Octrees
    - » BSP trees



## Binary Space Partition (BSP) Tree

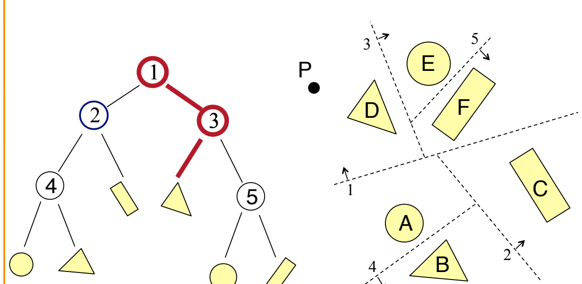
- Recursively partition space by planes
  - Every cell is a convex polyhedron






## Binary Space Partition (BSP) Tree

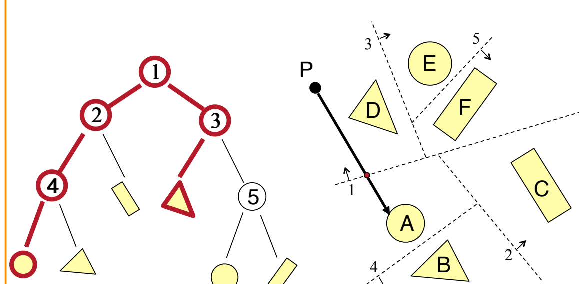
- Simple recursive algorithms
  - Example: point finding






## Binary Space Partition (BSP) Tree

- Trace rays by recursion on tree
  - BSP construction enables simple front-to-back traversal



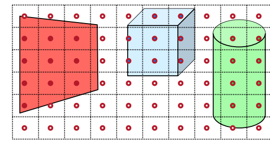
## Binary Space Partition (BSP) Tree

```
RayTreeIntersect(Ray ray, Node node, double min, double max)
{
    if (Node is a leaf)
        return intersection of closest primitive in cell, or NULL if none
    else
        dist = distance of the ray point to split plane of node
        near_child = child of node that contains the origin of Ray
        far_child = other child of node
        if the interval to look is on near side
            return RayTreeIntersect(ray, near_child, min, max)
        else if the interval to look is on far side
            return RayTreeIntersect(ray, far_child, min, max)
        else if the interval to look is on both side
            if (RayTreeIntersect(ray, near_child, min, dist)) return …;
            else return RayTreeIntersect(ray, far_child, dist, max)
}
```



## 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.





## Acceleration

- Intersection acceleration techniques are important
  - Bounding volume hierarchies
  - Spatial partitions
- General concepts
  - Sort objects spatially
  - Make trivial rejections quick
  - Utilize coherence when possible

Expected time is sub-linear in number of primitives



## Summary

- Writing a simple ray casting renderer is easy
  - Generate rays
  - Intersection tests
  - Lighting calculations

```
Image RayCast(Scene scene, int width, int height)
{
    Image image = new Image(width, height);
    for (int i = 0; i < width; i++) {
        for (int j = 0; j < height; j++) {
            Ray ray = ConstructRayThroughPixel(scene.camera, i, j);
            Intersection hit = FindIntersection(ray, scene);
            image[i][j] = GetColor(scene, ray, hit);
        }
    }
    return image;
}
```

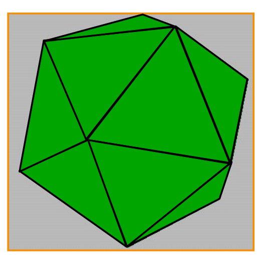


## Heckbert’s business card ray tracer

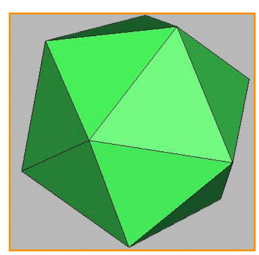
```
typedef struct{double x,y,z}vec;vec U,black,amb={.02,.02,.02};struct sphere{ vec cen,color;
double rad,kd,ks,kt,kl,ir}*s,*best,sph[]={0.,6.,.5,1.,1.,1.,.9, .05,.2,.85,0.,1.7,-1.,8.,-.5,1.,.5,.2,1.,
.7,.3,0.,.05,1.2,1.,8.,-.5,.1,.8,.8, 1.,.3,.7,0.,0.,1.2,3.,-6.,15.,1.,.8,1.,7.,0.,0.,0.,.6,1.5,-3.,-3.,12.,
.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!*/
```



## Next Time is Illumination!



Without Illumination



With Illumination