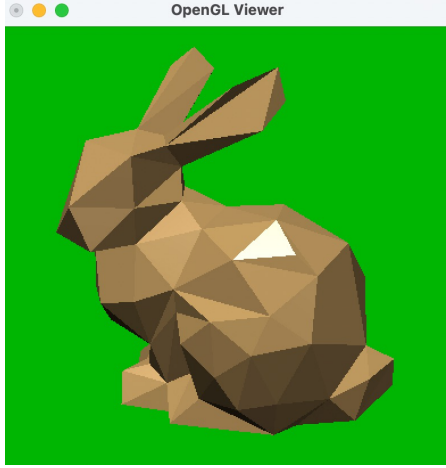
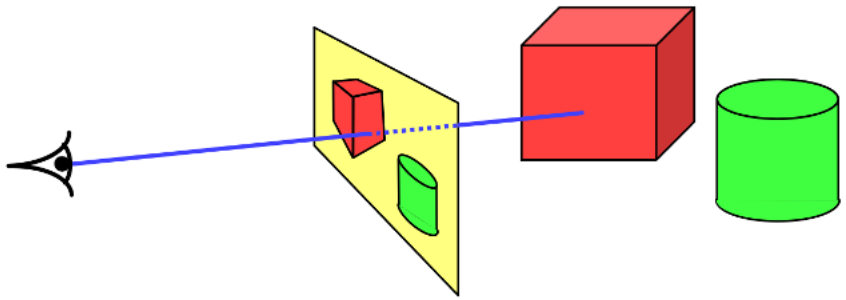


Ray Tracing

Welcome to embark on the journey on RAY TRACING!



Peter Hu

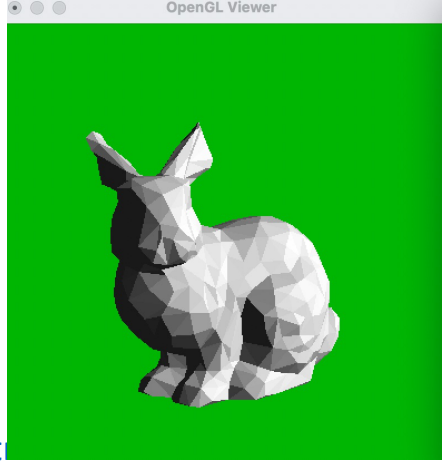
Huawei Technologies Research & Development

GPU Team

3 May 2023

HUAWEI

With Sincere Thanks to



Lai Wei's Ray Tracing Wiki

Other Useful Reference for Ray Tracing (and broader CG):

Fundamentals of Computer Graphics by Marschner & Shirley, CRC Press 2015 (4th edition)

MIT EECS 6.837 Computer Graphics (solid Math intro) https://groups.csail.mit.edu/graphics/classes/6.837/F04/calendar.ht

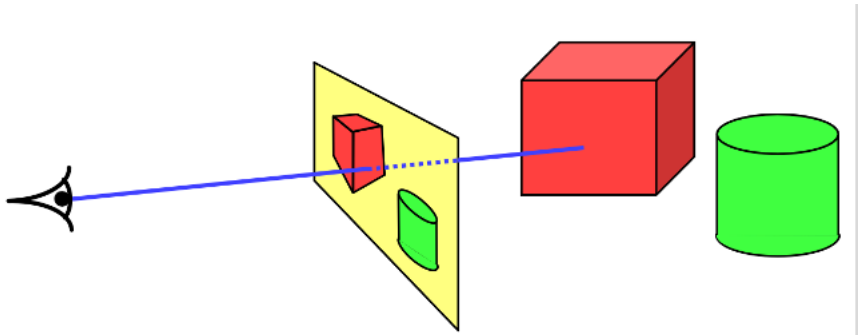
CMU Computer Graphics : Spring 2022 (cmu.edu)

Cambridge IA Intro to Graphics https://www.cl.cam.ac.uk/teaching/2223/Graphics/

Learn OpenGL https://learnopengl.com

Vulkan_Ray_Tracing_Overview_Apr21 Khronos Template 2015 (slides)

DirectX Specs https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html

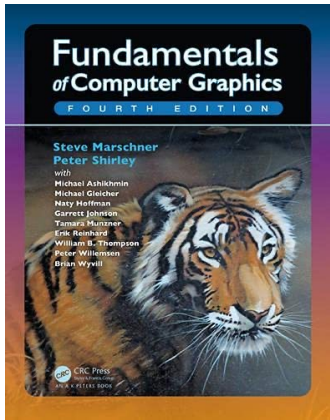


About the Cover

The cover image is from *Tiger in the Water* by J. W. Baker (brushed and air-brushed acrylic on canvas, 16" by 20", www.jwbart.com).

The subject of a tiger is a reference to a wonderful talk given by Alain Fournier (1943–2000) at a workshop at Cornell University in 1998. His talk was an evocative verbal description of the movements of a tiger. He summarized his point:

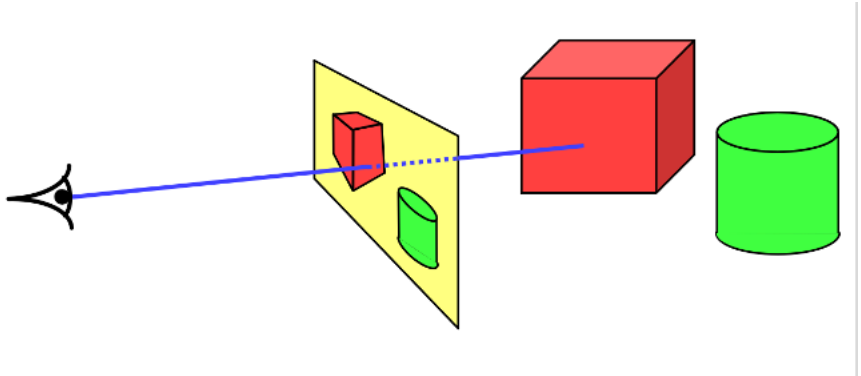
> Even though modelling and rendering in computer graphics have been improved tremendously in the past 35 years, we are still not at the point where we can model automatically a tiger swimming in the river in all its glorious details. By automatically I mean in a way that does not need careful manual tweaking by an artist/expert.
>
> The bad news is that we have still a long way to go.
>
> The good news is that we have still a long way to go.



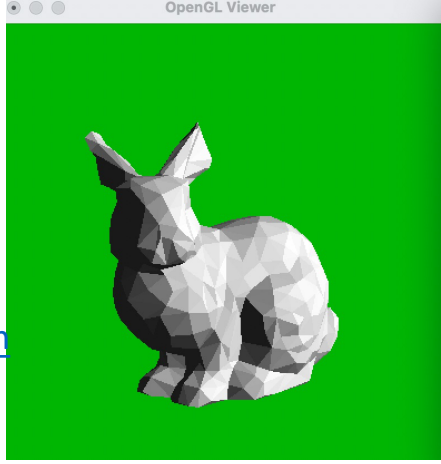
About me

Computer Graphics is most attractive to me at the moment, with lots of other interaction of Math and CS as well.

- Have engaged in course MIT 6.837, especially with focus on Ray-object intersection and Ray Tracer.
- Have attended Cambridge Part IA Computer Graphics
- Implemented a Ray Tracer myself without Graphics API , also with OpenGL experience.
- Currently, interested in research in ray tracing related, like BVH in spatial data structure as an acceleration methods



My Ray Tracer C++, OpenGL As a part of MIT6.837 Project.



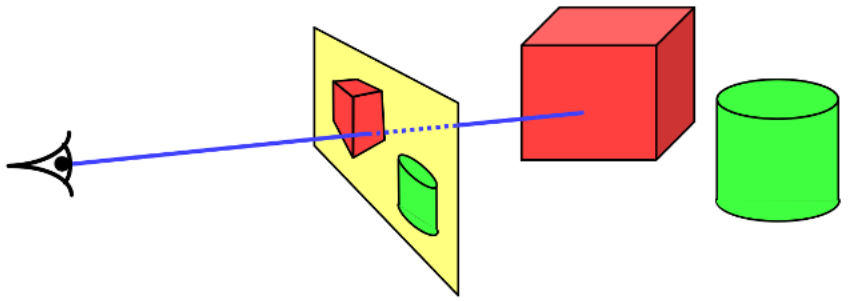
https://github.com/PeterHUistyping/MIT6.837-CG-Fall2004-Assignment/blob/main/MyProject/assignment4_Ray-Tracer/ray/raytracer.h

https://github.com/PeterHUistyping/MIT6.837-CG-Fall2004-Assignment

Lots more example of Graphics generated by my own Ray Tracer or OpenGL.

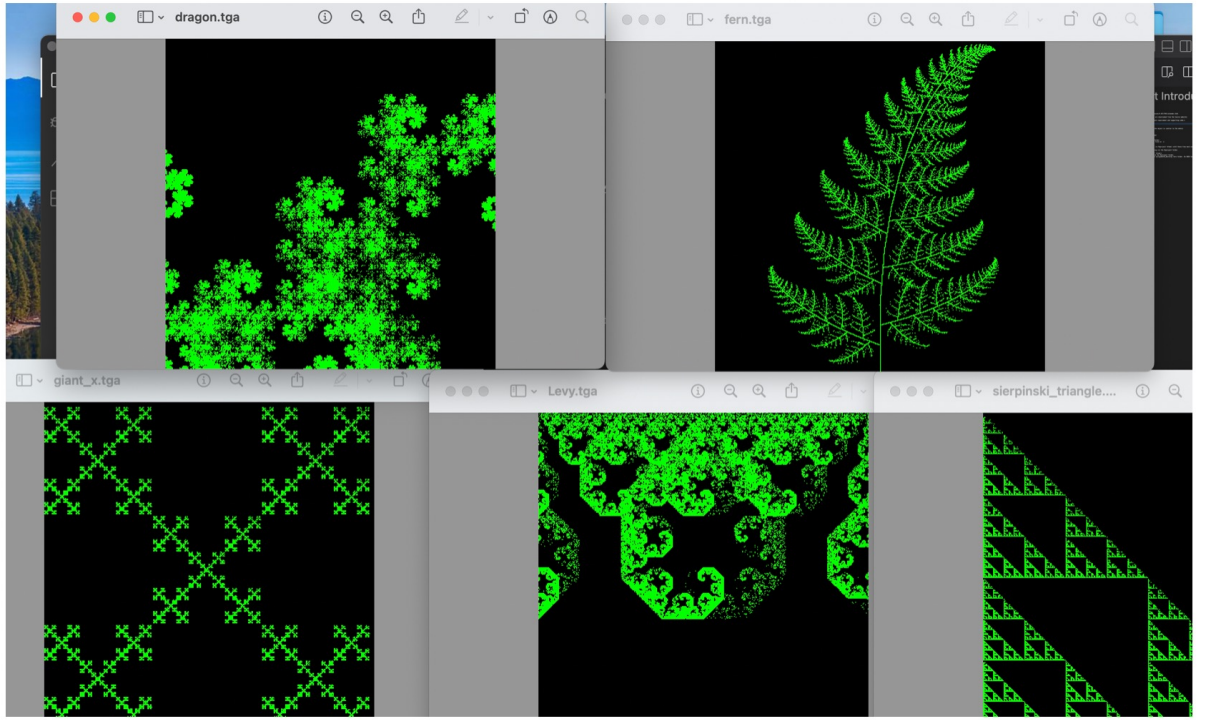
```
class RayTracer
{
public:
    RayTracer()
```

- assignment0_Barnsley fern
- assignment1_Sphere-Ray
- assignment2_Tri-Ray
- assignment3_Open-GL
- assignment4_Ray-Tracer
- assignment5_Voxel-Rendering
- assignment6_Grid-Acceleration+Solid-Textures
- assignment7_Supersampling+Antialiasing



MIT6.837-CG-Fall2004-Assignment

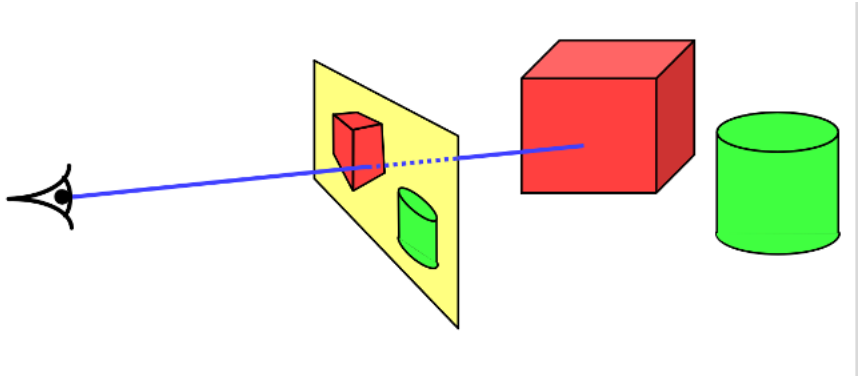
Computer Graphics



Recent Ongoing Research Focus

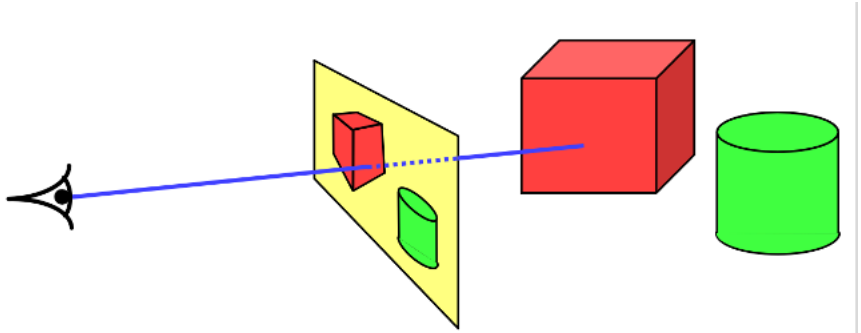
- Ray Traversal in BVH
- BVH Construction
- BVH Algorithm

Any suggestions and idea on RT are warmly welcome!



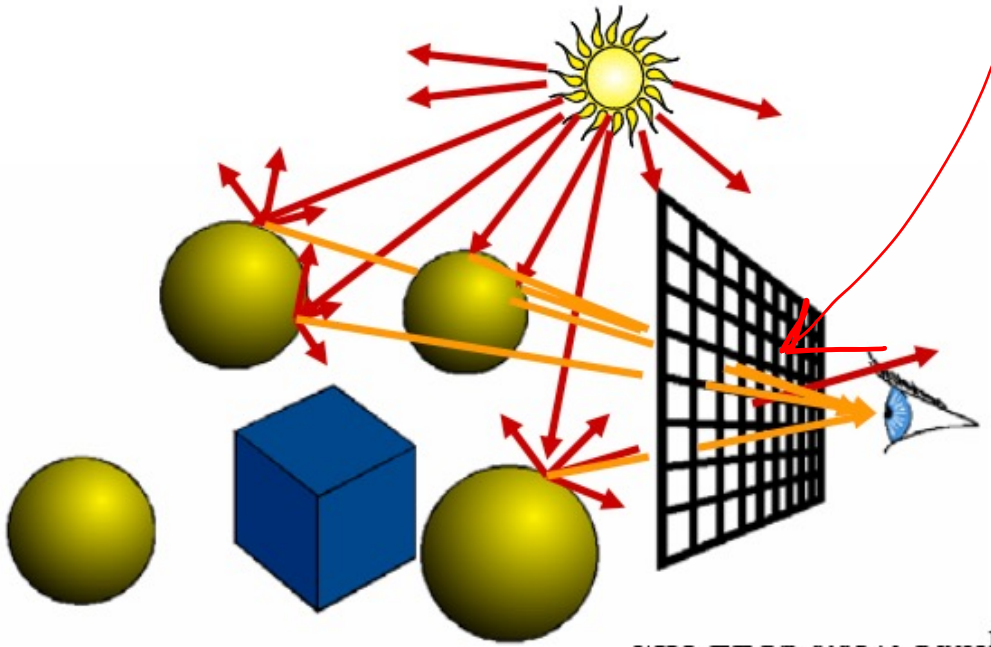
Part 1: Ray Tracing Algorithm

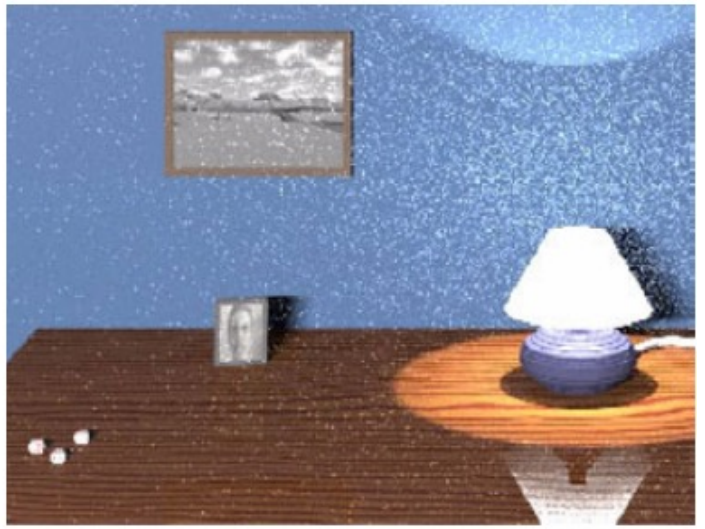
Motivation, Global Illumination, Coding: Step-by-step explanation



Forward Ray Tracing

- Start from the light source
  - But low probability to reach the eye
- What can we do about it?
  - Always send a ray to the eye…. still not efficient





Motivation

Real world mechanism-- Forward Ray Tracing is actually NOT efficient

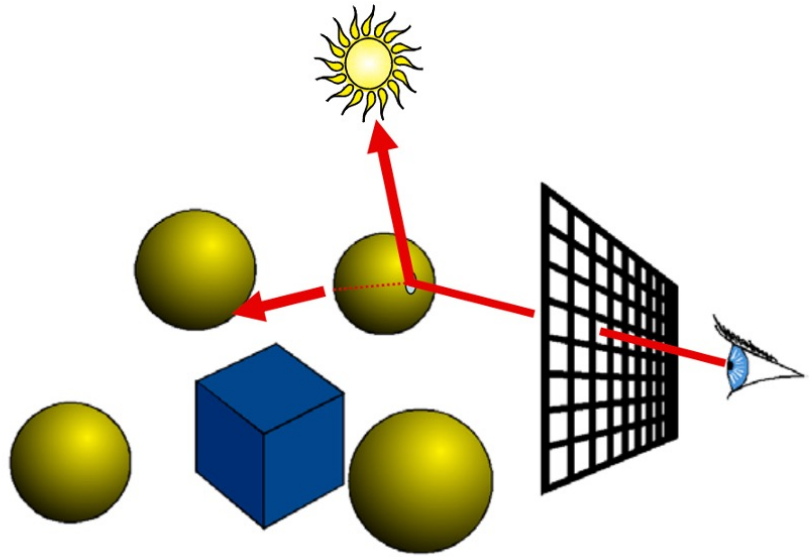
```
Lights {
    numLights 1
    DirectionalLight {
        direction 0 -1 0
        color 0.8 0.8 0.8
    }
}


Group {
    numObjects 2

    MaterialIndex 0
    Sphere {
        center 0 0 0
        radius 1
    }

    MaterialIndex 1
    Plane {
        normal 0.01 1 0.01
        offset -1
    }


}
```

Ray Tracing Algorithm

Ray tracing is a rendering technique used in CG to create realistic images by simulating the way light interacts with objects in the virtual world.

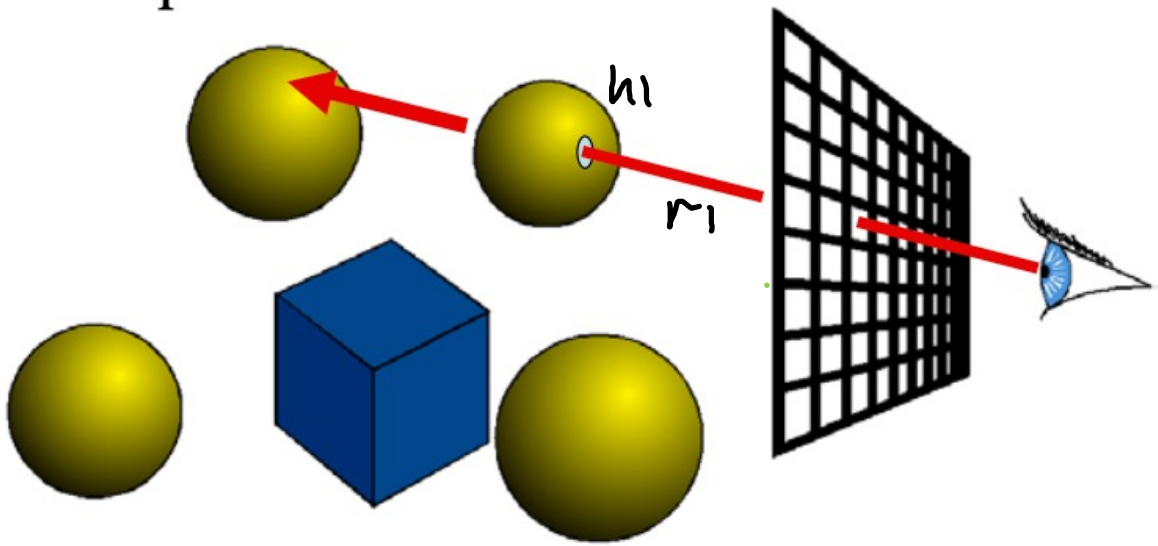
The technique traces the ***path of light*** from the eye/camera to the scene(backward), where it interacts with various objects and surfaces, and then back to the camera to produce the final image.

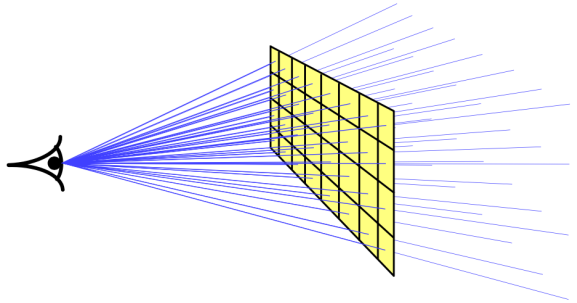
Reflections/refractions of light are simulated by generating a new set of rays based on material properties of objects.



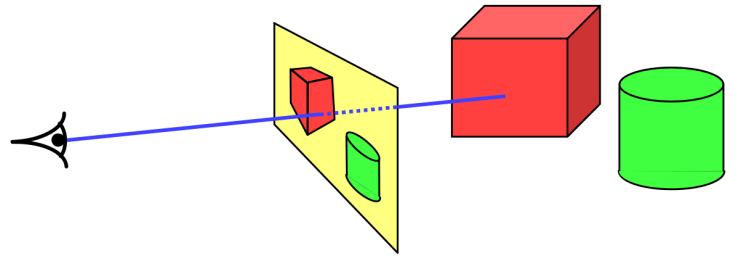
Ray Casting

- For every pixel
  construct a ray from the eye
  - For every object in the scene
    - Find intersection with the ray
    - Keep if closest





shoot a ray through each pixel



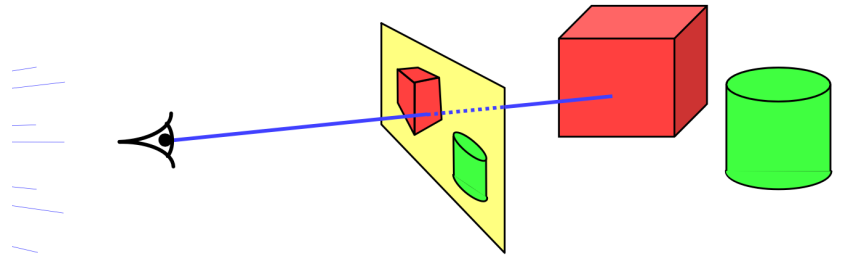
whatever the ray hits determines the colour of that pixel

```
Lights {
    numLights 1
    DirectionalLight {
        direction 0 -1 0
        color 0.8 0.8 0.8
    }
}
```

```
Group {
    numObjects 2

    MaterialIndex 0
    Sphere {
        center 0 0 0
        radius 1
    }

    MaterialIndex 1
    Plane {
        normal 0.01 1 0.01
        offset -1
    }

}
```

First Hit Problem

Given a scene defined by a set of N primitives and a ray r, find the closest point of intersection of r with the scene

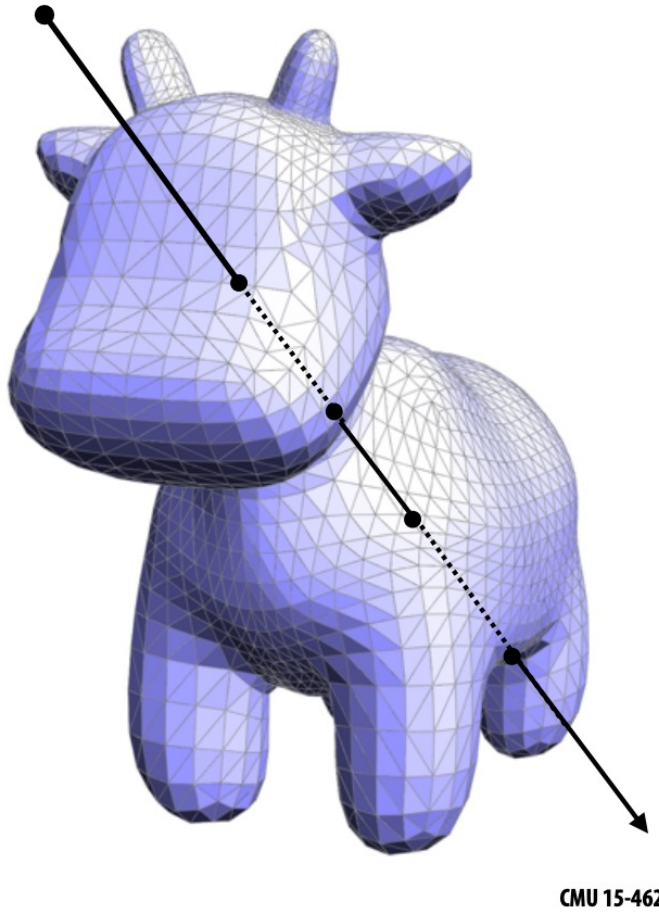


whatever the ray hits determines the colour of that pixel

"Find the first primitive the ray hits"

Naïve algorithm?

1. **Intersect ray with every triangle**
2. **Keep the closest hit point**



Complexity? $O(N)$

Can we do better? Bounding box!

```
struct HitInfo {
   Primitive* prim;  // which primitive did the ray hit?
   float t;    // at what t value?
};
```

CMU 15-462/662

void rayCasting( )

```
{
    Image image(width, height); // image.SetAllPixels(parser->getBackgroundColor());
    Vec2f p;                    //(0,0)->(1,1) 1024*1024 Generate Points on the sceen coordinate
    float pixel = 1024;
    for (float i = 0; i < pixel; i++)
    { // load the camera
        for (float j = 0; j < pixel; j++)   // FOR every pixel (i, j) in the screen plane:
        {
            Vec3f pcolor;
            p.Set(i / pixel, j / pixel); // Ray - generateRay(p)  //~Camera
            Hit h;
            Ray ray = parser->getCamera()->generateRay(p);
            pcolor += ray_tracer->traceRay(ray, epsilon, 0, 1, 0, h);
            image.SetPixel(width * i / pixel, height * j / pixel, pcolor);
        }
    }
    image.SaveTGA(output_file);
}
```

continue...

https://github.com/PeterHUistyping/MIT6.837-CG-Fall2004-Assignment/blob/main/MyProject/assignment4_Ray-Tracer/main.C

Does Ray Tracing Simulate Physics? ✗

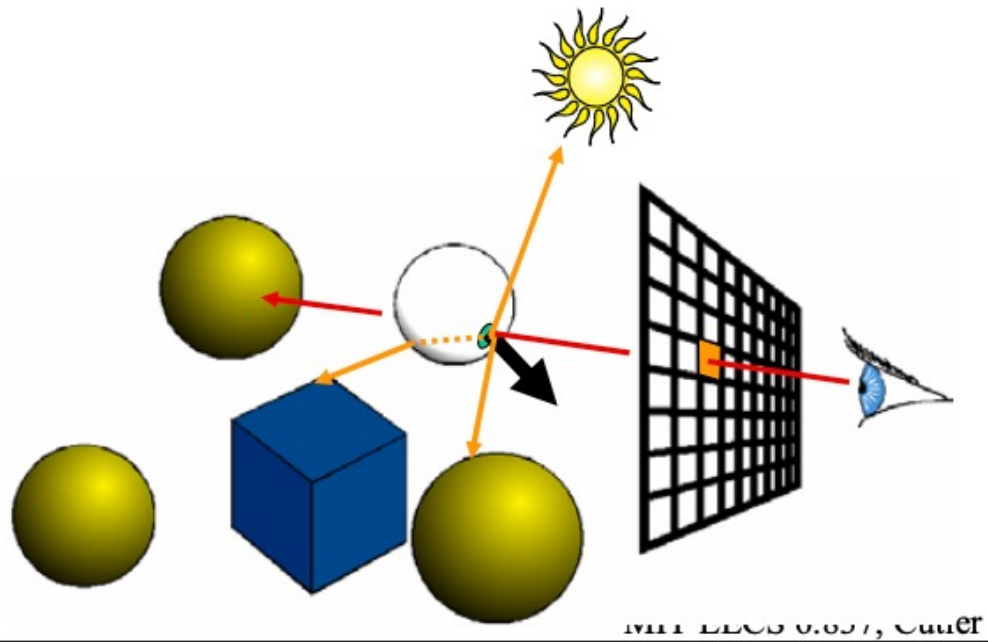
Motivation

BACKWARD Ray Tracing

Only consider the screen space

Efficient!

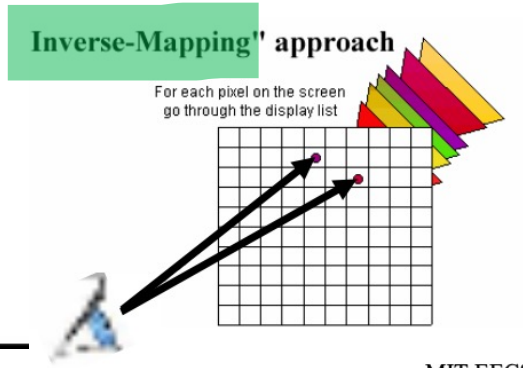
- Photons go from the light to the eye, not the other way
- What we do is *backward ray tracing*

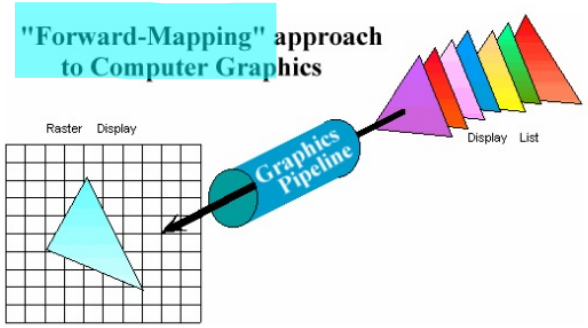


MIT EECS 6.837, Cutler and Durand

Ray Casting vs. Rendering Pipeline

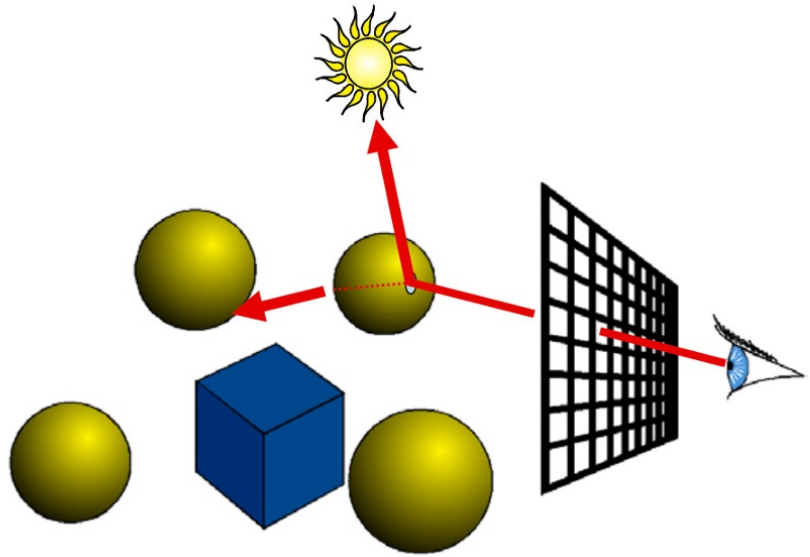
Ray Casting	Rendering Pipeline
`For each pixel`	`For each triangle`
`For each object`	`For each pixel`
Send pixels to the scene	Project scene to the pixels
Discretize first	Discretize last





Ray Tracing Algorithm

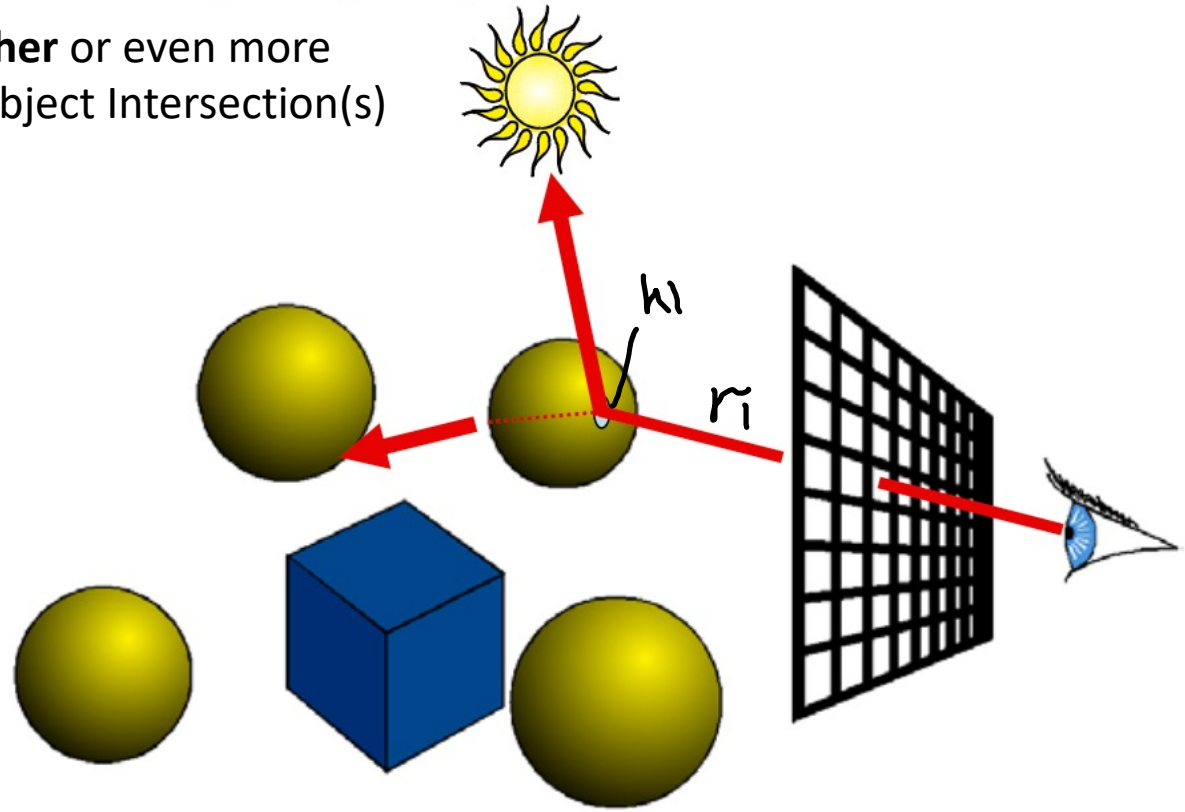
Ray tracing is a rendering technique used in CG to create realistic images by simulating the way light interacts with objects in the virtual world.
The technique traces the ***path of light*** from the eye/camera to the scene(backward), where it interacts with various objects and surfaces, and then back to the camera to produce the final image.
Reflections/refractions of light are simulated by generating ***a new set of rays*** based on material properties of objects.



Ray Tracing

- Shade (interaction of light and material)
- Secondary rays (shadows, reflection, refraction

another or even more
ray-object Intersection(s)

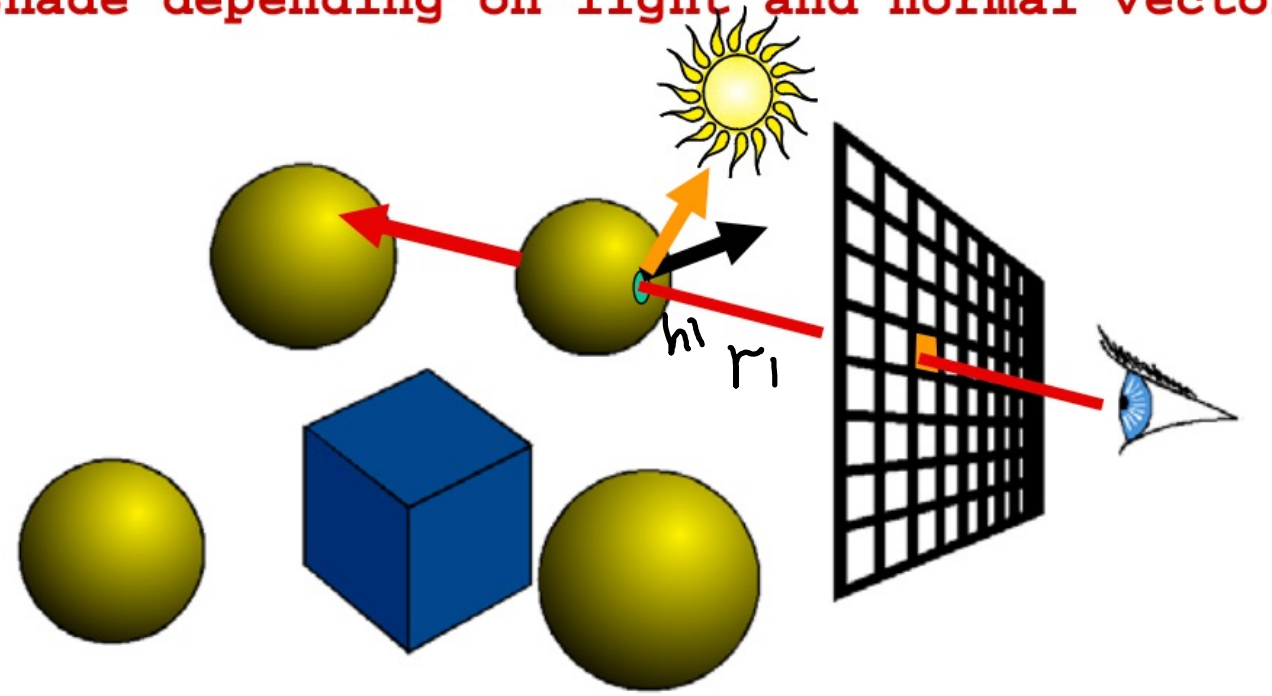


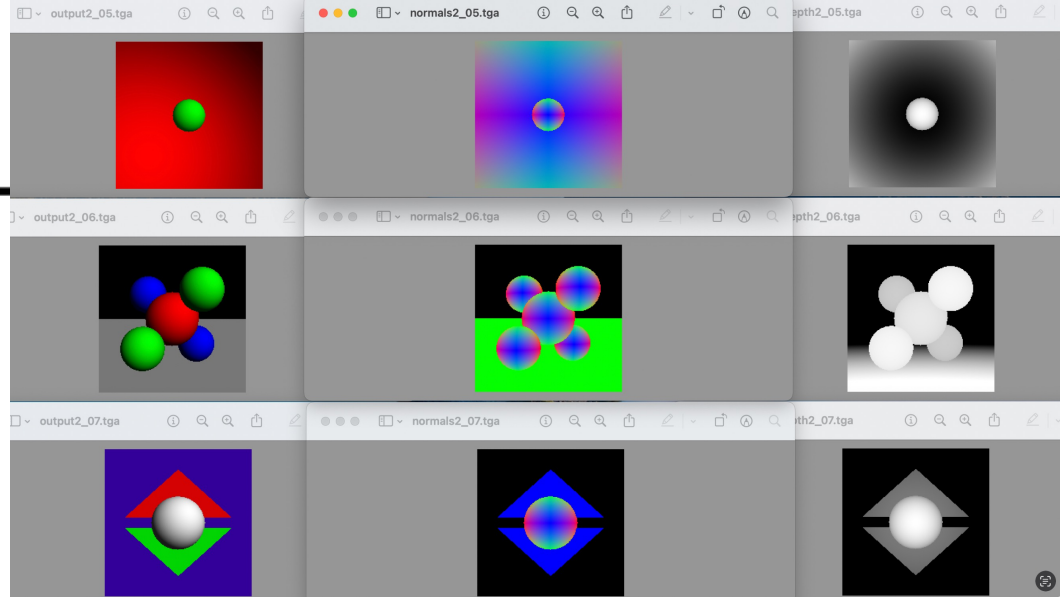

```
Lights {
    numLights 1
    DirectionalLight {
        direction 0 -1 0
        color 0.8 0.8 0.8
    }
}


Group {
    numObjects 2

    MaterialIndex 0
    Sphere {
        center 0 0 0
        radius 1
    }

    MaterialIndex 1
    Plane {
        normal 0.01 1 0.01
        offset -1
    }


}
```

Shading

```
For every pixel
    Construct a ray from the eye
    For every object in the scene
        Find intersection with the ray
        Keep if closest
```

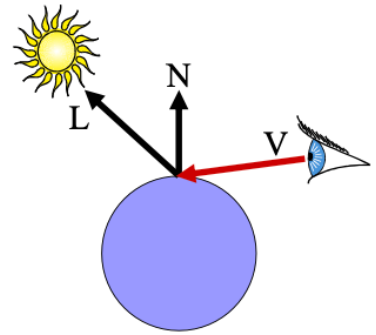
Shade depending on light and normal vector



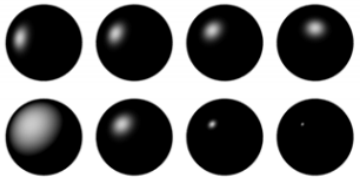


A Note on Shading

- Surface/Scene Characteristics:
  - surface normal
  - direction to light
  - viewpoint
- Material Properties
  - Diffuse (matte)
  - Specular (shiny)
  - …



Diffuse sphere

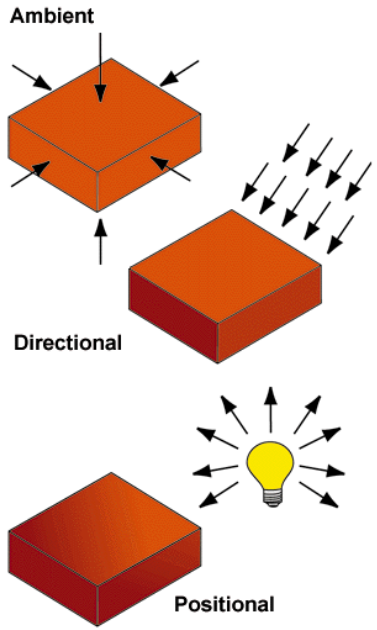


Specular spheres

Shading

Sum of three components:
1. “ambient” +
2. diffuse reflection +
3. specular reflection

1. Ambient Illumination
- Represents the reflection of all indirect illumination.
- This is a total hack!
- Avoids the complexity of global illumination.




```
pcolor += am; // ambient color
```

More specifically, pcolor += k_a . am
where k_a is a *vec3*
material-related ambient reflection coefficient

```
Background {
    color 0 0 1
    ambientLight 0.2 0.2 0.2
}
```

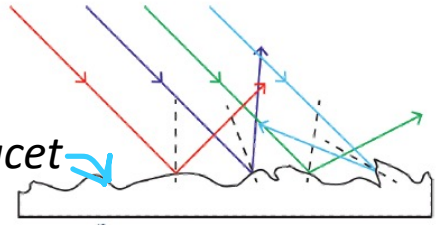
```
if (!parser->getGroup()->intersectShadowRay(r2, h2, epsilon_shadow)) // epsilon self-shadowing
{                                                                    // Diffuse  Color + Specular Color
    pcolor += hit.getMaterial()->Shade(ray, hit, dirToLight, lcolor);
    RayTree::AddShadowSegment(r2, 0, h2.getT());
}
```

Implemented by the function of another class

https://github.com/PeterHUistyping/MIT6.837-CG-Fall2004-Assignment/blob/main/MyProject/assignment4_Ray-Tracer/ray/material.h

Shading

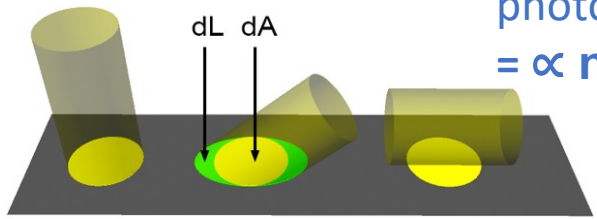
Sum of three components:
1. "ambient" +
2. *diffuse reflection* +
3. *specular reflection*



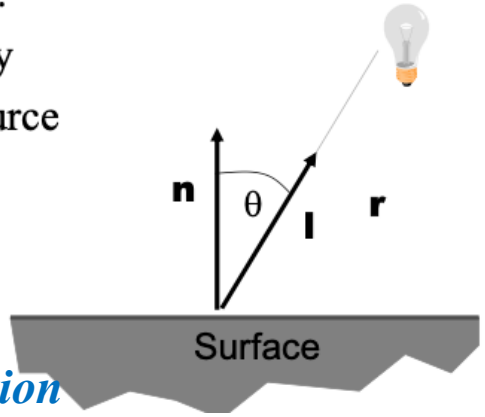
2. Ideal Diffuse Reflection

- Single Point Light Source
  - k_d: diffuse coefficient.
  - **n**: Surface normal.
  - **l**: Light direction.
  - L_i: Light intensity
  - r: Distance to source

$$L_o = k_d(\mathbf{n}\cdot\mathbf{l})\frac{L_i}{r^2}$$

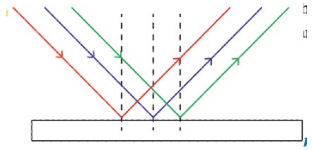


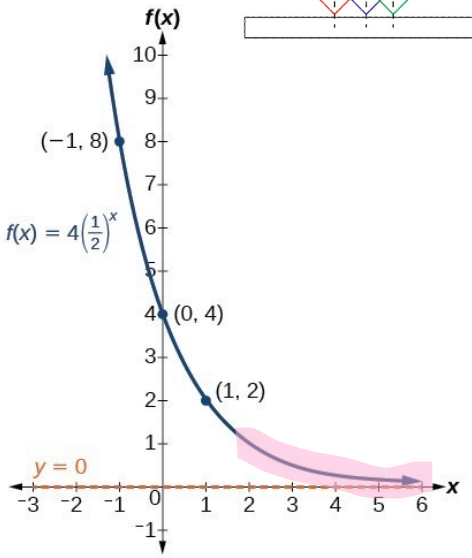
Inversely proportional to photon **projection area** = ∝ **n.l** dot product




```
if (normal.Dot3(dirToLight) > 0)
  d = normal.Dot3(dirToLight);

lcolor *= d;
lcolor = lcolor * diffuseColor;
pcolor += lcolor; // diffuse color
```

Ideal Specular material





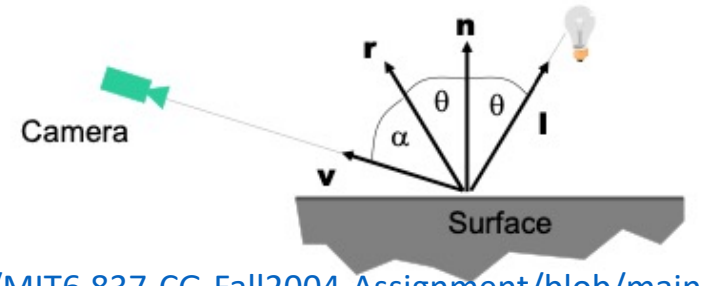
3. Specular Reflection

– Depends on the angle between the ideal reflection direction and the viewer direction α.

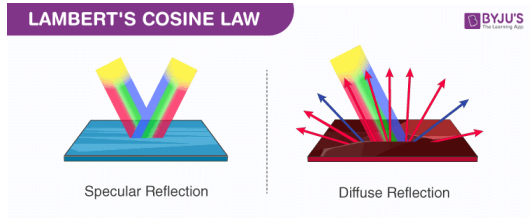
Parameters
- k_s: specular reflection coefficient
- q : specular reflection exponent

$$L_o = k_s(\cos\alpha)^q\frac{L_i}{r^2}$$

$$L_o = k_s(\mathbf{v}\cdot\mathbf{r})^q\frac{L_i}{r^2}$$




```
h = dirToLight - ray.getDirection(); // negative of ray dir
h.Normalize();
d = h.Dot3(hit.getNormal());
if (d < 0)
{
  d = 0; // not fix
}
d = pow(d, exponent);  // q>1
l = lcolor * d;        // ignore /r^2
l = l * specularColor; // specular color
```

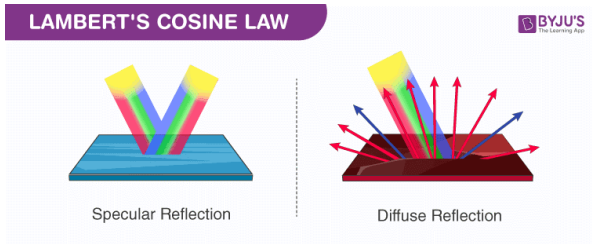



```
Materials {
    numMaterials 2
    PhongMaterial {
        diffuseColor 1 0 0
    }
    PhongMaterial {
        diffuseColor 0 1 0
    }
}
```

https://github.com/PeterHUistyping/MIT6.837-CG-Fall2004-Assignment/blob/main/MyProject/assignment4_Ray-Tracer/ray/material.h

Shading

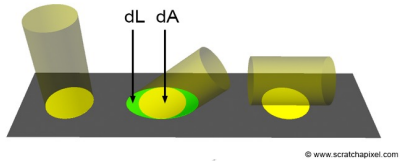
Sum of three components:
1. “ambient” +
2. diffuse reflection +
3. specular reflection

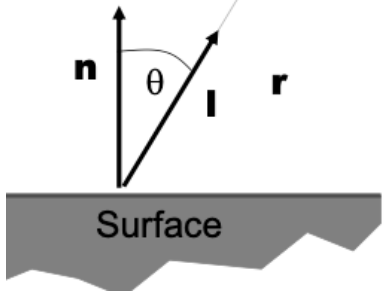


2. Ideal Diffuse Reflection

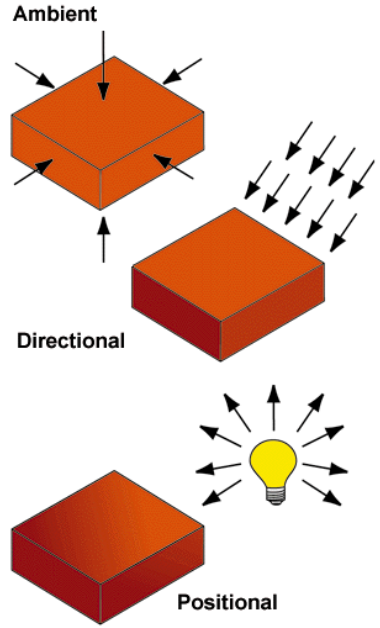
- Single Point Light Source
  - k_d: diffuse coefficient.
  - **n**: Surface normal.
  - **l**: Light direction.
  - L_i: Light intensity
  - r: Distance to source

$$L_o = k_d(\mathbf{n}\cdot\mathbf{l})\frac{L_i}{r^2}$$






```
Materials {
    numMaterials 2
    PhongMaterial {
        diffuseColor 1 0 0
    }
    PhongMaterial {
        diffuseColor 0 1 0
    }
}
```



3. Specular Reflection

– Depends on the angle between the ideal reflection direction and the viewer direction α.

- Parameters
  - k_s: specular reflection coefficient
  - q : specular reflection exponent

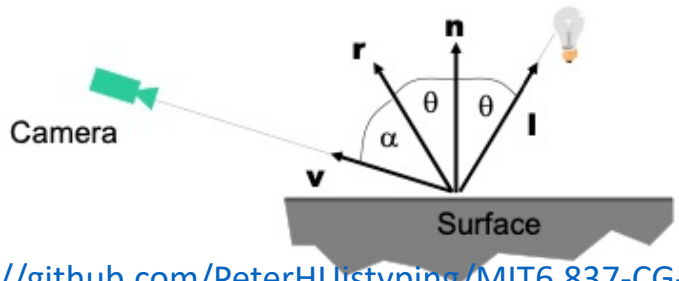
$$L_o = k_s(\cos\alpha)^q\frac{L_i}{r^2}$$

$$L_o = k_s(\mathbf{v}\cdot\mathbf{r})^q\frac{L_i}{r^2}$$

```
if (!parser->getGroup()->intersectShadowRay(r2, h2, epsilon_shadow)) // epsilon self-shadowing
{
    pcolor += hit.getMaterial()->Shade(ray, hit, dirToLight, lcolor);
    RayTree::AddShadowSegment(r2, 0, h2.getT());
}
```

// Diffuse Color + Specular Color

Implemented by the function of another class



1. Ambient Illumination

- Represents the reflection of all indirect illumination.
- This is a total hack!
- Avoids the complexity of global illumination.

```
pcolor += am; // ambient color
```

```
Background {
    color 0 0 1
    ambientLight 0.2 0.2 0.2
}
```

https://github.com/PeterHUistyping/MIT6.837-CG-Fall2004-Assignment/blob/main/MyProject/assignment4_Ray-Tracer/ray/material.h

```
Vec3f Shade(const Ray &ray, const Hit &hit, const Vec3f &dirToLight, const Vec3f &lightColor) // const
{
  Vec3f normal = hit.getNormal();
  if (shadeback)
  {
    if (hit.getNormal().Dot3(ray.getDirection()) > 0)
      normal.Negate(); // back shading
  }

  Vec3f lcolor = lightColor, pcolor;
  float d = 0;
  assert(fabs(ray.getDirection().Length() - 1) < 1e-6);
  if (normal.Dot3(dirToLight) > 0)
    d = normal.Dot3(dirToLight);

  lcolor *= d;
  lcolor = lcolor * diffuseColor;
  pcolor += lcolor; // diffuse color
                    //--------------------------------------------------------//
  Vec3f h, l;
  lcolor = lightColor;
  assert(fabs(dirToLight.Length() - 1) < 1e-6);
  assert(fabs(ray.getDirection().Length() - 1) < 1e-6);
  h = dirToLight - ray.getDirection(); // negative of ray dir
  h.Normalize();
  d = h.Dot3(hit.getNormal());
  if (d < 0)
  {
    d = 0; // not fix
  }
  d = pow(d, exponent);  // q>1
  l = lcolor * d;        // ignore /r^2
  l = l * specularColor; // specular color
  /*if (d < 0)
  {
    l *= dirToLight.Dot3(hit.getNormal()); // fix
    // multiplied by the dot product of the normal and direction to the light
  }*/
  pcolor += l;

  return pcolor;
}
```

tance

Point Light Source

ffuse coefficient.

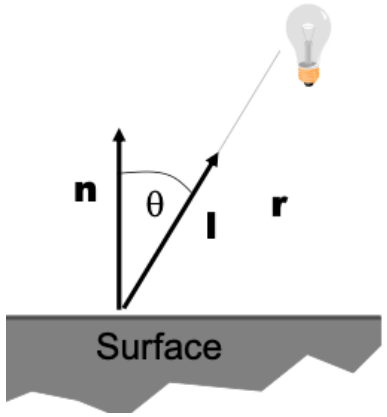
rface normal.

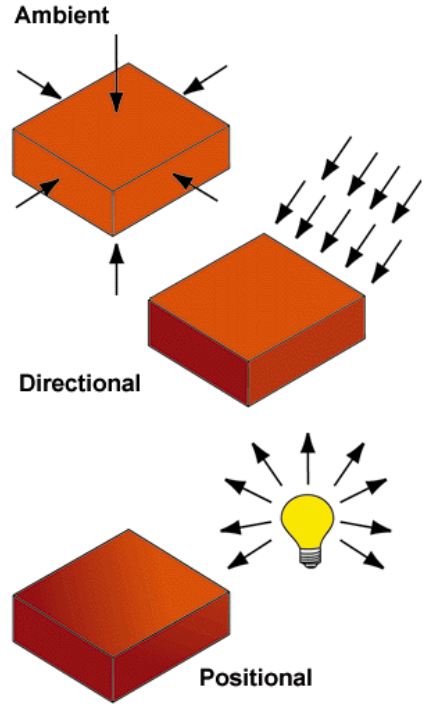
ht direction.

ight intensity

stance to source

$$L_o = k_d(\mathbf{n}\cdot\mathbf{l})\frac{L_i}{r^2}$$






```
rials {
umMaterials 2
PhongMaterial {
    diffuseColor 1 0 0
}
PhongMaterial {
    diffuseColor 0 1 0
}
```

```
)->intersectShadowRay(r2, h2, epsilon_shadow)) // epsilon self-shadowing
                                               // Diffuse  Color + Specular Color
aterial()->Shade(ray, hit, dirToLight, lcolor);
Segment(r2, 0, h2.getT());
```

Implemented by the function of another class

Illumination

ts the reflection of all indirect illumination.

otal hack!

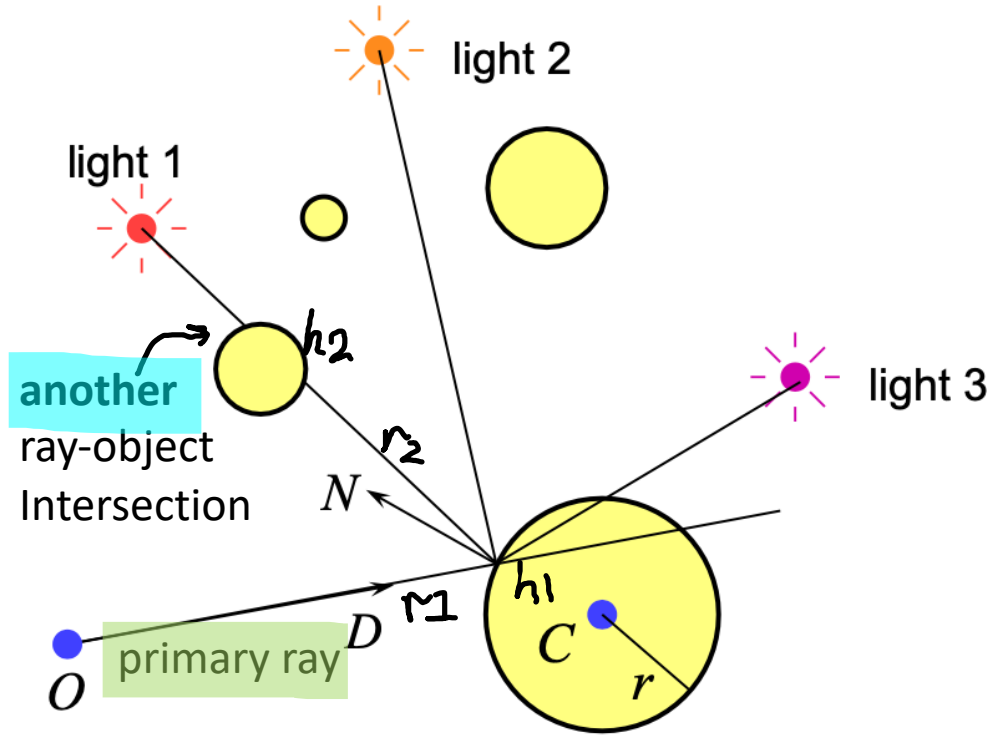
e complexity of global illumination.

```
+= am; // ambient color
```

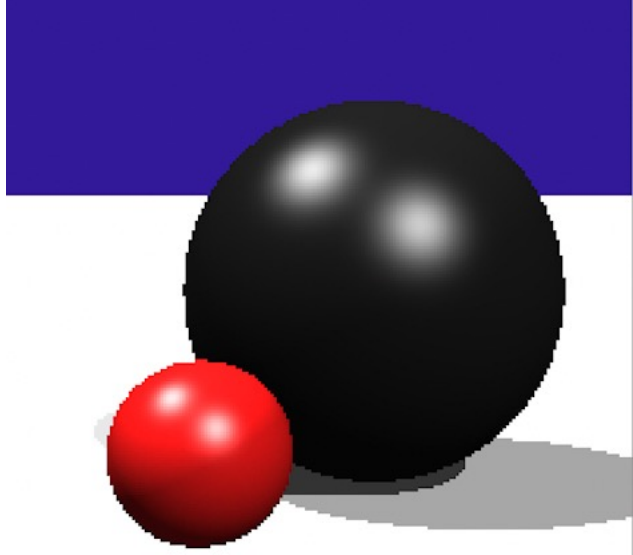
```
Background {
    color 0 0 1
    ambientLight 0.2 0.2 0.2
}
```

https://github.com/PeterHUistyping/MIT6.837-CG-Fall2004-Assignment/blob/main/MyProject/assignment4_Ray-Tracer/ray/material.h

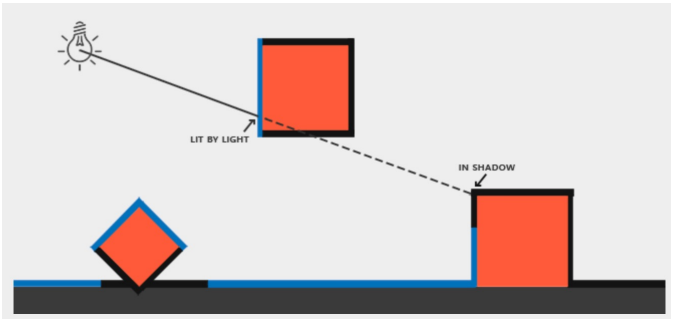
Ray tracing: shadows



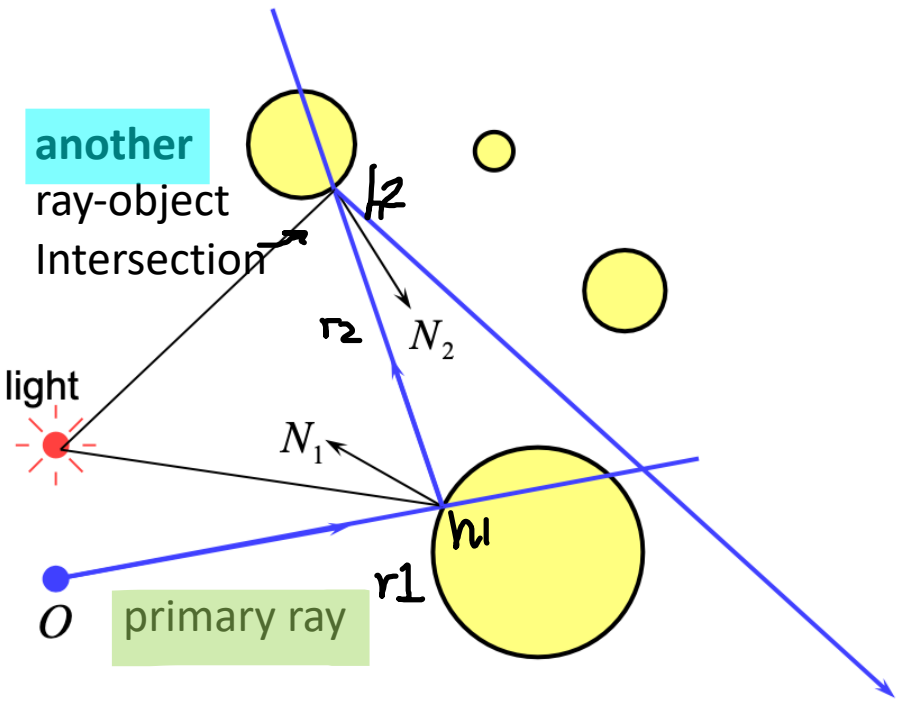
- because you are tracing rays from the intersection point to the light, you can check whether another object is between the intersection and the light and is hence casting a shadow
  - also need to watch for self-shadowing



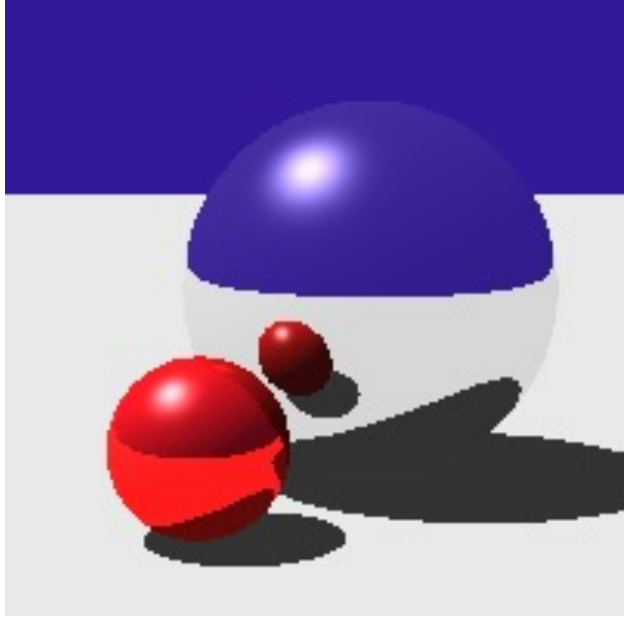
Ray tracing can get **shadows** by ray casting, otherwise shadow mapping. ->
(Depth buffer from light)
https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping



Ray tracing: reflection

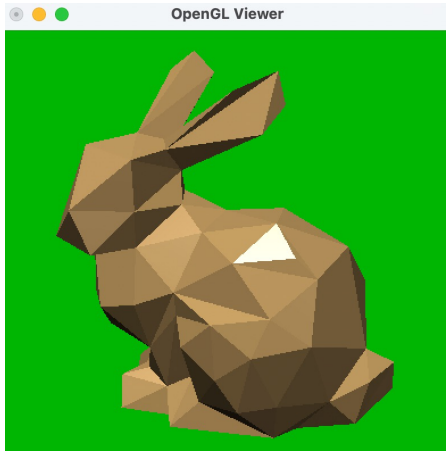


- if a surface is totally or partially reflective then new rays can be spawned to find the contribution to the pixel's colour given by the reflection
  - this is perfect (mirror) reflection

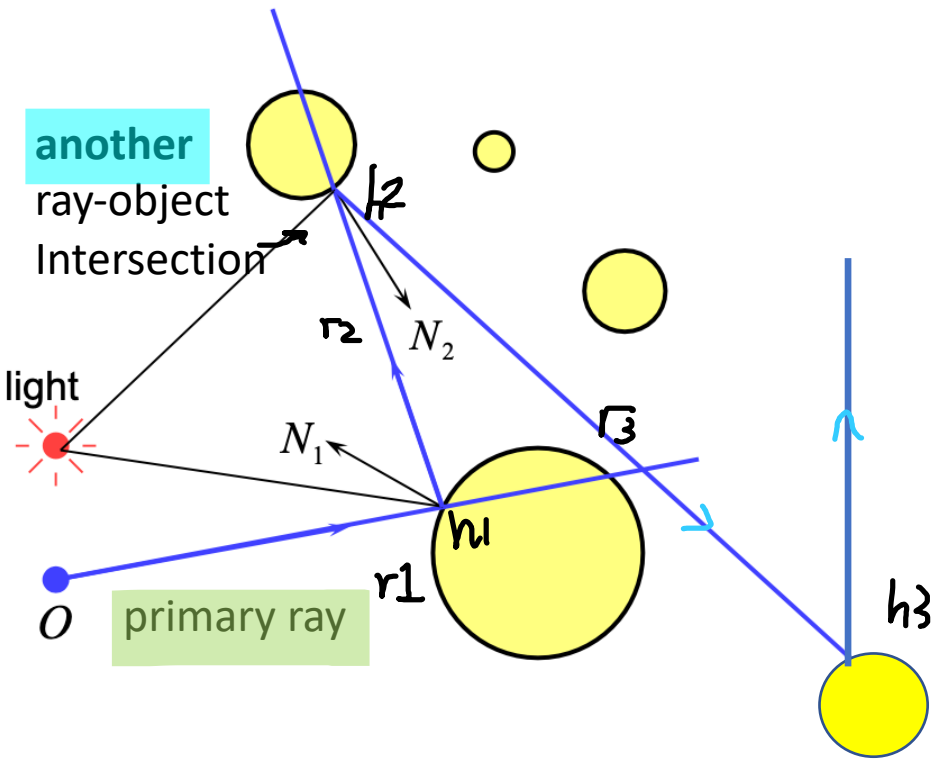


bounces = 1

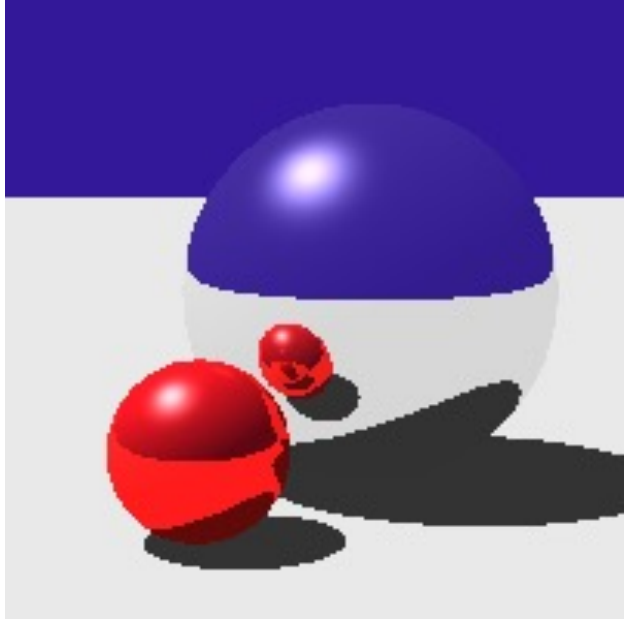
	Surface Examined	ID	Material of the Surface Examined	Reflection Coefficient [%]
Cloister	Portico flooring	a	Red ceramic tiles	12
	Terrace flooring	b	Red ceramic tiles	12
	Vertical walls of the portico	c	White painted lime plaster	82
	Yard facades	d	Orange painted lime plaster	29
	Columns and bases	e	Marble	71
	Yard surface	f	Cobblestone	28



Ray tracing: reflection



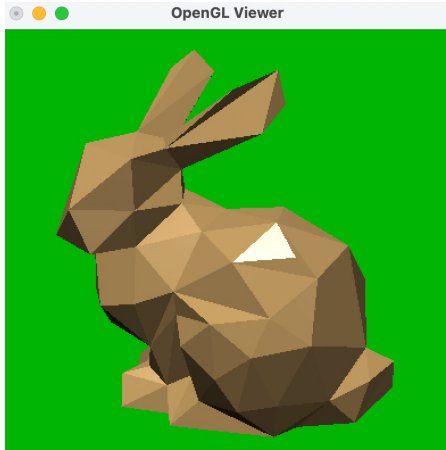
- if a surface is totally or partially reflective then new rays can be spawned to find the contribution to the pixel's colour given by the reflection
  - this is perfect (mirror) reflection



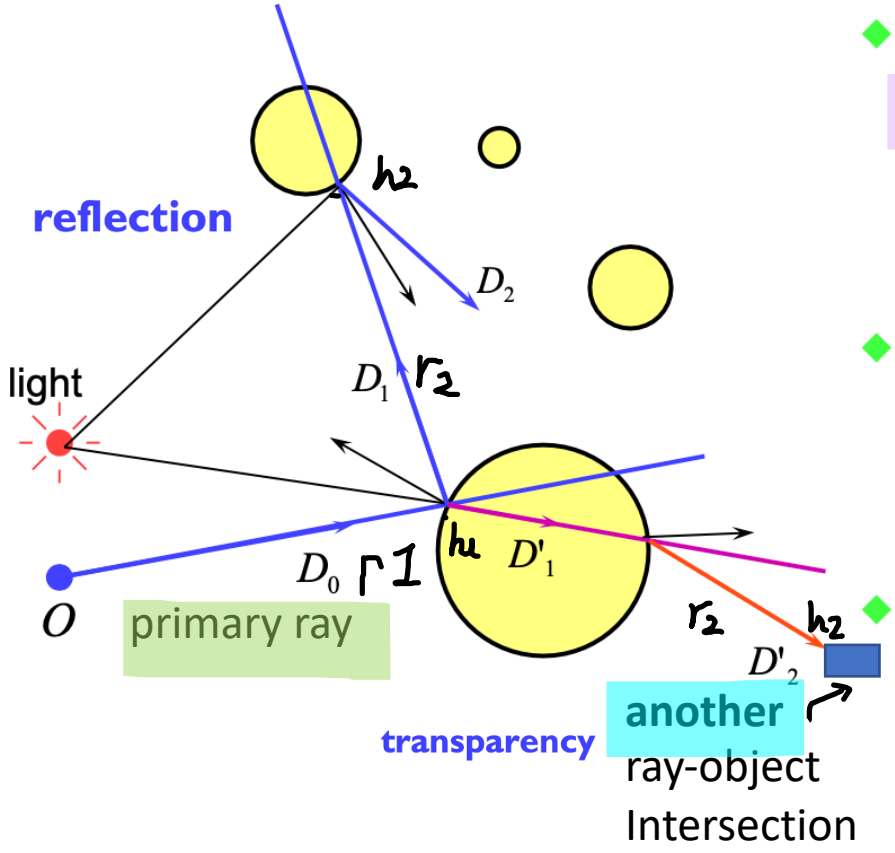
bounces = 2

and
can be set to infinite bounces☺

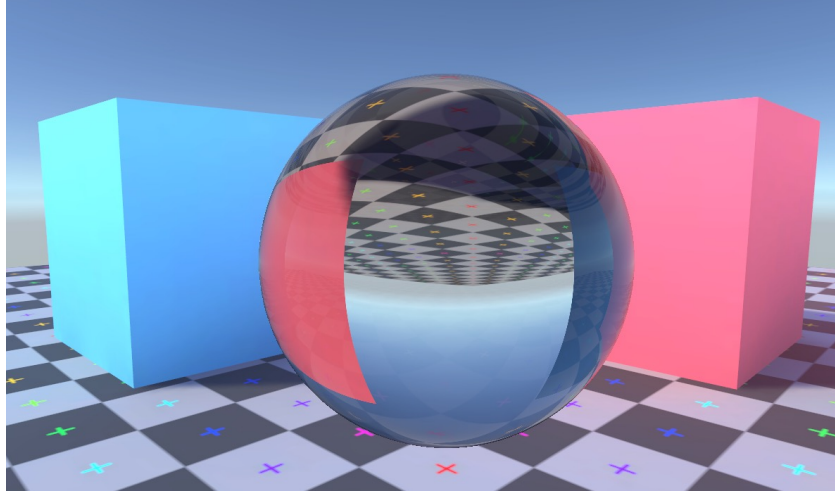
	Surface Examined	ID	Material of the Surface Examined	Reflection Coefficient [%]
Cloister	Portico flooring	a	Red ceramic tiles	12
	Terrace flooring	b	Red ceramic tiles	12
	Vertical walls of the portico	c	White painted lime plaster	82
	Yard facades	d	Orange painted lime plaster	29
	Columns and bases	e	Marble	71
	Yard surface	f	Cobblestone	28

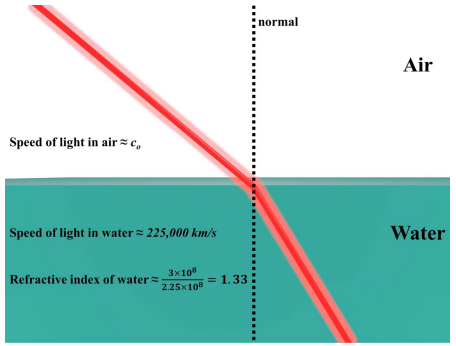


Ray tracing: transparency & refraction



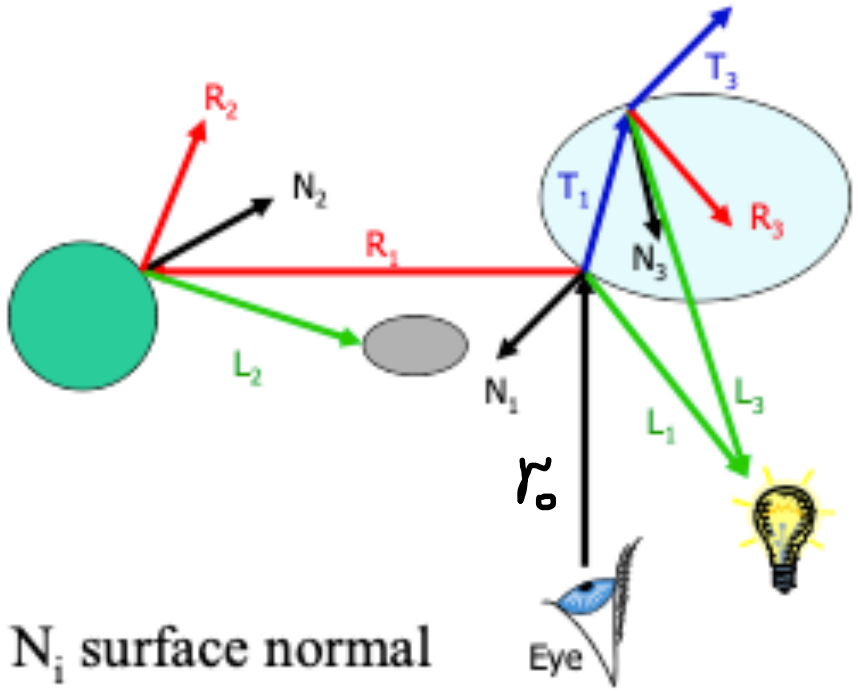
- objects can be totally or partially transparent
  - this allows objects behind the current one to be seen through it
- transparent objects can have refractive indices
  - bending the rays as they pass through the objects
- transparency + reflection means that a ray can split into two parts





Material	Index of Refraction (n)
Vacuum	1.000
Air	1.000277
Water	1.333333
Ice	1.31
Glass	About 1.5
Diamond	2.417

The Ray Tree

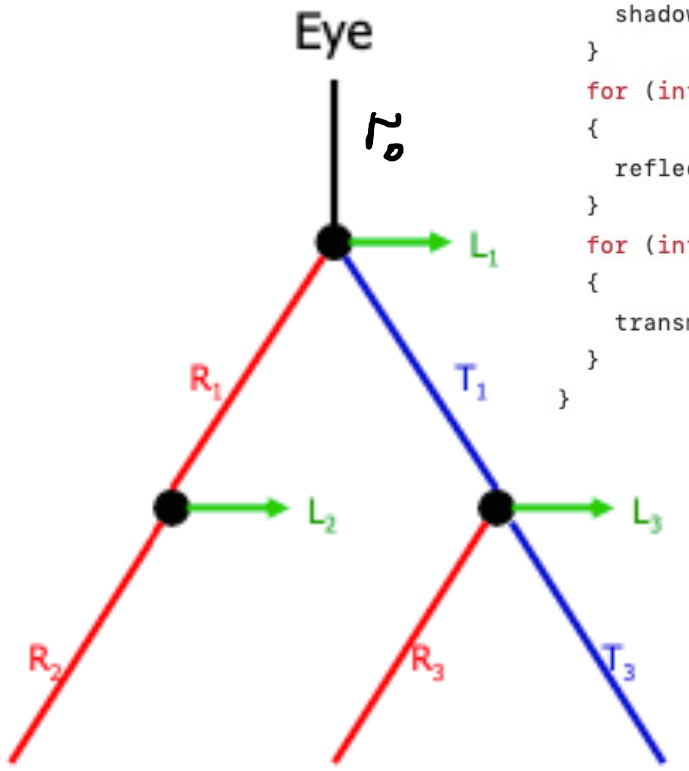


N_i surface normal

R_i reflected ray

L_i shadow ray

T_i transmitted (refracted) ray



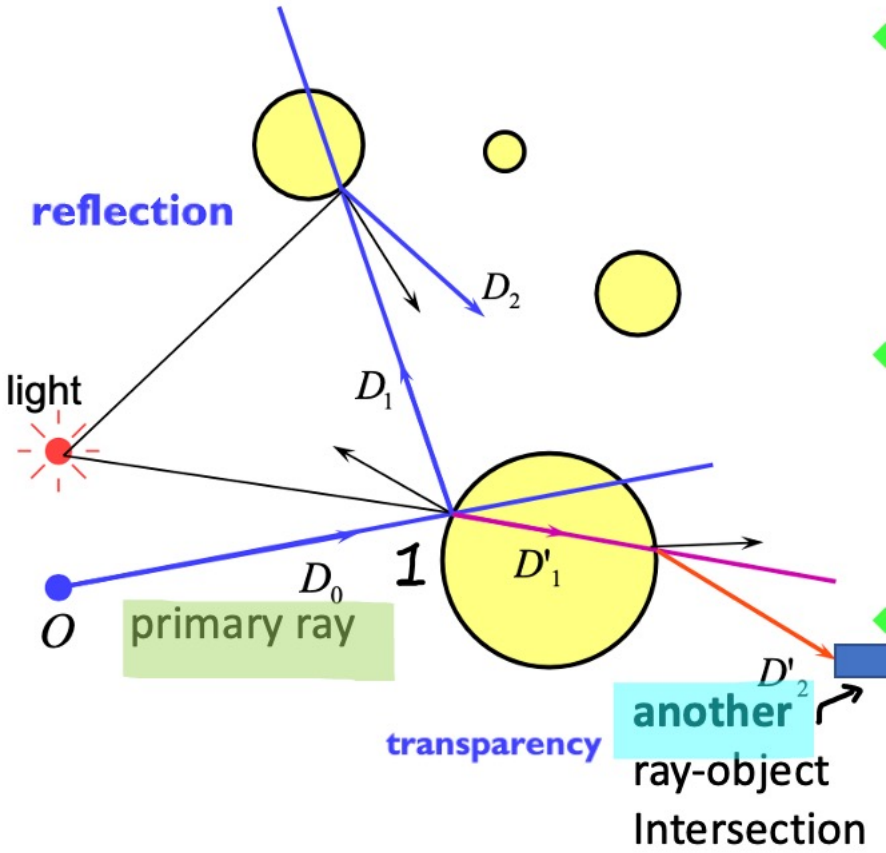
Complexity?

```
void RayTree::Print()
{
  main_segment.Print("main       ");
  int i;
  for (int i = 0; i < shadow_segments.getNumSegments(); i++)
  {
    shadow_segments.getSegment(i).Print("shadow     ");
  }
  for (int i = 0; i < reflected_segments.getNumSegments(); i++)
  {
    reflected_segments.getSegment(i).Print("reflected  ");
  }
  for (int i = 0; i < transmitted_segments.getNumSegments(); i++)
  {
    transmitted_segments.getSegment(i).Print("transmitted");
  }
}
```

https://github.com/PeterHUistyping/MIT6.837-CG-Fall2004-Assignment/blob/main/MyProject/assignment4_Ray-Tracer/ray/rayTree.C

Global illumination e.g. shadows, reflections and refractions

- Monte-Carlo Ray-tracing
  - Send **tons** of indirect rays



Local illumination

Simulate the light that comes directly from a light source (**direct illumination**)

How do we dealt with this?

– **Ambient term** to fake some uniform indirect light

(a single constant term, cheat!)

```
Vec3f am = parser->getAmbientLight();
am = am * hit.getMaterial()->getDiffuseColor();
pcolor += am; // ambient color
```

Global illumination

Simulate all light inter-reflections (**indirect lighting**)

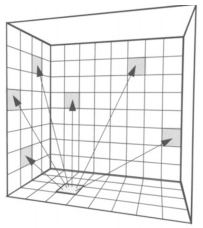
subsequent cases in which light rays from the same source are reflected by other surfaces in the scene, whether reflective or not.

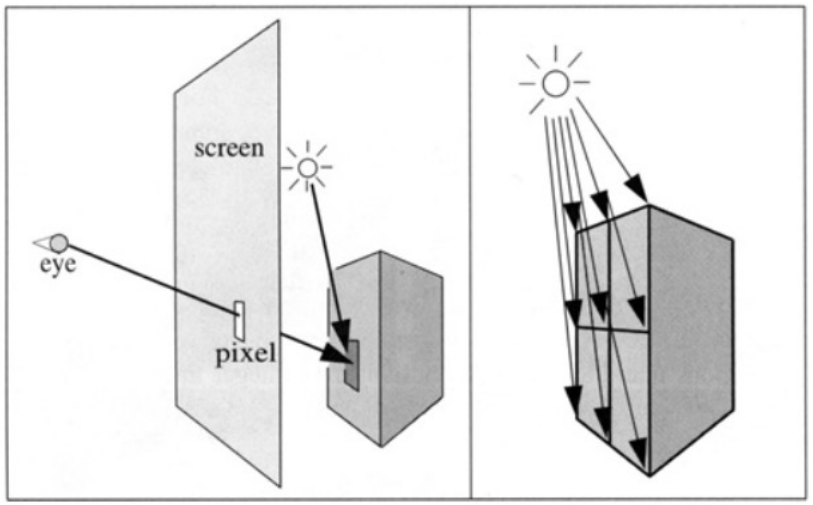
– e.g. in a room, a lot of the light is indirect: it is reflected by walls.

Alternative approaches for global illumination

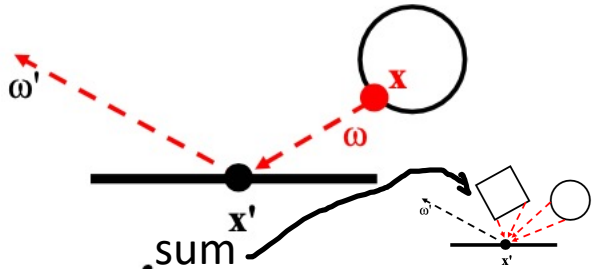
- Monte-Carlo Ray-tracing
  - Send **tons** of indirect rays
- Radiosity Bx: total rate of energy leaving a surface
  - View independent
  - Diffuse only

Or more https://en.wikipedia.org/wiki/Global_illumination





The Rendering Equation



G: Geometric relationship between the two surfaces ----- greatest transfer of light energy

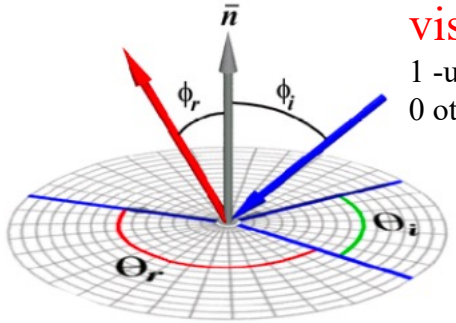
radiance

$$L(x',\omega') = E(x',\omega') + \int \rho_{x'}(\omega,\omega') L(x,\omega) G(x,x') V(x,x')\, dA$$

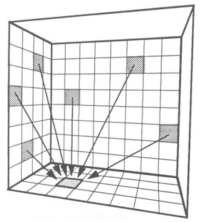
emitted radiance

(light source)

scale the contribution by $\rho_{x'}(\omega,\omega')$, the reflectivity (BRDF) of the surface at x'



visibility:
1 -unobstructed along ω
0 otherwise



Ray Tracing
image-space
based on camera position

Radiosity
object-space
view-independent
can pre compute complex lighting to allow interactive walkthroughs

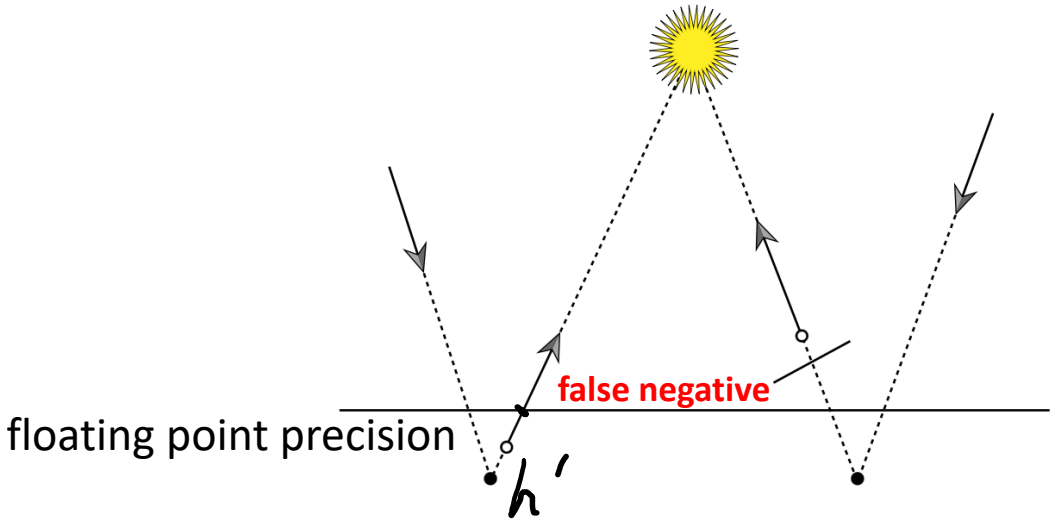
$$L(x',\omega') = E(x',\omega') + \int \rho_{x'}(\omega,\omega') L(x,\omega) G(x,x') V(x,x')\, dA$$

Radiosity assumption:
perfectly diffuse surfaces (not directional)

$$B_{x'} = E_{x'} + \rho_{x'} \int B_x\, G(x,x') V(x,x')$$

Avoid self-intersection using ε (secondary ray)

```
if (t > 0.0001f) ray.t = min( ray.t, t );
```



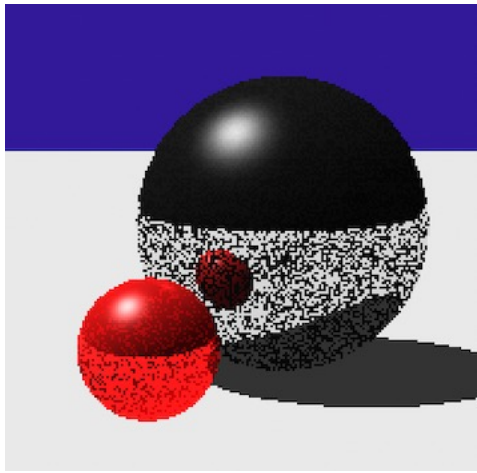
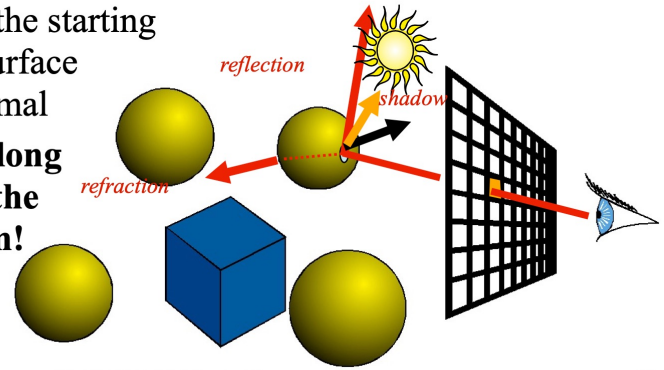


Figure 3.39: Geometric Settings for Rounding-Error Issues That Can Cause Visible Errors in Images. The incident ray on the left intersects the surface. On the left, the computed intersection point (black circle) is slightly below the surface and a too-low "epsilon" offsetting the origin of the shadow ray leads to an incorrect self-intersection, as the shadow ray origin (white circle) is still below the surface; thus the light is incorrectly determined to be occluded. On the right a too-high "epsilon" causes a valid intersection to be missed as the ray's origin is past the occluding surface.

https://www.pbr-book.org/3ed-2018/Shapes/Managing_Rounding_Error

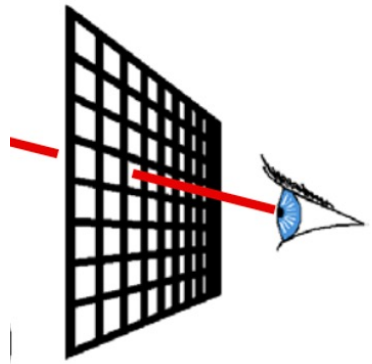
The evil ε

- In ray tracing, do NOT report intersection for rays starting at the surface
  - Secondary rays will start at the surfaces
  - Requires epsilons
  - Best to nudge the starting point off the surface e.g., along normal
  - **Best: nudge along the direction the ray came from!**




```
if (!parser->getGroup()->intersectShadowRay(r2, h2, epsilon_shadow)) // epsilon self-shadowing
```

Ray Tracing Algorithm



select an eye point and a screen plane

FOR every pixel in the screen plane:

determine the **ray** from the eye through the pixel's centre

FOR each object (sphere, cubes, etc) in the scene

IF the object is intersected by the ray

// Keep the intersection if it is closest

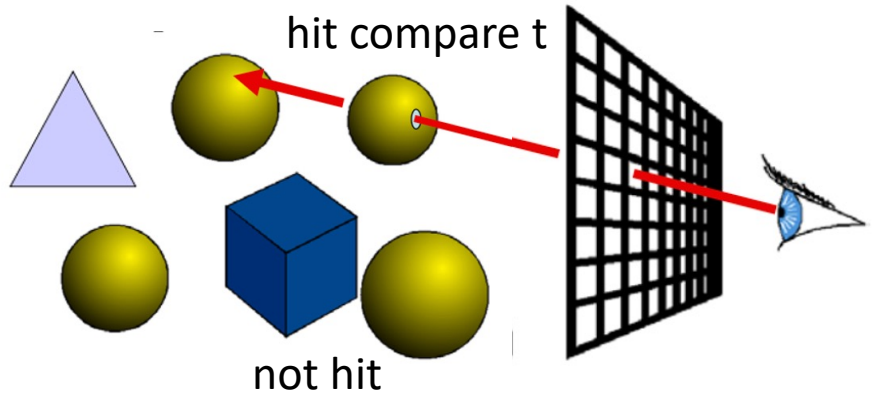
IF the intersection is the closest to the eye (so far)

record intersection point and object

// after all the iterations, we have found the closest intersection

set pixel's colour to that of the object at the closest intersection point

Ray Tracing Algorithm



select an eye point and a screen plane

FOR every pixel in the screen plane:

determine the **ray** from the eye through the pixel's centre

vec3color rayTracing (ray){

FOR each object (sphere, cubes, etc) in the scene
IF the object is intersected by the ray

// Keep the closest intersection

IF the intersection is the closest to the eye (so far)

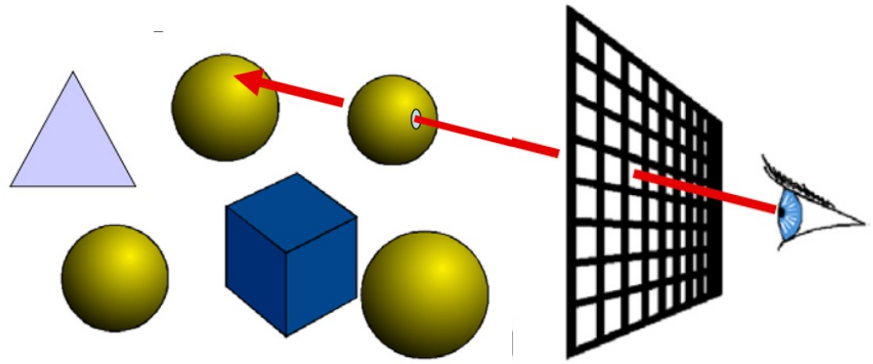
record intersection point and object

// After all the iterations, we have found the **closest intersection**

return **pixel's colour** to closest intersected object's color.

}

Ray Tracing Algorithm



select an eye point and a screen plane

FOR every pixel in the screen plane:

determine the **ray** from the eye through the pixel's centre

vec3color rayTracing (ray){

FOR each object (sphere, cubes, etc) in the scene

IF the object is intersected by the ray *primary ray*

// Keep the closest intersection

IF the intersection is the closest to the eye (so far)

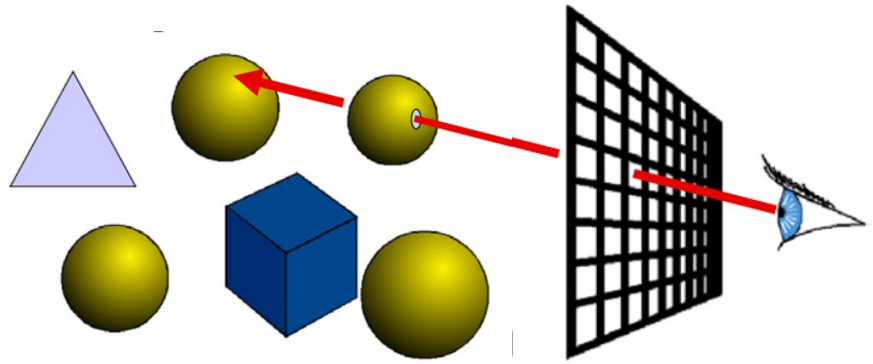
record intersection point and object

// After all the iterations, we have found the **closest intersection**

return **pixel's colour** to closest intersected object's color.

}

Ray Tracing Algorithm



select an eye point and a screen plane

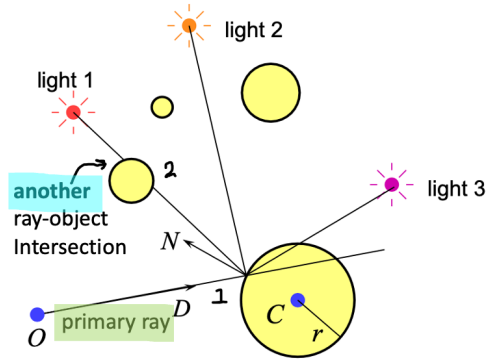
FOR every pixel in the screen plane:

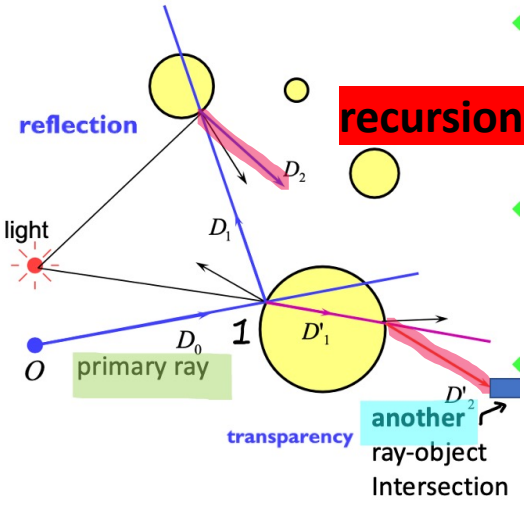
determine the **ray** from the eye through the pixel's centre

```
vec3color rayTracing ( ray ){
    Intersect all objects                          primary ray
    color = ambient term
    For every light
        cast shadow ray          secondary ray     IF the object is intersected by the ray
        color += local shading term
    If mirror
        color += color_refl *
        trace reflected ray                        secondary ray
    If transparent                                 IF the object is intersected by the ray
        color += color_trans *
        trace transmitted ray                      secondary ray
    return pixel's colour to closest intersected object's color.
}
```

Stopping criteria:

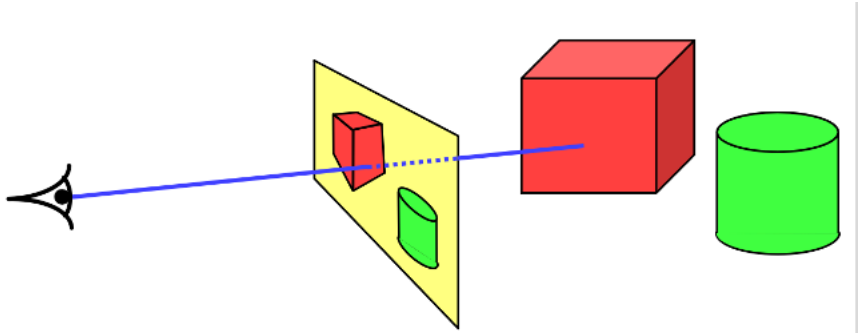
- Recursion depth
  - Stop after a number of bounces
- Ray contribution
  - Stop if reflected / transmitted contribution becomes too small



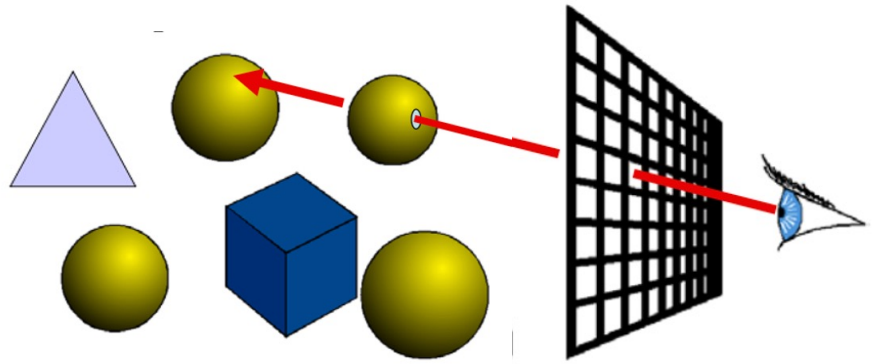


Ray Tracing Algorithm

Coding: Step by step explanation



Ray Tracing Algorithm



select an eye point and a screen plane

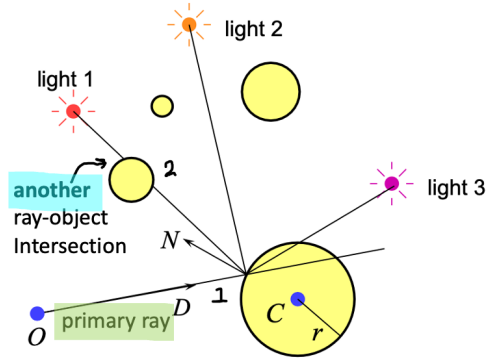
FOR every pixel in the screen plane:

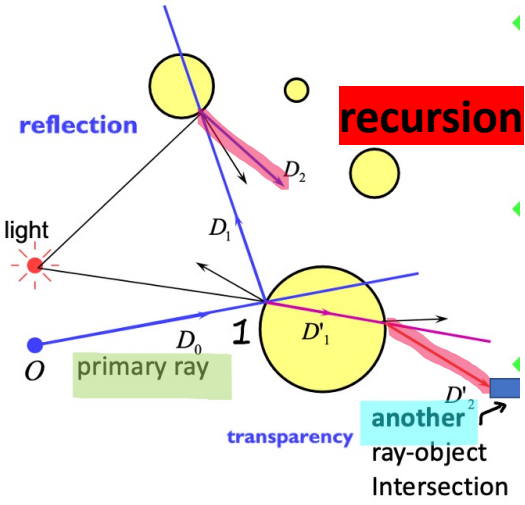
determine the **ray** from the eye through the pixel's centre

```
vec3color rayTracing ( ray ){
    Intersect all objects
    color = ambient term
    For every light
        cast shadow ray
        color += local shading term
    If mirror
        color += color_refl *
                 trace reflected ray
    If transparent
        color += color_trans *
                 trace transmitted ray
    return pixel's colour to closest intersected object's color.
}
```

Stopping criteria:

- Recursion depth
  - Stop after a number of bounces
- Ray contribution
  - Stop if reflected / transmitted contribution becomes too small





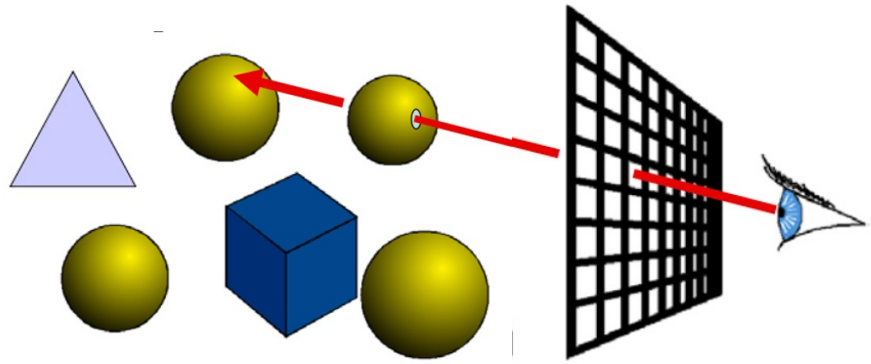
void rayTracing( )

```
{
    Image image(width, height); // image.SetAllPixels(parser->getBackgroundColor());
    Vec2f p;                    //(0,0)->(1,1) 1024*1024 Generate Points on the sceen coordinate
    float pixel = 1024;
    for (float i = 0; i < pixel; i++)
    { // load the camera
        for (float j = 0; j < pixel; j++)   // FOR every pixel (i, j) in the screen plane:
        {
            Vec3f pcolor;
            p.Set(i / pixel, j / pixel); // Ray - generateRay(p)  //~Camera
            Hit h;
            Ray ray = parser->getCamera()->generateRay(p);
            pcolor += ray_tracer->traceRay(ray, epsilon, 0, 1, 0, h);
            image.SetPixel(width * i / pixel, height * j / pixel, pcolor);
        }
    }
    image.SaveTGA(output_file);
}
```

continue...

https://github.com/PeterHUistyping/MIT6.837-CG-Fall2004-Assignment/blob/main/MyProject/assignment4_Ray-Tracer/main.C

Ray Tracing Algorithm



select an eye point and a screen plane

FOR every pixel in the screen plane:

determine the **ray** from the eye through the pixel's centre

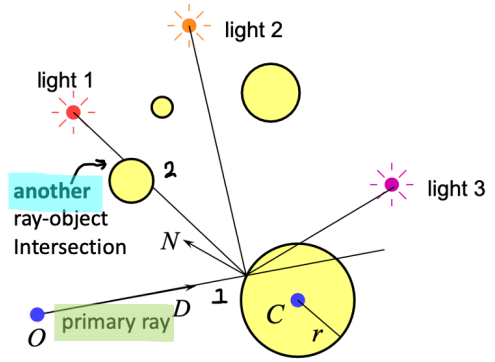
```
vec3color rayTracing ( ray ){
    Intersect all objects
    color = ambient term
    For every light
        cast shadow ray
        color += local shading term
    If mirror
        color += color_refl *
                 trace reflected ray
    If transparent
        color += color_trans *
                 trace transmitted ray
    return pixel's colour to closest intersected object's color.
}
```

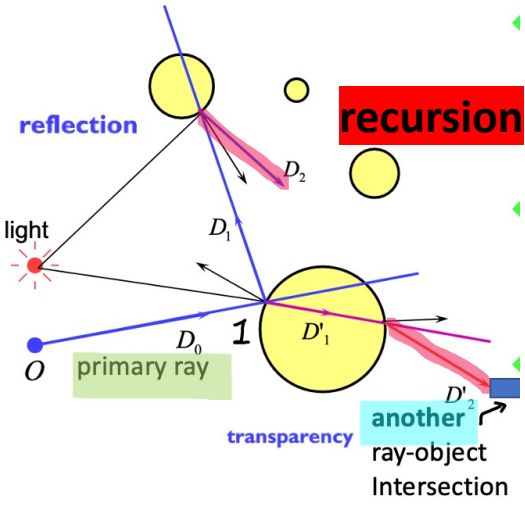
primary ray

secondary ray

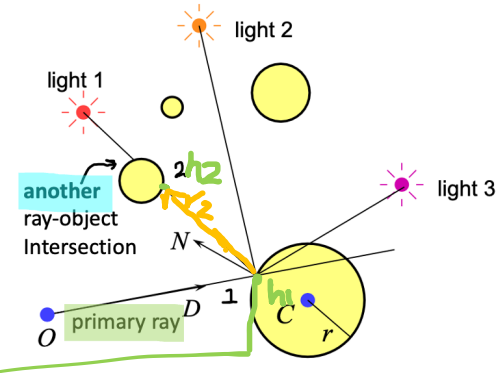
Stopping criteria:

- Recursion depth
  - Stop after a number of bounces
- Ray contribution
  - Stop if reflected / transmitted contribution becomes too small





```
Vec3f traceRay(Ray &ray, float tmin, int bounces, float weight, float indexOfRefraction, Hit &hit) const
{
    Vec3f pcolor;                                          // point_color
    if (bounces > max_bounces || weight <= cutoff_weight) // end of recursion
        return pcolor;
    RayTracer tracer_temp;
    if (parser->getGroup()->intersect(ray, hit, epsilon_shadow)) // parser->getCamera()->getTMin()
    {
        Vec3f am = parser->getAmbientLight();
        am = am * hit.getMaterial()->getDiffuseColor();
        pcolor += am; // ambient color                                    Shading
        indexOfRefraction = hit.getMaterial()->getIndexOfRefraction();
        RayTree::SetMainSegment(ray, 0, hit.getT());
        int lnum = parser->getNumLights();
        for (int l = 0; l < lnum; l++) // for every light
        {
            Vec3f dirToLight, lcolor, normal_temp;
            float distanceToLight = 0;
            parser->getLight(l)->getIllumination(hit.getIntersectionPoint(), dirToLight, lcolor, distanceToLight);
            if (shadows)
            {
                Ray r2(hit.getIntersectionPoint(), dirToLight); // Shadow
                Hit h2(distanceToLight, NULL, normal_temp);
                if (!parser->getGroup()->intersectShadowRay(r2, h2, epsilon_shadow)) // epsilon self-shadowing
                {                                                                    // if no other object is in the way, do shading
                    pcolor += hit.getMaterial()->Shade(ray, hit, dirToLight, lcolor); // Diffuse  Color + Specular Color
                    RayTree::AddShadowSegment(r2, 0, h2.getT());
                }
            }
```

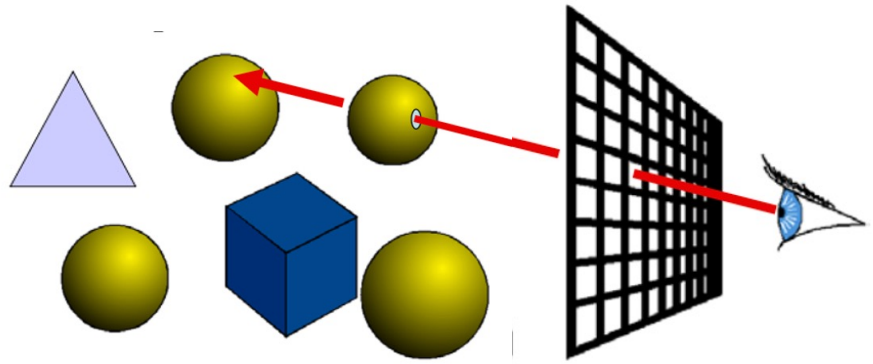


Implemented by the function of another class

continue...

https://github.com/PeterHUistyping/MIT6.837-CG-Fall2004-Assignment/blob/main/MyProject/assignment4_Ray-Tracer/ray/raytracer.h

Ray Tracing Algorithm



select an eye point and a screen plane

FOR every pixel in the screen plane:

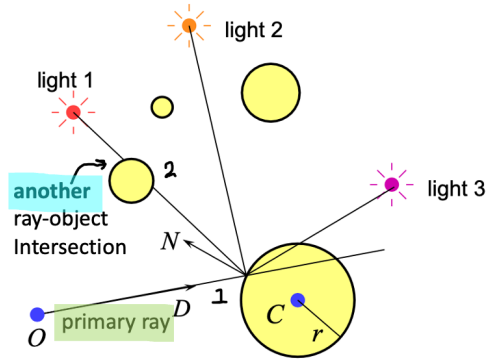
determine the **ray** from the eye through the pixel's centre

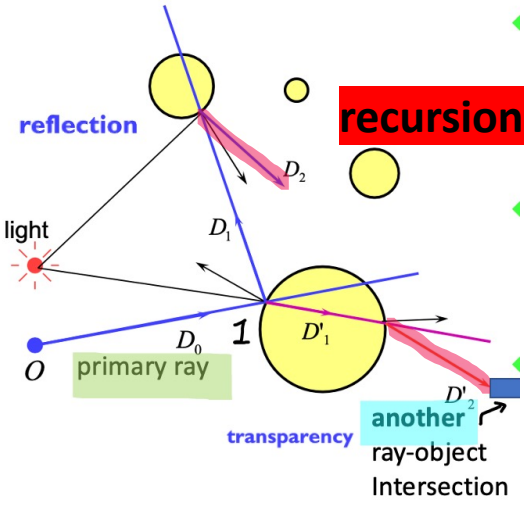
```
vec3color rayTracing ( ray ){
   Intersect all objects
   color = ambient term
   For every light
      cast shadow ray
      color += local shading term
   If mirror
      color += color_refl *
               trace reflected ray
   If transparent
      color += color_trans *
               trace transmitted ray
   return pixel's colour to closest intersected object's color.
}
```

secondary ray

Stopping criteria:

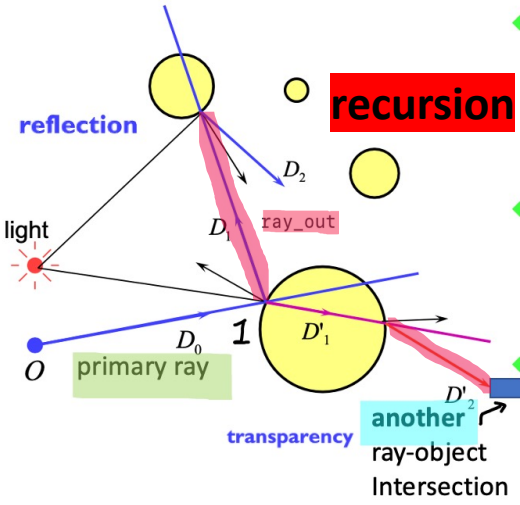
- Recursion depth
  - Stop after a number of bounces
- Ray contribution
  - Stop if reflected / transmitted contribution becomes too small





```
Vec3f traceRay(Ray &ray, float tmin, int bounces, float weight, float indexOfRefraction, Hit &hit) const
{

    if (hit.getMaterial()->getReflectiveColor().Length() != 0 && bounces < max_bounces)
    { // reflection
        Vec3f normal = hit.getNormal();
        if (shadeback)
        {
            if (hit.getNormal().Dot3(ray.getDirection()) > 0)
                normal.Negate(); // back shading
        }
        assert(fabs(normal.Length() - 1) < 1e-6);
        assert(fabs(ray.getDirection().Length() - 1) < 1e-6);
        Vec3f out = tracer_temp.mirrorDirection(normal, ray.getDirection());
        Ray ray_out(hit.getIntersectionPoint(), out);
        Hit h;
        pcolor += traceRay(ray_out, epsilon, bounces + 1, weight, indexOfRefraction, h) * hit.getMaterial()->getReflectiveColor();
        RayTree::AddReflectedSegment(ray_out, 0, h.getT());
    }
```

vec3color



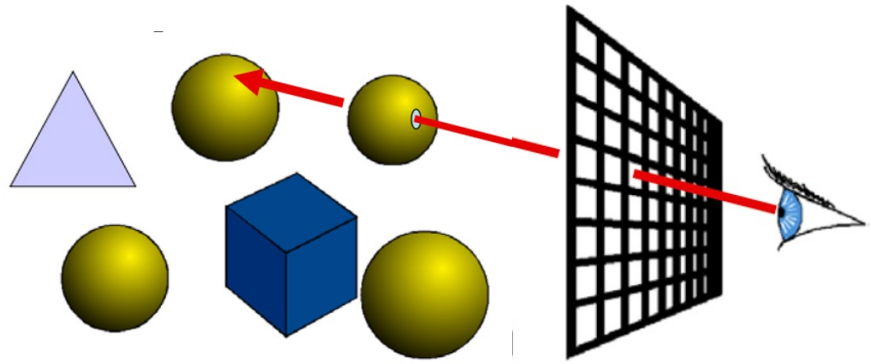
$color_{refl}$

Surface Examined		ID	Material of the Surface Examined	Reflection Coefficient [%]
Cloister	Portico flooring	a	Red ceramic tiles	12
	Terrace flooring	b	Red ceramic tiles	12
	Vertical walls of the portico	c	White painted lime plaster	82
	Yard facades	d	Orange painted lime plaster	29
	Columns and bases	e	Marble	71
	Yard surface	f	Cobblestone	28

continue...

https://github.com/PeterHUistyping/MIT6.837-CG-Fall2004-Assignment/blob/main/MyProject/assignment4_Ray-Tracer/ray/raytracer.h

Ray Tracing Algorithm



select an eye point and a screen plane

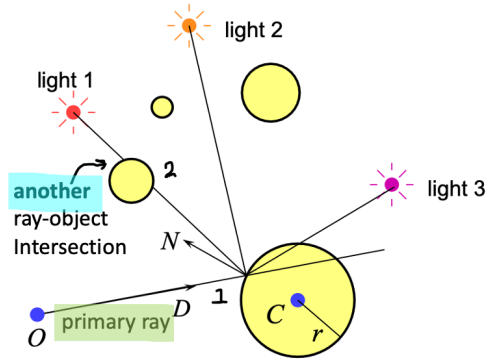
FOR every pixel in the screen plane:

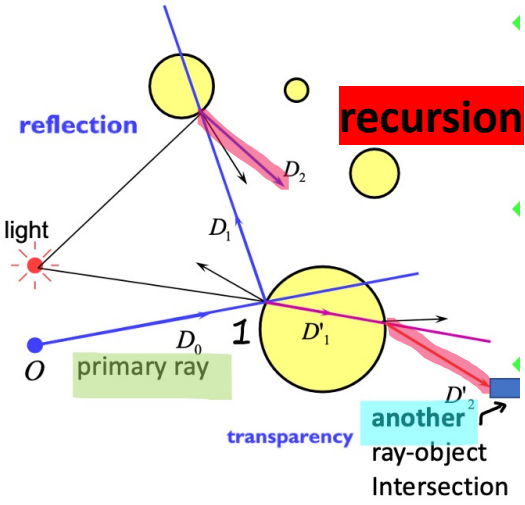
determine the **ray** from the eye through the pixel's centre

```
vec3color rayTracing ( ray ){
   Intersect all objects
   color = ambient term
   For every light
      cast shadow ray
      color += local shading term
   If mirror
      color += color_refl *
               trace reflected ray
   If transparent
      color += color_trans *
               trace transmitted ray   secondary ray
   return & set pixel's colour to closest intersected object's color.
}
```

Stopping criteria:

- Recursion depth
  - Stop after a number of bounces
- Ray contribution
  - Stop if reflected / transmitted contribution becomes too small



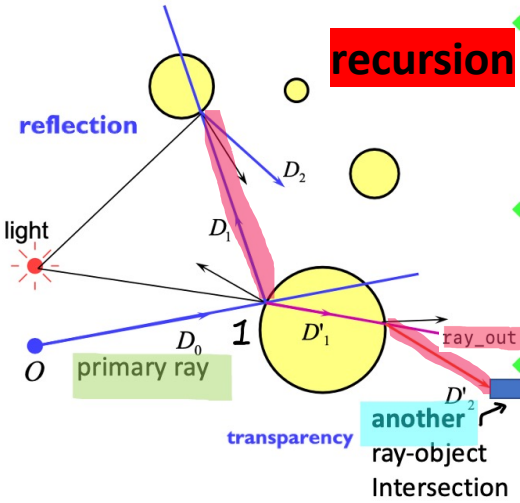


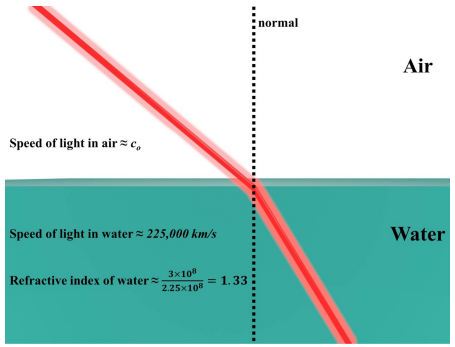
```
Vec3f traceRay(Ray &ray, float tmin, int bounces, float weight, float indexOfRefraction, Hit &hit) const
{

  if (hit.getMaterial()->getTransparentColor().Length() != 0 && bounces < max_bounces)
  { // refraction
      Vec3f transmitted;
      Vec3f normal = hit.getNormal();
      assert(fabs(hit.getNormal().Length() - 1) < 1e-6);
      assert(fabs(ray.getDirection().Length() - 1) < 1e-6);
      if (hit.getNormal().Dot3(ray.getDirection()) < 0) // air -> object
      {
          if (shadeback)
          {
              if (hit.getNormal().Dot3(ray.getDirection()) > 0)
                  normal.Negate(); // back shading
          }
          if (tracer_temp.transmittedDirection(normal, ray.getDirection(), 1, indexOfRefraction, transmitted))
          {
              Ray ray_out(hit.getIntersectionPoint(), transmitted);
              Hit h;
              Vec3f RefractionColor;
              RefractionColor = traceRay(ray_out, epsilon, bounces + 1, weight, indexOfRefraction, h) * hit.getMaterial()->getTransparentColor();
              pcolor += RefractionColor;
              RayTree::AddTransmittedSegment(ray_out, 0, h.getT());
          }
      }

    return pcolor;
}
```

recursion

vec3color

$color_{trans}$



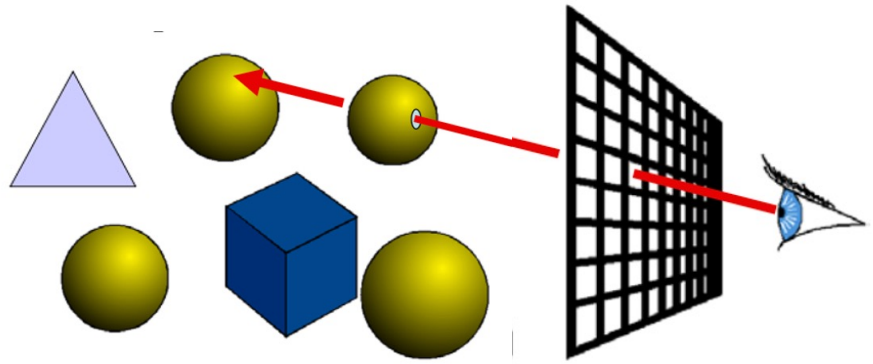


Material	Index of Refraction (n)
Vacuum	1.000
Air	1.000277
Water	1.333333
Ice	1.31
Glass	About 1.5
Diamond	2.417

End!

https://github.com/PeterHUistyping/MIT6.837-CG-Fall2004-Assignment/blob/main/MyProject/assignment4_Ray-Tracer/ray/raytracer.h

Ray Tracing Algorithm



select an eye point and a screen plane

FOR every pixel in the screen plane:

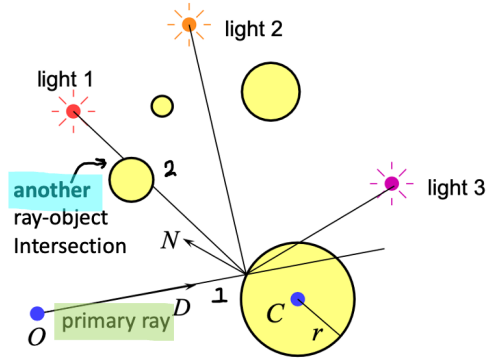
determine the **ray** from the eye through the pixel's centre

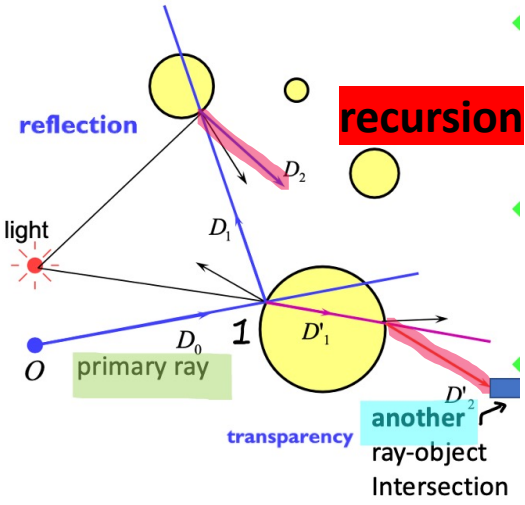
recursion of Ray Tracing Algo

```
vec3color rayTracing ( ray ){
    Intersect all objects                          primary ray
    color = ambient term
    For every light
        cast shadow ray        secondary ray   IF the object is intersected by the ray
        color += local shading term
    If mirror
        color += color_refl *
                 trace reflected ray            secondary ray
                                                IF the object is intersected by the ray
    If transparent
        color += color_trans *
                 trace transmitted ray          secondary ray
    return pixel's colour to closest intersected object's color.
}
```

Stopping criteria:

- Recursion depth
  - Stop after a number of bounces
- Ray contribution
  - Stop if reflected / transmitted contribution becomes too small

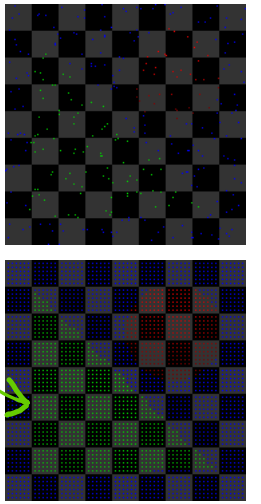


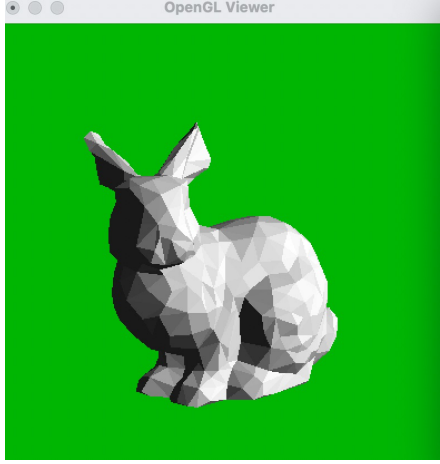


Ray Tracing Algorithm Analysis

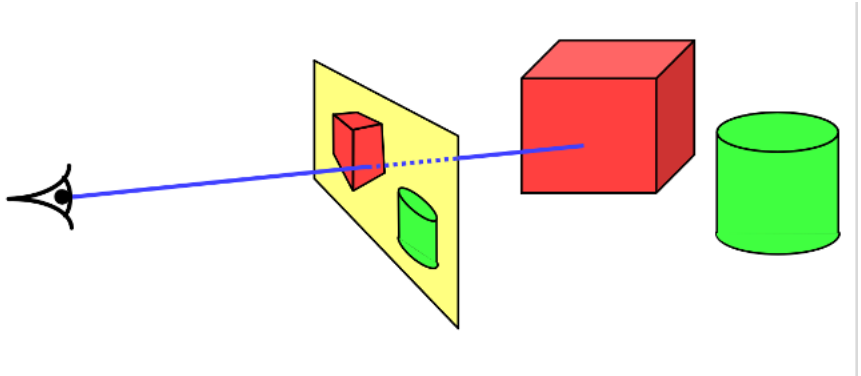
- Ray casting
- Lots of primitives
- Recursive
- Distributed Ray Tracing Effects
  - Soft shadows
  - Anti-aliasing
  - Glossy reflection
  - Motion blur
  - Depth of field

cost ≈ height * width *
num primitives *
intersection cost *
size of recursive ray tree *
num shadow rays *
num supersamples *
num glossy rays *
num temporal samples *
num focal samples *
. . .



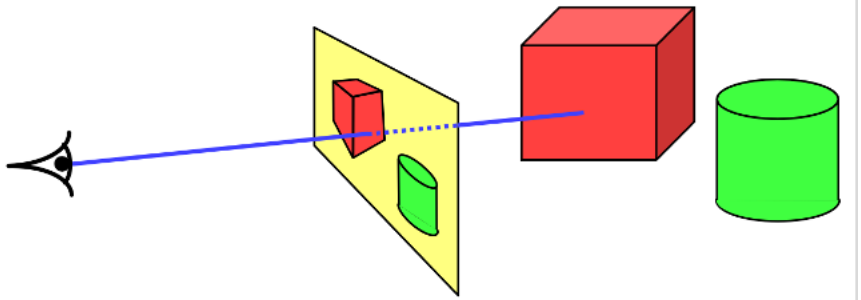


Q&A



Part 2: Ray-Object Intersection

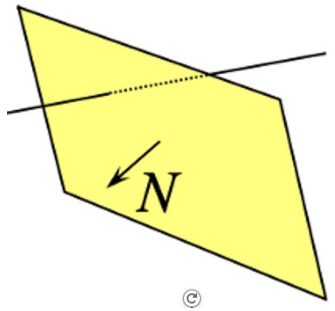
Plane, Sphere, Cube / Axis-Aligned Bounding Boxes(AABBs), Triangle



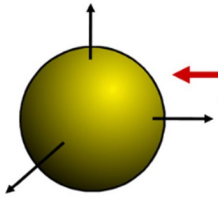
IF the object is intersected by the ray

Object

Object Representation

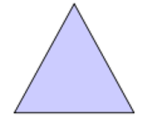


- Plane



- Sphere

- Cube / Axis-Aligned Bounding Boxes(AABBs)

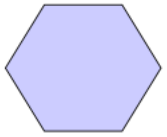


- Triangle (Frequently USED)

Others: MIT EECS 6.837 Computer Graphics (solid Math intro)
https://groups.csail.mit.edu/graphics/classes/6.837/F04/calendar.html

- Ray-Bunny Intersection & extra topics…

- Ray-Polygon Intersection



IF the object is intersected by the ray

Ray

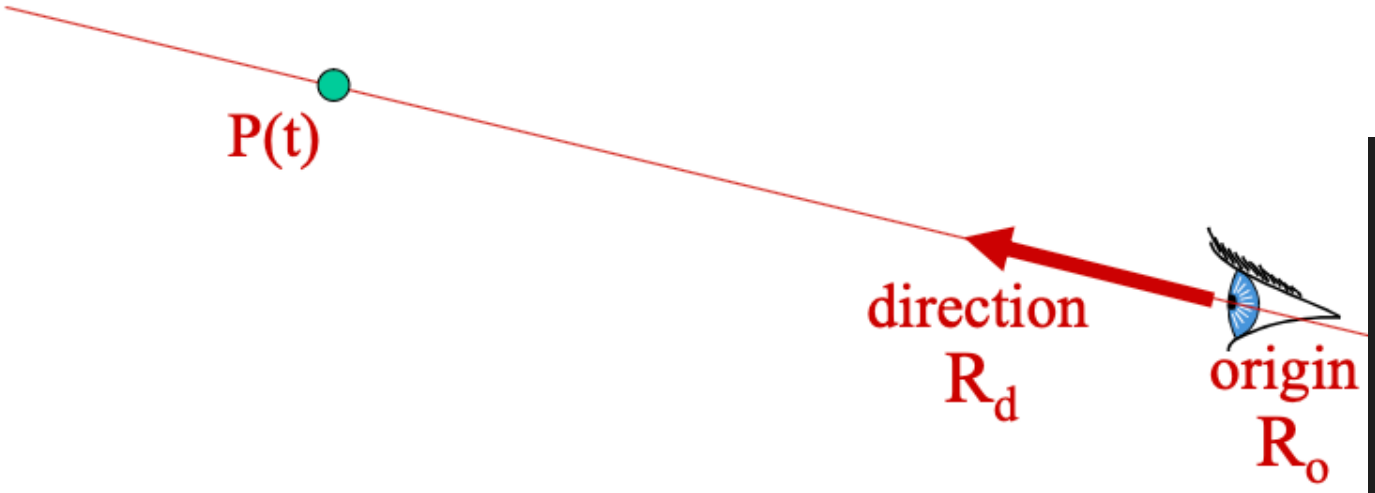
Recall: Ray Representation

- Parametric line
- $P(t) = R_o + t * R_d$
- Explicit representation

Ray - Object Intersection

intersection point - P(t)

Math problem of Solving t

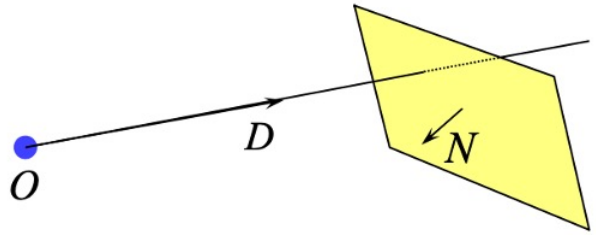


MIT EECS 6.837, Cutler and Durand

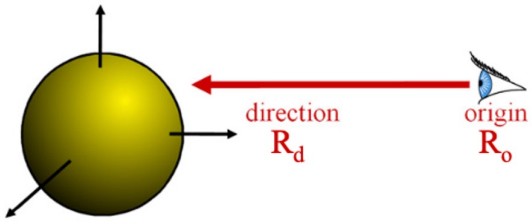
```
// Ray equation: The point P on the ray P(t) = o + t.D
class Ray{
public:
    // Origin of the ray : 3D coordinate
    const float o[3];
    // Direction of the ray: 3D vector
    const float D[3];
    // the closest intersection in the form of P(t) = o + t.D
    // SET to Infinity initially.
    const float t;
};
```

Object Representation

- Plane

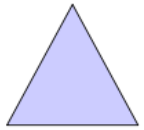


- Sphere



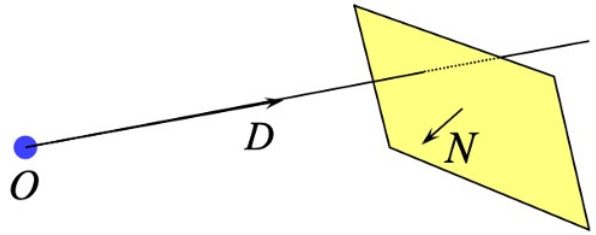
- Cube / Axis-Aligned Bounding Boxes(AABBs)
  - Ray-Box Intersection

- Triangle (Frequently USED)
  - Ray-Triangle Intersection

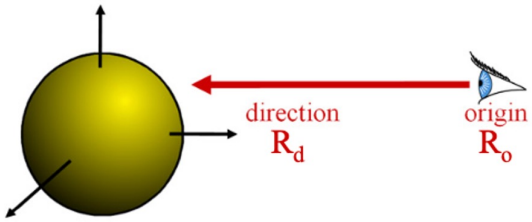


Plane Representation

- **Plane**

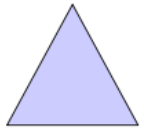


- Sphere



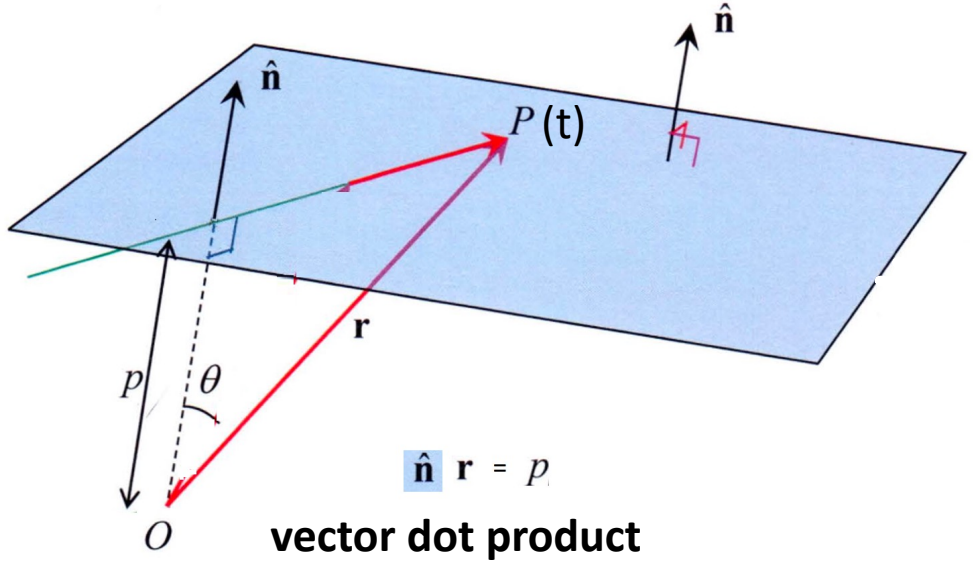
- Cube / Axis-Aligned Bounding Boxes(AABBs)
  - Ray-Box Intersection

- Triangle (Frequently USED)
  - Ray-Triangle Intersection



Plane Representation

Geometric meaning of the plane equation



vector dot product

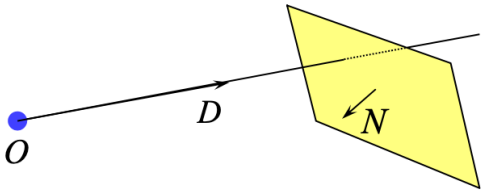
```
/* Plane equation:
The point P on the plane P(t) = n.P(t) + D =0
*/
class Plane{
public:
    // Normal of the plane
    const float n[3];
    // The second parameter of the plane
    const float D[3];
};
```

$$P(t) = R_o + t * R_d$$

$$H(P) = n \cdot P + D = 0$$

```
MaterialIndex 1
Plane {
    normal 0.01 1 0.01
    offset -1
}
```

Distance of origin to plane



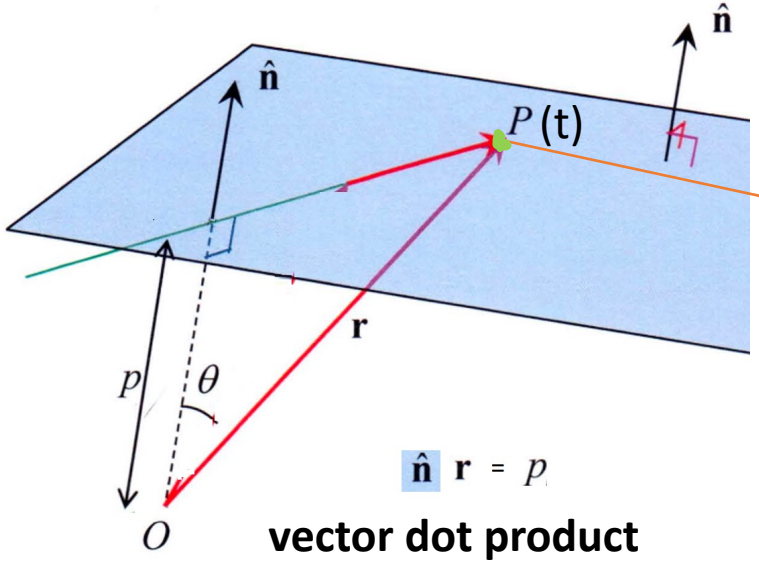
ray: $P = O + sD, s \geq 0$

plane: $P \cdot N + d = 0$

$$s = -\frac{d + N \cdot O}{N \cdot D}$$

Plane Representation

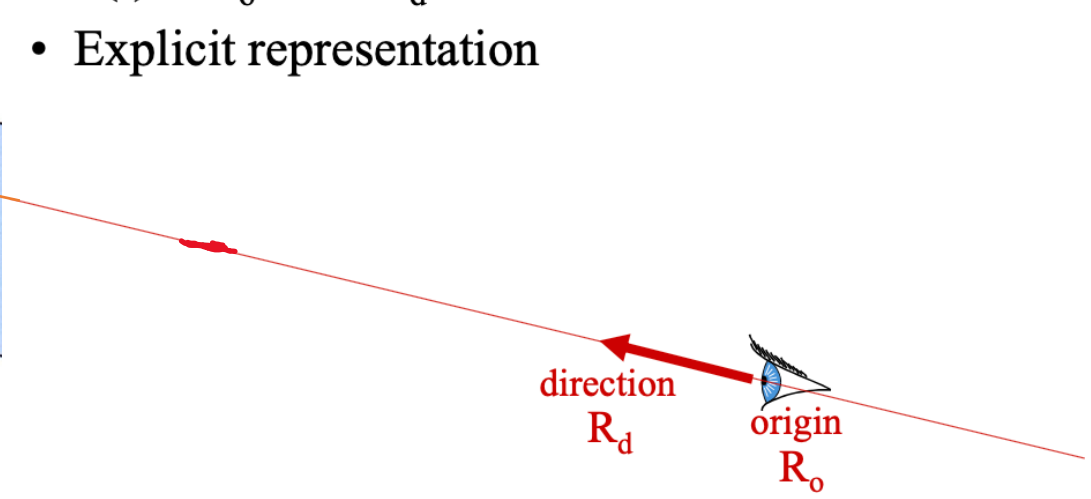
Geometric meaning of the plane equation



vector dot product

Recall: Ray Representation

- Parametric line
- $P(t) = R_o + t * R_d$
- Explicit representation



$$P(t) = R_o + t * R_d$$

$$H(P) = n \cdot P + D = 0$$

IF the object is intersected by the ray

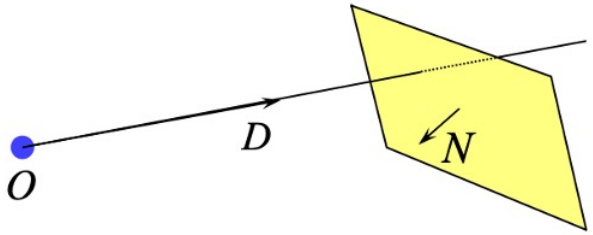
Ray - Object Intersection

Math problem of Solving t

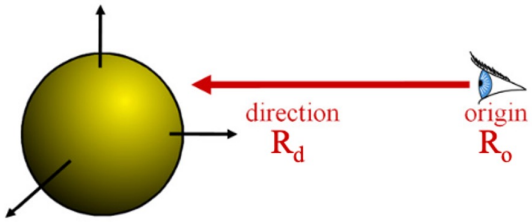
$$t = -(D + n \cdot R_o) / n \cdot R_d$$

Sphere Representation

- Plane

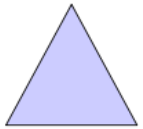


- **Sphere**



- Cube / Axis-Aligned Bounding Boxes(AABBs)
  - Ray-Box Intersection

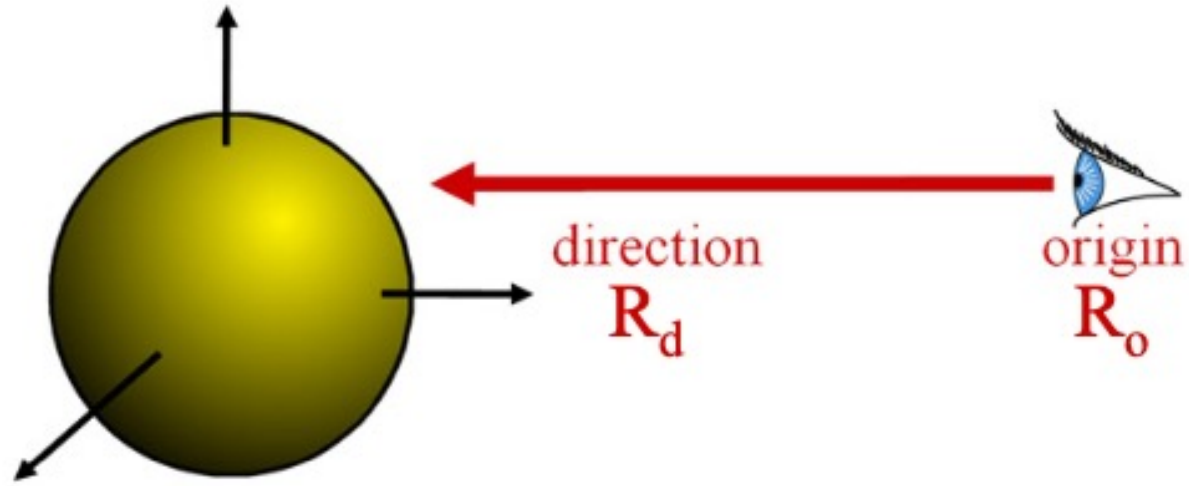
- Triangles (Frequently USED)
  - Ray-Triangle Intersection



Sphere Representation?

- Implicit sphere equation
  - Assume centered at origin (easy to translate)
  - $H(P) = P \cdot P - r^2 = 0$

```
MaterialIndex 0
Sphere {
    center 0 0 0
    radius 1
}
```




```
/* Sphere equation:
The point P on the sphere P(t).P(t) = r*r
*/
class Sphere{
public:
    // Radius of the sphere
    const float r;
};
```

$P(t) = R_o + t*R_d$ $\quad H(P) = P \cdot P - r^2 = 0$

$R_d \cdot R_d t^2 + 2R_d \cdot R_o t + R_o \cdot R_o - r^2 = 0$

Quadratic: $at^2 + bt + c = 0$

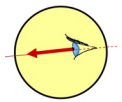
IF the object is intersected by the ray

Ray - Object Intersection

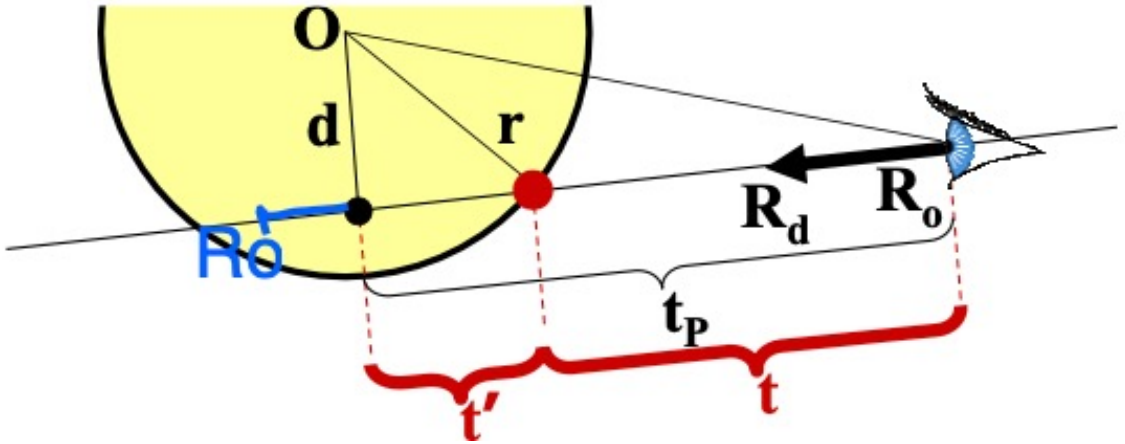
Math problem of Solving t

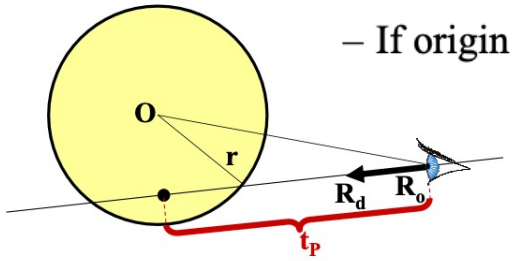
Sphere Representation?

Geometric Ray-Sphere Intersection

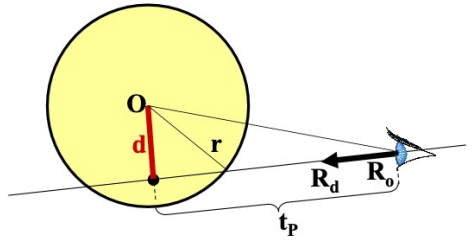


- Is ray origin **inside/outside/on** sphere?
- Find closest point to sphere center, $t_P = -R_o \cdot R_d$
- Find squared distance: $d^2 = R_o \cdot R_o - t_P^2$
- Find distance (t') from closest point (t_P) to correct intersection: $t'^2 = r^2 - d^2$
  - If origin outside sphere $\rightarrow t = t_P - t'$
  - If origin inside sphere $\rightarrow t = t_P + t'$





– If origin outside & $t_P < 0$ → **no hit**



– If $d^2 > r^2$ → **no hit**

$P(t) = R_o + t*R_d$ $\quad H(P) = P \cdot P - r^2 = 0$

$R_d \cdot R_d t^2 + 2R_d \cdot R_o t + R_o \cdot R_o - r^2 = 0$

Quadratic: $at^2 + bt + c = 0$

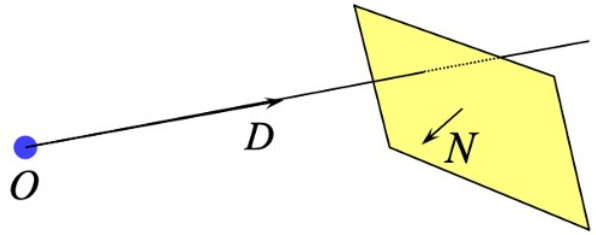
IF the object is intersected by the ray

Ray - Object Intersection

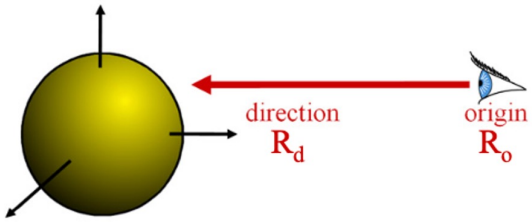
Math problem of Solving t

Cube Representation

- Plane

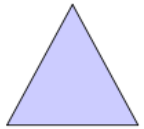


- Sphere



- **Cube** / Axis-Aligned Bounding Boxes(AABBs)
  - Ray-Box Intersection

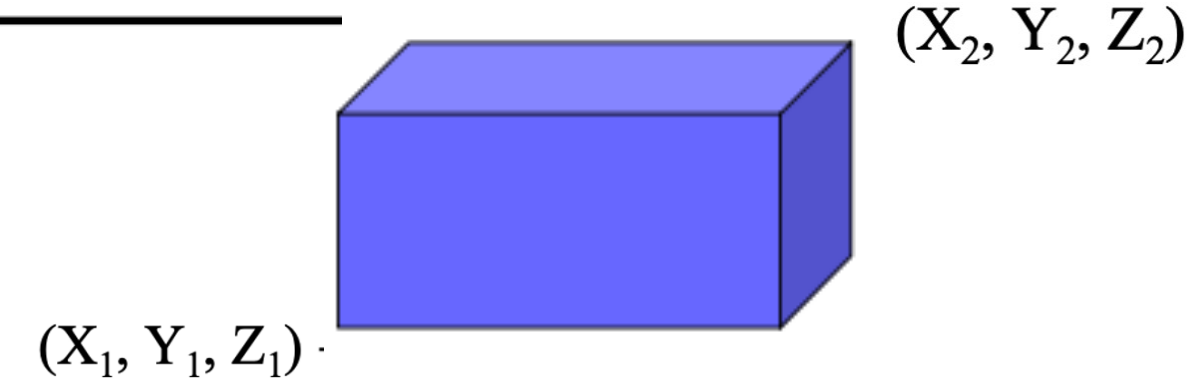
- Triangle (Frequently USED)
  - Ray-Triangle Intersection

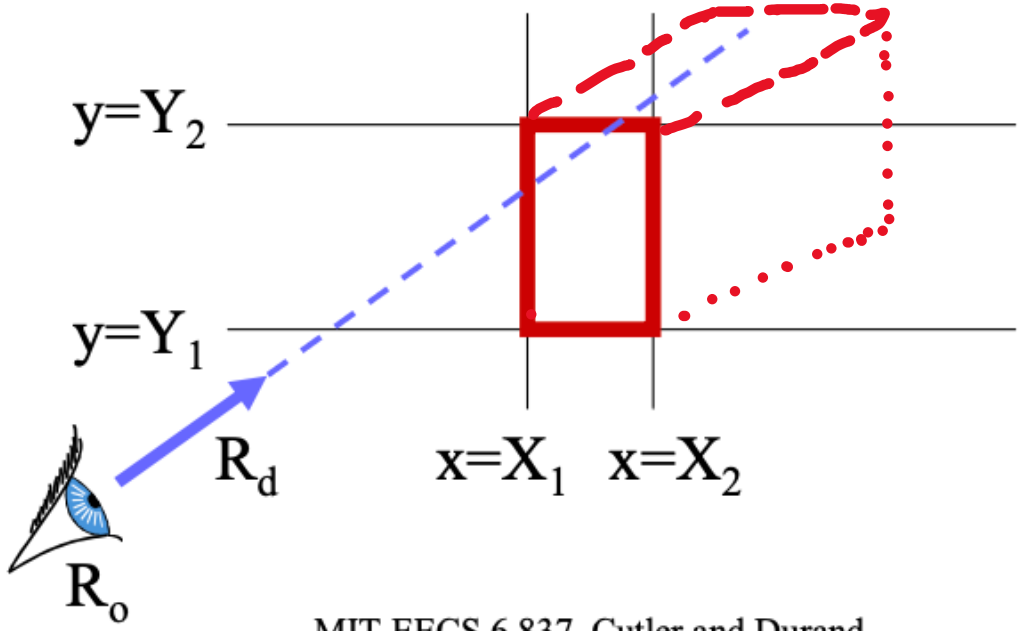


Ray-Box Intersection

Cube or Axis-Aligned Bounding Boxes (AABBs)

- Axis-aligned
- Box: $(X_1, Y_1, Z_1) \rightarrow (X_2, Y_2, Z_2)$
- Ray: $P(t) = R_o + tR_d$





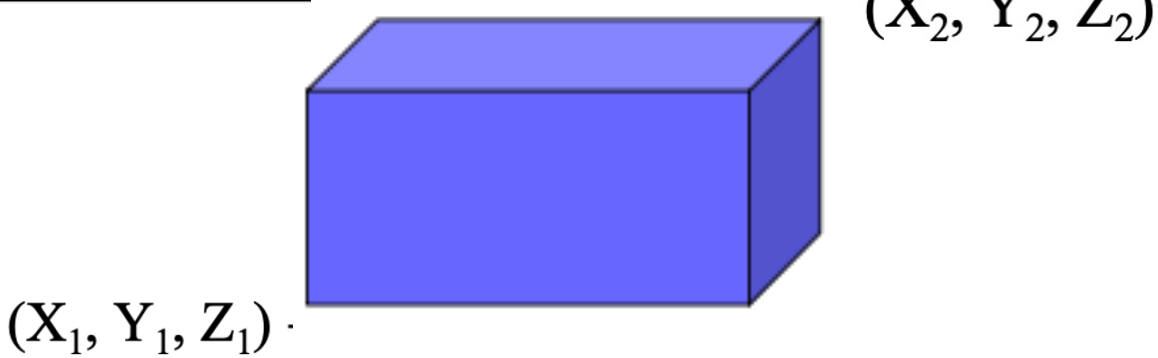
MIT EECS 6.837, Cutler and Durand

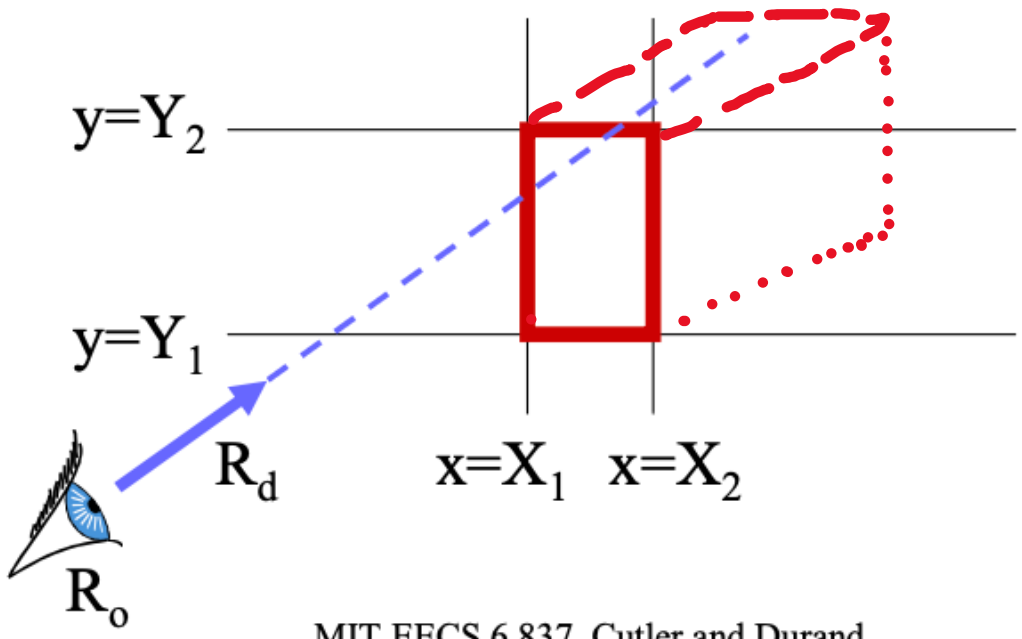
```
class Box{
public:
    // Buttom left point of the box : 3D coordinate
    const float bmin[3];
    // Top right point of the box: 3D coordinate
    const float bmax[3];
};
```

Ray-Box Intersection

Cube or Axis-Aligned Bounding Boxes (AABBs)

- Axis-aligned
- Box: $(X_1, Y_1, Z_1) \rightarrow (X_2, Y_2, Z_2)$
- Ray: $P(t) = R_o + tR_d$





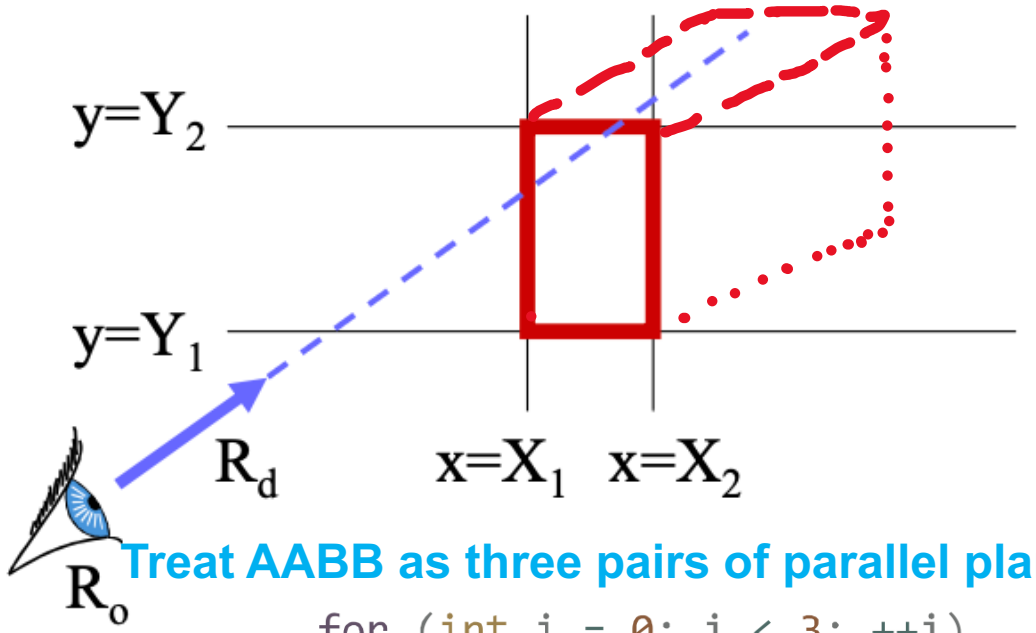
Naïve Ray-Box Intersection

- 6 plane equations: compute all intersections
- Return closest intersection inside the box
  - Verify intersections are on the correct side of each plane: $Ax+By+Cz+D < 0$

Ray-Box Intersection

Cube or Axis-Aligned Bounding Boxes (AABBs)

- Axis-aligned
- Box: $(X_1, Y_1, Z_1) \rightarrow (X_2, Y_2, Z_2)$
- Ray: $P(t) = R_o + tR_d$



Treat AABB as three pairs of parallel planes

```
for (int i = 0; i < 3; ++i)
```

Naïve Ray-Box Intersection

- 6 plane equations: compute all intersections
- Return closest intersection inside the box
  - Verify intersections are on the correct side of each plane: $Ax+By+Cz+D < 0$

Reducing Total Computation

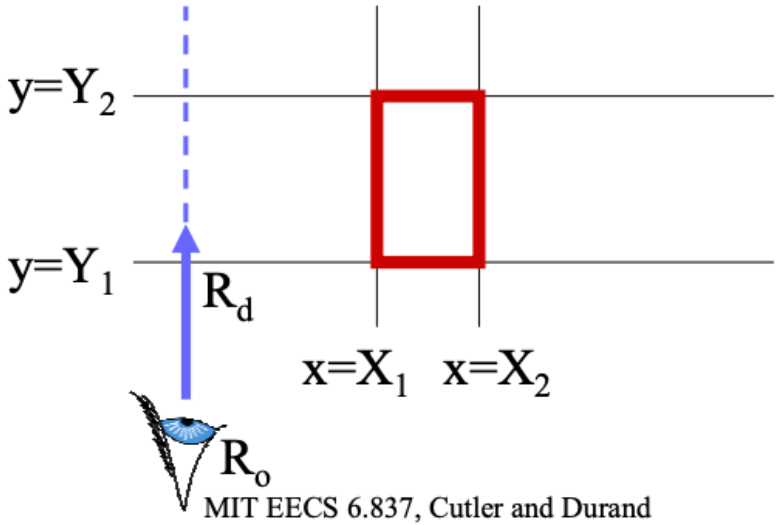
- Pairs of planes have the same normal
- Normals have only one non-0 component
- Do computations one dimension at a time

Ray-Box Intersection

Cube or Axis-Aligned Bounding Boxes (AABBs)

Test if Parallel

- If $R_{dx} = 0$ (ray is parallel) AND
  $R_{ox} < X_1$ or $R_{ox} > X_2$ → **no intersection**

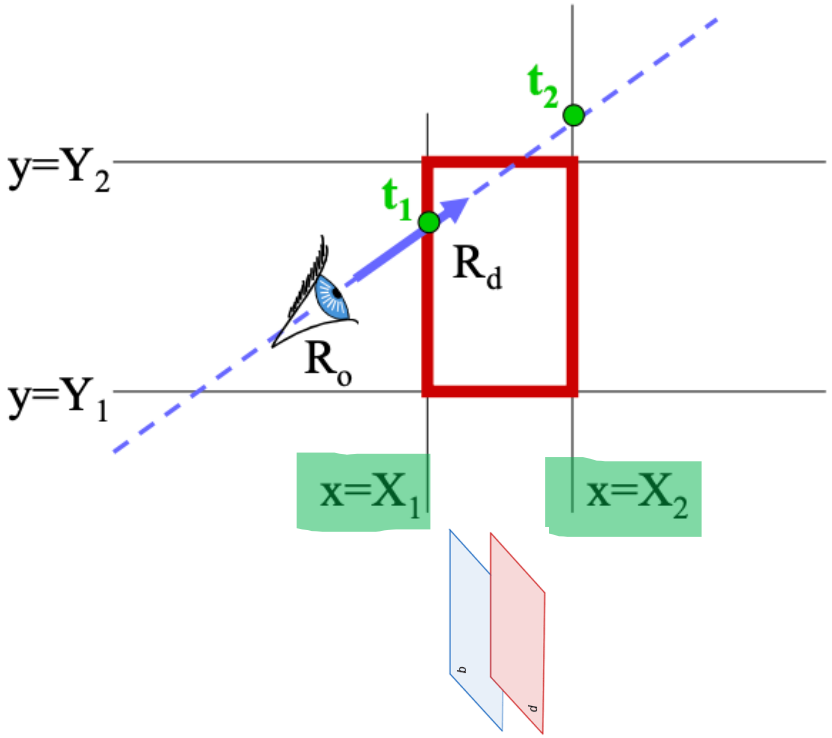


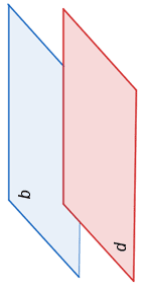
Ray-Box Intersection

Cube or Axis-Aligned Bounding Boxes (AABBs)

Find Intersections Per Dimension

- Calculate intersection distance t_1 and t_2
  - $t_1 = (X_1 - R_{ox}) / R_{dx}$
  - $t_2 = (X_2 - R_{ox}) / R_{dx}$





"first" intersection point
"exit" intersection point
of the parallel plane

```
float t1 = (b.bmin[i] - ray.o[i]) / ray.D[i];
float t2 = (b.bmax[i] - ray.o[i]) / ray.D[i];
```

local t in each dimension
t1, t2 for x axis
Similarly,
t3, t4 for y axis
t5, t6 for z axis

Ray - Object ***Intersection***

Math problem of Solving t

Cube or Axis-Aligned Bounding Boxes (AABBs)

IF the object is intersected by the ray

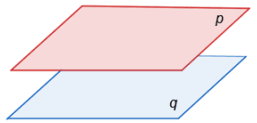
Maintain t_{near} & t_{far}

"first" intersection point
"exit" intersection point of the box

An intersection can only happen if the ray goes "into" the bounding region for all t values before it exits any of them.

- Closest & farthest intersections *on the object*
  - If $t_1 > t_{near}$, $t_{near} = t_1$
  - If $t_2 < t_{far}$, $t_{far} = t_2$

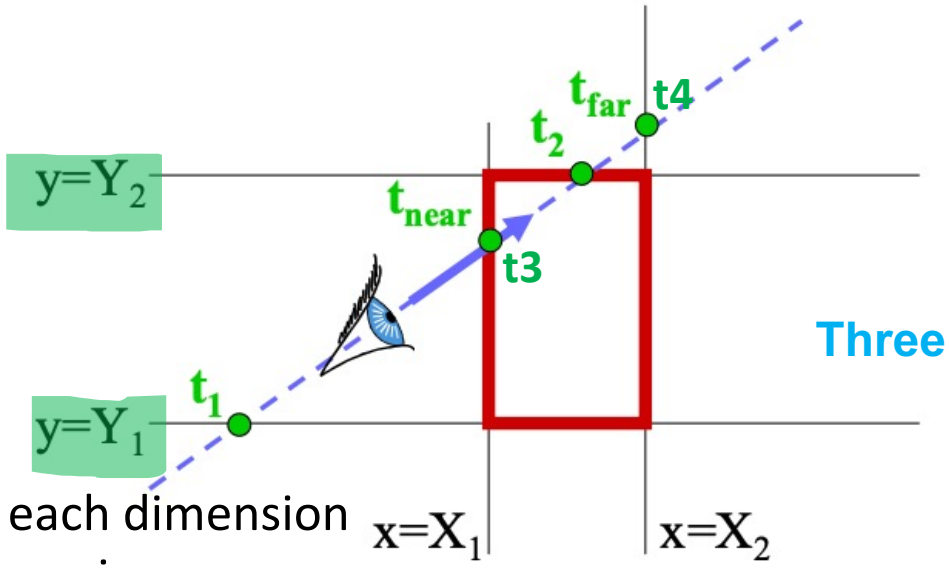
"first" intersection point
"exit" intersection point of the box



of each parallel plane

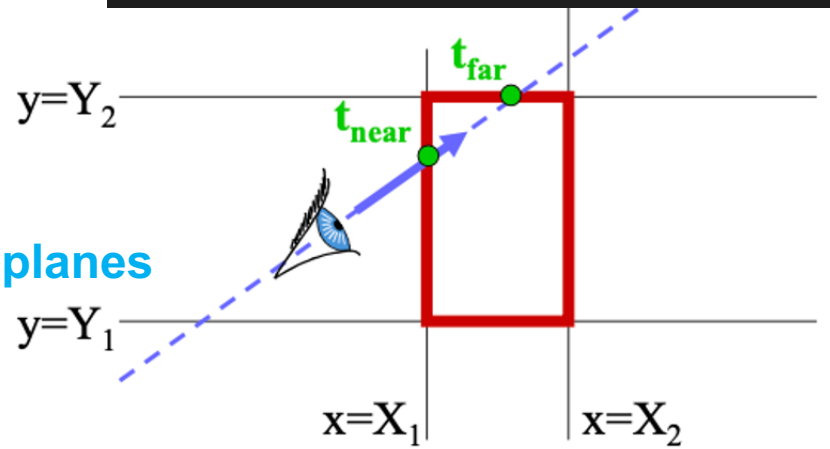
```
float t_near = max(t_near,min(t1,t2));
float t_far  = min(t_far, max(t1,t2));
```

max



3D to 1D

Three pairs of parallel planes
TO
the line of ray



local t in each dimension
t1, t2 for x axis
t3, t4 for y axis
t5, t6 for z axis

```
t_near = max(max(min(t1,t2),min(t3,t4)),min(t5,t6));
t_far  = min(min(max(t1,t2),max(t3,t4)),max(t5,t6));
```

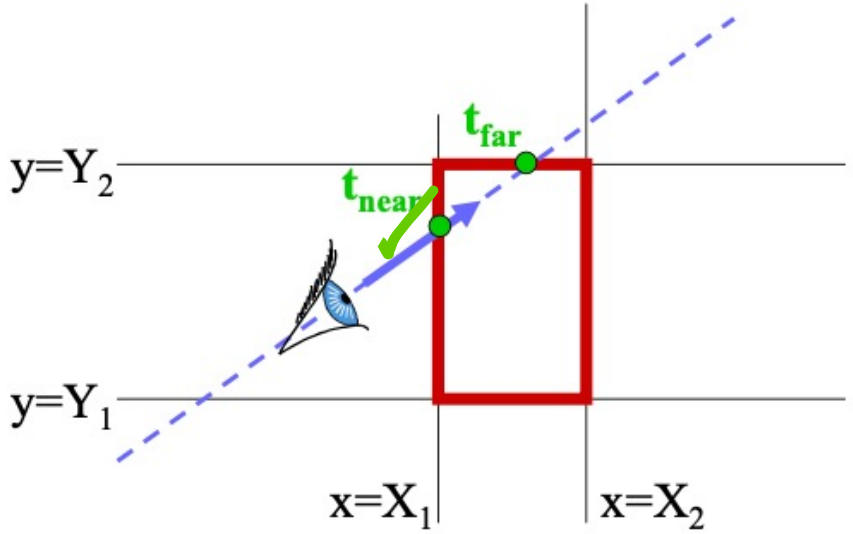
Ray-Box Intersection

Cube or Axis-Aligned Bounding Boxes (AABBs)

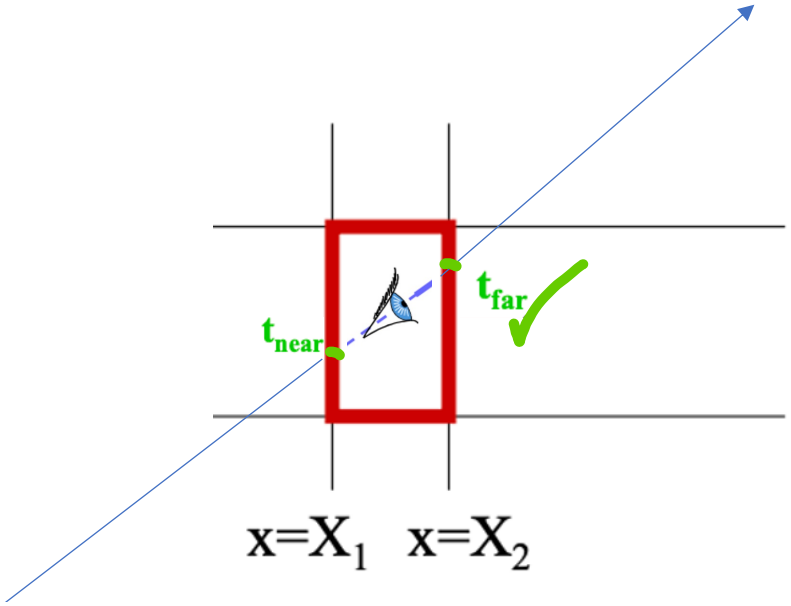
Return the Correct Intersection

- If $t_{near} > t_{min}$ → **closest intersection at t_{near}**
- Else → **closest intersection at t_{far}**

An intersection can only happen if the ray goes "into" the bounding region for all t values before it exits any of them.



closest intersection at t_{near}

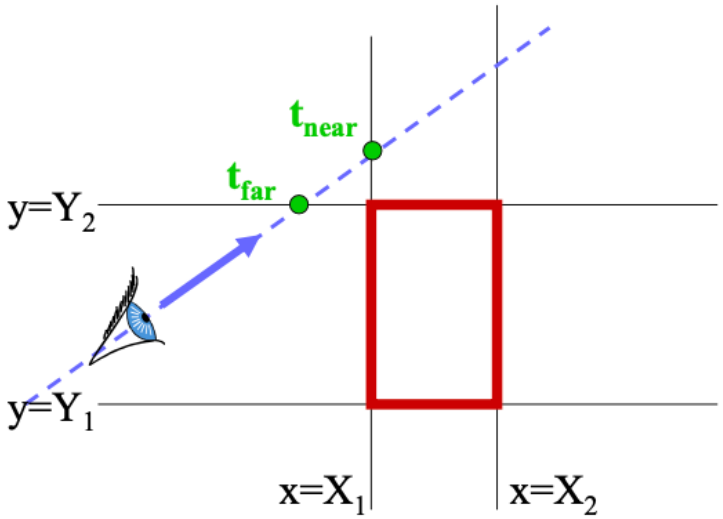


closest intersection at t_{far}

Cube or Axis-Aligned Bounding Boxes (AABBs)

Is there an Intersection?

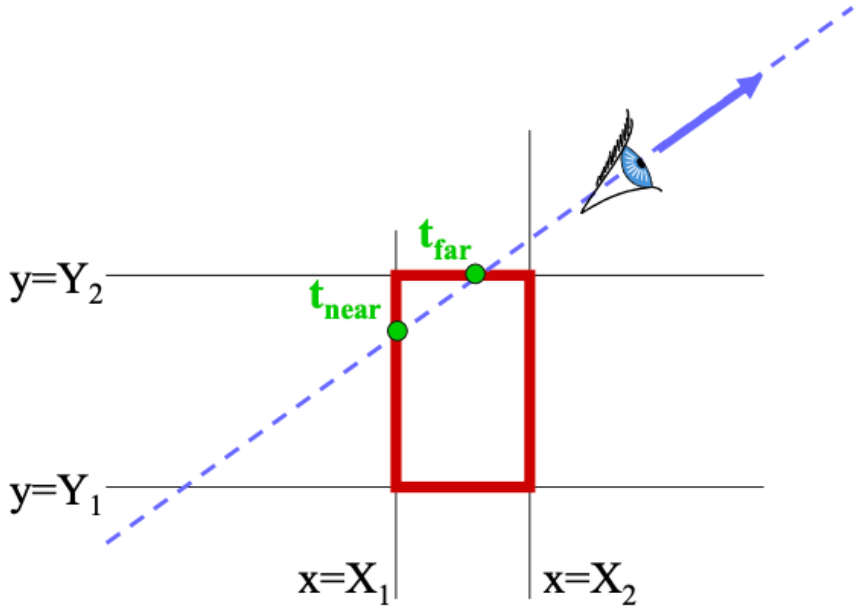
- If $t_{near} > t_{far}$ → **box is missed**



MIT EECS 6.837, Cutler and Durand

Is the Box Behind the Eyepoint?

- If $t_{far} < t_{min}$ → **box is behind**




```
return t_far >= t_near && t_near < ray.t && t_far > 0;
```

```
bool IntersectAABB( Ray & ray, const Box & b )
```

Cube or Axis-Aligned Bounding Boxes (AABBs)

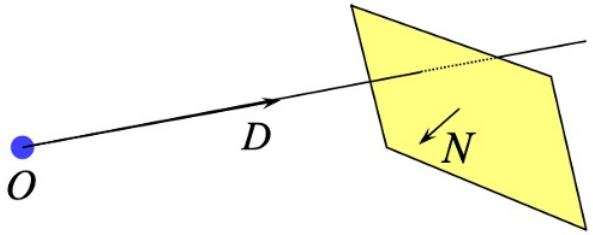
```
bool IntersectAABB( Ray & ray, const Box & b ) {
    float t_near, t_far;
    for (int i = 0; i < 3; ++i) {
        float t1 = (b.bmin[i] - ray.o[i]) / ray.D[i];
        float t2 = (b.bmax[i] - ray.o[i]) / ray.D[i];
        t_near = max(t_near,min(t1,t2));
        t_far  = min(t_far,max(t1,t2));
    }
    return t_far >= t_near && t_near < ray.t && t_far > 0;
}
```

Ray-Box Intersection Summary

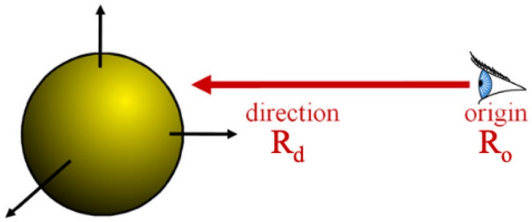
- For each dimension,
  - If $R_{dx} = 0$ (ray is parallel) AND
    $R_{ox} < X_1$ or $R_{ox} > X_2$ → **no intersection**
- For each dimension, calculate intersection distances t_1 and t_2
  - $t_1 = (X_1 - R_{ox}) / R_{dx}$ $\quad$ $t_2 = (X_2 - R_{ox}) / R_{dx}$
  - If $t_1 > t_2$, swap
  - Maintain t_{near} and t_{far} (closest & farthest intersections so far)
  - If $t_1 > t_{near}$, $t_{near} = t_1$ $\quad$ If $t_2 < t_{far}$, $t_{far} = t_2$
- If $t_{near} > t_{far}$ → **box is missed**
- If $t_{far} < t_{min}$ → **box is behind**
- If $t_{near} > t_{min}$ → **closest intersection at t_{near}**
- Else → **closest intersection at t_{far}**

Triangle Representation

- Plane

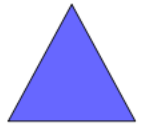


- Sphere



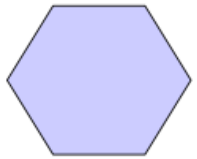
- Cube / Axis-Aligned Bounding Boxes(AABBs)
  - Ray-Box Intersection

- **Triangle** (Frequently USED)
  - Ray-Triangle Intersection

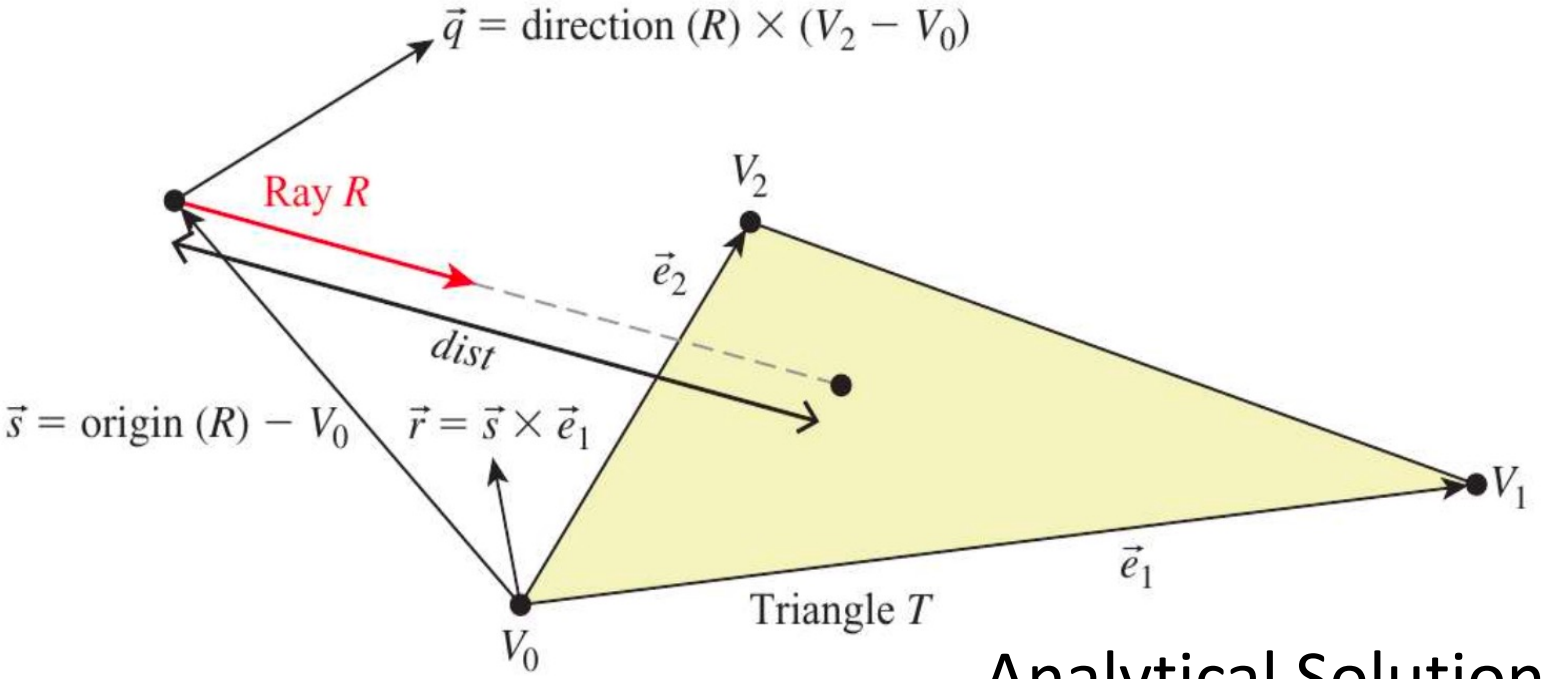


Ray-Triangle Intersection

- Use ray-polygon
- Ray-Polygon Intersection

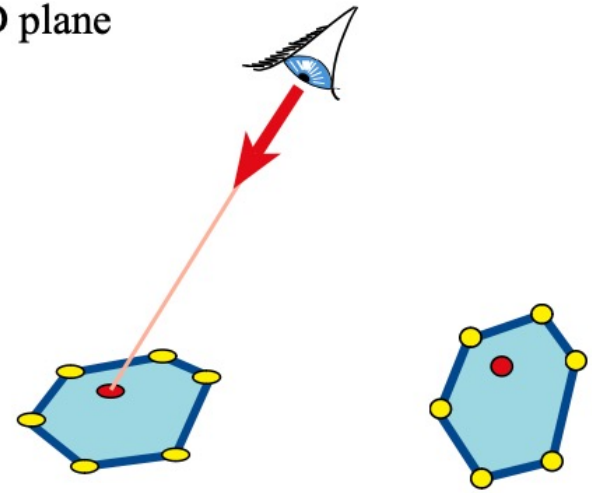


```
class Tri{
public:
    // Three points of the triangle: 3D coordinate
    const float a[3];
    const float b[3];
    const float c[3];
};
```



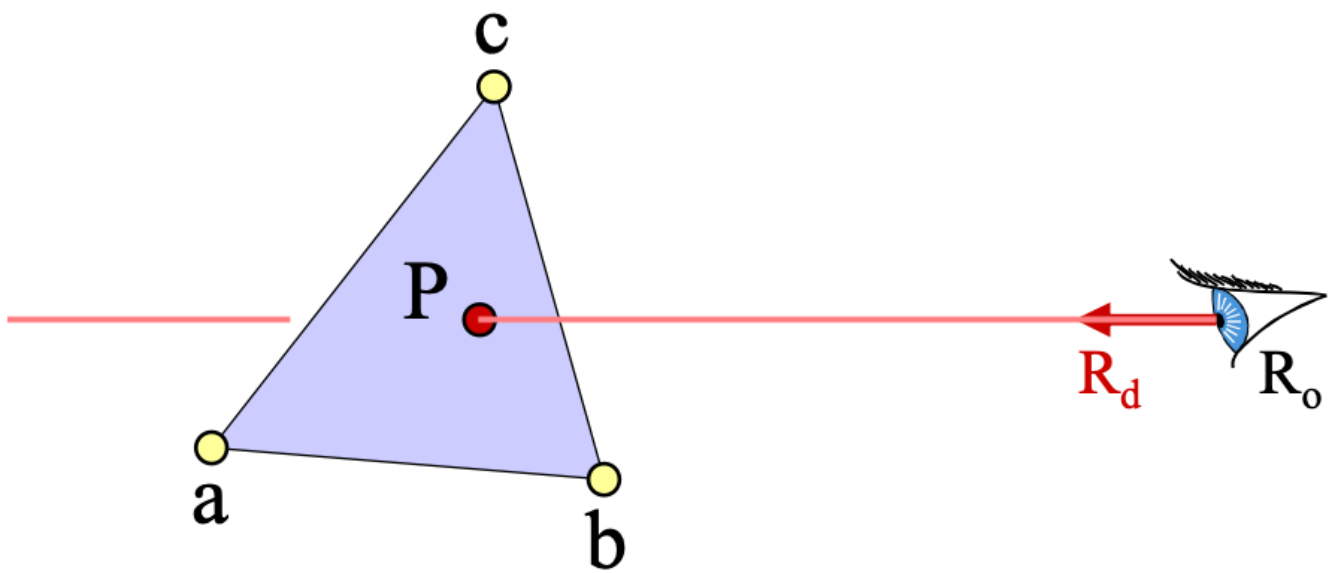
Analytical Solution

- Ray-plane intersection
- Test if intersection is in the polygon
  - Solve in the 2D plane



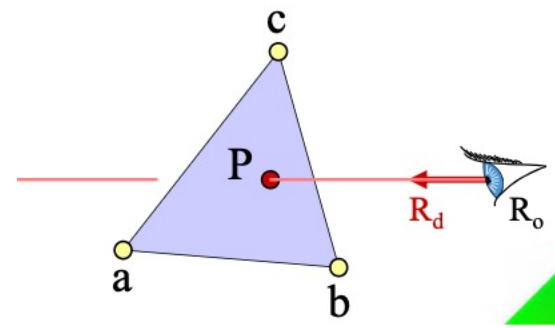
Ray-Triangle Intersection

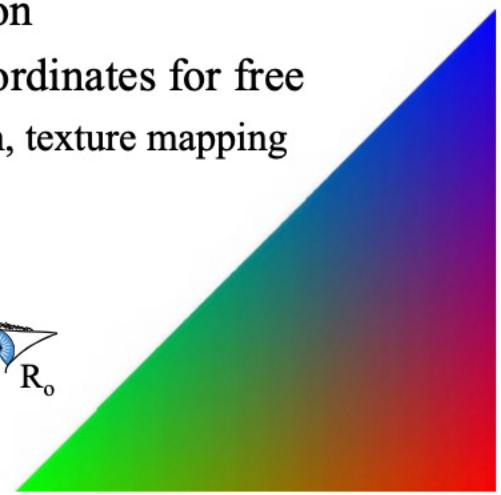
- Use ray-polygon
- Or try to be smarter
  - Use barycentric coordinates



Advantages of Barycentric Intersection

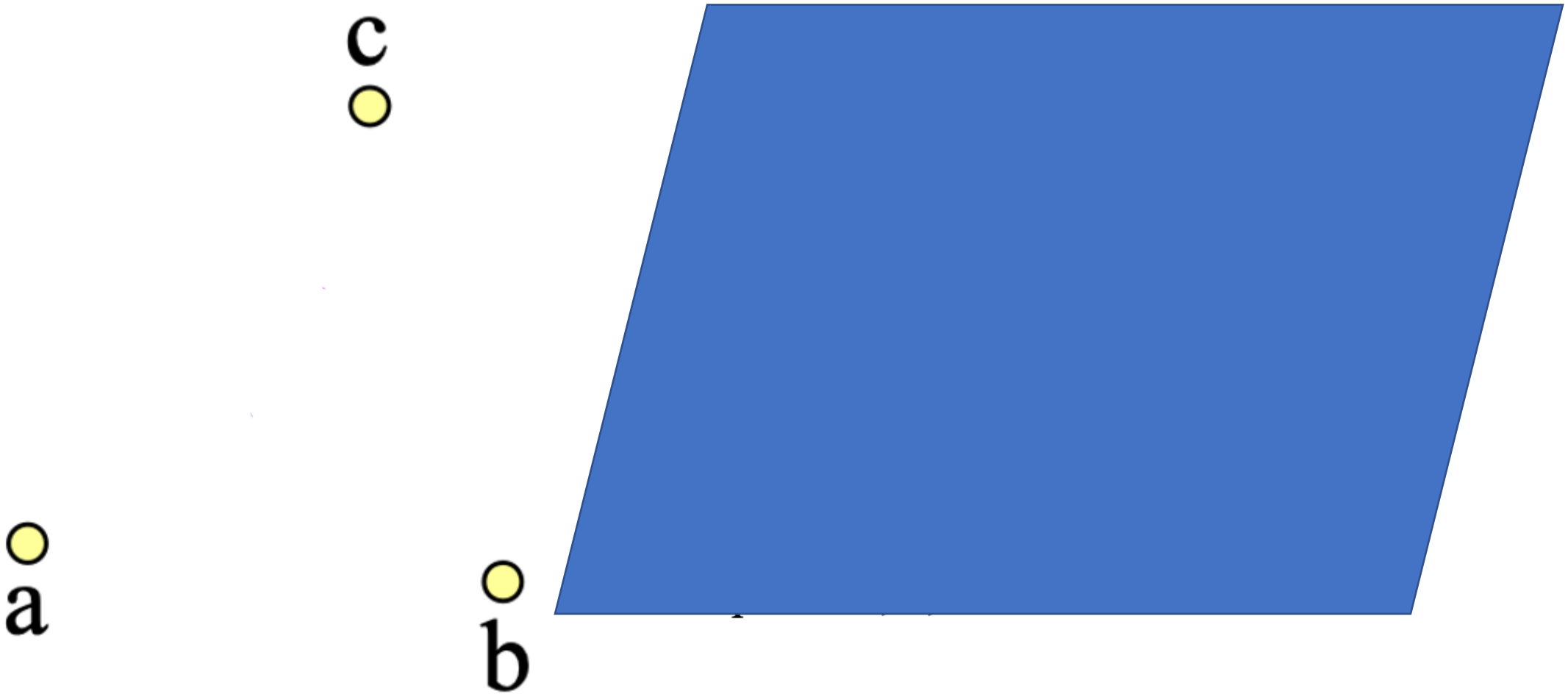
- Efficient
- Stores no plane equation
- Get the barycentric coordinates for free
  - Useful for interpolation, texture mapping





Barycentric Definition of a Plane

[Möbius, 1827]

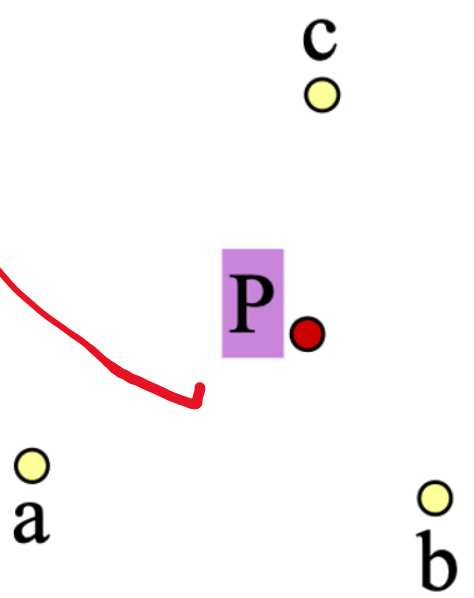


Barycentric Definition of a Plane

- $P(\alpha, \beta, \gamma) = \alpha a + \beta b + \gamma c$ with $\alpha + \beta + \gamma = 1$
- Is it explicit or implicit?

[Möbius, 1827]

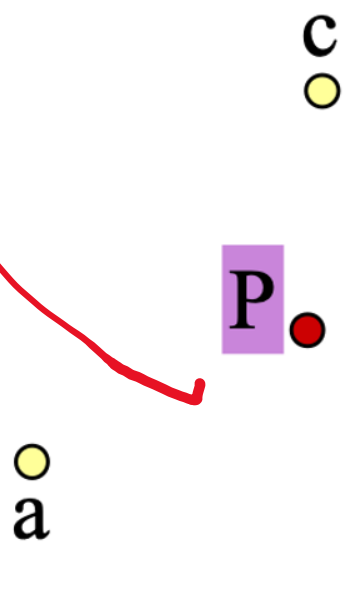
```
class Tri{
public:
    // Three points of the triangle: 3D coordinate
    const float a[3];
    const float b[3];
    const float c[3];
};
```



P is the *barycenter*:
the single point upon which the plane would balance if weights of size α, β, & γ are placed on points a, b, & c.

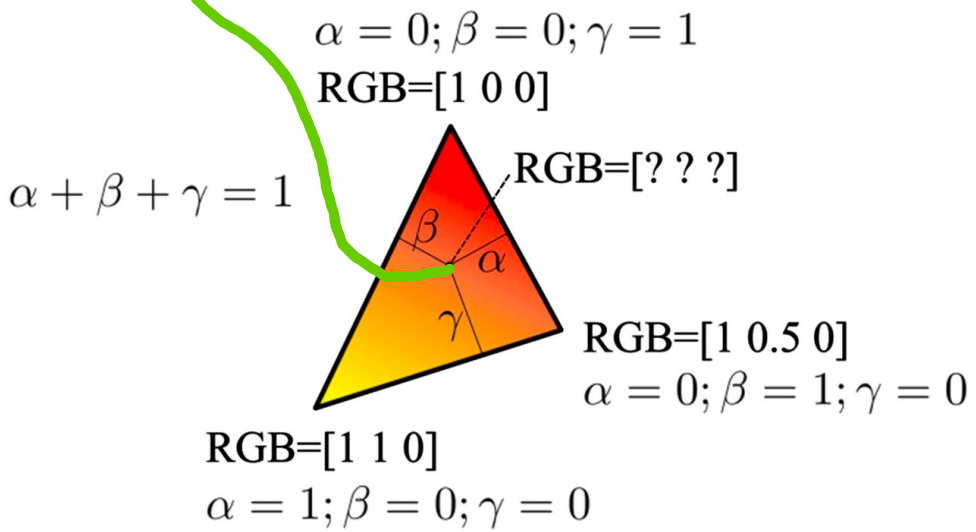
Barycentric Definition of a Plane

- $P(\alpha, \beta, \gamma) = \alpha a + \beta b + \gamma c$ with $\alpha + \beta + \gamma = 1$
- Is it explicit or implicit?



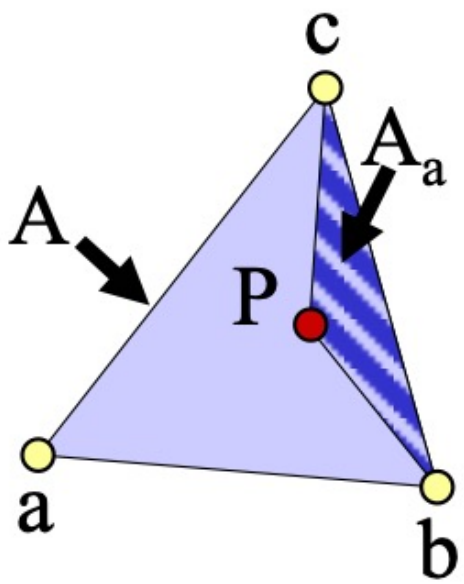
Barycentric Definition of a Triangle

- $P(\alpha, \beta, \gamma) = \alpha a + \beta b + \gamma c$ with $\alpha + \beta + \gamma = 1$
- AND $0 < \alpha < 1$ & $0 < \beta < 1$ & $0 < \gamma < 1$



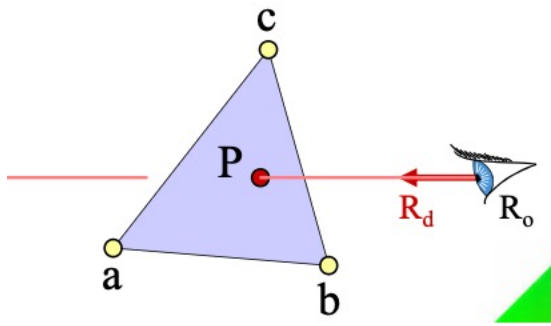
How Do We Compute α, β, γ ?

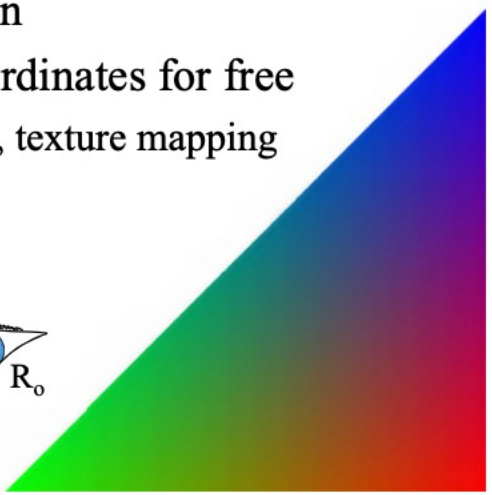
- Ratio of opposite sub-triangle area to total area
  - $\alpha = A_a/A$ $\beta = A_b/A$ $\gamma = A_c/A$
- Use signed areas for points outside the triangle



Advantages of Barycentric Intersection

- Efficient
- Stores no plane equation
- Get the barycentric coordinates for free
  - Useful for interpolation, texture mapping





Simplify

using this constraint

- Since $\alpha + \beta + \gamma = 1$, we can write $\alpha = 1 - \beta - \gamma$

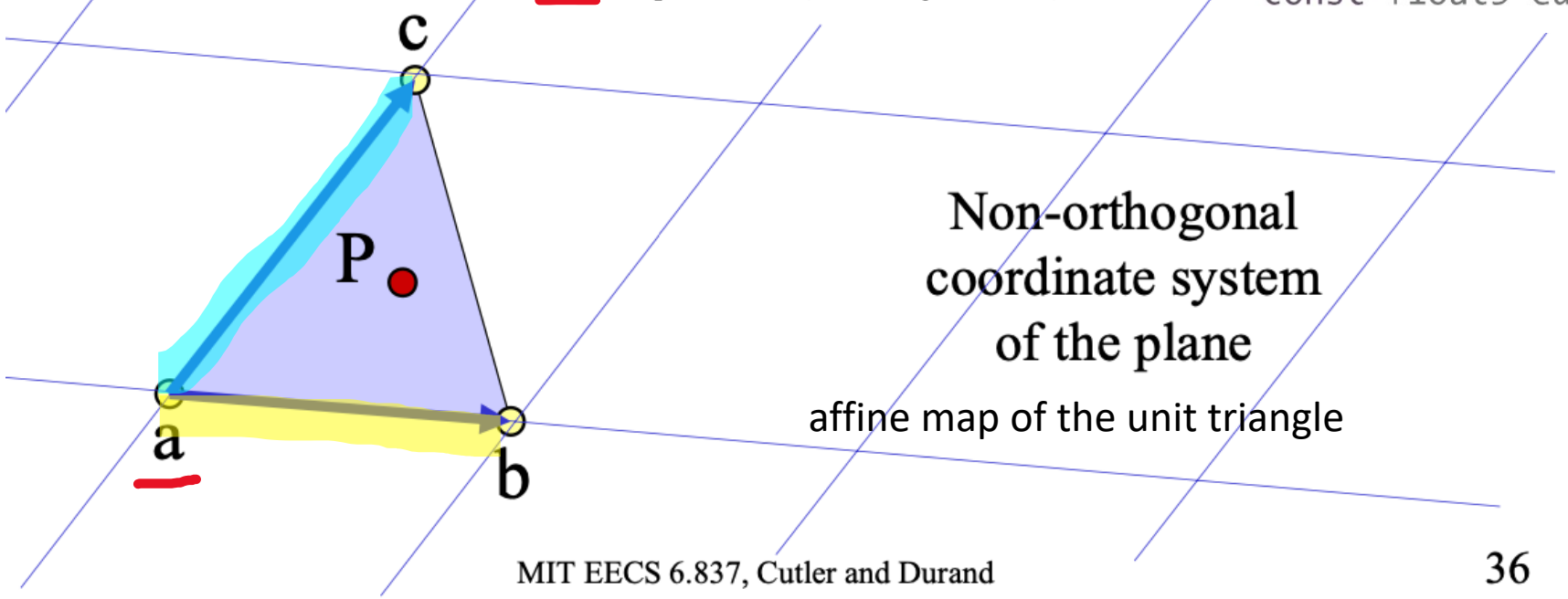
$$P(\alpha, \beta, \gamma) = \alpha a + \beta b + \gamma c$$

rewrite

$$P(\beta, \gamma) = (1-\beta-\gamma)a + \beta b + \gamma c$$

$$= a + \beta(b-a) + \gamma(c-a)$$

```
const float3 edge1 = tri.vertex1 - tri.vertex0;
const float3 edge2 = tri.vertex2 - tri.vertex0;
```



Non-orthogonal coordinate system of the plane

affine map of the unit triangle

Ray - Object ***Intersection***

IF the object is intersected by the ray

Math problem of Solving t

- Since $\alpha + \beta + \gamma = 1$, we can write $\alpha = 1 - \beta - \gamma$

$$P(\alpha, \beta, \gamma) = \alpha a + \beta b + \gamma c$$

rewrite

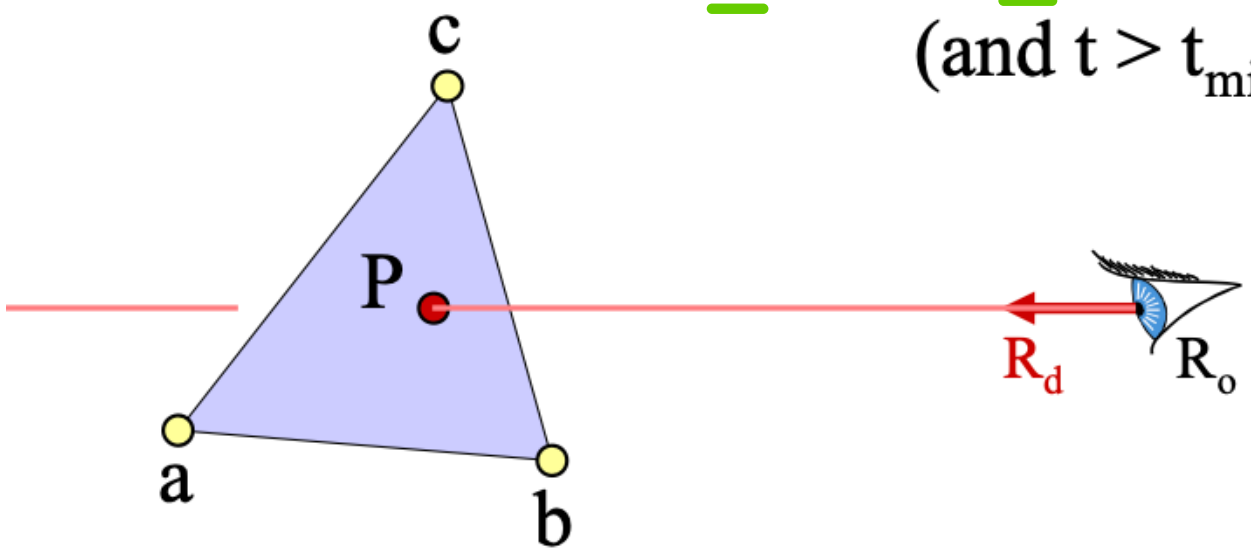
$$P(\beta, \gamma) = (1-\beta-\gamma)a + \beta b + \gamma c$$

$$= a + \beta(b-a) + \gamma(c-a)$$

(and $t > t_{min} \ldots$)

$$P(t) = R_o + t * R_d$$

$$R_o + t * R_d = a + \beta(b-a) + \gamma(c-a)$$



Ray - Object Intersection

Math problem of Solving t

$$R_o + t * R_d = a + \beta(b-a) + \gamma(c-a)$$

Intersection if $\beta + \gamma < 1$ & $\beta > 0$ & $\gamma > 0$

Intersection with Barycentric Triangle

- $R_o + t * R_d = a + \beta(b-a) + \gamma(c-a)$

$$\left.\begin{aligned} R_{ox} + tR_{dx} &= a_x + \beta(b_x - a_x) + \gamma(c_x - a_x) \\ R_{oy} + tR_{dy} &= a_y + \beta(b_y - a_y) + \gamma(c_y - a_y) \\ R_{oz} + tR_{dz} &= a_z + \beta(b_z - a_z) + \gamma(c_z - a_z) \end{aligned}\right\}$$

3D space
3 equations,
3 unknowns

- Regroup & write in matrix form:

edge1 edge2

$$\begin{bmatrix} a_x - b_x & a_x - c_x & R_{dx} \\ a_y - b_y & a_y - c_y & R_{dy} \\ a_z - b_z & a_z - c_z & R_{dz} \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} a_x - R_{ox} \\ a_y - R_{oy} \\ a_z - R_{oz} \end{bmatrix}$$

Ray - Object Intersection

Math problem of Solving t

IF the object is intersected by the ray

$$R_o + t * R_d = a + \beta(b-a) + \gamma(c-a)$$

Intersection if $\beta + \gamma < 1$ & $\beta > 0$ & $\gamma > 0$

Intersection with Barycentric Triangle

- $R_o + t * R_d = a + \beta(b-a) + \gamma(c-a)$

$$\begin{aligned} R_{ox} + tR_{dx} &= a_x + \beta(b_x - a_x) + \gamma(c_x - a_x) \\ R_{oy} + tR_{dy} &= a_y + \beta(b_y - a_y) + \gamma(c_y - a_y) \\ R_{oz} + tR_{dz} &= a_z + \beta(b_z - a_z) + \gamma(c_z - a_z) \end{aligned}$$

3D space
3 equations,
3 unknowns

- Regroup & write in matrix form:

edge1 edge2

$$\begin{bmatrix} a_x - b_x & a_x - c_x & R_{dx} \\ a_y - b_y & a_y - c_y & R_{dy} \\ a_z - b_z & a_z - c_z & R_{dz} \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} a_x - R_{ox} \\ a_y - R_{oy} \\ a_z - R_{oz} \end{bmatrix}$$

$$\begin{cases} 3x + 2y + z = 7 \\ 2x - 3y + 2z = 3 \\ 4x - y - 6z = -9 \end{cases}$$

Ray - Object Intersection

Math problem of Solving t

IF the object is intersected by the ray

$$R_o + t * R_d = a + \beta(b-a) + \gamma(c-a)$$

Intersection if $\beta + \gamma < 1$ & $\beta > 0$ & $\gamma > 0$

Intersection with Barycentric Triangle

- $R_o + t * R_d = a + \beta(b-a) + \gamma(c-a)$

$$\begin{aligned} R_{ox} + tR_{dx} &= a_x + \beta(b_x - a_x) + \gamma(c_x - a_x) \\ R_{oy} + tR_{dy} &= a_y + \beta(b_y - a_y) + \gamma(c_y - a_y) \\ R_{oz} + tR_{dz} &= a_z + \beta(b_z - a_z) + \gamma(c_z - a_z) \end{aligned}$$

3D space
3 equations,
3 unknowns

- Regroup & write in matrix form:

edge1 edge2

$$\begin{bmatrix} a_x - b_x & a_x - c_x & R_{dx} \\ a_y - b_y & a_y - c_y & R_{dy} \\ a_z - b_z & a_z - c_z & R_{dz} \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} a_x - R_{ox} \\ a_y - R_{oy} \\ a_z - R_{oz} \end{bmatrix}$$

$$Ax = b$$

$$\begin{cases} 3x + 2y + z = 7 \\ 2x - 3y + 2z = 3 \\ 4x - y - 6z = -9 \end{cases}$$

$$x_i = \frac{\det \mathbf{A}_i}{\det \mathbf{A}},$$

where Ai is the matrix obtained by replacing the i-th column of A by column vector y.

Ray - Object Intersection

Math problem of Solving t

IF the object is intersected by the ray

$$R_o + t * R_d = a + \beta(b-a) + \gamma(c-a)$$

Intersection if $\beta + \gamma < 1$ & $\beta > 0$ & $\gamma > 0$

Intersection with Barycentric Triangle

- $R_o + t * R_d = a + \beta(b-a) + \gamma(c-a)$

$$\begin{aligned} R_{ox} + tR_{dx} &= a_x + \beta(b_x - a_x) + \gamma(c_x - a_x) \\ R_{oy} + tR_{dy} &= a_y + \beta(b_y - a_y) + \gamma(c_y - a_y) \\ R_{oz} + tR_{dz} &= a_z + \beta(b_z - a_z) + \gamma(c_z - a_z) \end{aligned}$$

3D space
3 equations,
3 unknowns

- Regroup & write in matrix form:

edge1 edge2

$$\begin{bmatrix} a_x - b_x & a_x - c_x & R_{dx} \\ a_y - b_y & a_y - c_y & R_{dy} \\ a_z - b_z & a_z - c_z & R_{dz} \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} a_x - R_{ox} \\ a_y - R_{oy} \\ a_z - R_{oz} \end{bmatrix}$$

Cramer's Rule

Generally, given that $\begin{cases} a_1x + b_1y + c_1z = k_1 \\ a_2x + b_2y + c_2z = k_2 \\ a_3x + b_3y + c_3z = k_3 \end{cases}$

we get: $\begin{pmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} k_1 \\ k_2 \\ k_3 \end{pmatrix}$ in matrix form:

and $x = \dfrac{\begin{vmatrix} k_1 & b_1 & c_1 \\ k_2 & b_2 & c_2 \\ k_3 & b_3 & c_3 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}}$, $y = \dfrac{\begin{vmatrix} a_1 & k_1 & c_1 \\ a_2 & k_2 & c_2 \\ a_3 & k_3 & c_3 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}}$, $z = \dfrac{\begin{vmatrix} a_1 & b_1 & k_1 \\ a_2 & b_2 & k_2 \\ a_3 & b_3 & k_3 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}}$

edge1 edge2

$$t = \frac{\begin{vmatrix} a_x - b_x & a_x - c_x & a_x - R_{ox} \\ a_y - b_y & a_y - c_y & a_y - R_{oy} \\ a_z - b_z & a_z - c_z & a_z - R_{oz} \end{vmatrix}}{|A|}$$

| | denotes the determinant

matrix form
Can be copied mechanically into code

Ray - Object Intersection

Math problem of Solving t

IF the object is intersected by the ray

$$R_o + t * R_d = a + \beta(b-a) + \gamma(c-a)$$

Intersection if $\beta + \gamma < 1$ & $\beta > 0$ & $\gamma > 0$

Intersection with Barycentric Triangle

- $R_o + t * R_d = a + \beta(b-a) + \gamma(c-a)$

$$\begin{aligned} R_{ox} + tR_{dx} &= a_x + \beta(b_x - a_x) + \gamma(c_x - a_x) \\ R_{oy} + tR_{dy} &= a_y + \beta(b_y - a_y) + \gamma(c_y - a_y) \\ R_{oz} + tR_{dz} &= a_z + \beta(b_z - a_z) + \gamma(c_z - a_z) \end{aligned}$$

3D space
3 equations,
3 unknowns

- Regroup & write in matrix form:

$$\begin{bmatrix} a_x - b_x & a_x - c_x & R_{dx} \\ a_y - b_y & a_y - c_y & R_{dy} \\ a_z - b_z & a_z - c_z & R_{dz} \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} a_x - R_{ox} \\ a_y - R_{oy} \\ a_z - R_{oz} \end{bmatrix}$$

(edge1, edge2)

Or

$$\begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} a_x - b_x & a_x - c_x & R_{dx} \\ a_y - b_y & a_y - c_y & R_{dy} \\ a_z - b_z & a_z - c_z & R_{dz} \end{bmatrix}^{-1} \begin{bmatrix} a_x - R_{ox} \\ a_y - R_{oy} \\ a_z - R_{oz} \end{bmatrix}$$

Cramer's Rule

- Used to solve for one variable at a time in system of equations

$$\beta = \frac{\begin{vmatrix} a_x - R_{ox} & a_x - c_x & R_{dx} \\ a_y - R_{oy} & a_y - c_y & R_{dy} \\ a_z - R_{oz} & a_z - c_z & R_{dz} \end{vmatrix}}{|A|} \qquad \gamma = \frac{\begin{vmatrix} a_x - b_x & a_x - R_{ox} & R_{dx} \\ a_y - b_y & a_y - R_{oy} & R_{dy} \\ a_z - b_z & a_z - R_{oz} & R_{dz} \end{vmatrix}}{|A|}$$

$$t = \frac{\begin{vmatrix} a_x - b_x & a_x - c_x & a_x - R_{ox} \\ a_y - b_y & a_y - c_y & a_y - R_{oy} \\ a_z - b_z & a_z - c_z & a_z - R_{oz} \end{vmatrix}}{|A|}$$

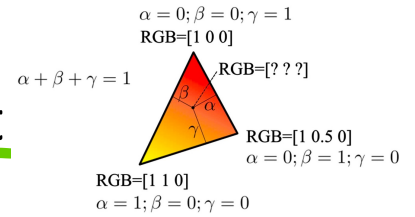
(edge1, edge2)

| | denotes the determinant

matrix form
Can be copied mechanically into code

Ray - Object Intersection

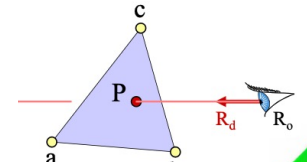
Math problem of Solving t



Advantages of Barycentric Intersection

- Efficient
- Stores no plane equation

Get the barycentric coordinates for free
– Useful for interpolation, texture mapping



$$R_o + t * R_d = a + \beta(b-a) + \gamma(c-a)$$

Intersection if $\beta + \gamma < 1$ & $\beta > 0$ & $\gamma > 0$

Intersection with Barycentric Triangle

- $R_o + t * R_d = a + \beta(b-a) + \gamma(c-a)$

$$\begin{aligned} R_{ox} + tR_{dx} &= a_x + \beta(b_x - a_x) + \gamma(c_x - a_x) \\ R_{oy} + tR_{dy} &= a_y + \beta(b_y - a_y) + \gamma(c_y - a_y) \\ R_{oz} + tR_{dz} &= a_z + \beta(b_z - a_z) + \gamma(c_z - a_z) \end{aligned}$$

3D space
3 equations,
3 unknowns

- Regroup & write in matrix form:

$$\begin{bmatrix} a_x - b_x & a_x - c_x & R_{dx} \\ a_y - b_y & a_y - c_y & R_{dy} \\ a_z - b_z & a_z - c_z & R_{dz} \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} a_x - R_{ox} \\ a_y - R_{oy} \\ a_z - R_{oz} \end{bmatrix}$$

edge1 edge2

Or

$$\begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} a_x - b_x & a_x - c_x & R_{dx} \\ a_y - b_y & a_y - c_y & R_{dy} \\ a_z - b_z & a_z - c_z & R_{dz} \end{bmatrix}^{-1} \begin{bmatrix} a_x - R_{ox} \\ a_y - R_{oy} \\ a_z - R_{oz} \end{bmatrix}$$

Cramer's Rule

- Used to solve for one variable at a time in system of equations

$$\beta = \frac{\begin{vmatrix} a_x - R_{ox} & a_x - c_x & R_{dx} \\ a_y - R_{oy} & a_y - c_y & R_{dy} \\ a_z - R_{oz} & a_z - c_z & R_{dz} \end{vmatrix}}{|A|} \qquad \gamma = \frac{\begin{vmatrix} a_x - b_x & a_x - R_{ox} & R_{dx} \\ a_y - b_y & a_y - R_{oy} & R_{dy} \\ a_z - b_z & a_z - R_{oz} & R_{dz} \end{vmatrix}}{|A|}$$

edge1 edge2

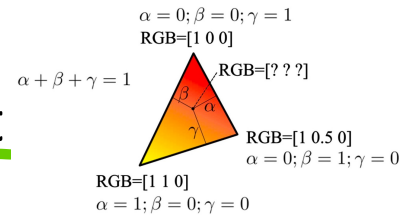
$$t = \frac{\begin{vmatrix} a_x - b_x & a_x - c_x & a_x - R_{ox} \\ a_y - b_y & a_y - c_y & a_y - R_{oy} \\ a_z - b_z & a_z - c_z & a_z - R_{oz} \end{vmatrix}}{|A|}$$

| | denotes the determinant

matrix form
Can be copied mechanically into code

Ray - Object Intersection

Math problem of Solving t

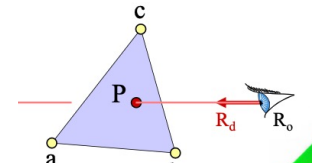


Advantages of Barycentric Intersection

- Efficient
- Stores no plane equation

Get the barycentric coordinates for free

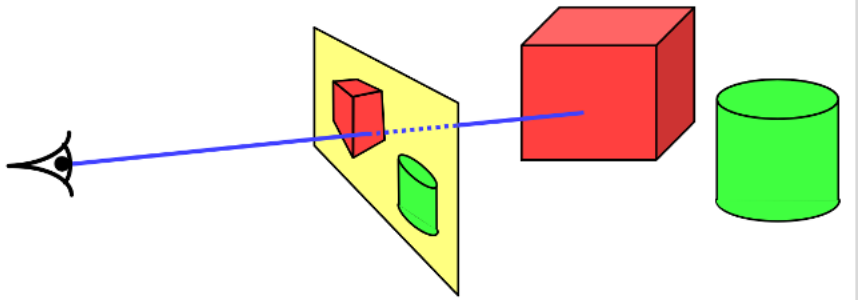
– Useful for interpolation, texture mapping



Coding: Step by step explanation

Möller–Trumbore intersection algorithm

Paper: Tomas Möller and Ben Trumbore, *Fast Minimum Storage Ray Triangle Intersection*, 1997



$$R_o + t * R_d = a + \beta(b-a) + \gamma(c-a)$$

Intersection if $\beta + \gamma < 1$ & $\beta > 0$ & $\gamma > 0$

Intersection with Barycentric Triangle

- $R_o + t * R_d = a + \beta(b-a) + \gamma(c-a)$

$$\begin{aligned} R_{ox} + tR_{dx} &= a_x + \beta(b_x - a_x) + \gamma(c_x - a_x) \\ R_{oy} + tR_{dy} &= a_y + \beta(b_y - a_y) + \gamma(c_y - a_y) \\ R_{oz} + tR_{dz} &= a_z + \beta(b_z - a_z) + \gamma(c_z - a_z) \end{aligned}$$

3D space
3 equations,
3 unknowns

- Regroup & write in matrix form:

$$A = \begin{bmatrix} a_x - b_x & a_x - c_x & R_{dx} \\ a_y - b_y & a_y - c_y & R_{dy} \\ a_z - b_z & a_z - c_z & R_{dz} \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} a_x - R_{ox} \\ a_y - R_{oy} \\ a_z - R_{oz} \end{bmatrix}$$

edge1 edge2

$$t = \frac{\begin{vmatrix} a_x - b_x & a_x - c_x & a_x - R_{ox} \\ a_y - b_y & a_y - c_y & a_y - R_{oy} \\ a_z - b_z & a_z - c_z & a_z - R_{oz} \end{vmatrix}}{|A|}$$

edge1 edge2

| | denotes the determinant

matrix form
Can be copied mechanically into code

IF the object is intersected by the ray

How to solve the determinant |A|?

$$R_o + t * R_d = a + \beta(b-a) + \gamma(c-a)$$

Intersection if $\beta + \gamma < 1$ & $\beta > 0$ & $\gamma > 0$

Intersection with Barycentric Triangle

- $R_o + t * R_d = a + \beta(b-a) + \gamma(c-a)$

$$\begin{aligned} R_{ox} + tR_{dx} &= a_x + \beta(b_x - a_x) + \gamma(c_x - a_x) \\ R_{oy} + tR_{dy} &= a_y + \beta(b_y - a_y) + \gamma(c_y - a_y) \\ R_{oz} + tR_{dz} &= a_z + \beta(b_z - a_z) + \gamma(c_z - a_z) \end{aligned}$$

3D space
3 equations,
3 unknowns

- Regroup & write in matrix form:

$$A = \begin{bmatrix} a_x - b_x & a_x - c_x & R_{dx} \\ a_y - b_y & a_y - c_y & R_{dy} \\ a_z - b_z & a_z - c_z & R_{dz} \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} a_x - R_{ox} \\ a_y - R_{oy} \\ a_z - R_{oz} \end{bmatrix}$$

edge1 edge2

```
const float3 h = cross( ray.Direction, edge2 );
const float a = dot( edge1, h );
```

Scalar Triple Product Formula

$$a \cdot (b \times c) = \det \begin{bmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{bmatrix} = \det \begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{bmatrix} = \det \begin{bmatrix} a & b & c \end{bmatrix}$$

$= |A|$

cuemath THE MATH EXPERT

IF the object is intersected by the ray

How to solve the determinant |A|?

$$R_o + t * R_d = a + \beta(b-a) + \gamma(c-a)$$

Intersection if $\beta + \gamma < 1$ & $\beta > 0$ & $\gamma > 0$

```
if (a > -0.0001f && a < 0.0001f)
    return false; // ray parallel to triangle
const float f = 1 / a;
```

denominator !=0
floating-point precision

```
const float3 h = cross( ray.Direction, edge2 );
const float a = dot( edge1, h );
```

$$A = \begin{bmatrix} a_x - b_x & a_x - c_x & R_{dx} \\ a_y - b_y & a_y - c_y & R_{dy} \\ a_z - b_z & a_z - c_z & R_{dz} \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} a_x - R_{ox} \\ a_y - R_{oy} \\ a_z - R_{oz} \end{bmatrix}$$

```
const float3 h = cross( ray.Direction, edge2 );
const float a = dot( edge1, h );
```

edge1 edge2 s = ray.O - tri.vertex0

$$t = \frac{\begin{vmatrix} a_x - b_x & a_x - c_x & a_x - R_{ox} \\ a_y - b_y & a_y - c_y & a_y - R_{oy} \\ a_z - b_z & a_z - c_z & a_z - R_{oz} \end{vmatrix}}{|A|}$$

| | denotes the determinant

matrix form
Can be copied mechanically into code

IF the object is intersected by the ray

$$R_o + t * R_d = a + \beta(b-a) + \gamma(c-a)$$

```
if (a > -0.0001f && a < 0.0001f)   denominator !=0
    return false; // ray parallel to triangle   floating-point precision
const float f = 1 / a;
const float3 s = ray.O - tri.vertex0;
const float u = f * dot( s, h );
if (u < 0 || u > 1) return false;
const float3 q = cross( s, edge1 );
const float v = f * dot( ray.Direction, q );
if (v < 0 || u + v > 1) return false;
const float t = f * dot( edge2, q );
```

Intersection if $\beta + \gamma < 1 \quad \& \quad \beta > 0 \quad \& \quad \gamma > 0$

```
const float3 h = cross( ray.Direction, edge2 );
const float a = dot( edge1, h );
```

- Used to solve for one variable at a time in system of equations

s = ray.O - tri.vertex0 edge2

$$\beta = \frac{\begin{vmatrix} a_x - R_{ox} & a_x - c_x & R_{dx} \\ a_y - R_{oy} & a_y - c_y & R_{dy} \\ a_z - R_{oz} & a_z - c_z & R_{dz} \end{vmatrix}}{|A|}$$

edge1 s = ray.O - tri.vertex0

$$\gamma = \frac{\begin{vmatrix} a_x - b_x & a_x - R_{ox} & R_{dx} \\ a_y - b_y & a_y - R_{oy} & R_{dy} \\ a_z - b_z & a_z - R_{oz} & R_{dz} \end{vmatrix}}{|A|}$$

edge1 edge2 s = ray.O - tri.vertex0

$$t = \frac{\begin{vmatrix} a_x - b_x & a_x - c_x & a_x - R_{ox} \\ a_y - b_y & a_y - c_y & a_y - R_{oy} \\ a_z - b_z & a_z - c_z & a_z - R_{oz} \end{vmatrix}}{|A|}$$

| | denotes the determinant

matrix form
Can be copied mechanically into code

IF the object is intersected by the ray

$$R_o + t * R_d = a + \beta(b-a) + \gamma(c-a)$$

```
if (a > -0.0001f && a < 0.0001f)
    return false; // ray parallel to triangle
const float f = 1 / a;
const float3 s = ray.O - tri.vertex0;
const float u = f * dot( s, h );
if (u < 0 || u > 1) return false;
const float3 q = cross( s, edge1 );
const float v = f * dot( ray.Direction, q );
if (v < 0 || u + v > 1) return false;
const float t = f * dot( edge2, q );
```

denominator !=0
floating-point precision

Intersection if $\beta + \gamma < 1 \quad \& \quad \beta > 0 \quad \& \quad \gamma > 0$

```
const float3 h = cross( ray.Direction, edge2 );
const float a = dot( edge1, h );
```

- Used to solve for one variable at a time in system of equations

s = ray.O - tri.vertex0 edge2

$$\beta = \frac{\begin{vmatrix} a_x - R_{ox} & a_x - c_x & R_{dx} \\ a_y - R_{oy} & a_y - c_y & R_{dy} \\ a_z - R_{oz} & a_z - c_z & R_{dz} \end{vmatrix}}{|A|}$$

edge1 s = ray.O - tri.vertex0

$$\gamma = \frac{\begin{vmatrix} a_x - b_x & a_x - R_{ox} & R_{dx} \\ a_y - b_y & a_y - R_{oy} & R_{dy} \\ a_z - b_z & a_z - R_{oz} & R_{dz} \end{vmatrix}}{|A|}$$

edge1 edge2 s = ray.O - tri.vertex0

$$t = \frac{\begin{vmatrix} a_x - b_x & a_x - c_x & a_x - R_{ox} \\ a_y - b_y & a_y - c_y & a_y - R_{oy} \\ a_z - b_z & a_z - c_z & a_z - R_{oz} \end{vmatrix}}{|A|}$$

| | denotes the determinant

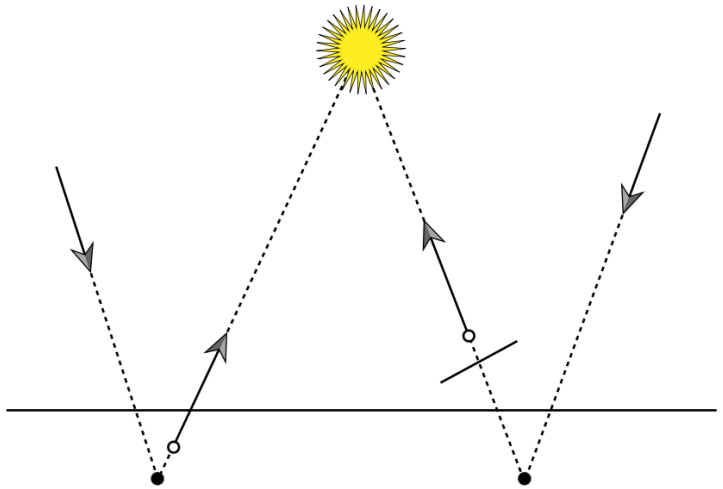
matrix form
Can be copied mechanically into code

Nearly All Done! Finally...

IF the object is intersected by the ray

Avoid self-intersection using ε (secondary ray)

```
if (t > 0.0001f) ray.t = min( ray.t, t );
```



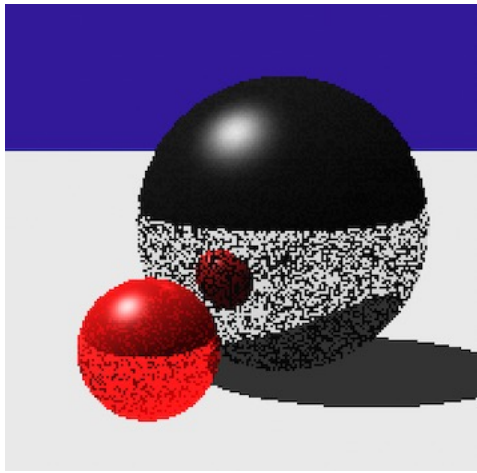
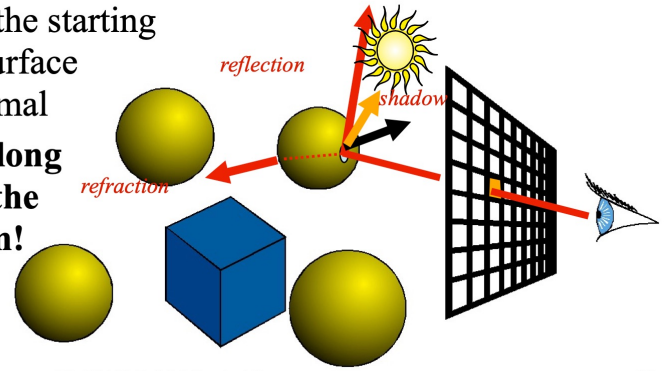


Figure 3.39: Geometric Settings for Rounding-Error Issues That Can Cause Visible Errors in Images. The incident ray on the left intersects the surface. On the left, the computed intersection point (black circle) is slightly below the surface and a too-low "epsilon" offsetting the origin of the shadow ray leads to an incorrect self-intersection, as the shadow ray origin (white circle) is still below the surface; thus the light is incorrectly determined to be occluded. On the right a too-high "epsilon" causes a valid intersection to be missed as the ray's origin is past the occluding surface.

https://www.pbr-book.org/3ed-2018/Shapes/Managing_Rounding_Error

The evil ε

- In ray tracing, do NOT report intersection for rays starting at the surface
  - Secondary rays will start at the surfaces
  - Requires epsilons
  - Best to nudge the starting point off the surface e.g., along normal
  - **Best: nudge along the direction the ray came from!**



Ray-triangle intersection—a different way

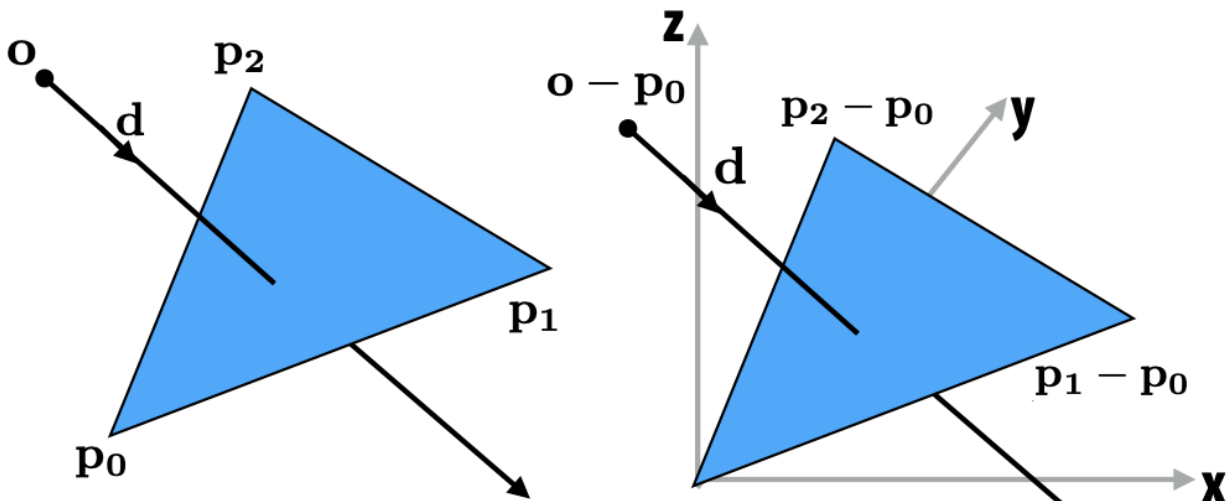
Plug parametric ray equation directly into equation for points on triangle:

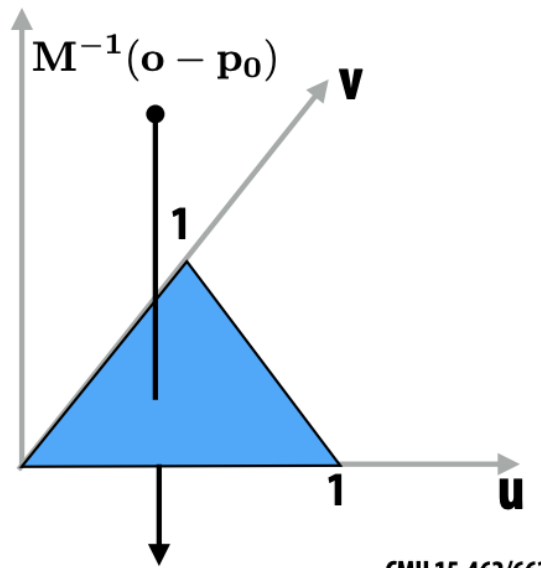
$$\mathbf{p_0} + u(\mathbf{p_1} - \mathbf{p_0}) + v(\mathbf{p_2} - \mathbf{p_0}) = \mathbf{o} + t\mathbf{d}$$

Solve for u, v, t:

$$\underbrace{\begin{bmatrix} \mathbf{p_1} - \mathbf{p_0} & \mathbf{p_2} - \mathbf{p_0} & -\mathbf{d} \end{bmatrix}}_{\mathbf{M}} \begin{bmatrix} u \\ v \\ t \end{bmatrix} = \mathbf{o} - \mathbf{p_0}$$

$\mathbf{M}^{-1}$ **transforms triangle back to unit triangle in u,v plane, and transforms ray's direction to be orthogonal to plane**





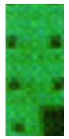
```
bool IntersectTri( Ray& ray, const Tri& tri )
{
    const float3 edge1 = tri.vertex1 - tri.vertex0;
    const float3 edge2 = tri.vertex2 - tri.vertex0;
    const float3 h = cross( ray.Direction, edge2 );
    const float a = dot( edge1, h );
    if (a > -0.0001f && a < 0.0001f)
        return false; // ray parallel to triangle
    const float f = 1 / a;
    const float3 s = ray.O - tri.vertex0;
    const float u = f * dot( s, h );
    if (u < 0 || u > 1) return false;
    const float3 q = cross( s, edge1 );
    const float v = f * dot( ray.Direction, q );
    if (v < 0 || u + v > 1) return false;
    const float t = f * dot( edge2, q );
    if (t > 0.0001f) ray.t = min( ray.t, t );
    return true;
}
```

All Done! Finally...

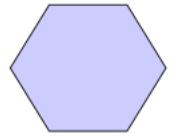
IF the object is intersected by the ray

- Other primitives
  - Boxes
  - Polygons
  - Triangles
  - IFS?

- Ray-Polygon Intersection

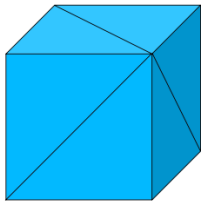


- Ray-Bunny Intersection & extra topics…

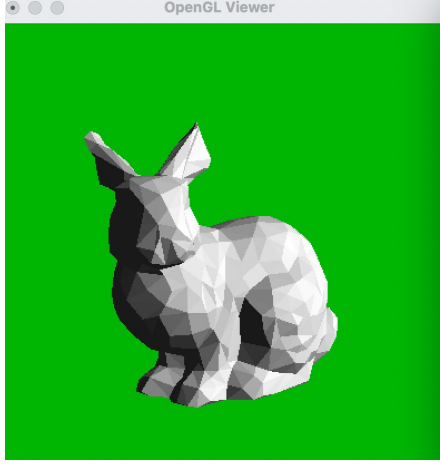


Triangle Meshes (.obj)

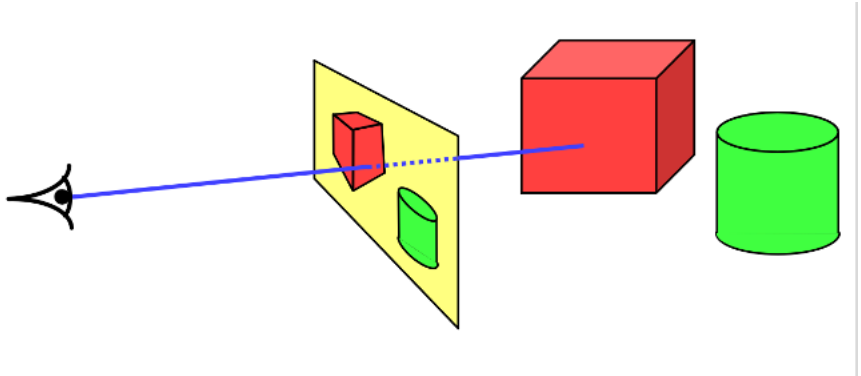
```
vertices
v -1 -1 -1
v  1 -1 -1
v -1  1 -1
v  1  1 -1
v -1 -1  1
v  1 -1  1
v -1  1  1
v  1  1  1
triangles
f 1 3 4
f 1 4 2
f 5 6 8
f 5 8 7
f 1 2 6
f 1 6 5
f 3 7 8
f 3 8 4
f 1 5 7
f 1 7 3
f 2 4 8
f 2 8 6
```



MIT EECS 6.837 Computer Graphics (solid Math intro) https://groups.csail.mit.edu/graphics/classes/6.837/F04/calendar.html

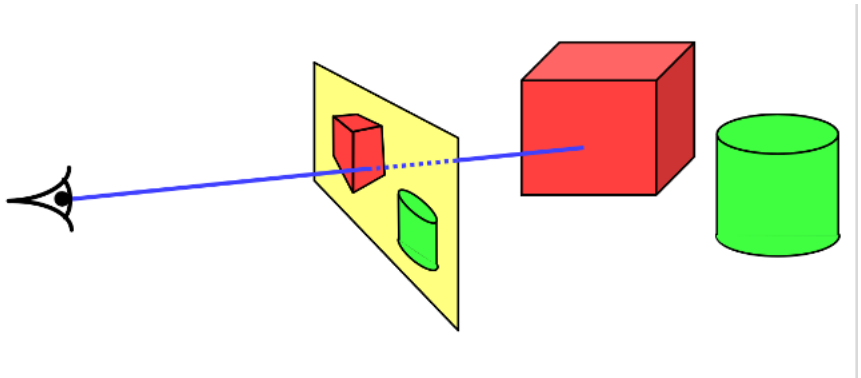


Q&A



Part 3: Improvements of RT

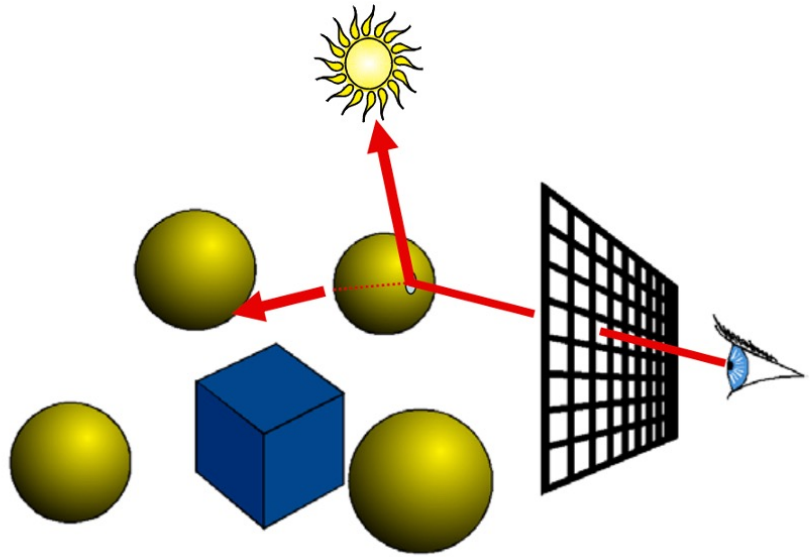
Acceleration Data Structure



Ray Tracing Algorithm

However, it is not abundantly present in simulations as it is computationally expensive.

Suppose an image has roughly one million triangles. If we compute all of the ray-triangle intersections, it requires about 10^14 ray-triangle intersection computations. In the year of 2010 ray-triangle intersection method could only achieve a few hundred million ray-triangle intersections (10^8) per second. In order to implement efficient ray tracing, we need to decrease the number of redundant ray-triangle intersections.



Ray Tracing Algorithm Analysis

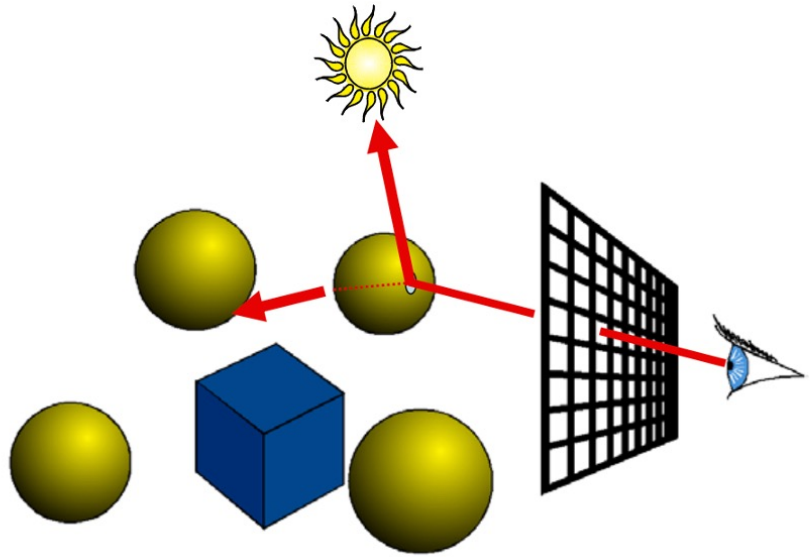
- Ray casting
- Lots of primitives
- Recursive
- Distributed Ray Tracing Effects
  - Soft shadows
  - Anti-aliasing
  - Glossy reflection
  - Motion blur
  - Depth of field

cost ≈ height * width *
num primitives *
intersection cost *
size of recursive ray tree *
num shadow rays *
num supersamples *
num glossy rays *
num temporal samples *
num focal samples *
. . .

can we reduce this?

Ray Tracing Algorithm

However, it is not abundantly present in simulations as it is computationally expensive. Suppose an image has roughly one million triangles. If we compute all of the ray-triangle intersections, it requires about **10^14** ray-triangle intersection computations. In the year of 2010 ray-triangle intersection method could only achieve a few hundred million ray-triangle intersections (**10^8**) per second. In order to implement efficient ray tracing, we need to decrease the number of redundant ray-triangle intersections.



Rasterization (Used in most Graphics Pipeline)

Ray Casting

```
For each pixel
    For each object
```

- Whole scene must be in memory

Send pixels to the scene

Discretize first

Rendering Pipeline

```
For each triangle
    For each pixel
```

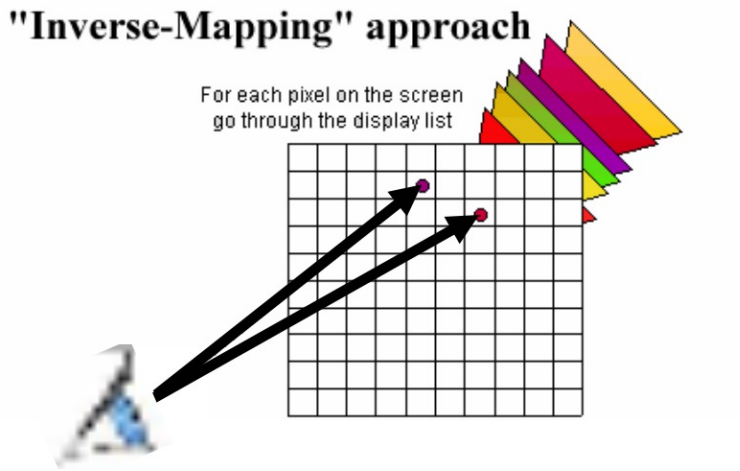
- Primitives processed one at a time

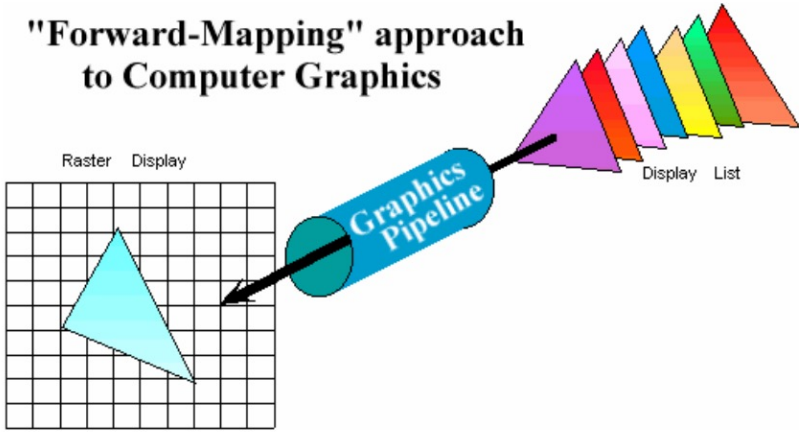
Project scene to the pixels

Discretize last

MVP transformation

Object -> World -> Screen coordinate





Acceleration Data Structures

for

Ray Tracing

Space subdivision

· Normal Grids

· Adaptive Grids

Triangle lists subdivision

· Bounding Volume Hierarchy

Rays spatial subdivision

·Ray space Hierarchy

Motivation

First Hit Problem

Solution

Given a scene defined by a set of N primitives and a ray r, find the closest point of intersection of r with the scene

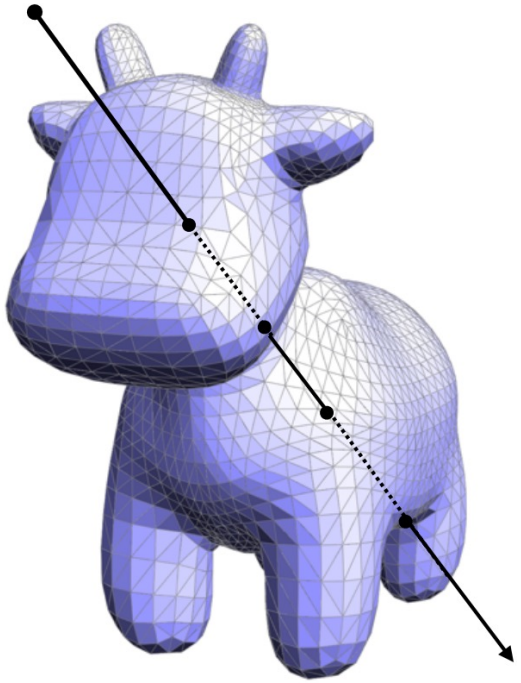
"Find the first primitive the ray hits"

Naïve algorithm?

1. Intersect ray with every triangle
2. Keep the closest hit point

Complexity? $O(N)$

Can we do better?



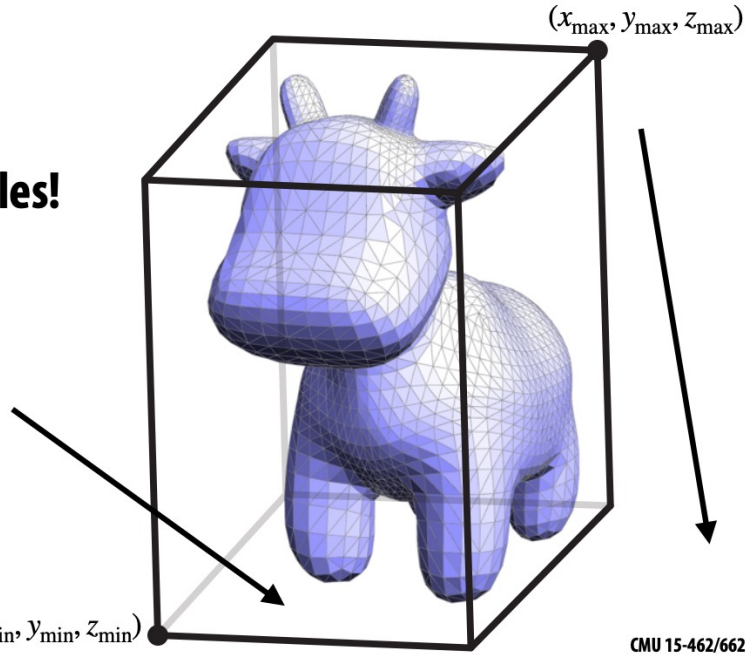
Bounding Box

- Precompute smallest "bounding box" around all primitives
  - Q: How?
  - A: Loop over vertices; keep max/min (x,y,z) coordinates
- Intersect ray with box
  - If it misses, we're done!
  - If it hits…try all triangles!

Did we actually do better?

No! Worst case is still O(N)

(Also: ray-box intersection?)

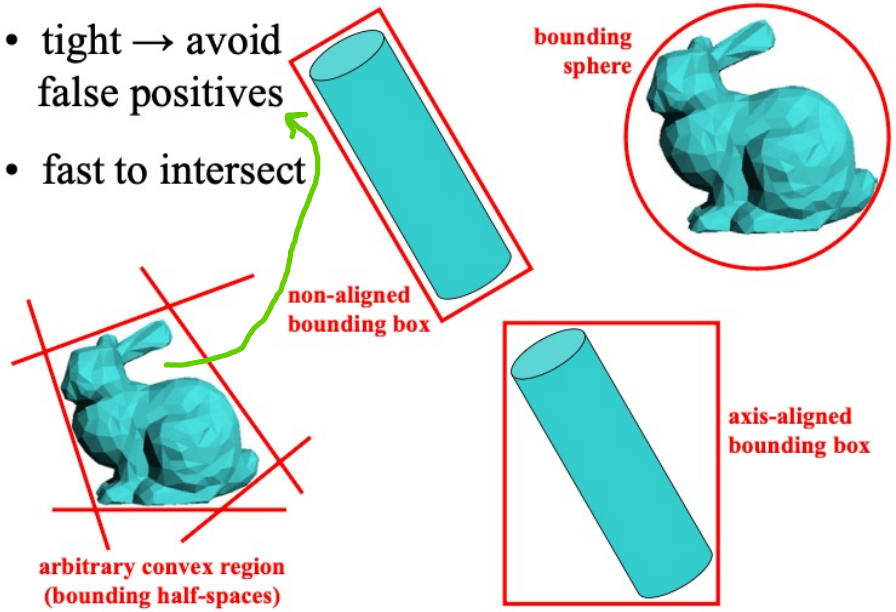


Ray-axis-aligned-box intersection

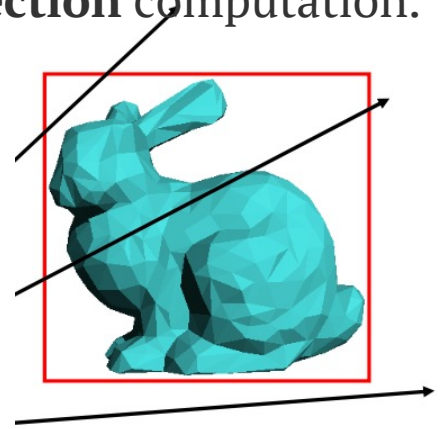
Bounding Volume

a closed volume that completely contains the union of the objects in the set.

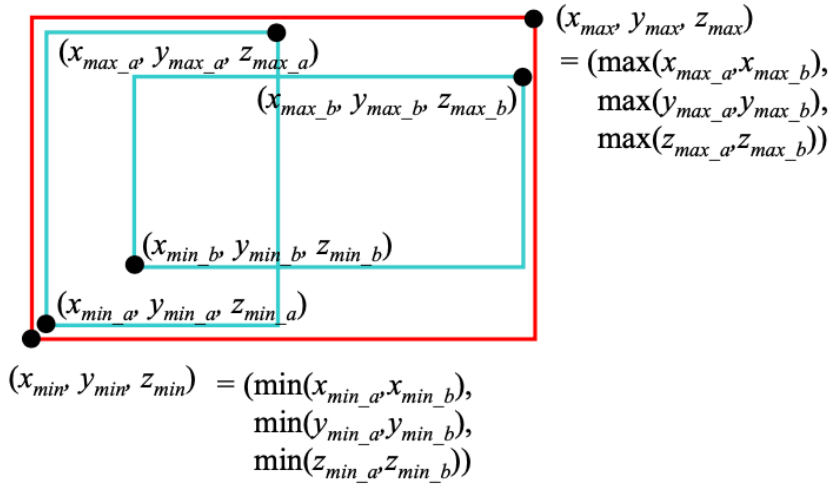
First check for an intersection with a conservative bounding region (like AABB)
Early reject



Axis Aligned **Bounding Box** (AABB) is commonly used due to its **efficiency** of the ray-AABB **intersection** computation.



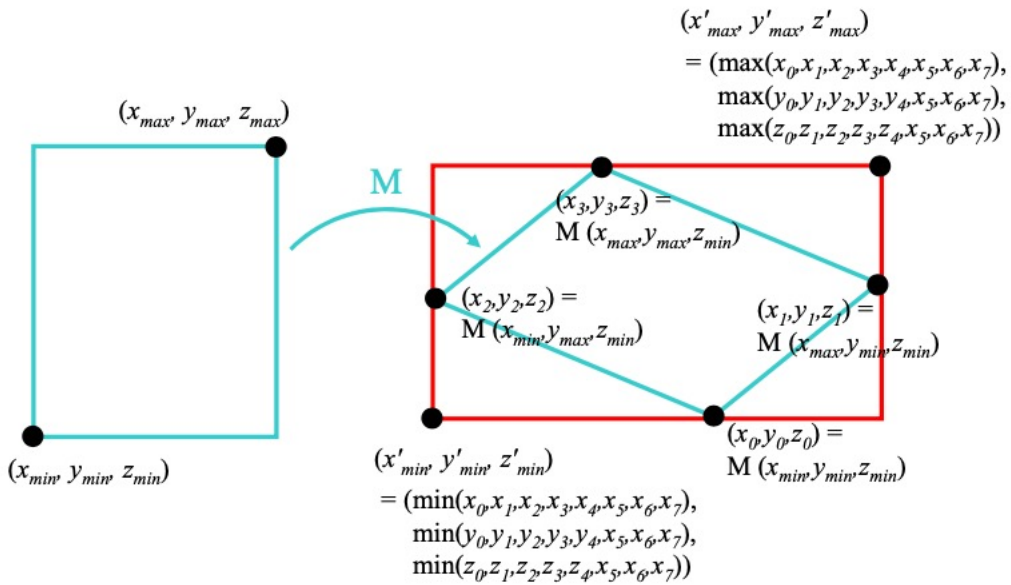
Others: MIT EECS 6.837 Computer Graphics (solid Math intro)
https://groups.csail.mit.edu/graphics/classes/6.837/F04/calendar.html

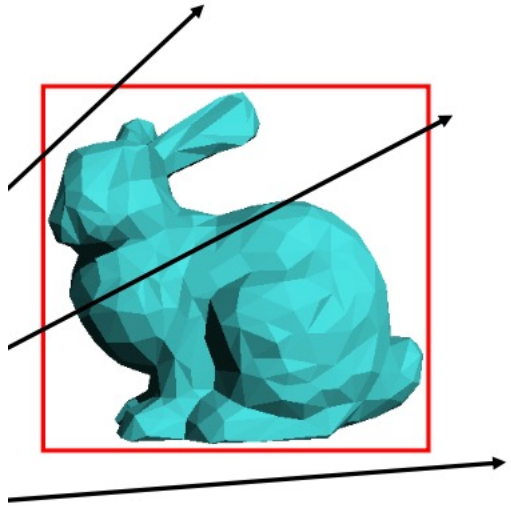


Bounding Boxes

First check for an intersection with a conservative bounding region (AABB)

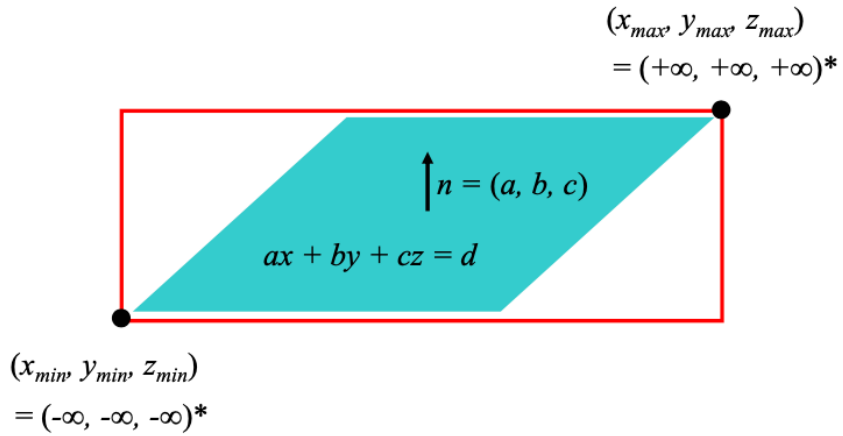
Early reject

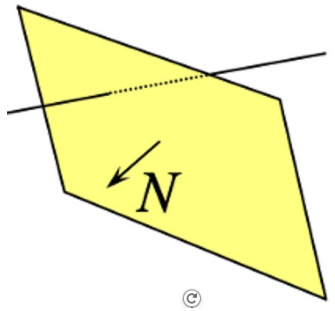




Object Representation

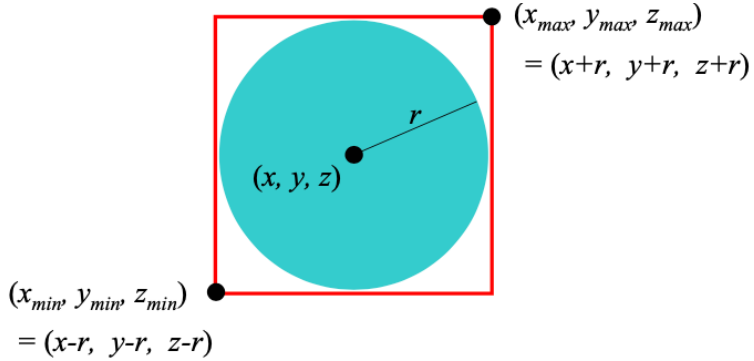
- **Plane**

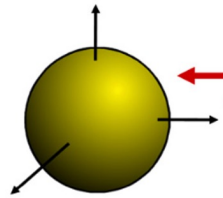




** unless n is exactly perpendicular to an axis*

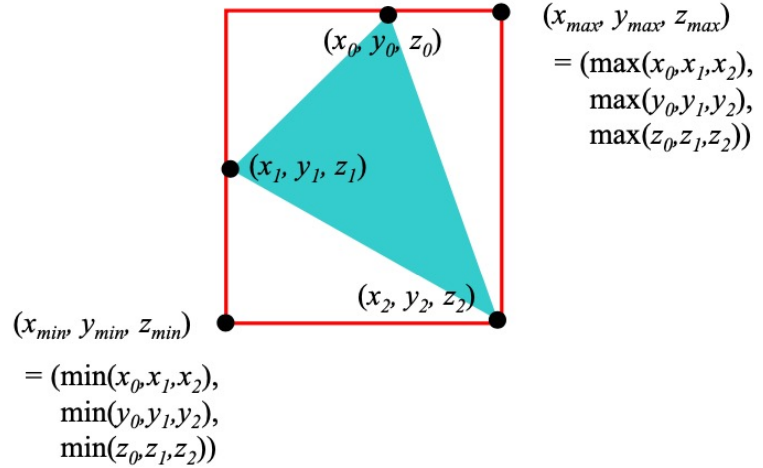
- **Sphere**

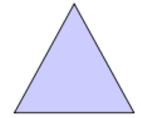




- ~~Cube /AABBs~~

- **Triangle (Frequently USED)**

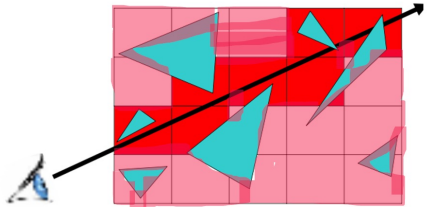


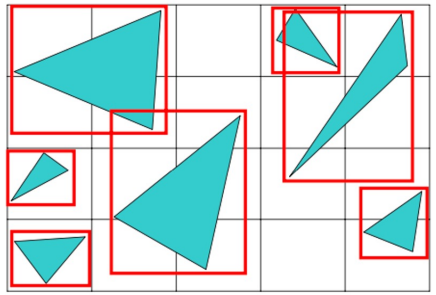


Normal Grids

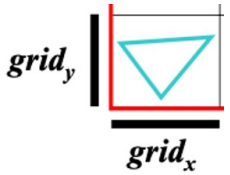
space subdivision

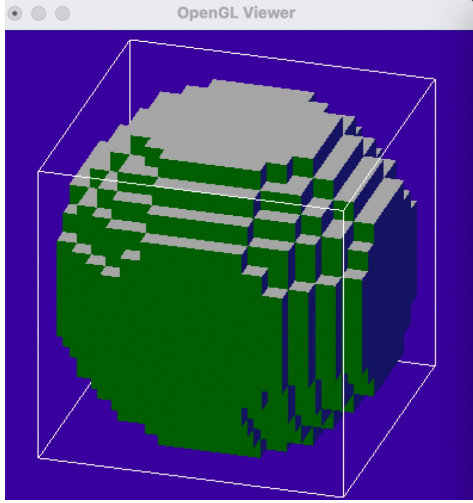
Using *pointers* to store the objects in each cell





resolution: nx * ny * nz





Bounding Box Pre-processing : O(n) *Traverse all the objects*

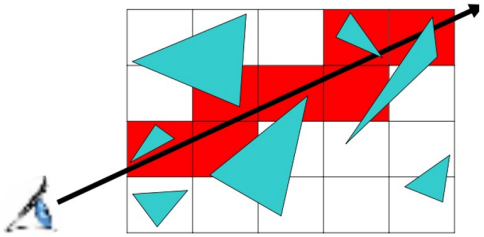
Bounding box of the ***scene***

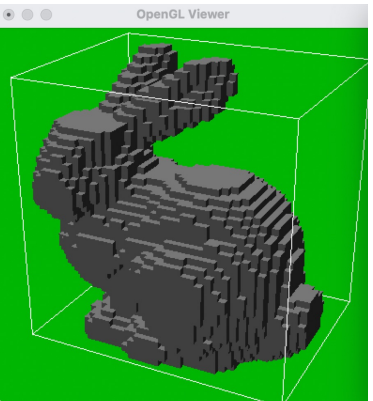
Voxel Rendering: for all grids

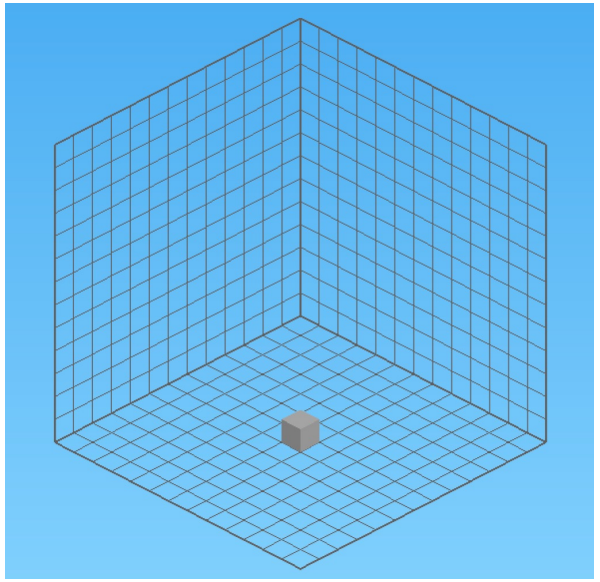
Simple, but nx * ny * nz is huge

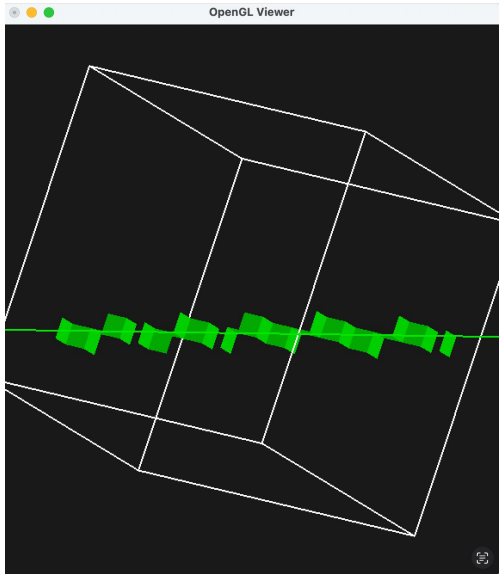
Ray Marching: only cells along the ray

Reduce # intersection !





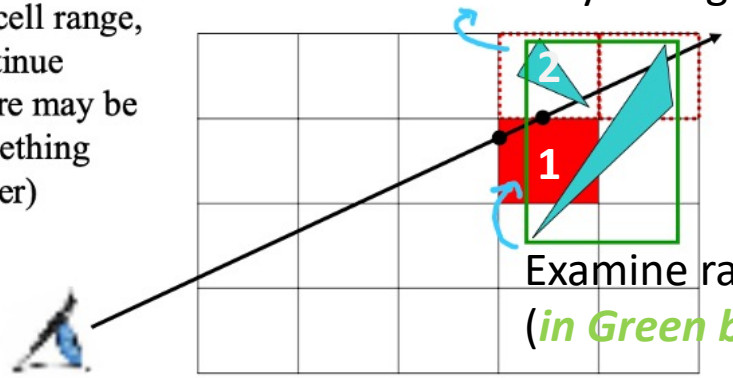




Ray Marching Visualization

- If intersection t is not within the cell range, continue (there may be something closer)

Examine ray-triangle(Red dashed box) intersection



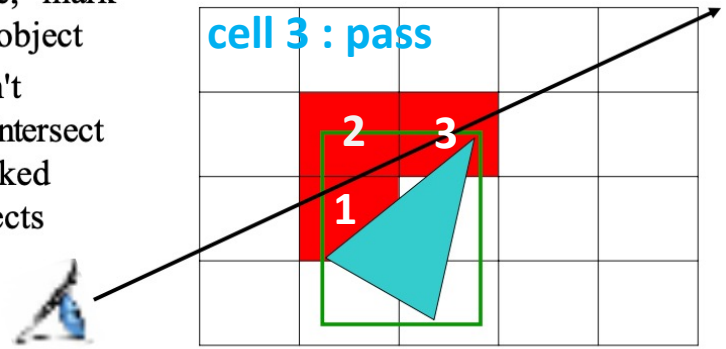
Examine ray-triangle (*in Green box*) intersection

cache the result

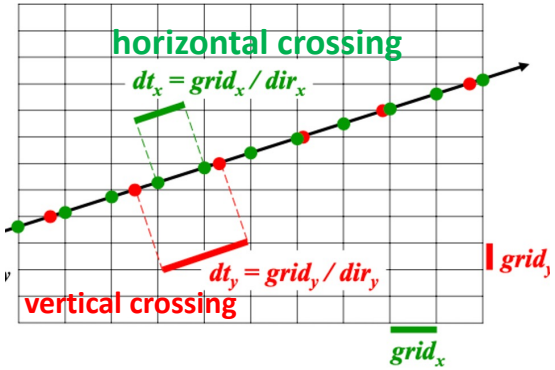
Preventing Repeated Computation

- Perform the computation once, "mark" the object
- Don't re intersect marked objects

cell 1 : not hit with triangle
cell 2 : pass (cache)
cell 3 : pass

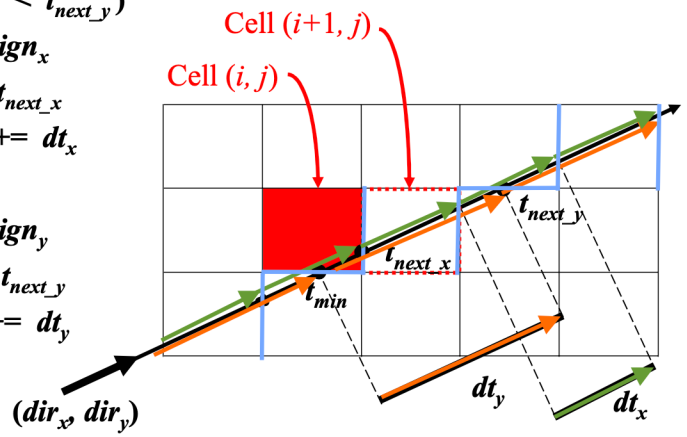


compare t next_x and t next_y



What's the Next Cell?

```
if (t_next_x < t_next_y)
    i += sign_x
    t_min = t_next_x
    t_next_x += dt_x
else
    j += sign_y
    t_min = t_next_y
    t_next_y += dt_y
```



Pseudo-Code

```
create grid
insert primitives into grid
for each ray r
  find initial cell c(i,j), t_min, t_next_x & t_next_y
  compute dt_x, dt_y, sign_x and sign_y
  while c != NULL
    for each primitive p in c
      intersect r with p
      if intersection in range found
        return
    c = find next cell
```

Reduce # primitives!

3D Digital differential analyser

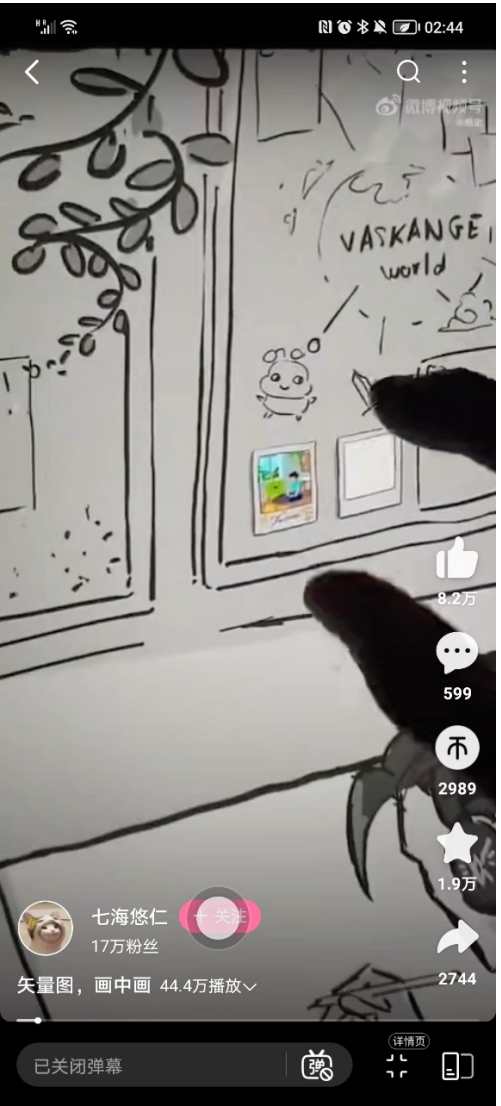
if $(dir_x > 0)$ $sign_x = 1$ *else* $sign_x = -1$
if $(dir_y > 0)$ $sign_y = 1$ *else* $sign_y = -1$

https://en.wikipedia.org/wiki/Digital_differential_analyzer_(graphics_algorithm)

similar: Bresenham's line algorithm

https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm

Sampling: small objects being missed completely

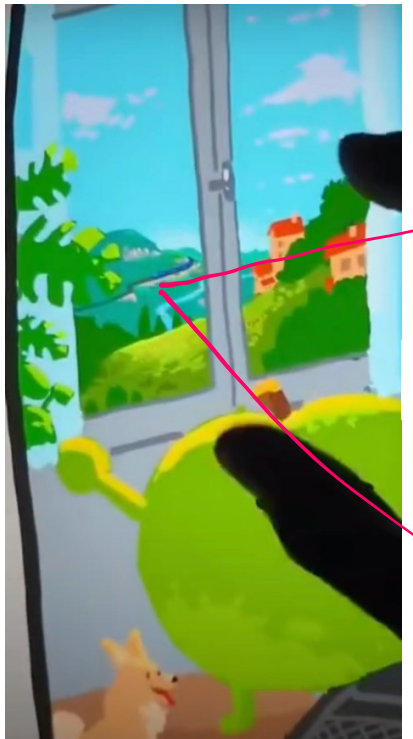


Disadvantage of Uniform Grids

Inflexible & "Teapot in a Stadium" problem

The difficulty to render small-scale details within a very large scene

large scene



small-scale

Disadvantage of Uniform Grids

Inflexible & "Teapot in a Stadium" problem

The difficulty to render small-scale details within a very large scene

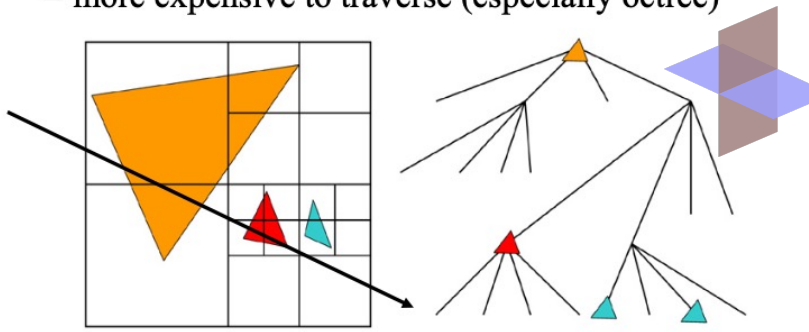
Adaptive Grids

spatial subdivision

space-partitioning

The ray traversal of k-dimensional tree implemented efficiently: a local **stack**.
Unfortunately, graphics hw doesn't have such on each element (vertex or pixel).
Solution: stack-less kD-tree traversal methods kD-restart and kD-backtrack.
by Foley et al. proposed.

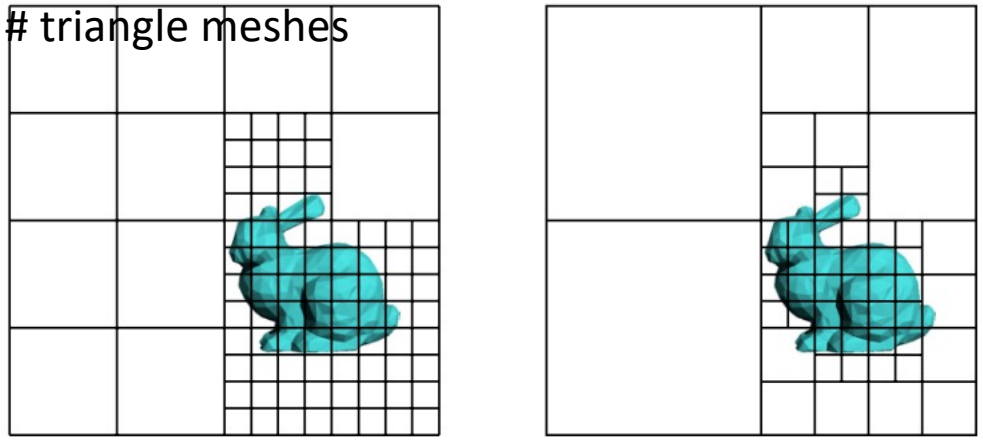
- Advantages?
  - grid complexity matches geometric density
- Disadvantages?
  - more expensive to traverse (especially octree)



Adaptive Grids

- Subdivide until each cell contains no more than n elements, or maximum depth d is reached

triangle meshes

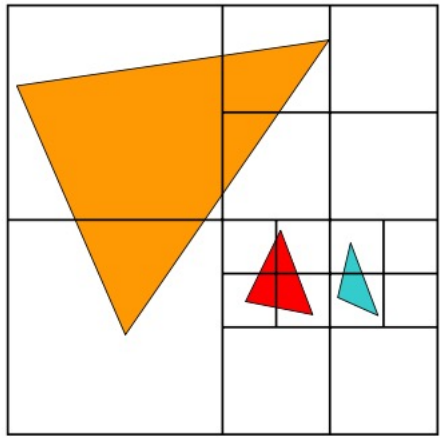


Nested Grids

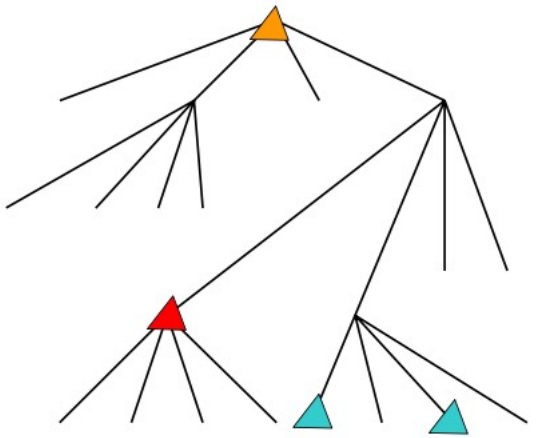
Octree/(Quadtree)

Primitives in an Adaptive Grid

- Can live at intermediate levels, or be pushed to lowest level of grid

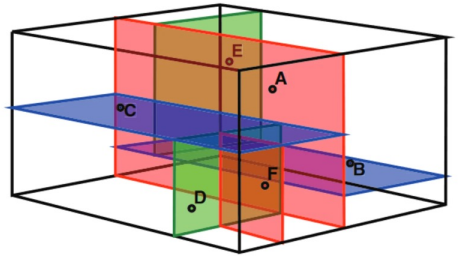


Octree/(Quadtree)

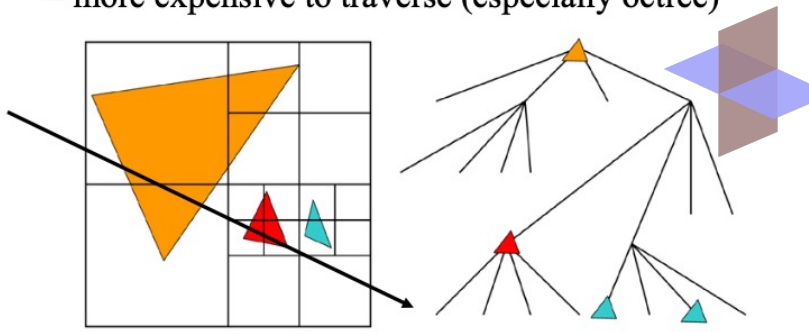


space-partitioning

The ray traversal of k-dimensional tree implemented efficiently: a local **stack**.
Unfortunately, graphics hw doesn't have such on each element (vertex or pixel).
Solution: stack-less kD-tree traversal methods kD-restart and kD-backtrack.
by Foley et al. proposed.

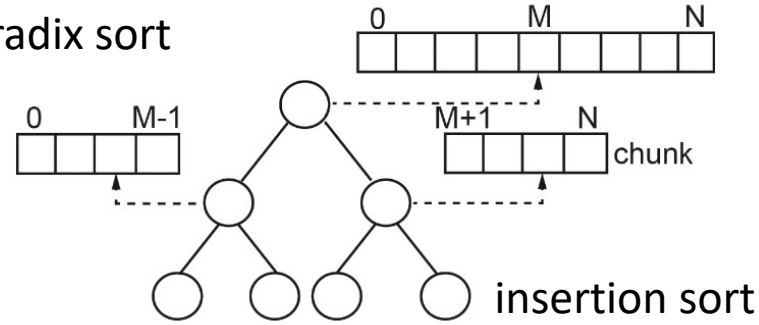


- Advantages?
  - grid complexity matches geometric density
- Disadvantages?
  - more expensive to traverse (especially octree)

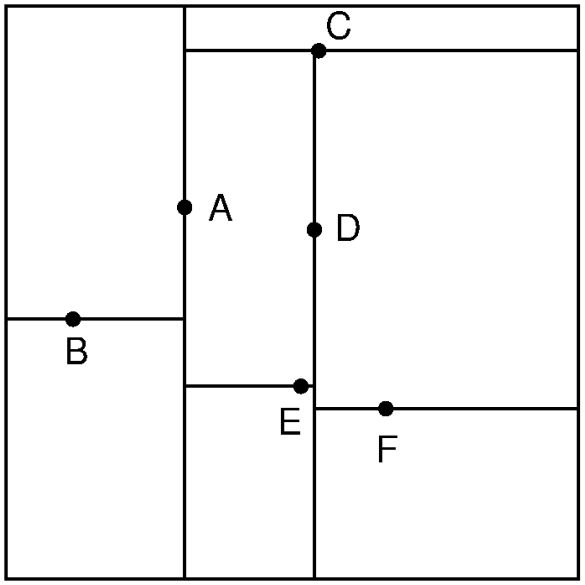


Adaptive Grids

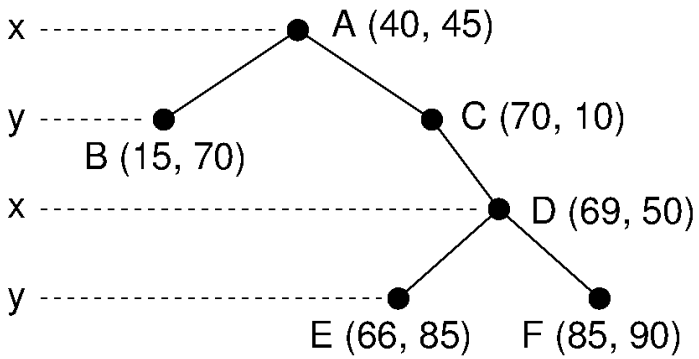
linear-time radix sort



insertion sort



(a)



(b)

Fig. 2. Level 1 handles the entire array, and splits on the median between 0 and N. Nodes in level 2 each handle corresponding sub-arrays ("chunks") from their parent. This pattern continues until each node only handles a single element.

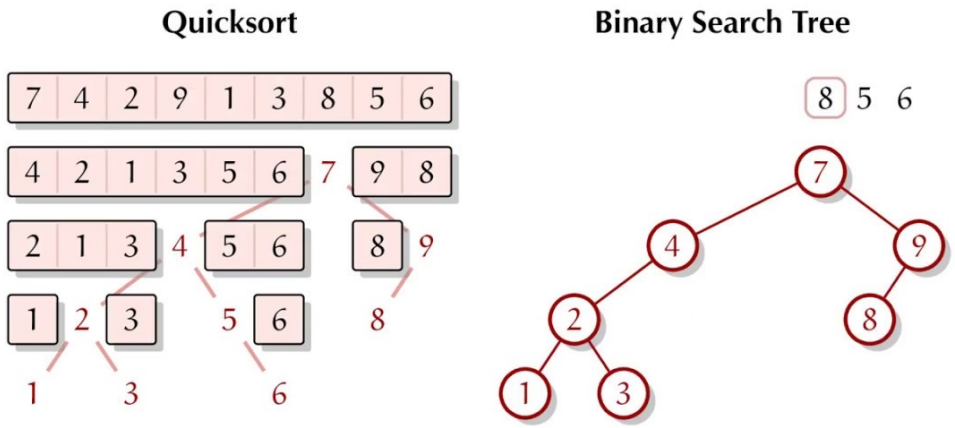
Parallel kd-Tree Construction on the GPU with an Adaptive Split and Sort Strategy
David Wehr ∗ and Rafael Radkowski †

Uniform Grids Adaptive Grids

*subdivide **space** into smaller sub-spaces*

Bounding Volume Hierarchy

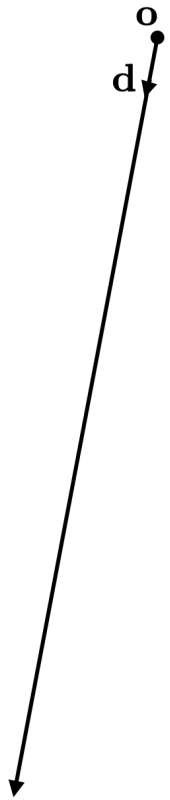
*subdivide a list of **triangles** into smaller subsets*



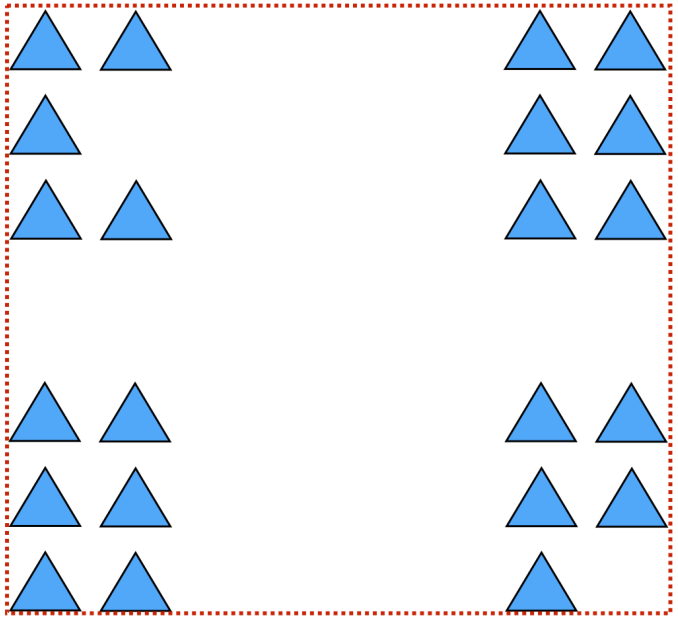
find the k-th largest number in an array

Simple case

Pre-processing
Bounding Box *Traverse all the objects* O(n)

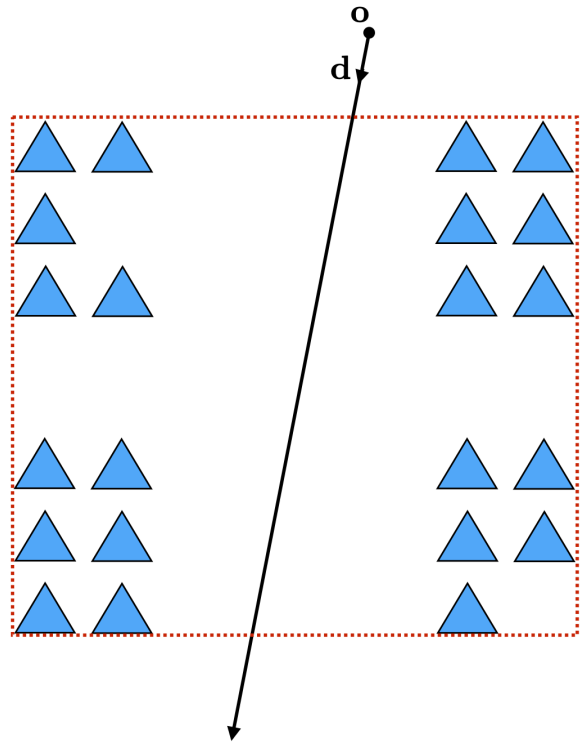


Ray misses bounding box of all primitives in scene



Cost (misses box):
preprocessing: O(n)
ray-box test: O(1)
amortized cost*: O(1)

Another (should be) simple case



Cost (hits box):
preprocessing: O(n)
ray-box test: O(1)
triangle tests: O(n)
amortized cost*: O(n)

Still no better than naïve algorithm (test all triangles)!

***over many ray-scene intersection tests**

Q: How can we do better?

A: Apply this strategy hierarchically.

- Find bounding box of objects
- Split objects into two groups
- Recurse

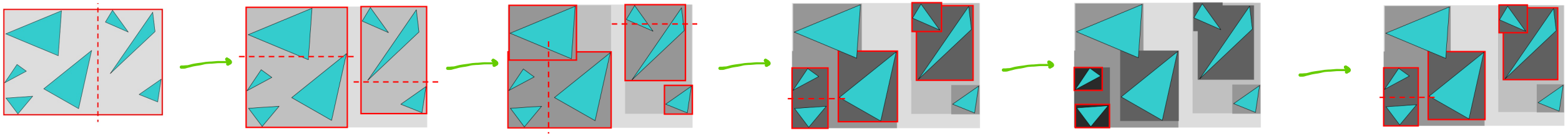
BVH

Construction

is the process of partition

Where to split objects?

- At midpoint *OR*
- Sort, and put half of the objects on each side *OR*
- Use modeling hierarchy



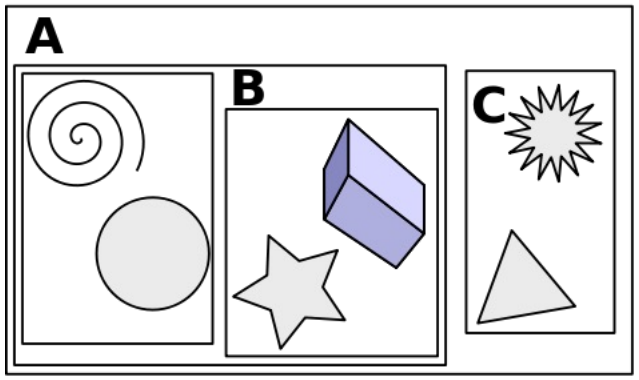
Bounding Volume Hierarchy

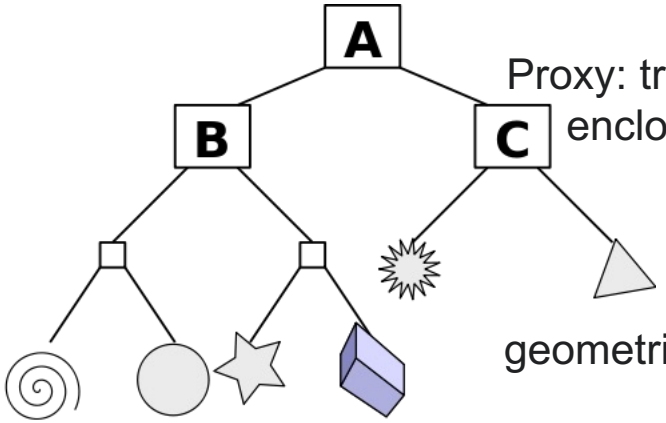
A tree structure on a set of geometric objects

Early reject

Intersection Test Performed : **O(log n)**

constrained by tree height.





Proxy: triangles grouped as small sets enclosed in larger bounding volumes

wrapped in bounding volumes

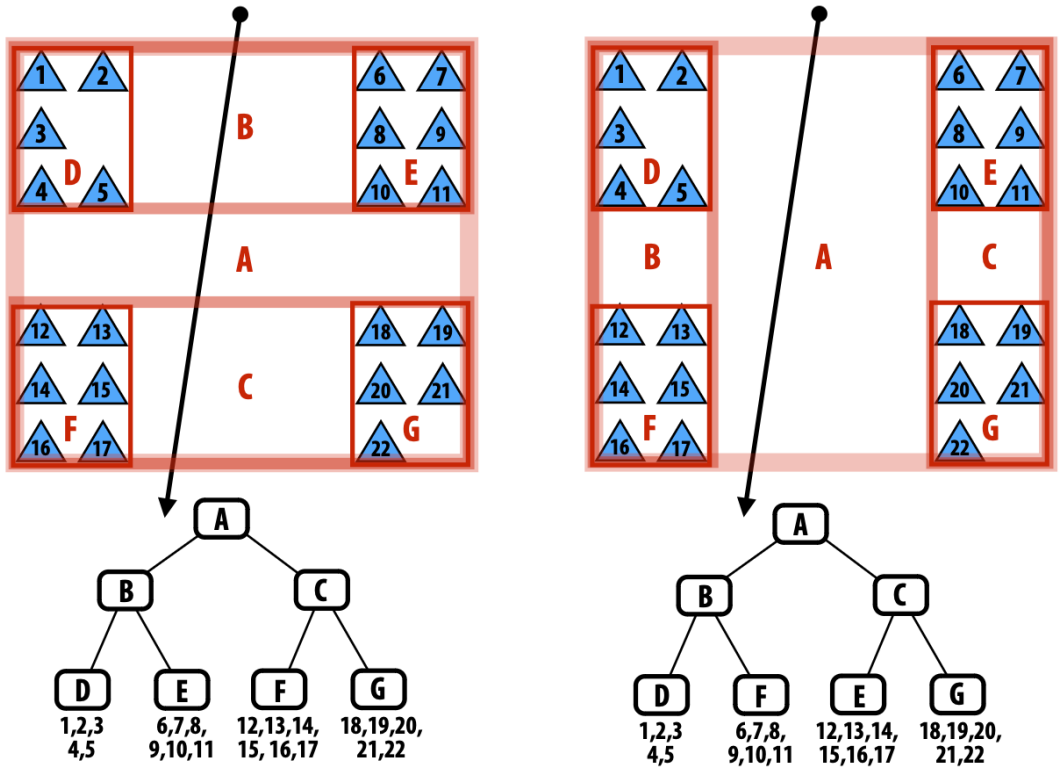
geometric objects

https://en.wikipedia.org/wiki/Bounding_volume_hierarchy

also used in collision detection

Bounding volume hierarchy (BVH)

- **Leaf nodes:**
  - **Contain small list of primitives**
- **Interior nodes:**
  - **Proxy for a large subset of primitives** Pretending the capture the geometry of all primitives it contains
  - **Stores bounding box for all primitives in subtree**



Left: two different BVH organizations of the same scene containing 22 primitives.

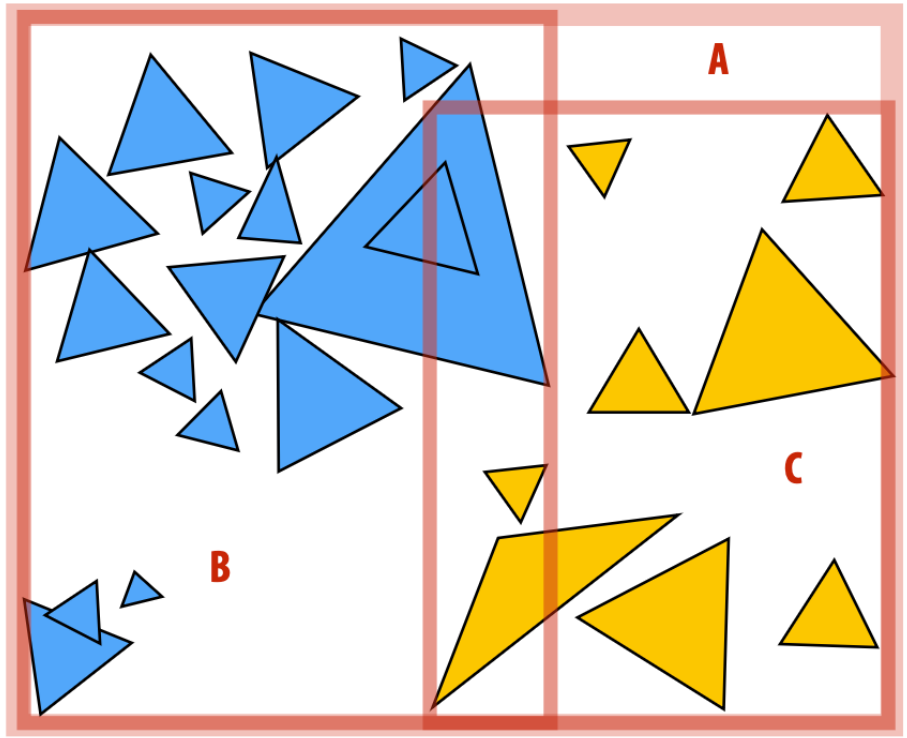
Is one BVH better than the other?

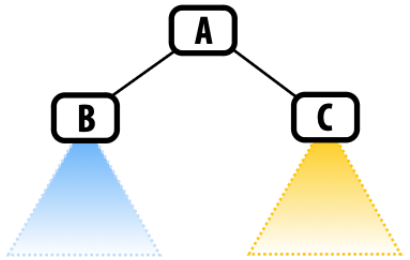
```
struct BVHNode {
   bool leaf;  // am I a leaf node?
   BBox bbox;  // min/max coords of enclosed primitives
   BVHNode* child1; // "left" child (could be NULL)
   BVHNode* child2; // "right" child (could be NULL)
   Primitive* primList; // for leaves, stores primitives
};
```


15462.courses.cs.cmu.edu/spring2022content/lectures/13_spatial/13_spatial_slides.pdf

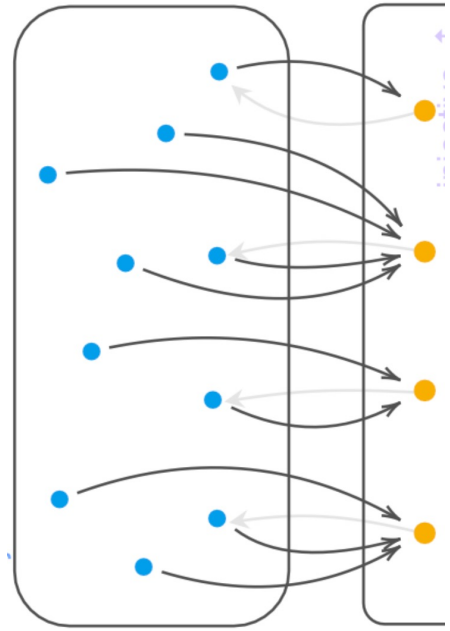
Another BVH example

- **BVH partitions each node's primitives into disjoints sets**
  - **Note: The sets can still be overlapping in space (below: child bounding boxes may overlap in space)**





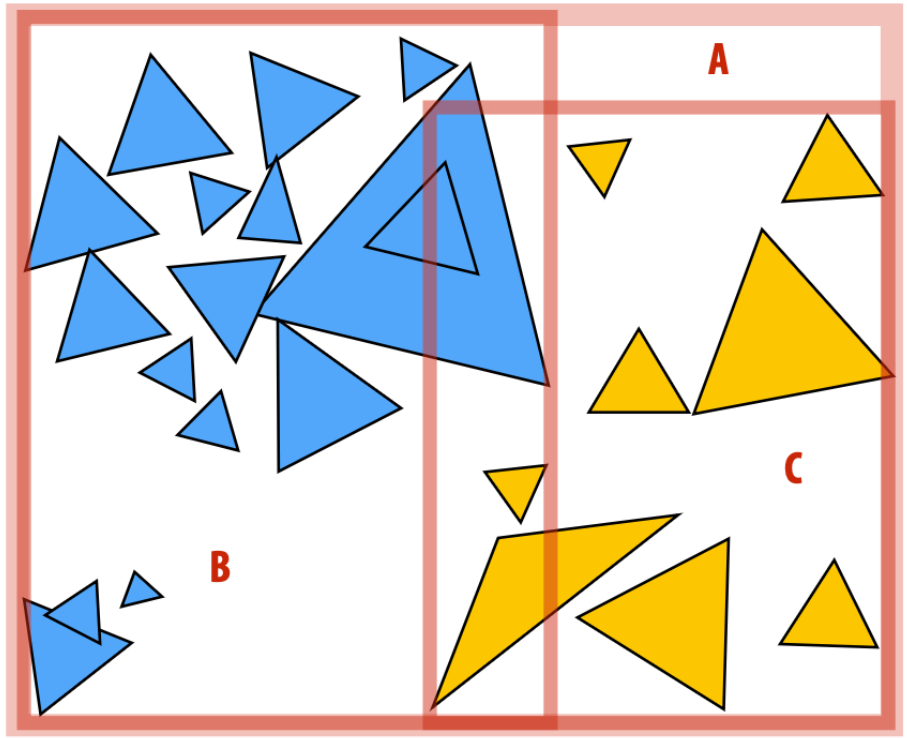
Mathematically: (Set Theory)
Having Triangle lists, Bounding Box Sets
Find a surjection onto the triangle
Triangle lists ↠ Bounding Box

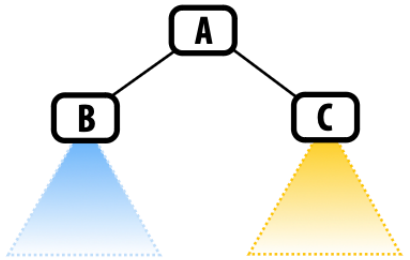


15462.courses.cs.cmu.edu/spring2022content/lectures/13_spatial/13_spatial_slides.pdf

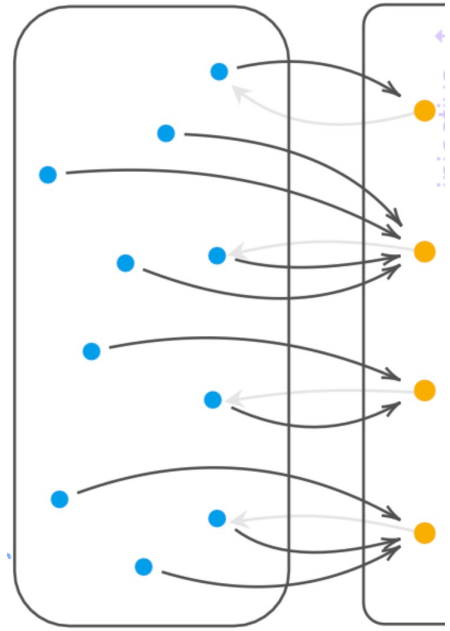
Another BVH example

- **BVH partitions each node's primitives into disjoints sets**
  - **Note: The sets can still be overlapping in space (below: child bounding boxes may overlap in space)**





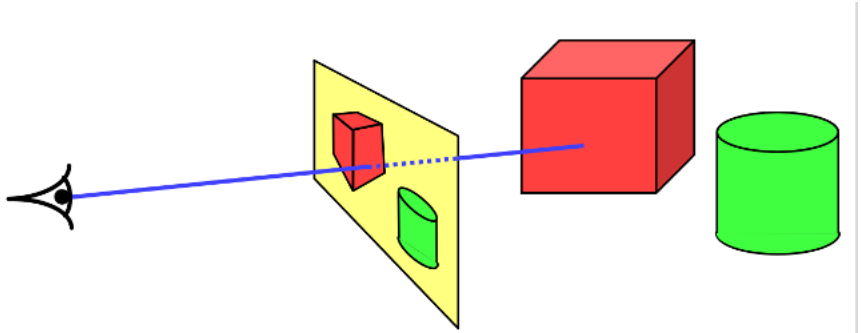
Mathematically: (Set Theory)
Having Triangle lists, Bounding Box Sets
Find a surjection onto the triangle
Triangle lists ↠ Bounding Box



15462.courses.cs.cmu.edu/spring2022content/lectures/13_spatial/13_spatial_slides.pdf

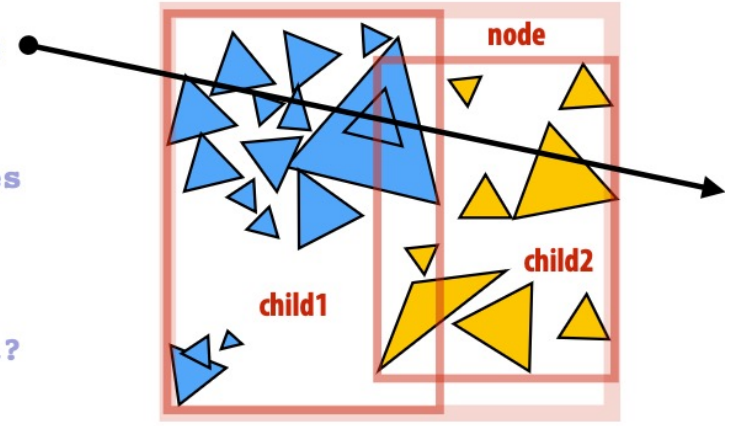
Coding: Step by step explanation

BVH Algorithm



Ray-scene intersection using a BVH

```
struct BVHNode {
   bool leaf;  // am I a leaf node?
   BBox bbox;  // min/max coords of enclosed primitives
   BVHNode* child1; // "left" child (could be NULL)
   BVHNode* child2; // "right" child (could be NULL)
   Primitive* primList; // for leaves, stores primitives
};

struct HitInfo {
   Primitive* prim;  // which primitive did the ray hit?
   float t;   // at what t value?
};

void find_closest_hit(Ray* ray, BVHNode* node, HitInfo* closest) {
   HitInfo hit = intersect(ray, node->bbox);  // test ray against node's bounding box
   if (hit.prim == NULL || hit.t > closest.t))
      return; // don't update the hit record

   if (node->leaf) {
      for (each primitive p in node->primList) {
         hit = intersect(ray, p);
         if (hit.prim != NULL && hit.t < closest.t) {
            closest.prim = p;
            closest.t = t;
         }
      }
   } else {
      find_closest_hit(ray, node->child1, closest);
      find_closest_hit(ray, node->child2, closest);
   }}
```



If LEAF_NODE
Intersect RAY with all the primitives (tri)
in this BB

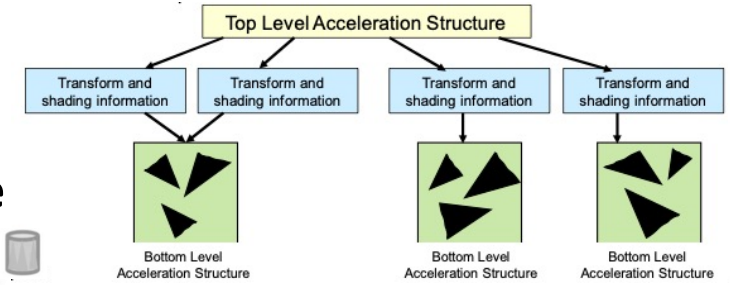
If NOT
Do the binary partition

Top Level **Acceleration Structure**

A set of instances of ***BLAS***

Bottom Level **Acceleration Structure**

A set of geometries of same type



Given a mesh with a BVH built in object space and 4×4 matrix that stores orientation and translation for the mesh, we can intersect its BVH in world space by applying the ***inverse transform*** *to the rays. [Fixed, Easier]*

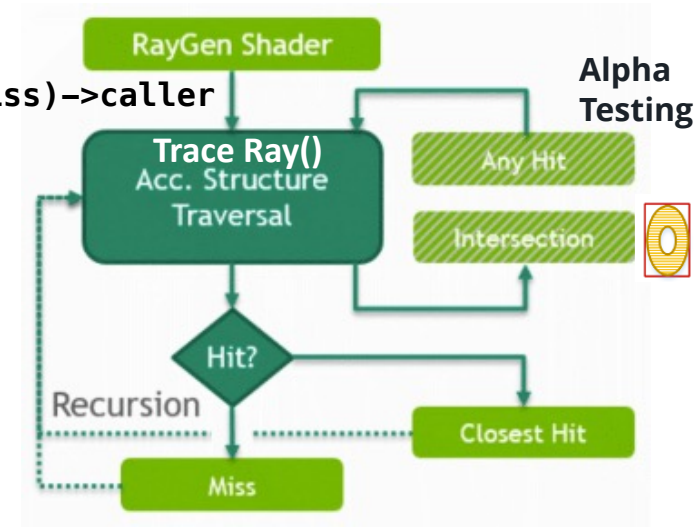
$$P(t) = R_o + t * R_d$$

TLAS *also includes*

An inverse matrix transform

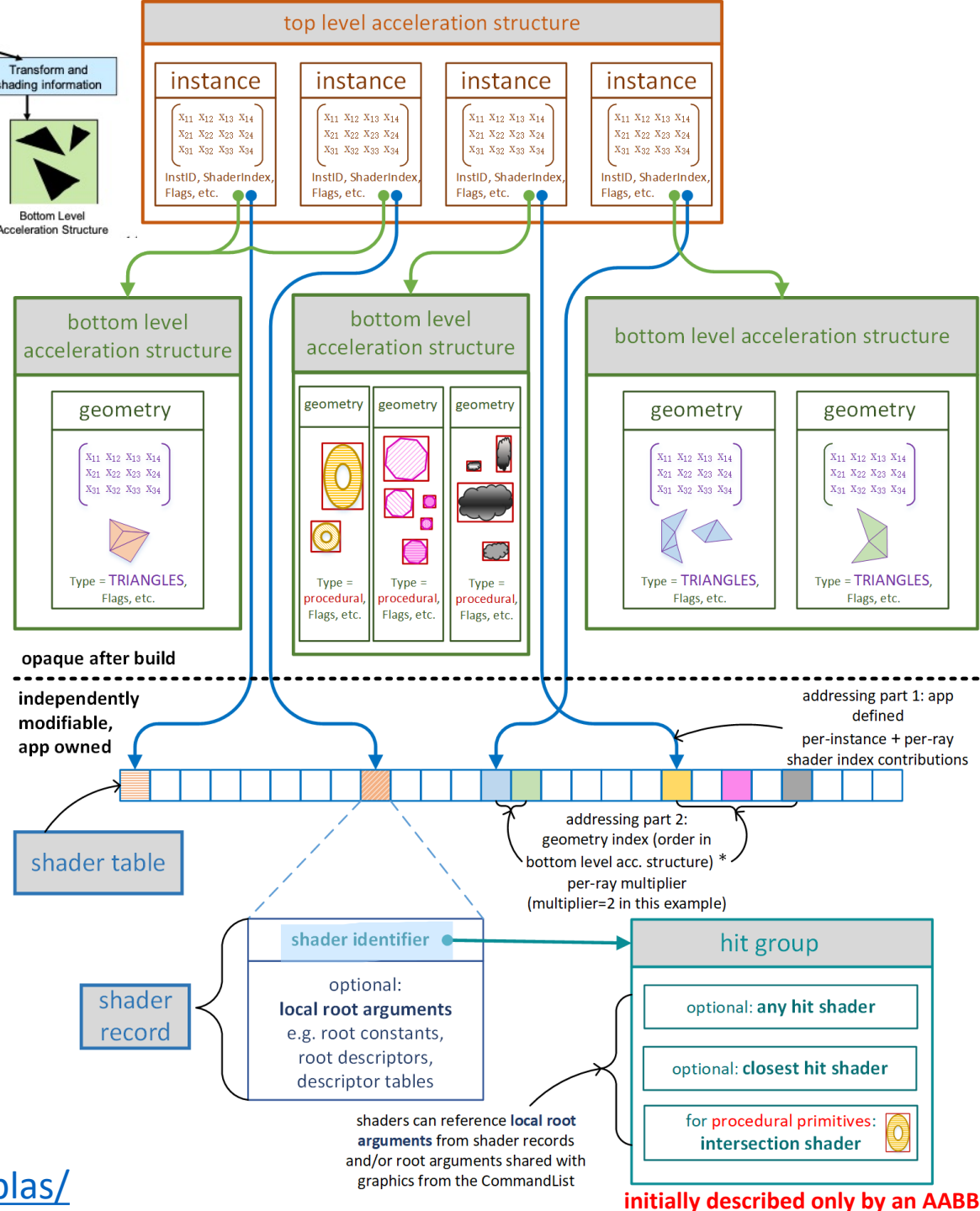
Unique instance ID to shaders (user defined)

```
caller->anyhit->(closesthit|miss)->caller
          ^  |
Usual way |__|
```



Alpha Testing

Environment Map



https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html

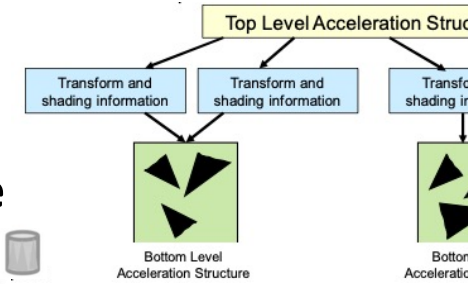
https://jacco.ompf2.com/2022/05/07/how-to-build-a-bvh-part-5-tlas-blas/

Top Level **A**cceleration **S**tructure

A set of instances of ***BLAS***

Bottom Level **A**cceleration **S**tructure

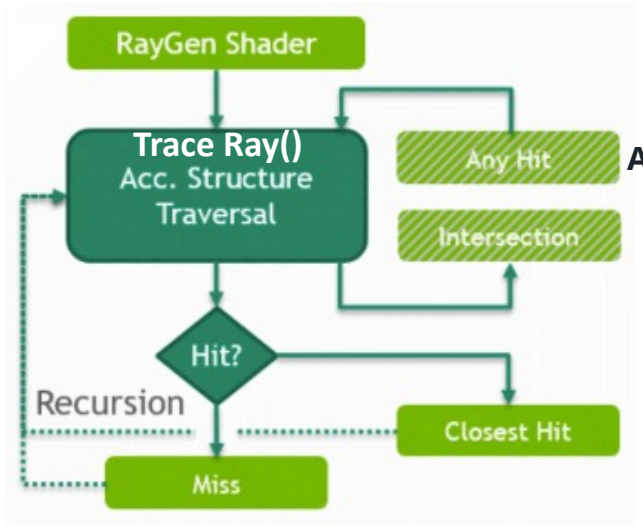
A set of geometries of same type



TLAS *also includes*

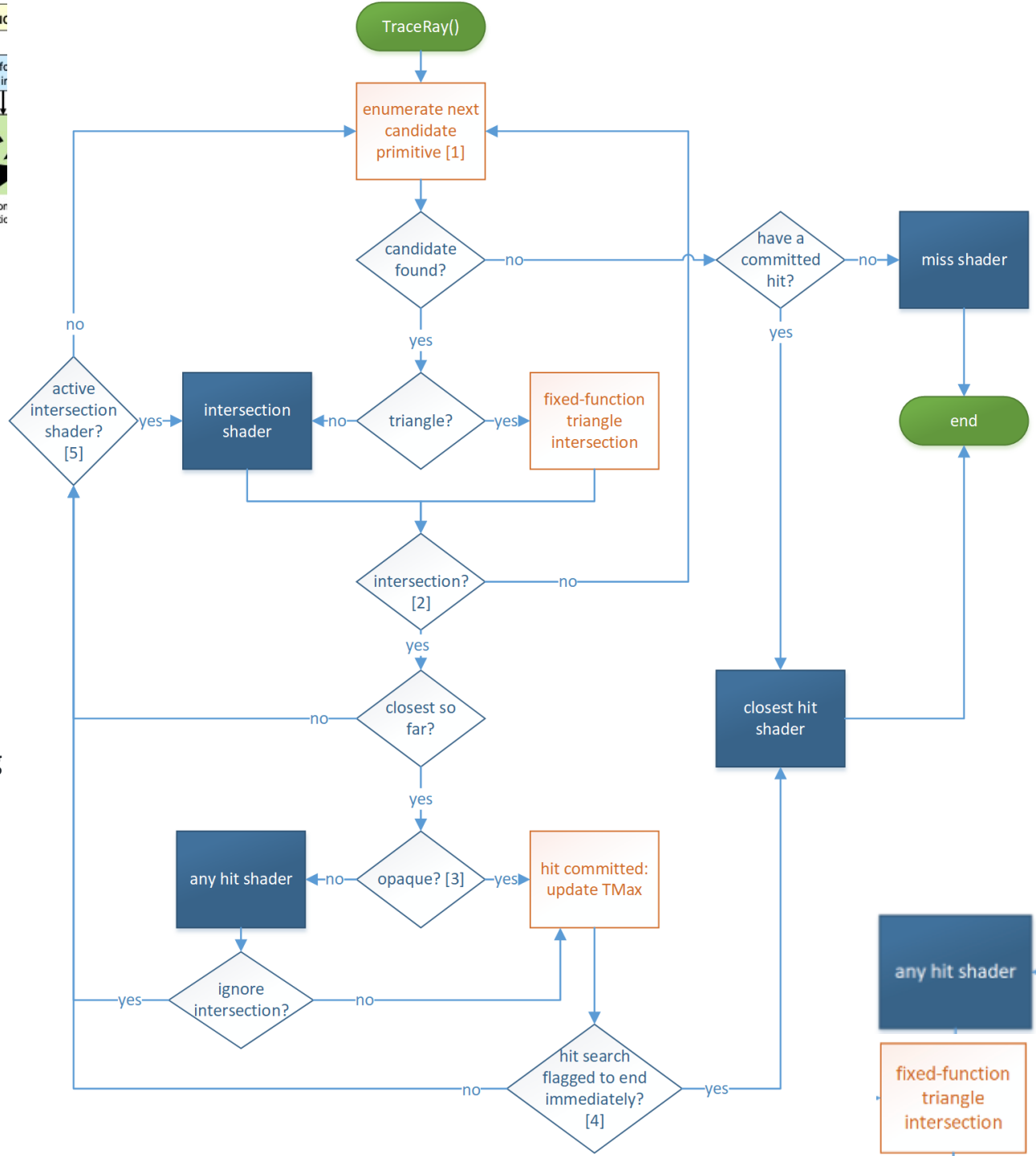
An inverse matrix transform

Unique instance ID to shaders (user defined)



Alpha testing

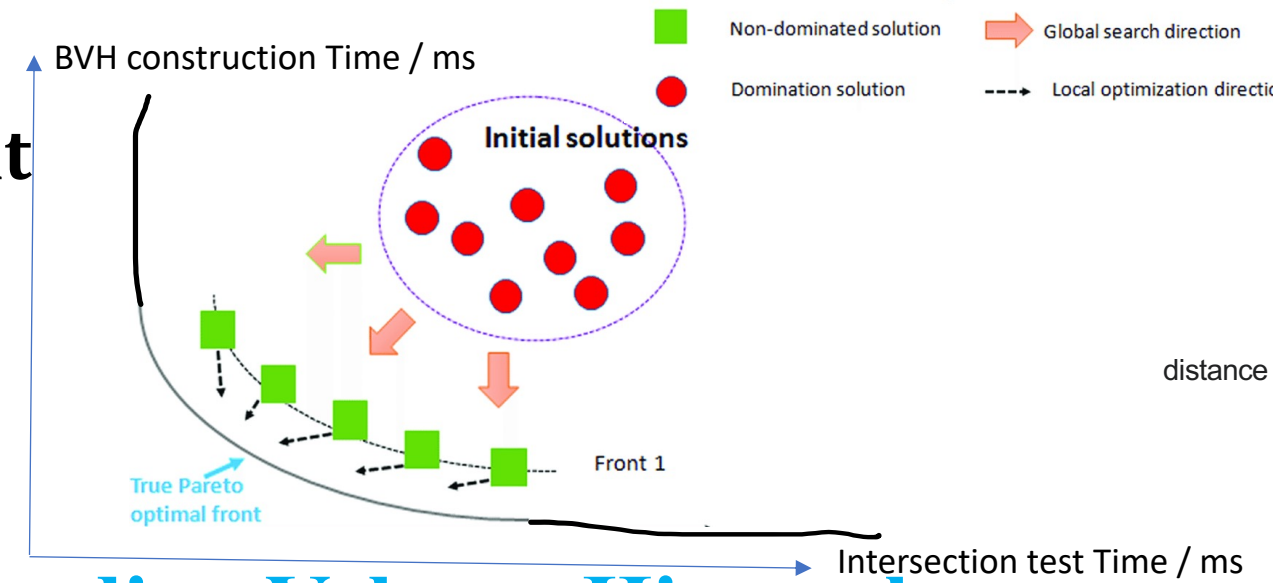
Environment Map

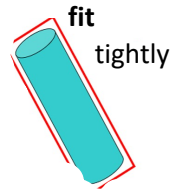


https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html

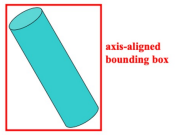
BVH design issues & Potential Improvement

Multi-objective / Pareto optimization
more than one objective function





simple shape;
intersection tests and
distance computations are simple and fast



Bounding Volume Hierarchy

a tree structure on a set of geometric objects

Build a BVH
Improvement on Ray traversal

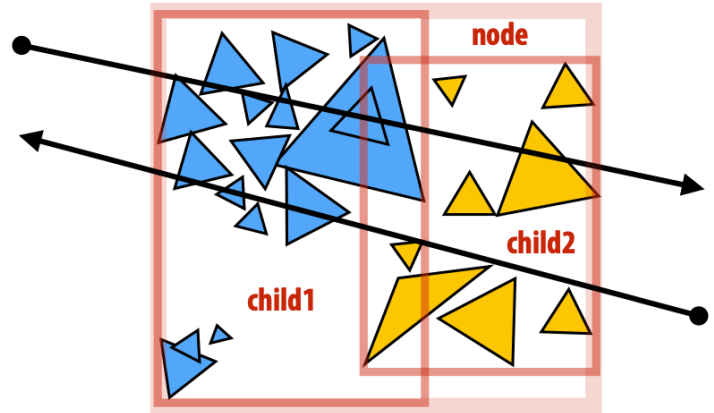
https://en.wikipedia.org/wiki/Bounding_volume_hierarchy

also used in collision detection

Improvement: "front-to-back" traversal

General strategy for improving performance:

Do traversal in a way that is likely to terminate "early"




```
void find_closest_hit(Ray* ray, BVHNode* node, HitInfo* closest)
{
   if (node->leaf) {
      // same as before
   } else {
      HitInfo hit1 = intersect(ray, node->child1->bbox);
      HitInfo hit2 = intersect(ray, node->child2->bbox);

      NVHNode* first = (hit1.t <= hit2.t) ? child1 : child2;
      NVHNode* second = (hit1.t <= hit2.t) ? child2 : child1;
      HitInfo secondHit = (hit1.t <= hit2.t) ? hit2 : hit1;

      find_closest_hit(ray, first, closest);
      if (secondHit.t < closest.t)
         find_closest_hit(ray, second, closest); // why might we still need to do this?
   }
```

"Front to back" traversal. Traverse to closest child node first. Why?

prune out branches

First Hit
Though not every time, but is the usual case that front will be the First Hit

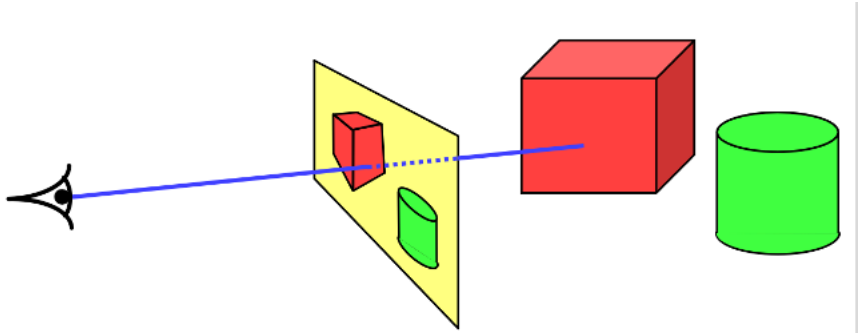
Because different BB overlap in space
(though in terms of triangles list is disjoint)

15462.courses.cs.cmu.edu/spring2022content/lectures/13_spatial/13_spatial_slides.pdf

Due to scene locality,
AI solution could have a go

Part 4: SAH Algorithm

Good Partition Principle, SAH, other advanced ideas



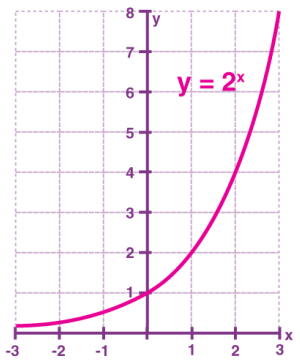
Other strategy for improving performance: Build a "better" BVH!

But for a given set of primitives, there are many possible BVHs…

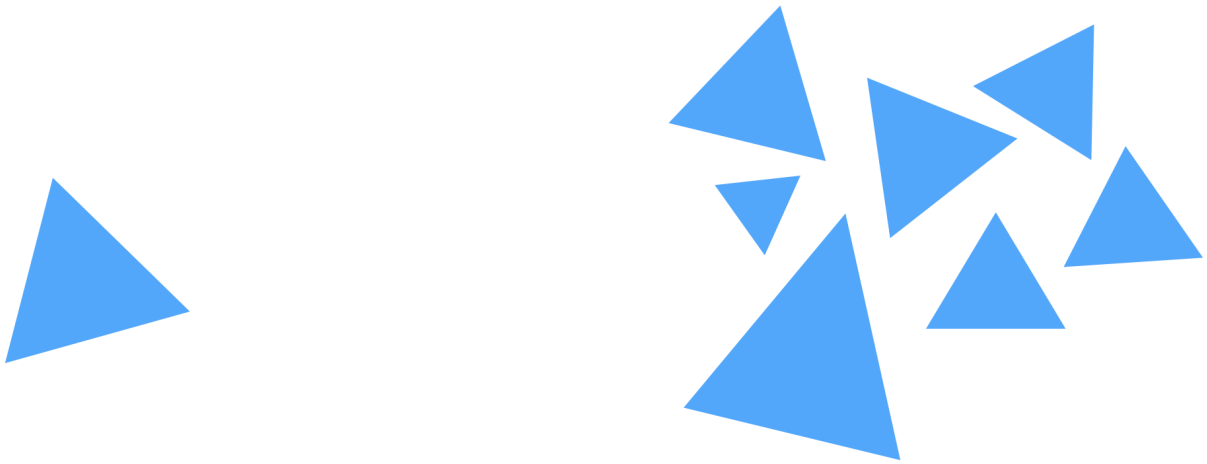
(2^N/2 ways to partition N primitives into two groups)

Exponential is horrible!

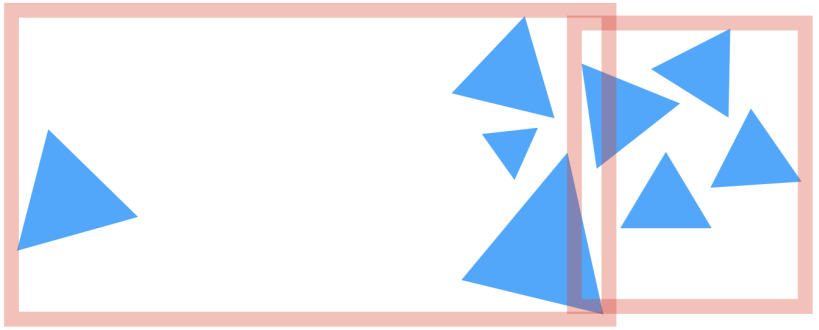
Q: How do we quickly build a high-quality BVH?



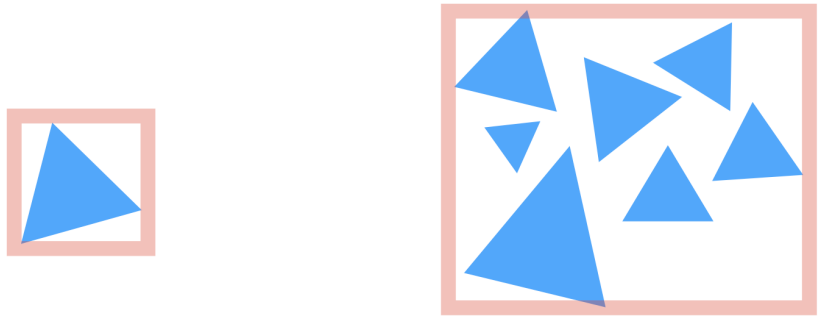
Good Partition Principle 2^{N-1} in total



False positive ✓



Partition into child nodes with equal numbers of primitives



Better partition

Intuition: want small bounding boxes (minimize overlap between children, avoid empty space)

15462.courses.cs.cmu.edu/spring2022content/lectures/13_spatial/13_spatial_slides.pdf

Good Partition Principle 2^{N-1} in total

A good partitioning minimizes the cost of finding the closest intersection of a ray with primitives in the node.

EASY CASE—for a leaf node:

$$C = \sum_{i=1}^{N} C_{\text{isect}}(i)$$

Where $C_{\text{isect}}(i)$ is the cost of ray-primitive intersection for primitive i in the node.

$$= NC_{\text{isect}}$$

(Common to assume all primitives have the same cost)

15462.courses.cs.cmu.edu/spring2022content/lectures/13_spatial/13_spatial_slides.pdf

Good Partition Principle 2^{N-1} in total

HARDER CASE—the expected cost of intersecting an interior node, given that the node's primitives are partitioned into child sets A and B:

$$C = C_{\text{trav}} + p_A C_A + p_B C_B$$

C_{trav} **is the cost of traversing an interior node (e.g., bounding box test)**

C_A **and** C_B **are the costs of intersection with the resultant child subtrees**

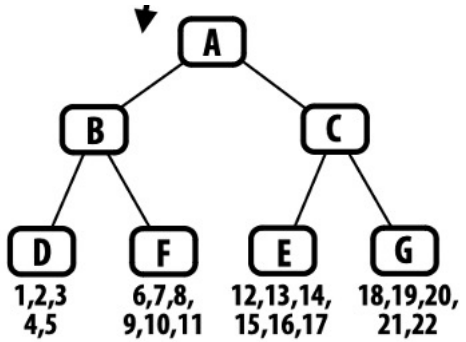
p_A **and** p_B **are the probability a ray intersects the bbox of the child nodes A and B**

Primitive count is common heuristic for child node costs:

$$C = C_{\text{trav}} + p_A N_A C_{\text{isect}} + p_B N_B C_{\text{isect}}$$

Remaining question: how do we get the probabilities p_A, p_B?

N_A the # primitives all the way down, including leaves



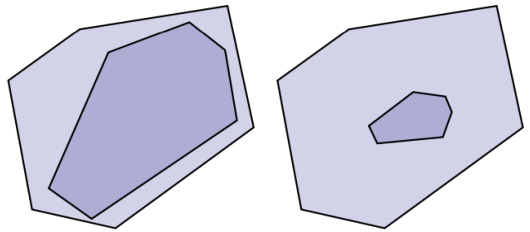
15462.courses.cs.cmu.edu/spring2022content/lectures/13_spatial/13_spatial_slides.pdf

Good Partition Principle 2^{N-1} in total

Estimating probabilities

- **For convex object A inside convex object B, the probability that a random ray that hits B also hits A is given by the ratio of the surface areas S_A and S_B of these objects.**

$$P(\text{hit}A|\text{hit}B) = \frac{S_A}{S_B}$$



Leads to surface area heuristic (SAH):

$$C = C_{\text{trav}} + \frac{S_A}{S_N} N_A C_{\text{isect}} + \frac{S_B}{S_N} N_B C_{\text{isect}}$$

Assumptions of the SAH (which may not hold in practice!):

- **Rays are randomly distributed**
- **No occlusion (i.e., one object blocking another)**

heuristic

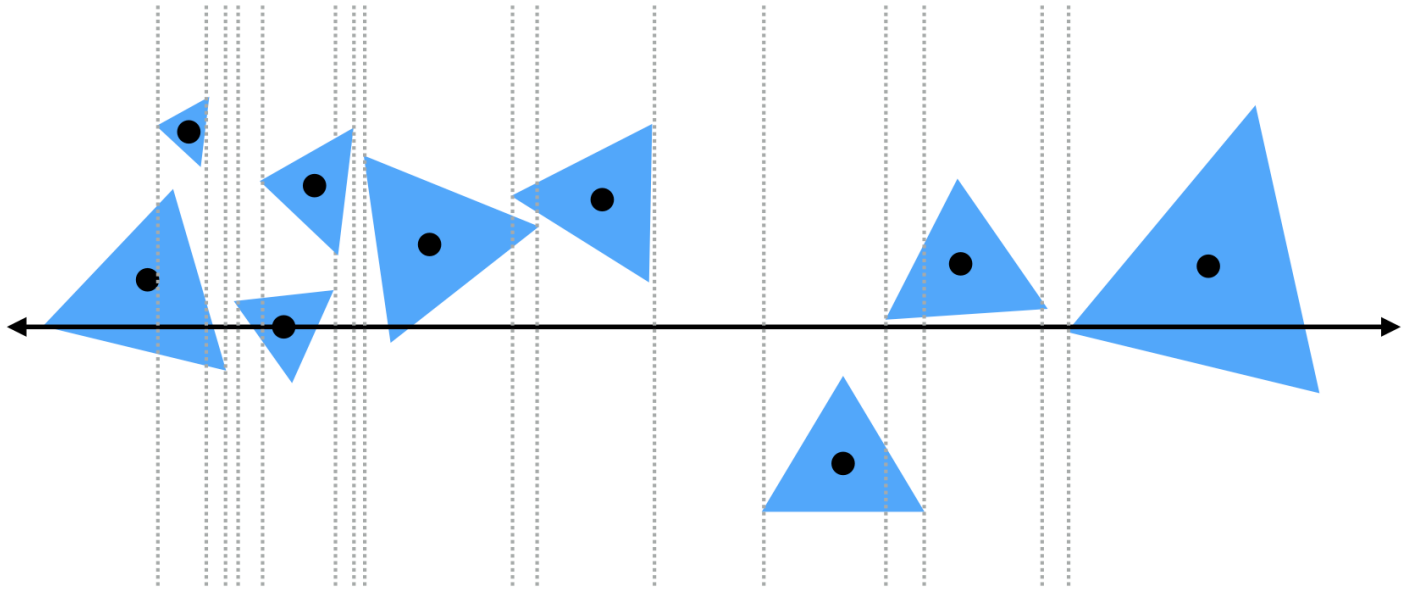
is a technique designed for solving a problem more quickly when **classic methods** are too *slow* for finding an approximate solution, or when classic methods *fail* to find any exact solution. This is achieved by trading optimality, completeness, accuracy, or precision for speed. In a way, it can be considered a shortcut.

15462.courses.cs.cmu.edu/spring2022content/lectures/13_spatial/13_spatial_slides.pdf

Good Partition Principle 2^{N-1} in total

Implementing partitions

- Constrain search for good partitions to axis-aligned spatial partitions
  - Choose an axis; choose a split plane on that axis
  - Partition primitives by the side of splitting plane their centroid lies
  - Cost estimate changes only when plane moves past triangle boundary
  - Have to consider rather large number of possible split planes…



15462.courses.cs.cmu.edu/spring2022content/lectures/13_spatial/13_spatial_slides.pdf

Uniform Grids **Adaptive Grids** *subdivide **space** into smaller sub-spaces*

Bounding Volume Hierarchy *subdivide a list of **triangles** into smaller subsets*

*spatial subdivision of **geometry***

often costly than ray tracing itself

Ray-Space Hierarchy

*spatial subdivision of **rays***

This approach is well-suited for dynamic scenes.

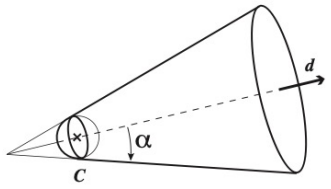


Figure 1: *We use a cone-sphere structure for our ray-space hierarchy. Each node is defined by a sphere and a cone.*

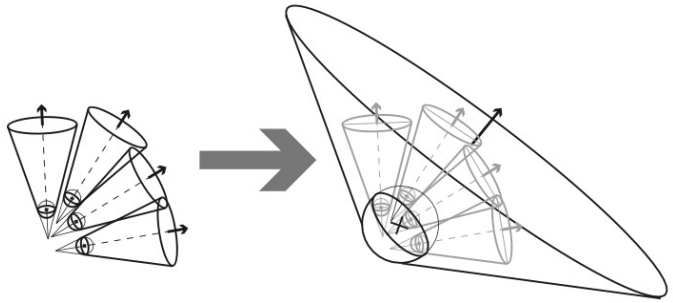


Figure 2: *The parent node is constructed as the enclosing cone-sphere for the four children.*

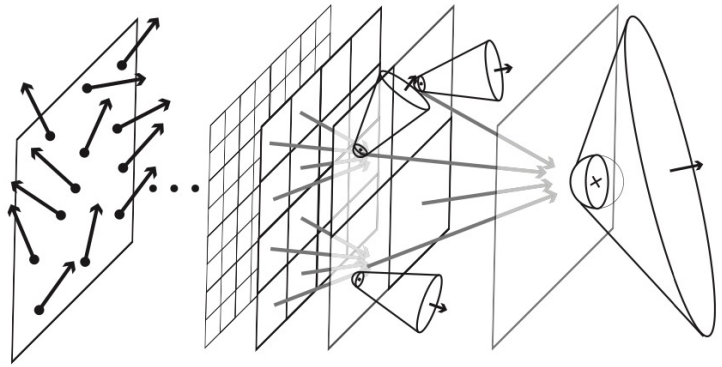


Figure 3: *We start with the set of secondary rays, and recursively build the enclosing cone-sphere for each hierarchical level.*

http://dx.doi.org/10.2312/EGWR/EGSR07/099-110

Whitted Ray-Tracing for Dynamic Scenes using a Ray-Space Hierarchy on the GPU

•D. Roger, Ulf Assarsson, Nicolas Holzschuch

Ray-Space Hierarchy

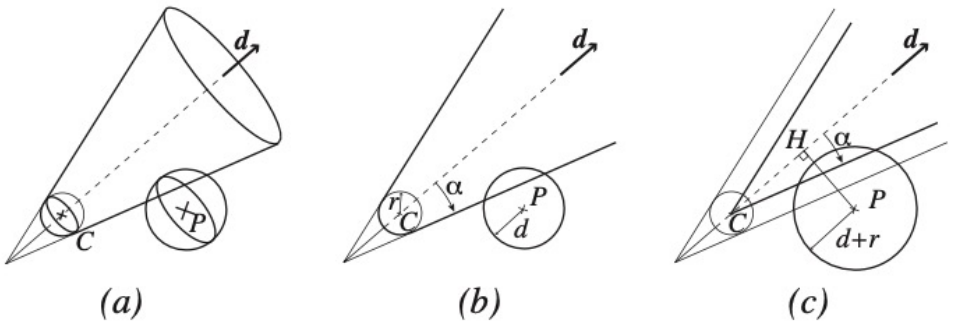


Figure 5: *Testing the intersection between a node and the bounding sphere of a triangle (a) reduces to a 2D problem (b). It is equivalent to testing the intersection between a reduced cone and an enlarged sphere (c).*

http://dx.doi.org/10.2312/EGWR/EGSR07/099-110

Whitted Ray-Tracing for Dynamic Scenes using a Ray-Space Hierarchy on the GPU

•D. Roger, Ulf Assarsson, Nicolas Holzschuch

With Sincere Thanks to

Lai Wei's Ray Tracing Wiki

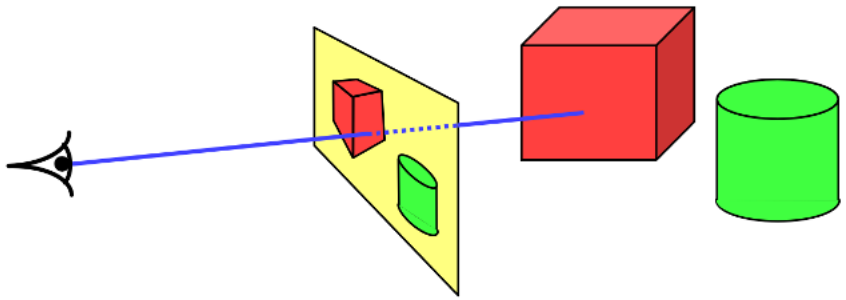
Other Useful Reference for Ray Tracing (and broader CG):

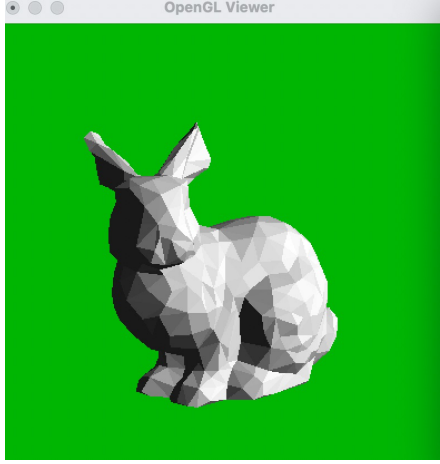
Fundamentals of Computer Graphics by Marschner & Shirley, CRC Press 2015 (4th edition)

MIT EECS 6.837 Computer Graphics (solid Math intro) https://groups.csail.mit.edu/graphics/classes/6.837/F04/calendar.html

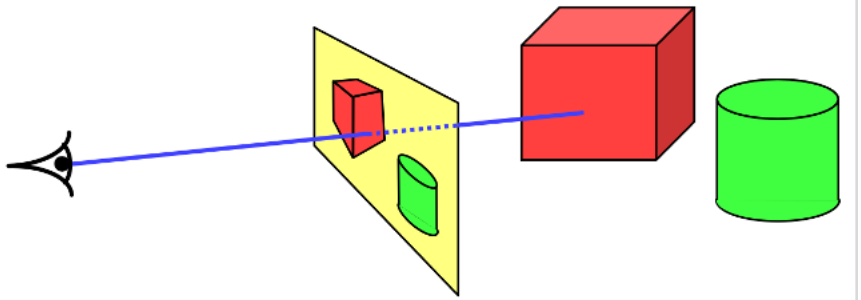
Cambridge IA Intro to Graphics https://www.cl.cam.ac.uk/teaching/2223/Graphics/

Vulkan_Ray_Tracing_Overview_Apr21 Khronos Template 2015 (slides)

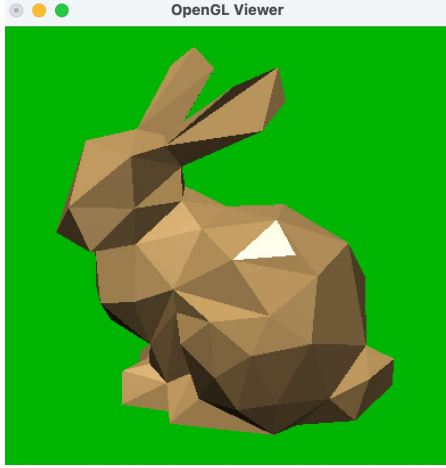




Q&A



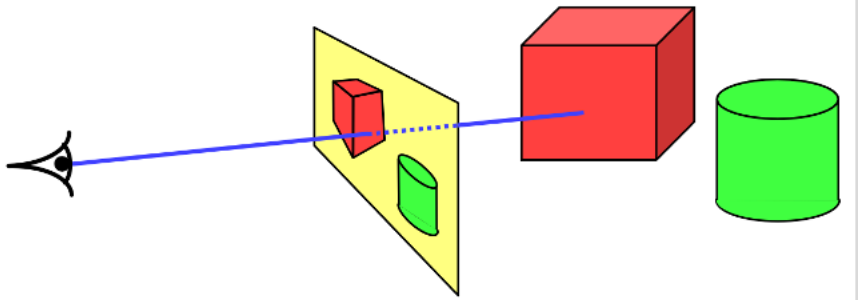
HUAWEI



Thank you for listening!

Wish you enjoy this journey on RAY TRACING!

Best of luck!



HUAWEI