

Speeding up real-time Ray Tracing

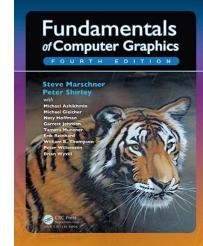
Welcome to embark on the journey on RAY TRACING!

Peter Hu

Revised 1 Nov 2023

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.

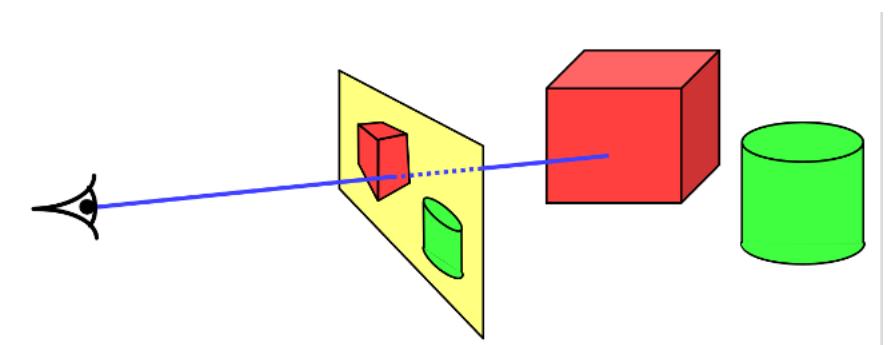
The bad news is that we have still a long way to go.

The good news is that we have still a long way to go.



Ray Tracing

Welcome to embark on the journey on RAY TRACING!



Peter Hu
Huawei Technologies Research & Development
GPU Team
3 May 2023





With Sincere Thanks to

Useful Reference for Ray Tracing (and broader CG):

Text Book

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



Courses

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

CMU CG [Computer Graphics : Spring 2022 \(cmu.edu\)](http://www.cs.cmu.edu/~barla/courses/15-463/)

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

Graphics API

Learn OpenGL <https://learnopengl.com>

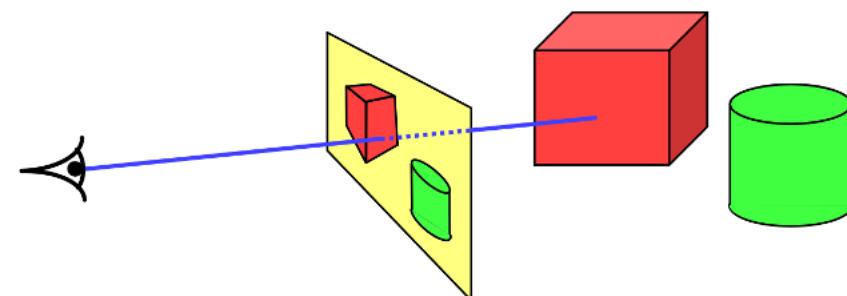
DirectX Specs <https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html>

Vulkan_Ray_Tracing_Overview_Apr21 [Khronos Template 2015 \(slides\)](#)

Code and Repo

5-star repo [knightcrawler25/GLSL-PathTracer: A GLSL Path Tracer \(github.com\)](https://github.com/knightcrawler25/GLSL-PathTracer)

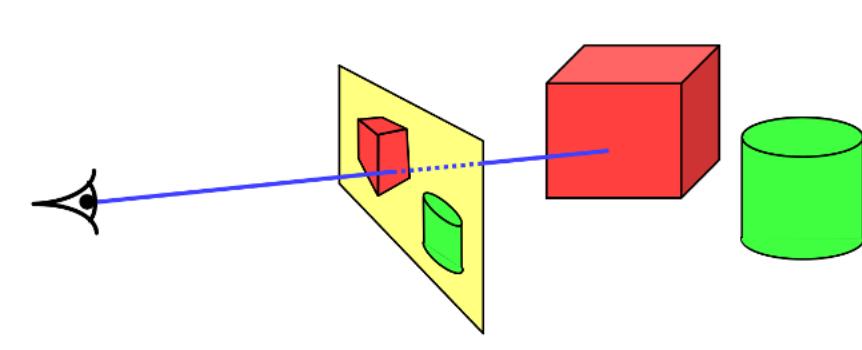
See more links attached in later slides...



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/PeterHUiTyping/MIT6.837-CG-Fall2004-Assignment/blob/main/MyProject/assignment4_Ray-Tracer/ray/raytracer.h

<https://github.com/PeterHUiTyping/MIT6.837-CG-Fall2004-Assignment>

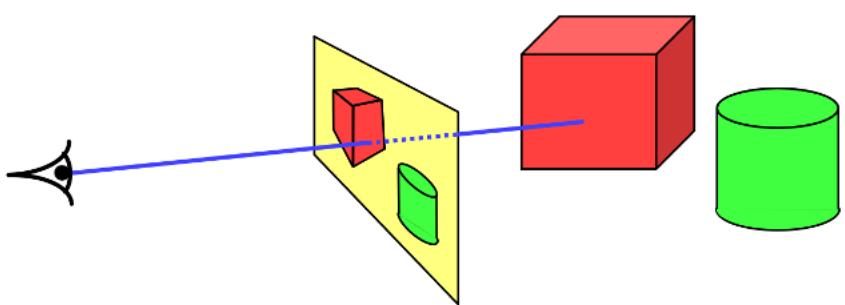
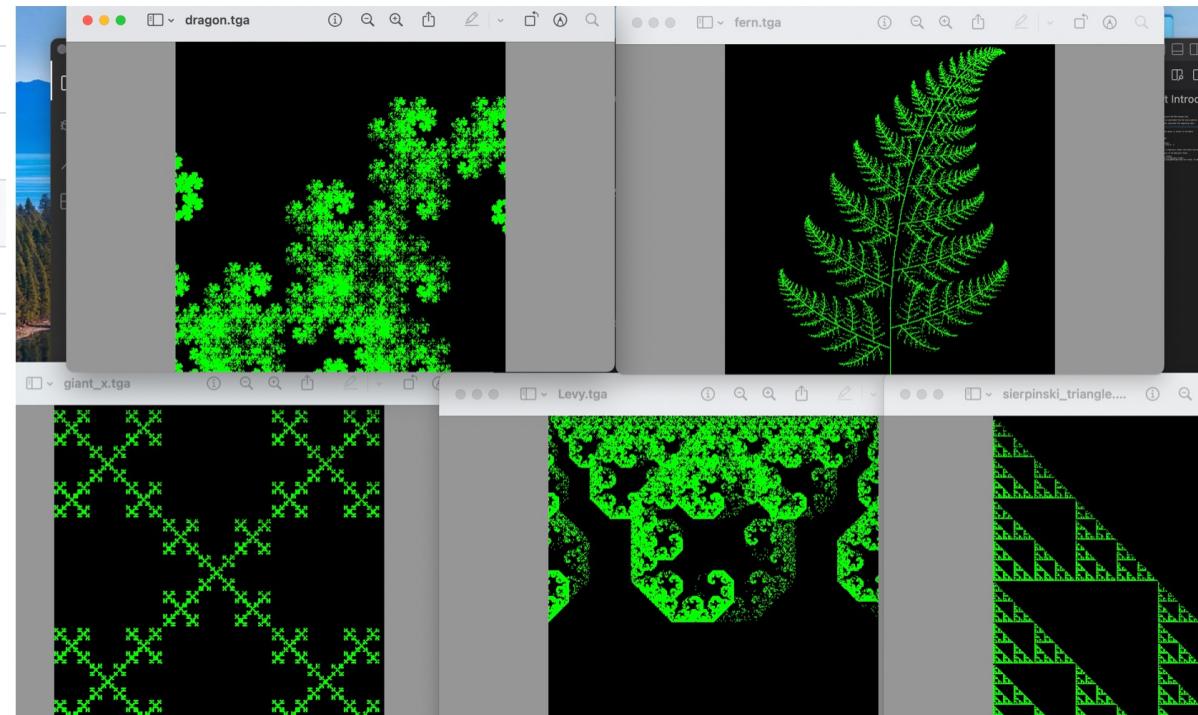
Lots more example of Graphics generated by my own Ray Tracer or OpenGL.

```
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

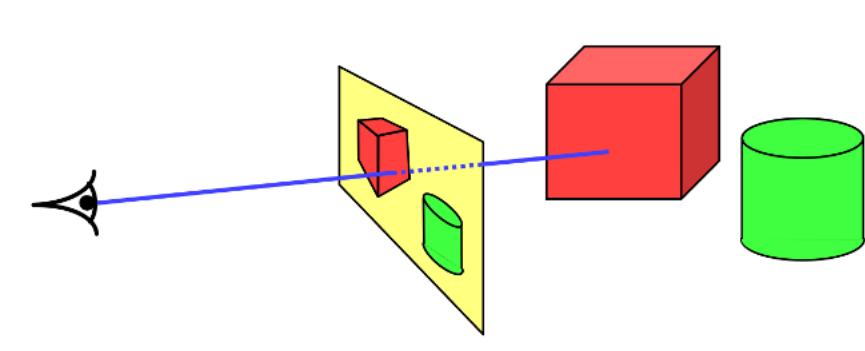
Computer Graphics



Recent Ongoing Research Focus

- BVH Algorithms
including BVH Construction and Traversal
- Other Acceleration Structures

Any suggestions and idea on RT are warmly welcome!



RTX
ON

Peter Hu

1 Nov 2023



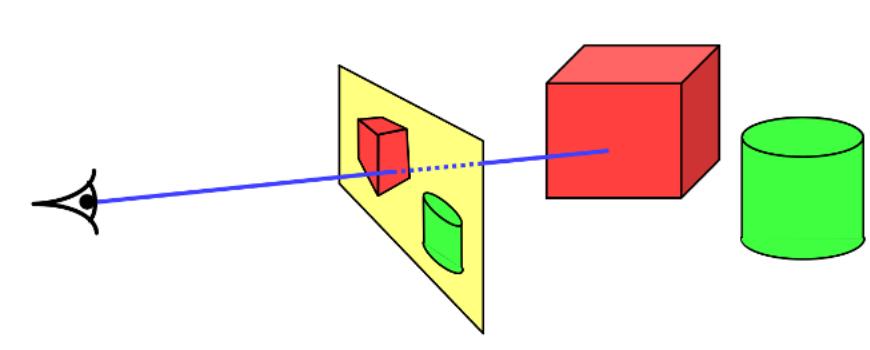
real-time Ray Tracing

*Application reflection,
refraction,
soft shadows,
scattering,
depth of field,
motion blur,
caustics,
ambient occlusion,
dispersion phenomena,
etc*

Physically-accurate
than tricks like
texture mapping

Part 1: Ray Tracing Algorithm

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



Why Ray Tracing is better than Rasterization?

Ray Casting

For each pixel

For each object

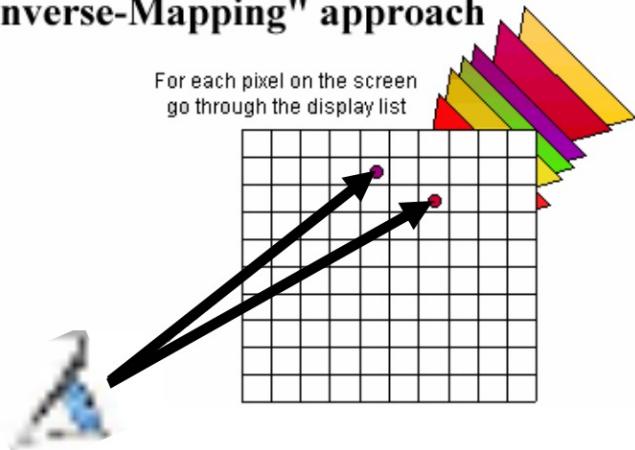
- Whole scene must be in memory

Send pixels to the scene

Discretize first

"Inverse-Mapping" approach

For each pixel on the screen
go through the display list



Rendering Pipeline

For each triangle

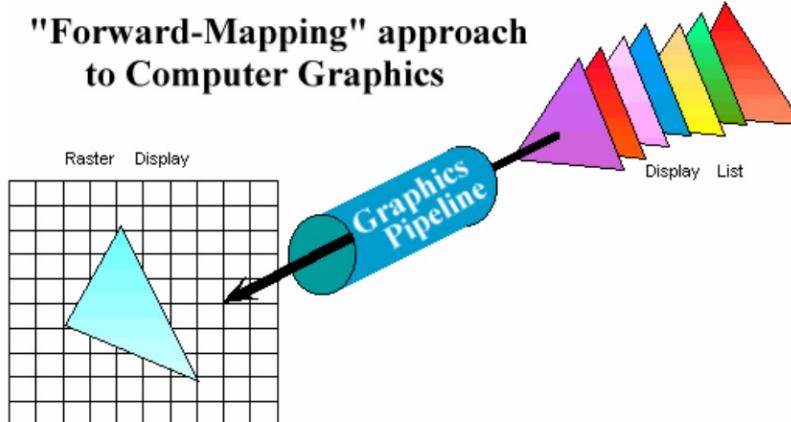
For each pixel

- Primitives processed one at a time

MVP transformation Projection

Object -> World -> Screen coordinate

"Forward-Mapping" approach to Computer Graphics

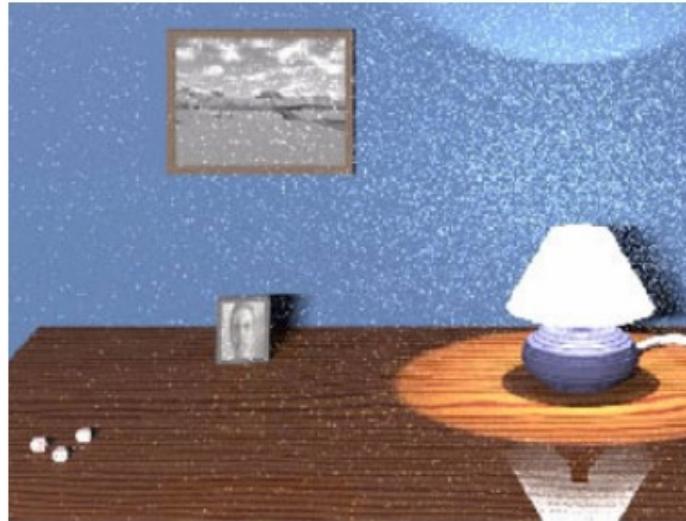
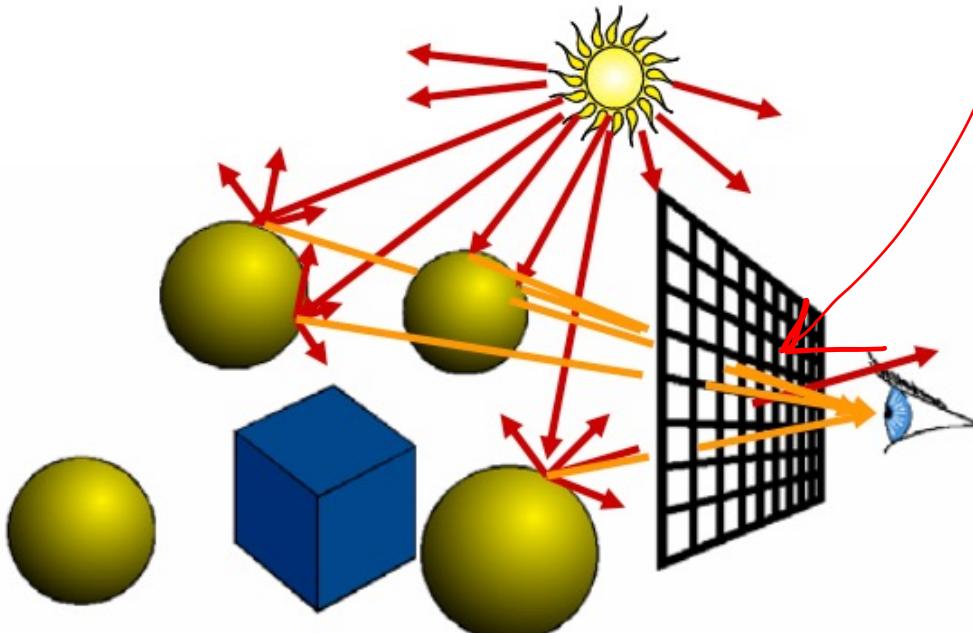


Forward Ray Tracing

Motivation

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

- 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



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

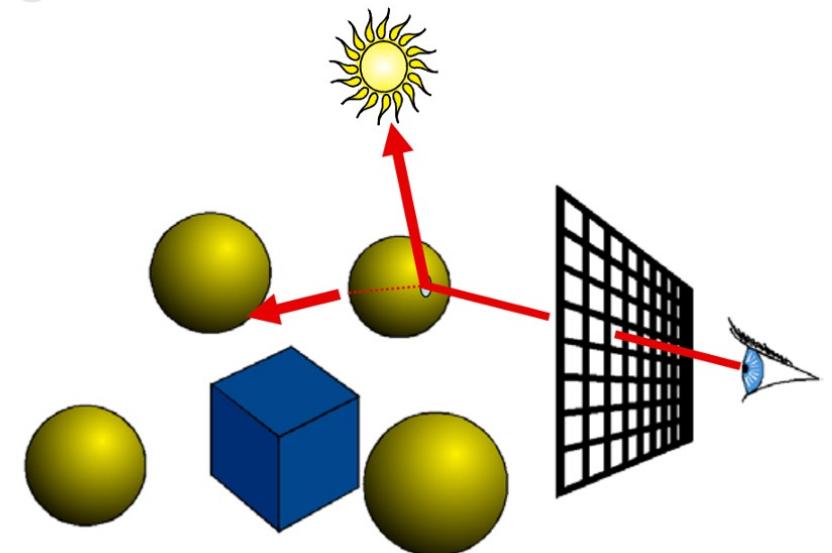
```
Group {  
    numObjects 2  
  
    MaterialIndex 0  
    Sphere {  
        center 0 0 0  
        radius 1  
    }  
  
    MaterialIndex 1  
    Plane {  
        normal 0.01 1 0.01  
        offset -1  
    }  
}
```

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.*

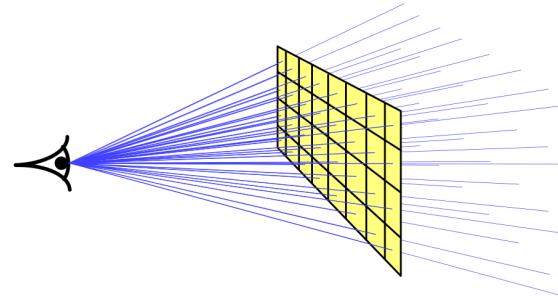
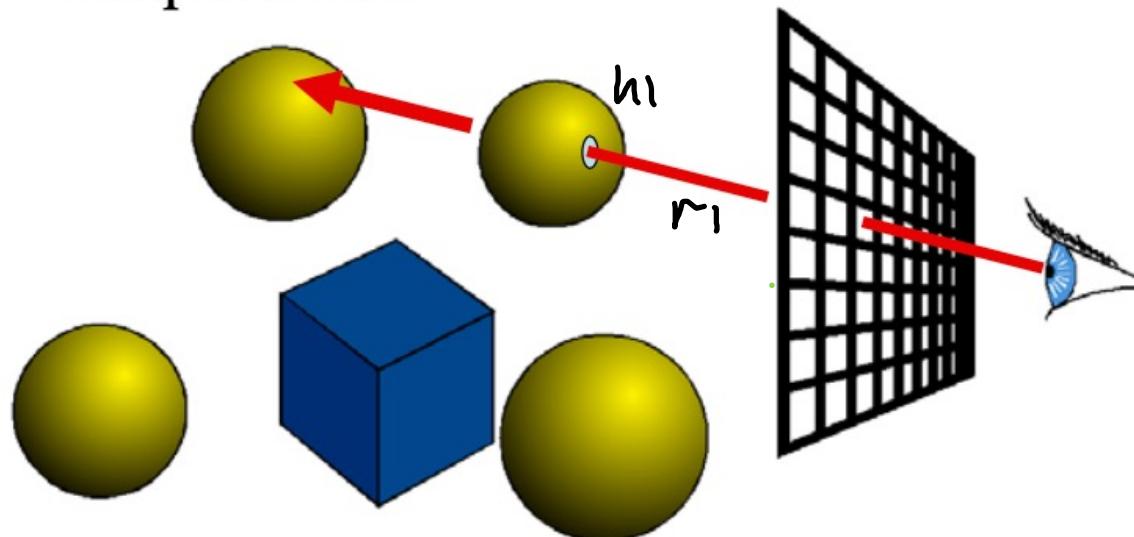
The technique traces the *path of light* from the eye/camera to the scene(backward), where it interacts with various objects and surfaces, and then back to the camera to produce the final image.

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

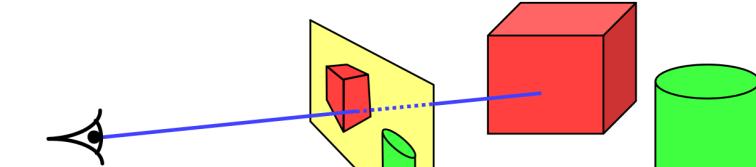


Ray Casting

- 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

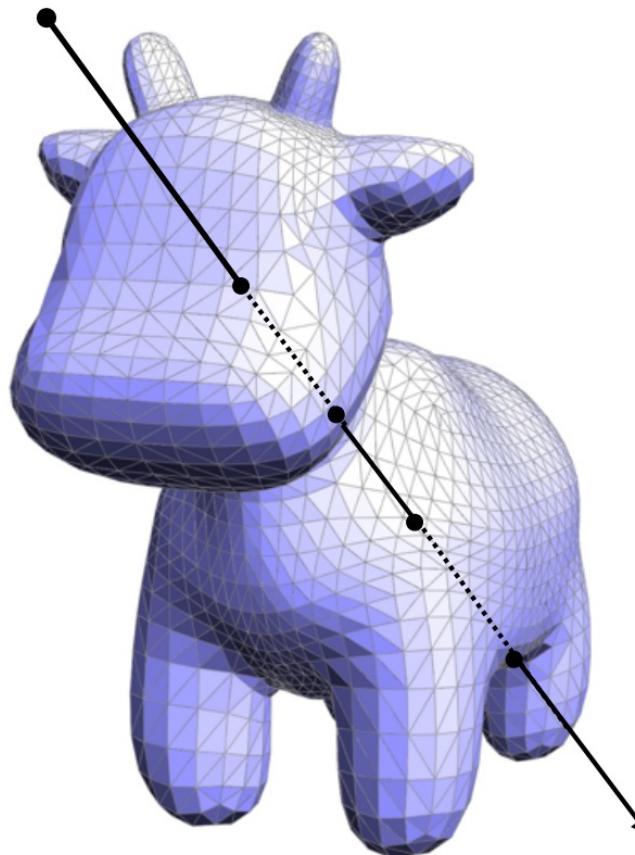
“Find the first primitive the ray hits”

Naïve algorithm?

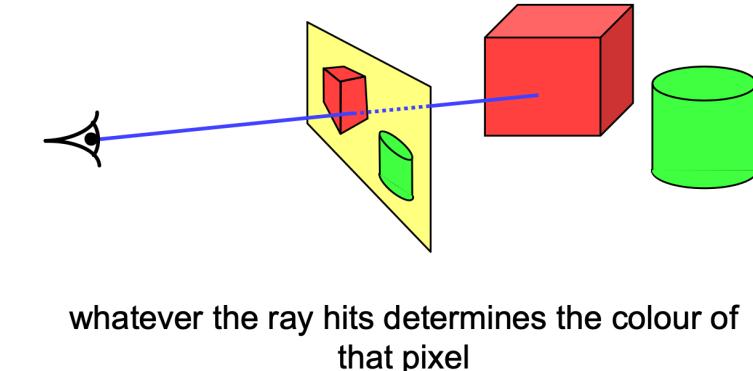
1. Intersect ray with every triangle
2. Keep the closest hit point

Complexity? $O(N)$

Can we do better? Bounding box!



CMU 15-462/662



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

```

void rayCasting( )
{
    Image image(width, height); // image.SetAllPixels(parser->getBackgroundColor());
    Vec2f p;                  // (0,0)->(1,1) 1024*1024 Generate Points on the screen 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/PeterHUi/typing/MIT6.837-CG-Fall2004-Assignment/blob/main/MyProject/assignment4_Ray-Tracer/main.C

Does Ray Tracing Simulate Physics?

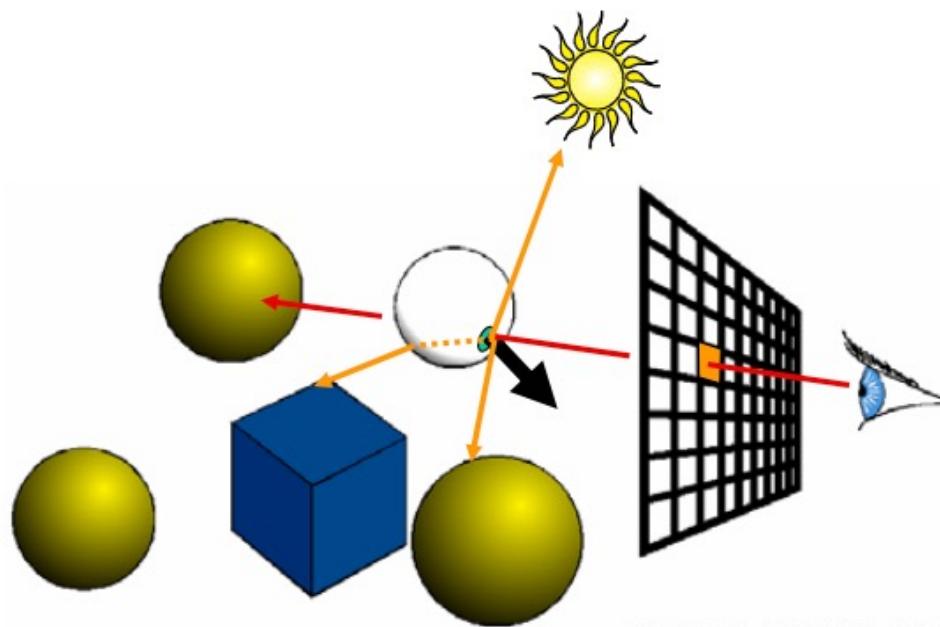
Motivation

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

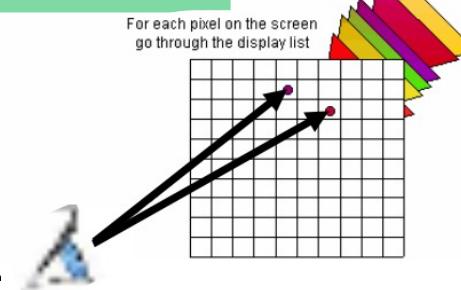
Ray Casting

For each pixel
For each object

Send pixels to the scene
Discretize first

Inverse-Mapping" approach

For each pixel on the screen
go through the display list

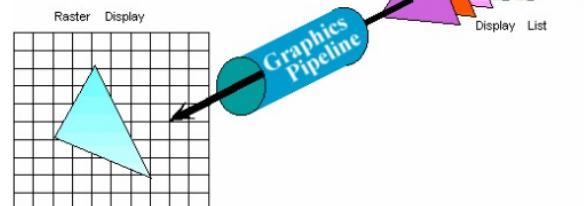


Rendering Pipeline

For each triangle
For each pixel

Project scene to the pixels
Discretize last

"Forward-Mapping" approach to Computer Graphics



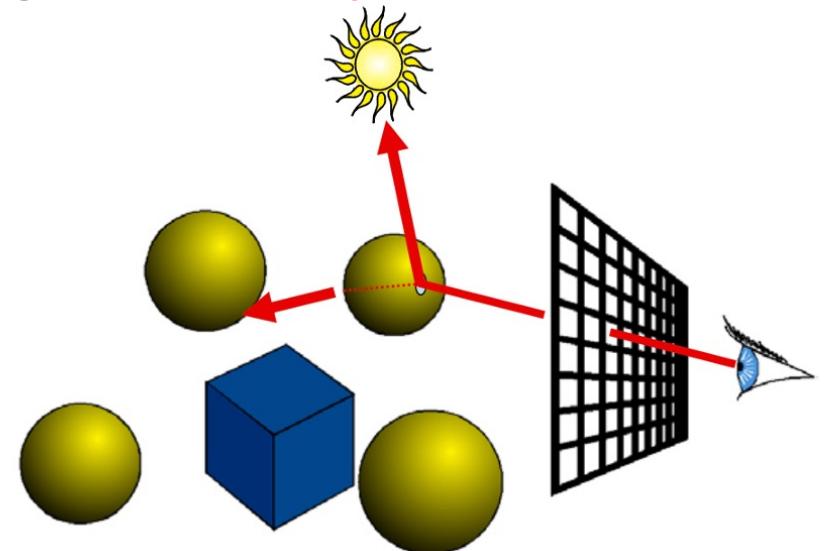
MIT EECS 6.837, Durand and Cutler

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.

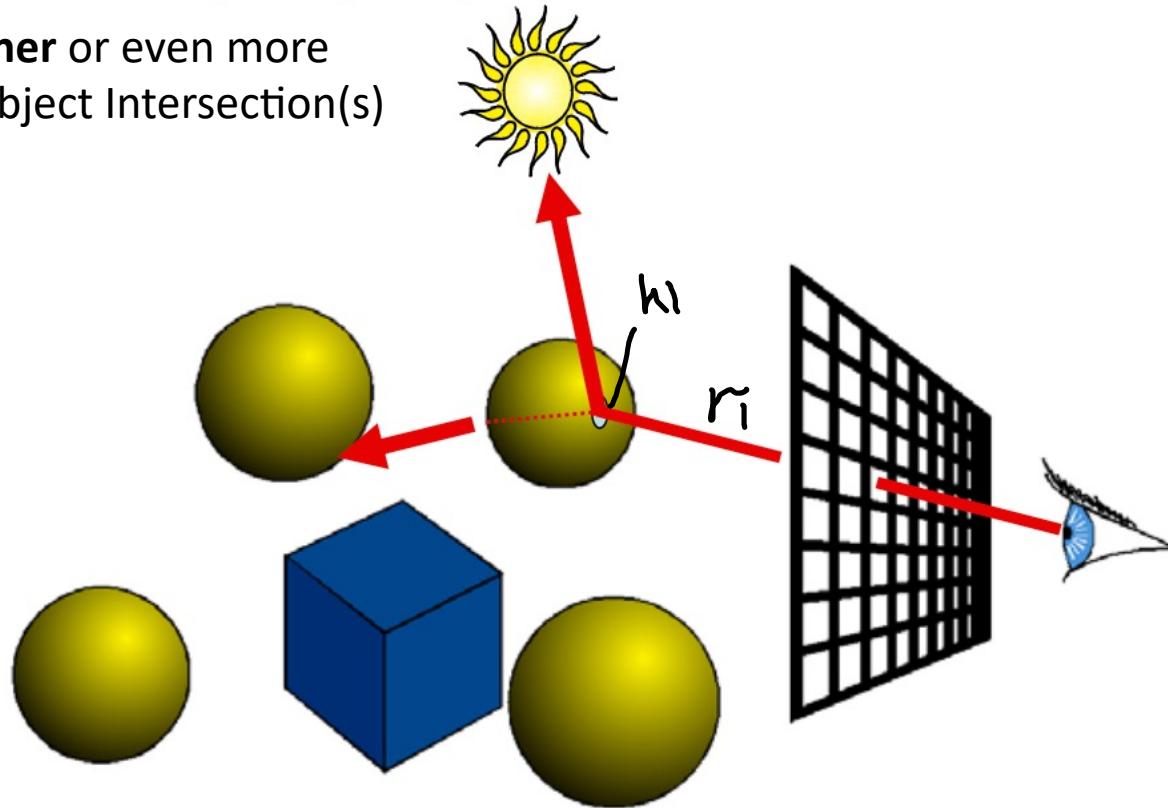
The technique traces the **path of light** from the eye/camera to the scene(backward), where it interacts with various objects and surfaces, and then back to the camera to produce the final image.

*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

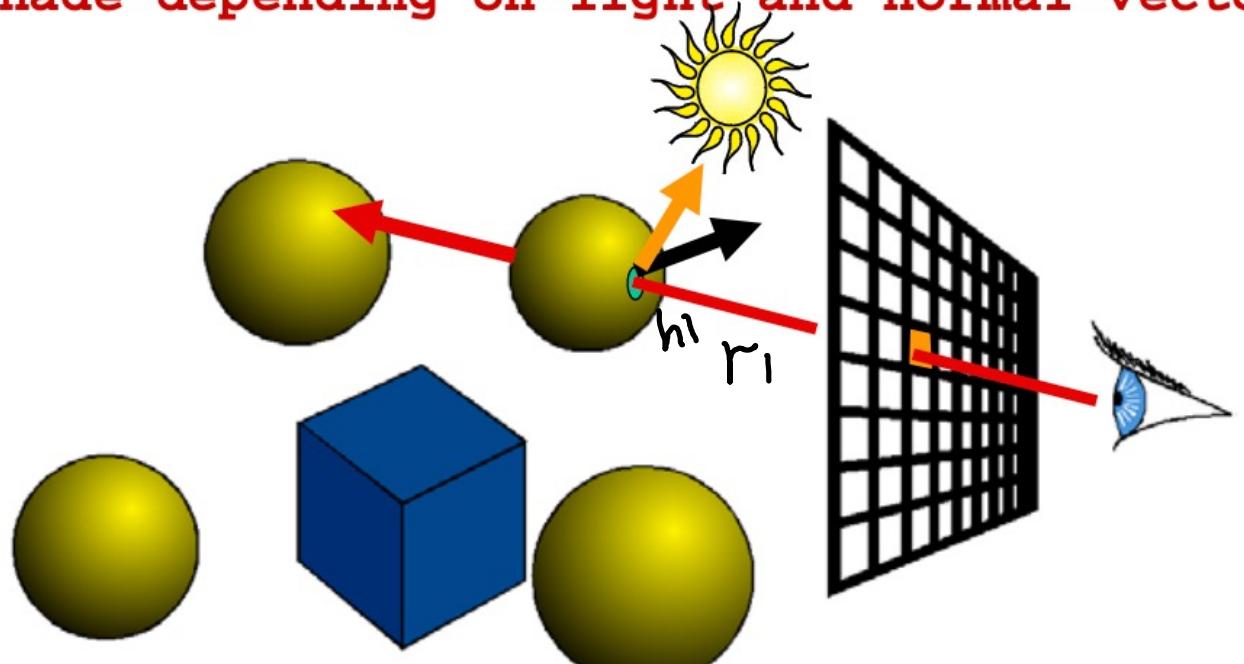
Construct a ray from the eye

For every object in the scene

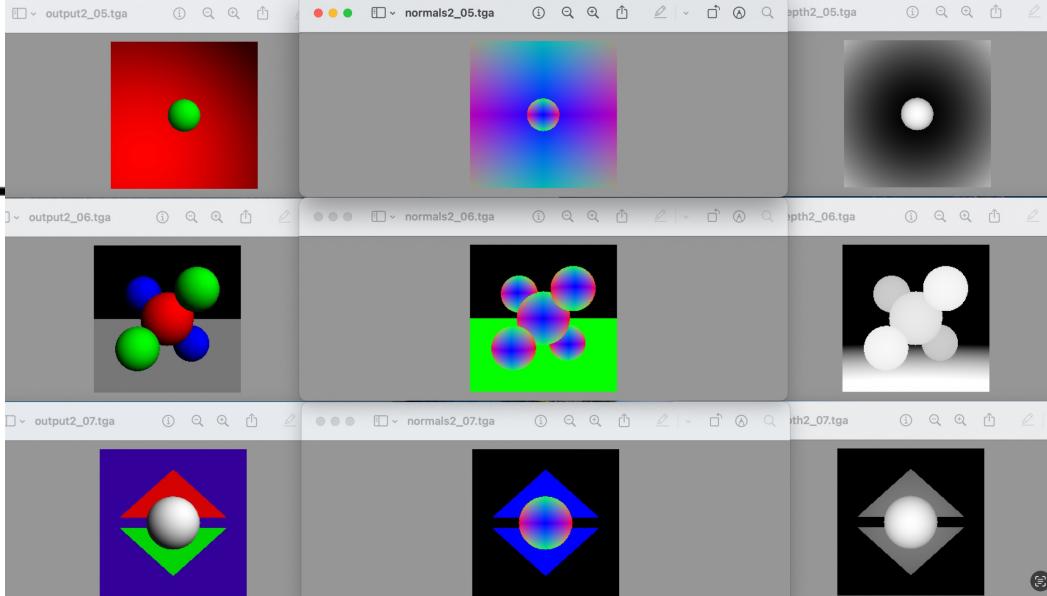
Find intersection with the ray

Keep if closest

Shade depending on light and normal vector

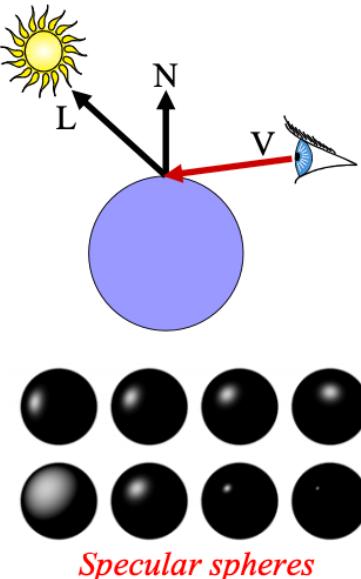


MIT EECS 6.837, Cutler and Durand



A Note on Shading

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



MIT EECS 6.837, Cutler and Durand

Shading

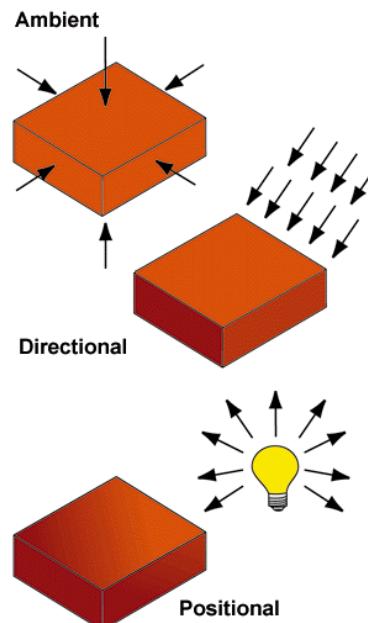
Sum of three components:

1. “ambient” +
2. diffuse reflection +
3. specular reflection

```
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  
}
```

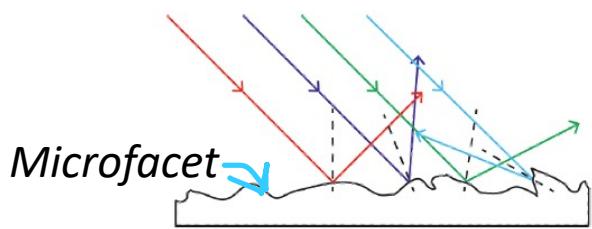


```
if (!parser->getGroup()->intersectShadowRay(r2, h2, epsilon_shadow)) // epsilon self-shadowing  
{  
    pcolor += hit.getMaterial()->Shade(ray, hit, dirToLight, lcolor); // Diffuse Color + Specular Color  
    RayTree::AddShadowSegment(r2, 0, h2.getT());  
}
```

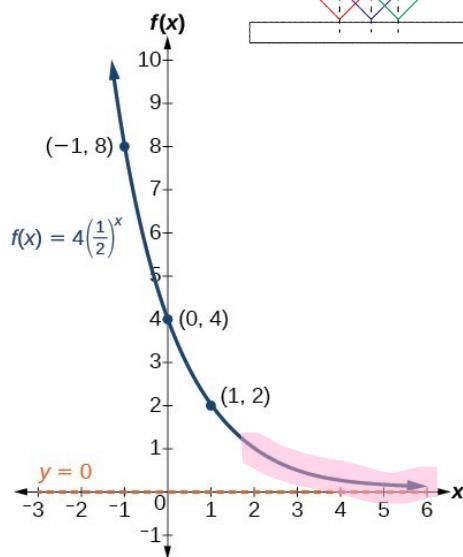
Shading

Sum of three components:

1. "ambient" +
- 2. diffuse reflection** +
- 3. specular reflection**



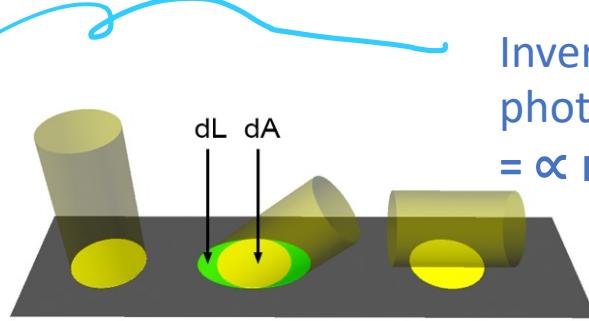
Ideal Specular material



2. Ideal Diffuse Reflection

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

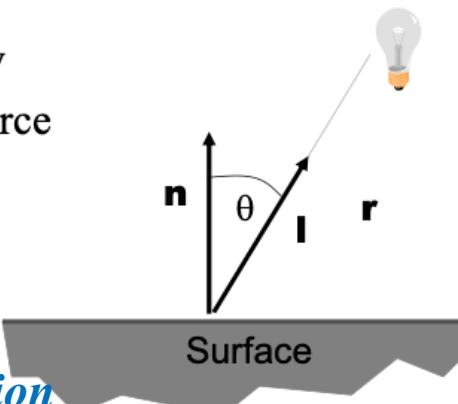
$$L_o = k_d (\mathbf{n} \cdot \mathbf{l}) \frac{L_i}{r^2}$$



Inversely proportional to photon projection area
 $= \propto \mathbf{n} \cdot \mathbf{l}$ dot product

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

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



3. Specular Reflection

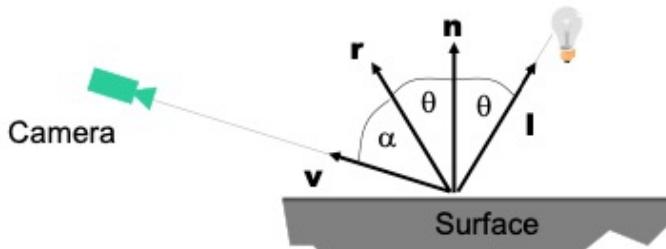
- Depends on the angle between the ideal reflection direction and the viewer direction α .

Parameters

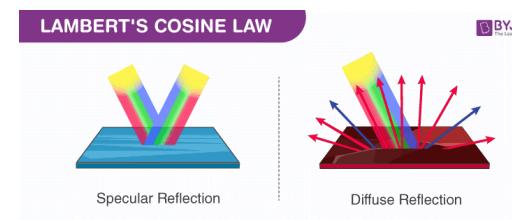
- k_s : specular reflection coefficient
- q : specular reflection exponent

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

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



```
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
```

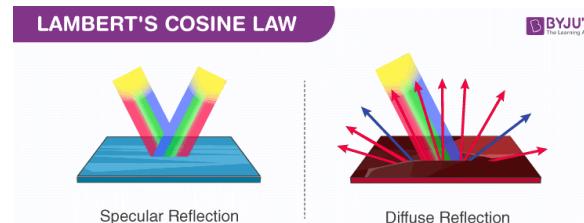


```
Materials {
    numMaterials 2
    PhongMaterial {
        diffuseColor 1 0 0
    }
    PhongMaterial {
        diffuseColor 0 1 0
    }
}
```

Shading

Sum of three components:

1. “ambient” +
2. diffuse reflection +
3. specular reflection



3. Specular Reflection

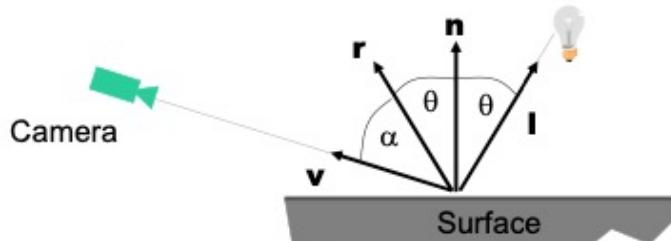
- Depends on the angle between the ideal reflection direction and the viewer direction α .

• Parameters

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

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

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

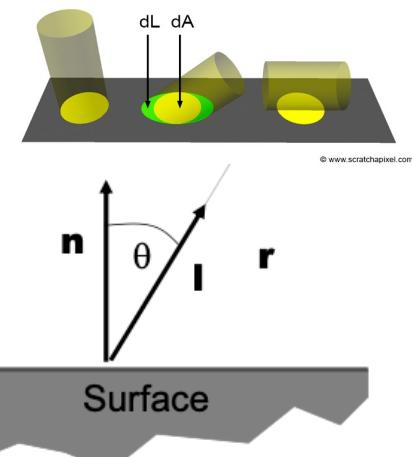


2. Ideal Diffuse Reflection

• Single Point Light Source

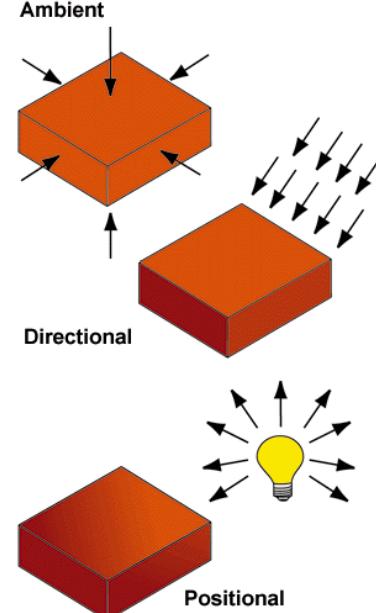
- k_d : diffuse coefficient.
- \mathbf{n} : Surface normal.
- \mathbf{l} : Light direction.
- L_i : Light intensity
- r : Distance to source

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



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

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



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  
}
```

```

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.

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

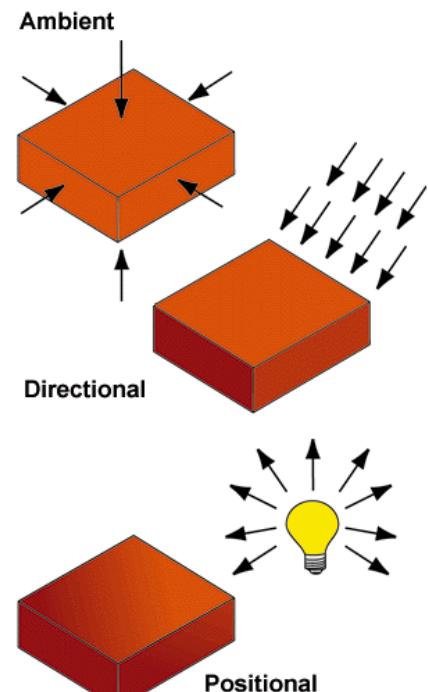
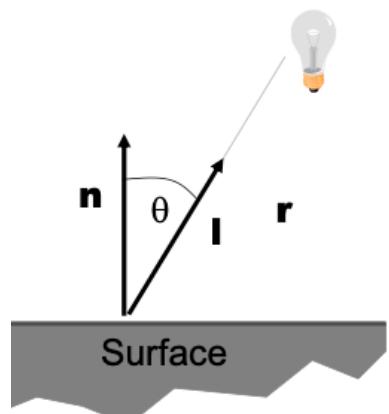
ht direction.

ight intensity

stance to source

```

Materials {
    numMaterials 2
    >hongMaterial {
        diffuseColor 1 0 0
    }
    >hongMaterial {
        diffuseColor 0 1 0
    }
}
```



```

    )->intersectShadowRay(r2, h2, epsilon_shadow)) // epsilon self-shadowing
                                                    // Diffuse Color + Specular Color
    material()->Shade(ray, hit, dirToLight, lcolor); // Implemented by the function of another class
    Segment(r2, 0, h2.getT());
    //-----//
    l += am; // ambient color
}
```

Illumination

ts the reflection of all indirect illumination.
total hack!

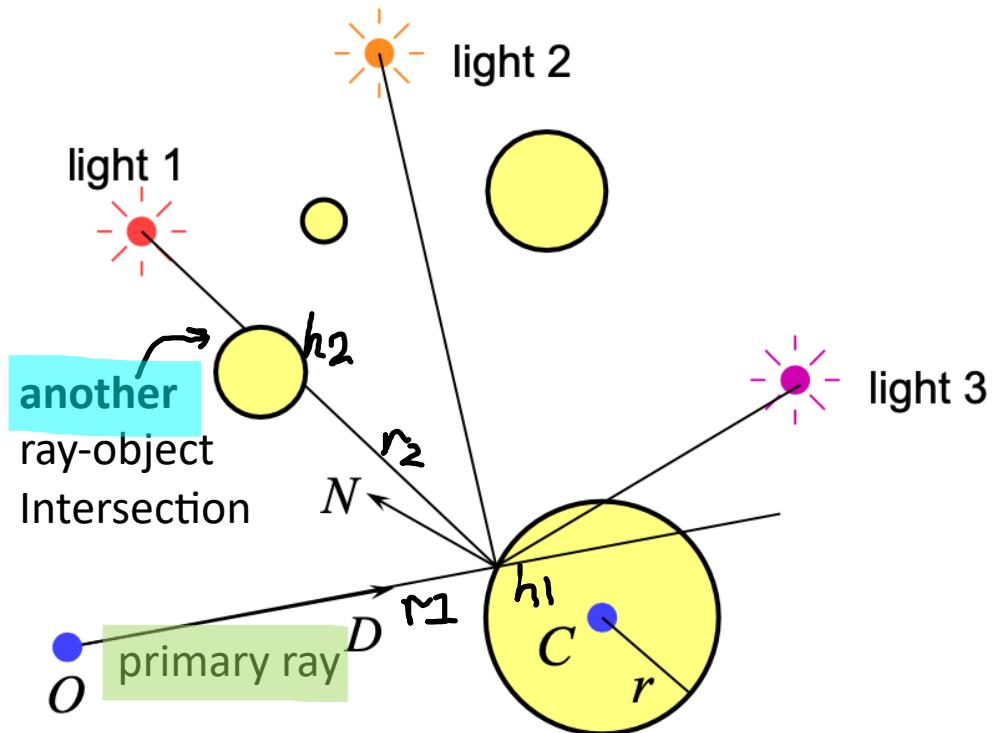
e complexity of global illumination.

+ am; // ambient color

```

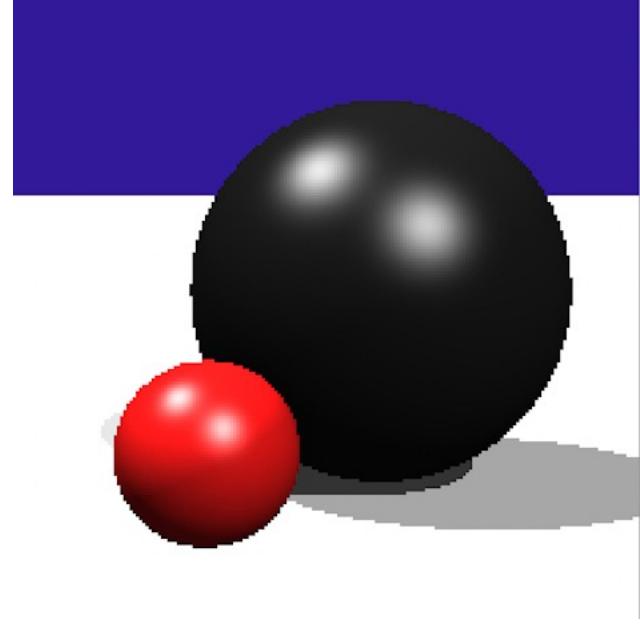
Background {
    color 0 0 1
    ambientLight 0.2 0.2 0.2
}
```

Ray tracing: shadows



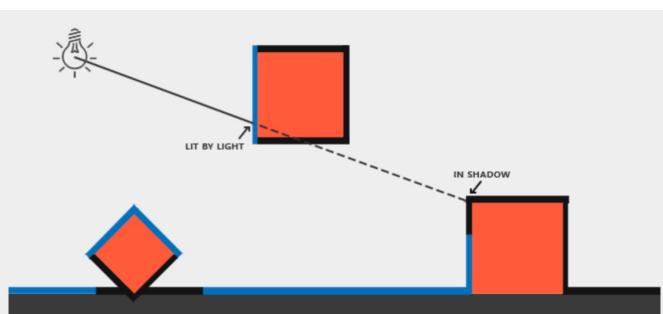
- because you are tracing rays from the intersection point to the light, you can check whether **another object** is between the intersection and the light and is hence casting a shadow

- also need to watch for self-shadowing

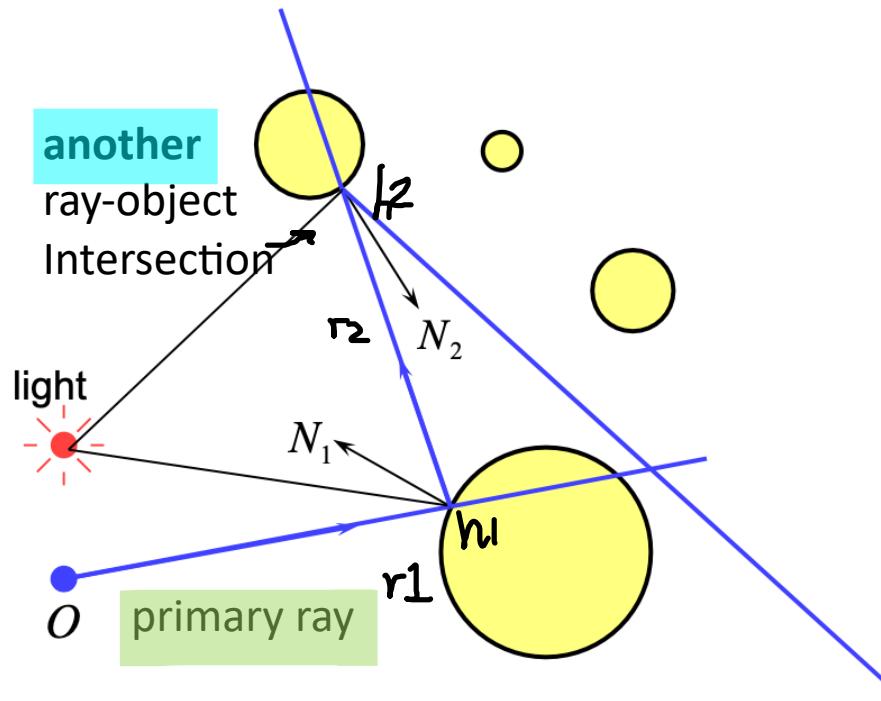


Ray tracing can get **shadows** by ray casting, otherwise **shadow mapping**. ->
(*Depth buffer from light*)

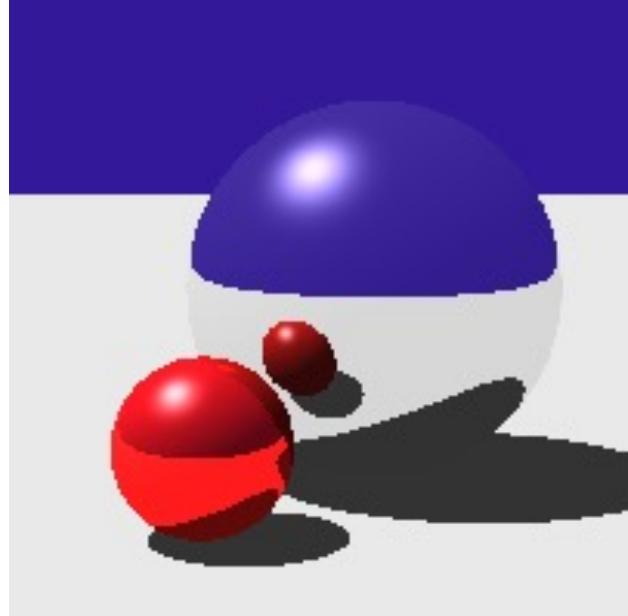
<https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping>



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



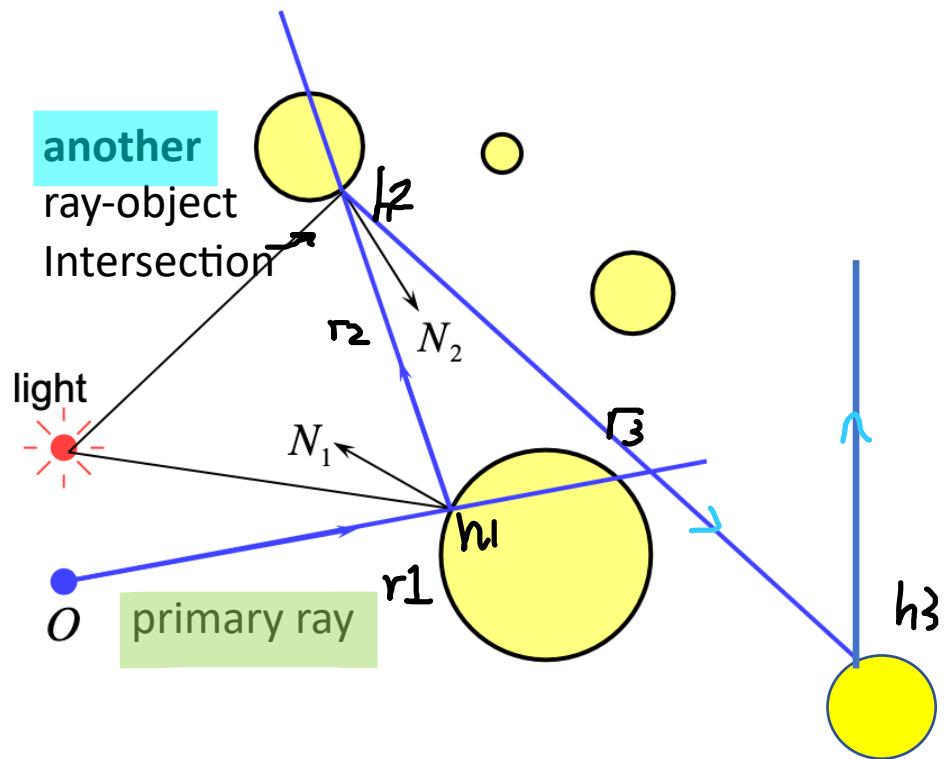
bounces = 1

	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

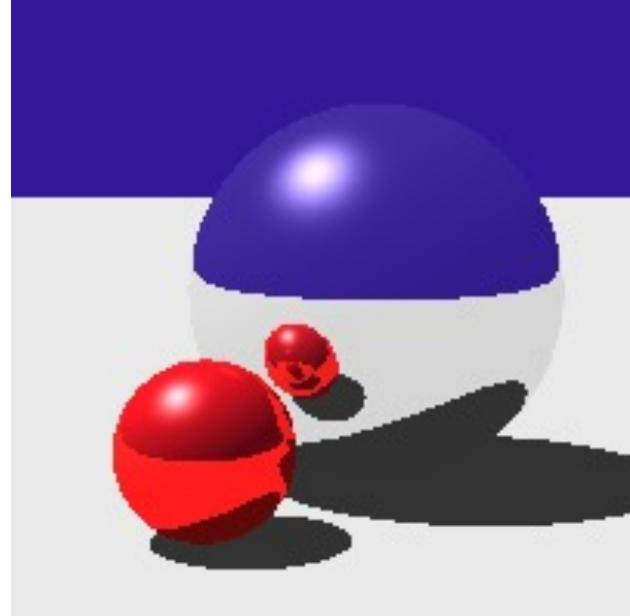
Adapted from ©Neil A Dodgson, Peter Robinson & Rafał Mantiuk, Cambridge University, 1996-2021



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



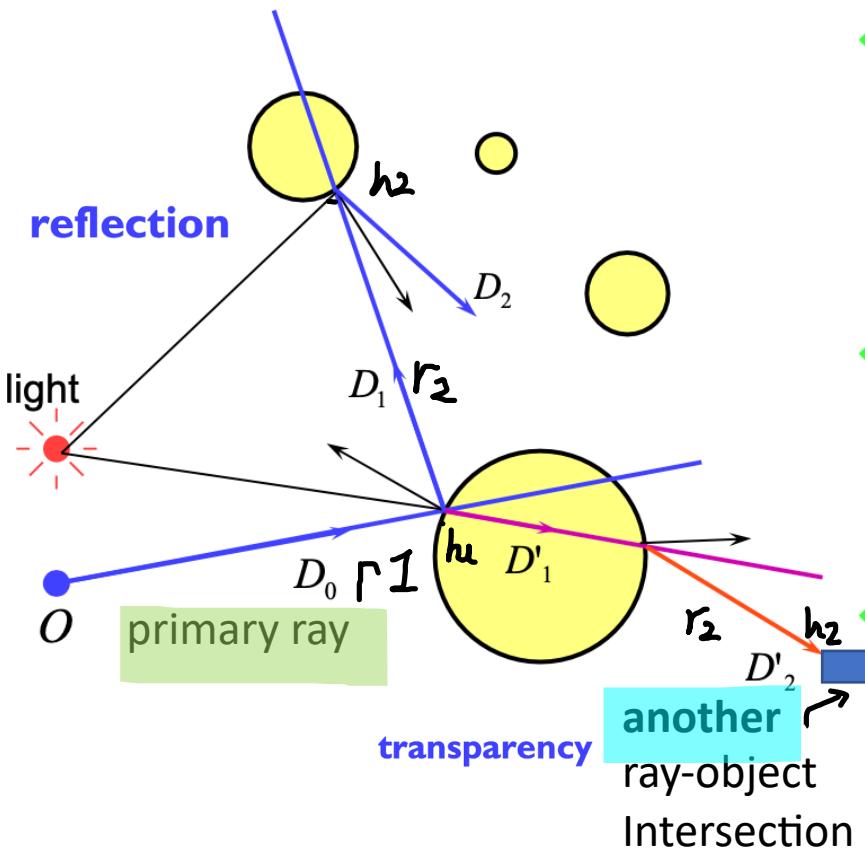
and
can be set to infinite bounces😊

Surface Examined	ID	Material of the Surface Examined	Reflection Coefficient [%]
Cloister	a	Red ceramic tiles	12
	b	Red ceramic tiles	12
	c	White painted lime plaster	82
	d	Orange painted lime plaster	29
	e	Marble	71
	f	Cobblestone	28

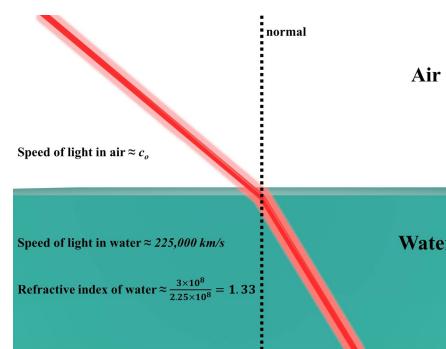
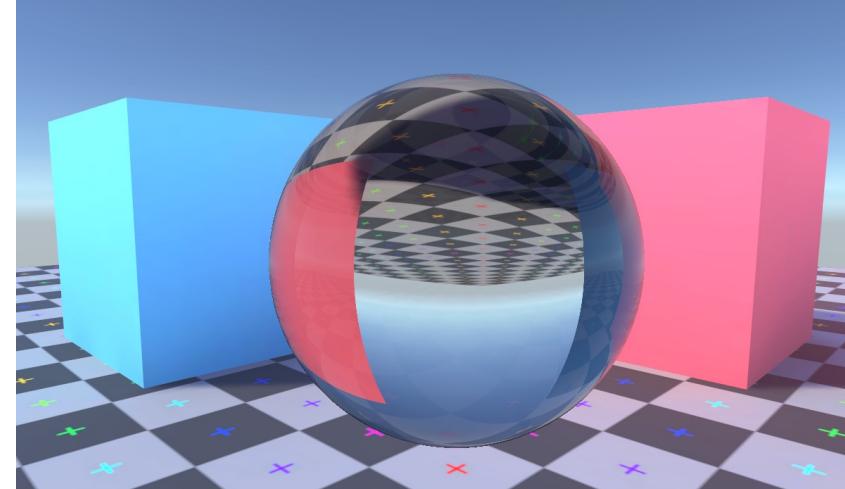
Adapted from ©Neil A Dodgson, Peter Robinson & Rafał Mantiuk, Cambridge University, 1996-2021



Ray tracing: transparency & refraction

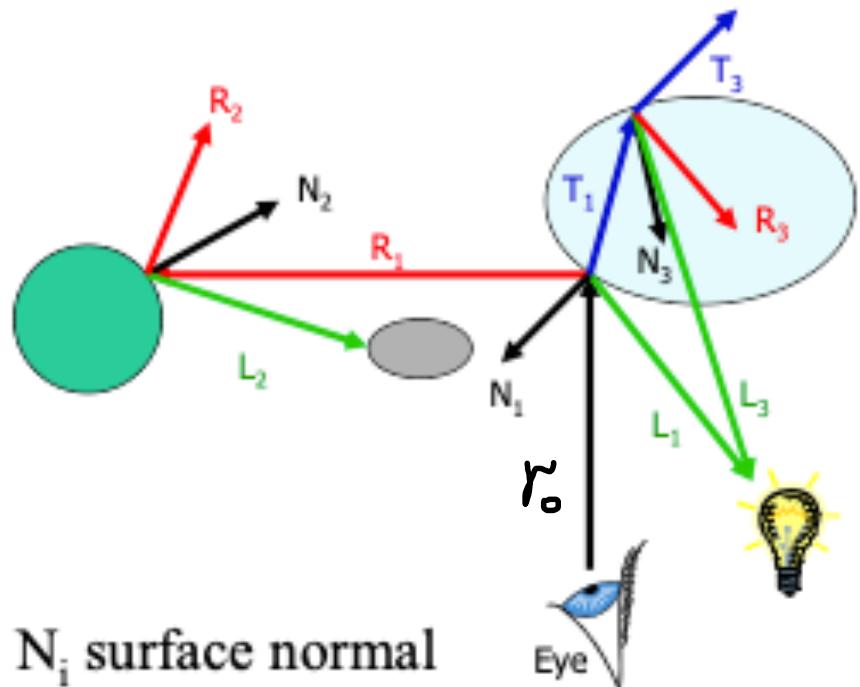


- objects can be **totally or partially transparent**
 - this allows objects behind the current one to be seen through it
- transparent objects can have **refractive indices**
 - bending the rays as they pass through the objects
- transparency + reflection means that a ray can split into two parts



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

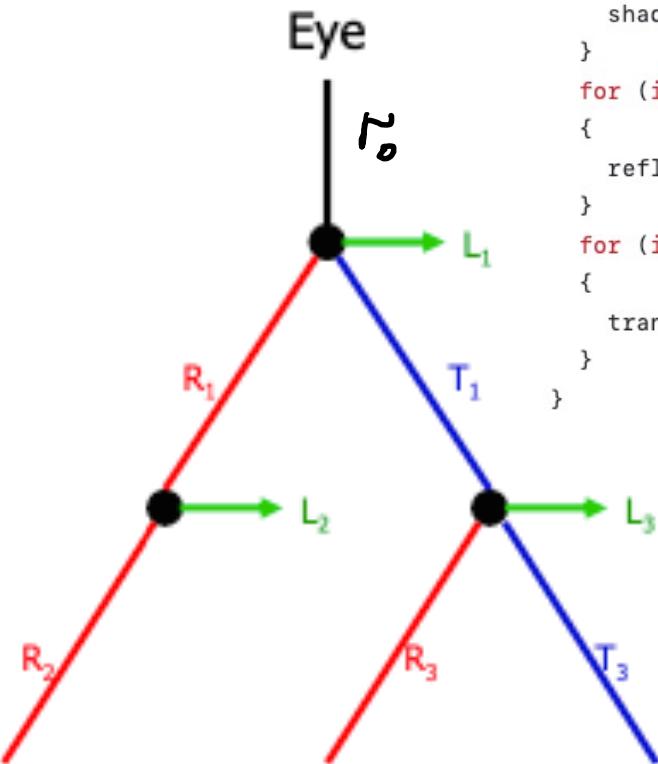


N_i surface normal

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");
    }
}
```

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!)

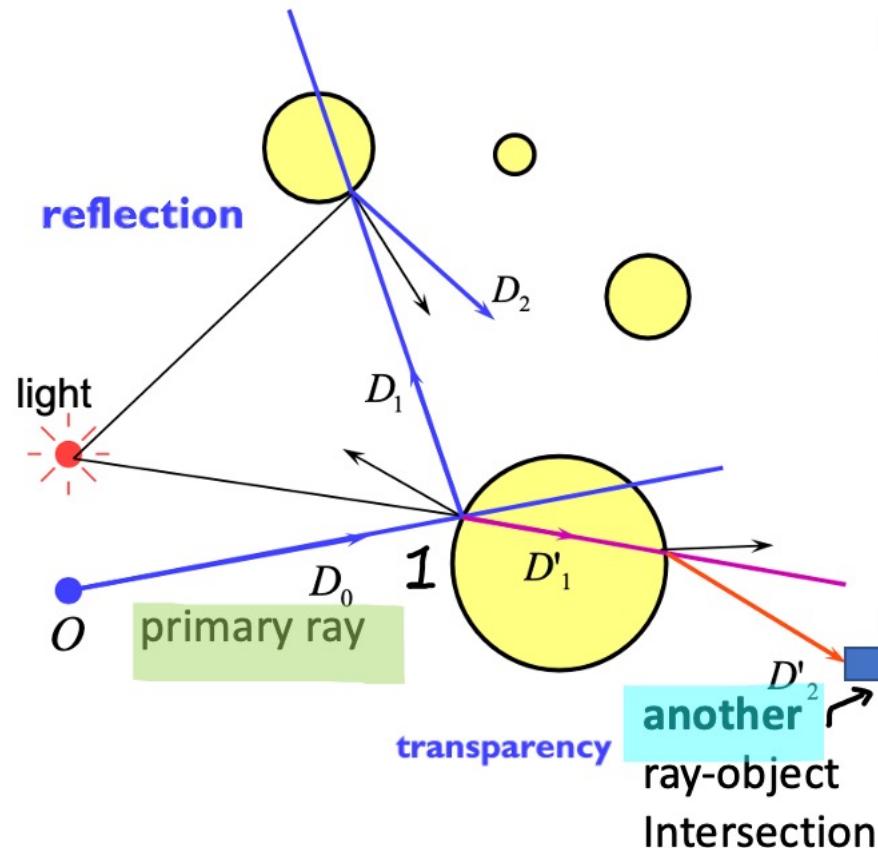
```
Vec3f am = parser->getAmbientLight();
am = am * hit.getMaterial()->getDiffuseColor();
pcolor += am; // ambient color
```

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.

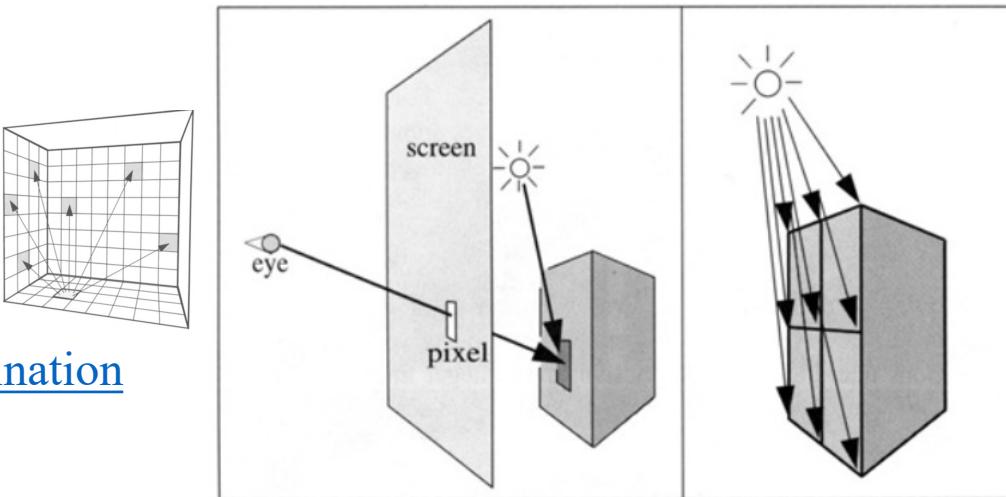
- e.g. in a room, a lot of the light is indirect: it is reflected by walls.



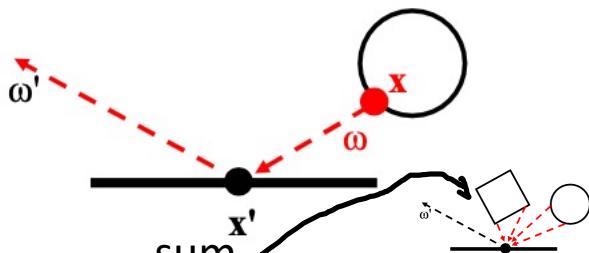
Alternative approaches for global illumination

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

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



The Rendering Equation



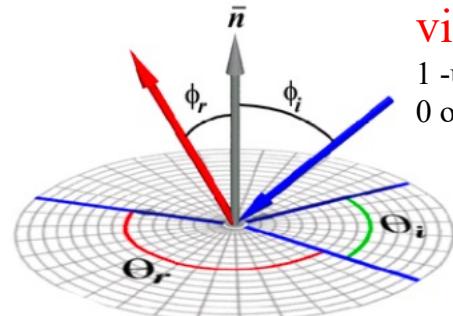
G: Geometric relationship
between the two surfaces

greatest transfer of light energy

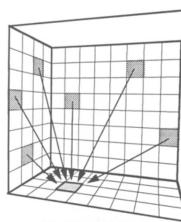
radiance

$$L(x',\omega') = E(x',\omega') + \int_{\text{emitted radiance}} \rho_x(\omega,\omega') L(x,\omega) G(x,x') V(x,x') dA$$

(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)

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

Avoid self-intersection using ϵ (secondary ray)

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

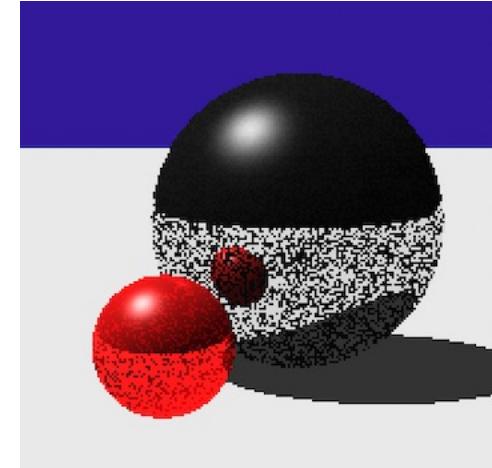
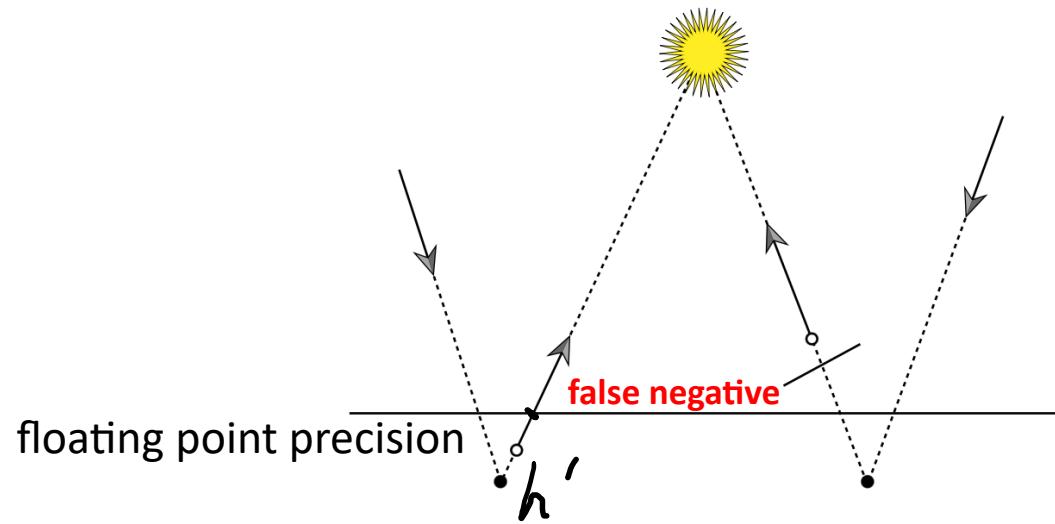
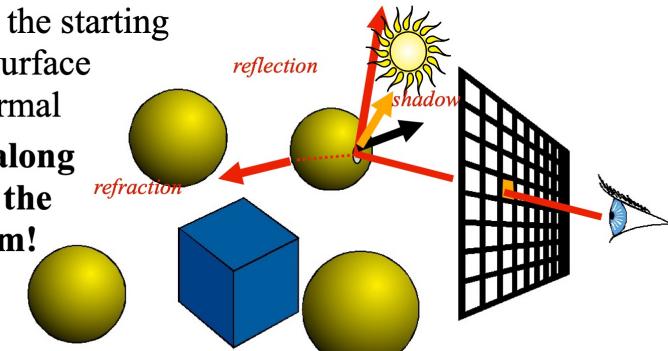


Figure 3.39: Geometric Settings for Rounding-Error Issues That Can Cause Visible Errors in Images. The incident ray on the left intersects the surface. On the left, the computed intersection point (black circle) is slightly below the surface and a too-low “epsilon” offsetting the origin of the shadow 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

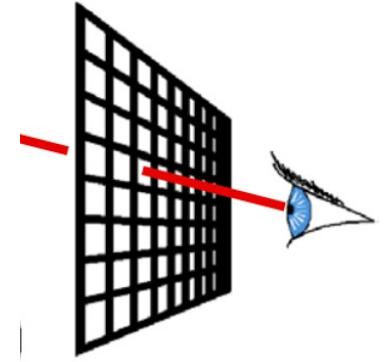
// Keep the intersection if it is closest

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

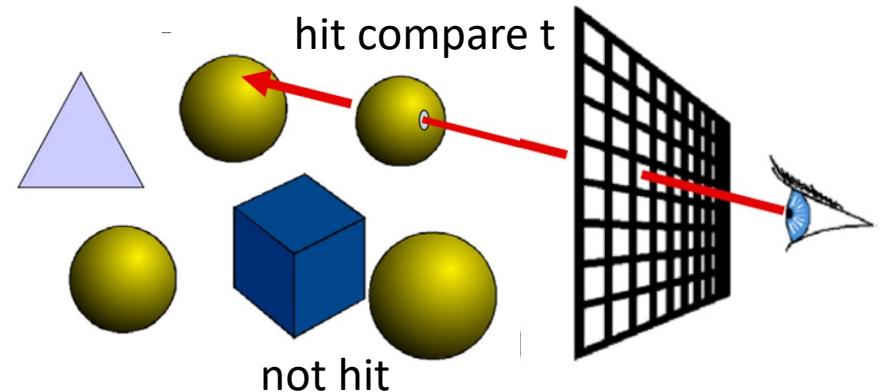
IF the intersection is the closest to the eye (so far)

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

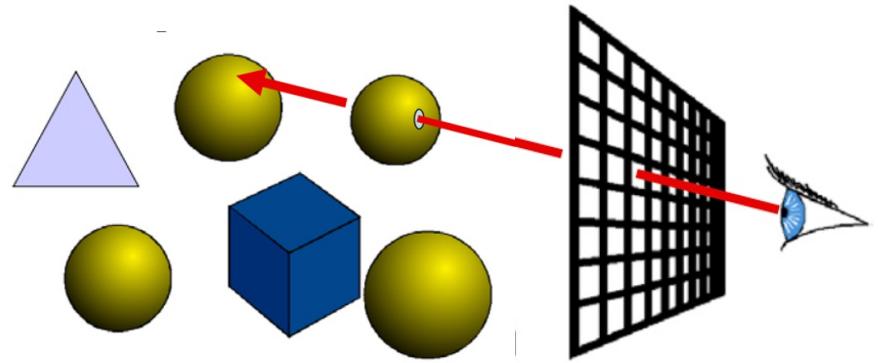
IF the intersection is the closest to the eye (so far)

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
```

```
    color = ambient term
```

```
    For every light
```

```
        cast shadow ray secondary ray
```

```
        color += local shading term
```

```
    If mirror
```

```
        color += colorrefl *
```

```
        trace reflected ray secondary ray
```

primary ray

```
    If transparent
```

```
        color += colortrans *
```

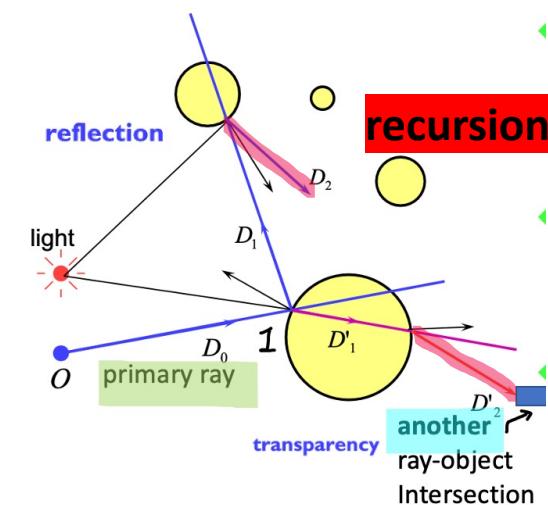
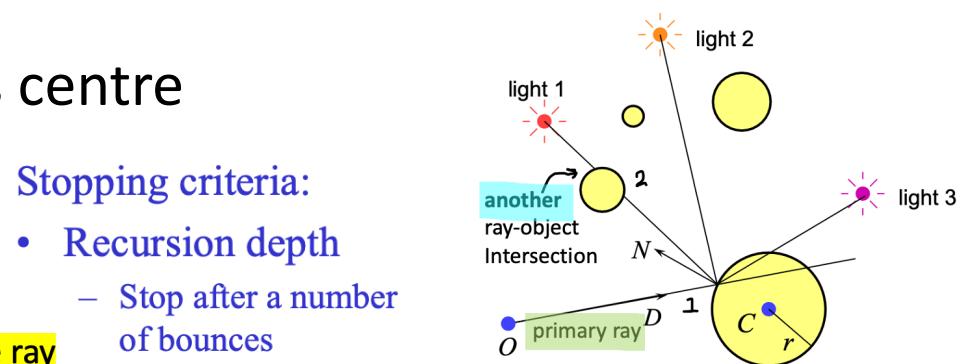
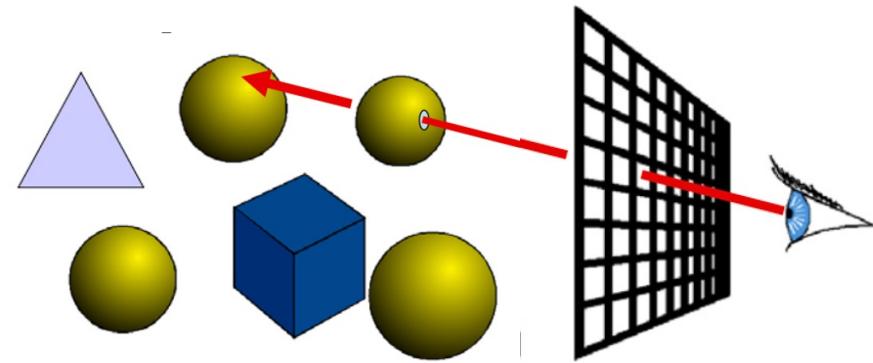
```
        trace transmitted ray secondary ray
```

secondary ray

IF the object is intersected by the ray

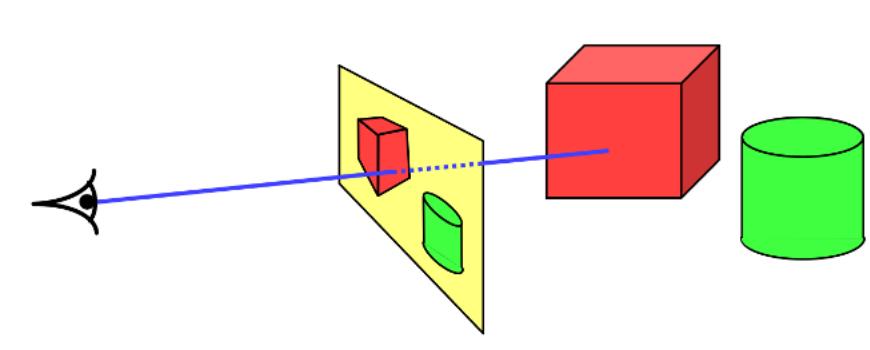
```
}
```

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



Ray Tracing Algorithm

Coding: Step by step explanation



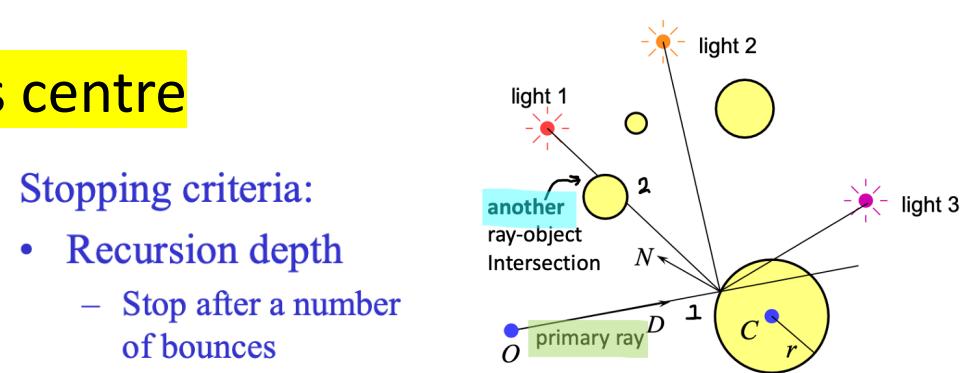
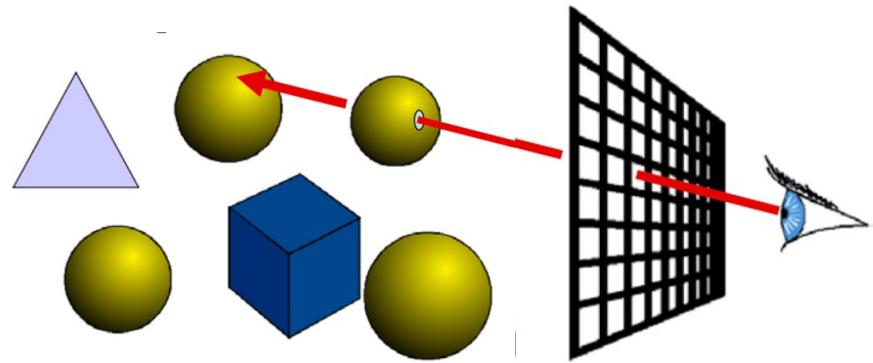
Ray Tracing Algorithm

select an eye point and a screen plane

FOR every pixel in the screen plane:

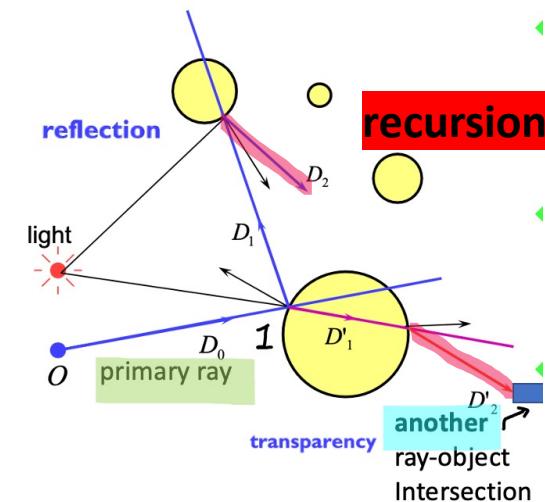
determine the ray from the eye through the pixel's centre

```
vec3color rayTracing ( ray ){  
    Intersect all objects  
    color = ambient term  
    For every light  
        cast shadow ray  
        color += local shading term  
    If mirror  
        color += colorrefl *  
        trace reflected ray  
    If transparent  
        color += colortrans *  
        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 screen 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/PeterHUiTyping/MIT6.837-CG-Fall2004-Assignment/blob/main/MyProject/assignment4_Ray-Tracer/main.C

Ray Tracing Algorithm

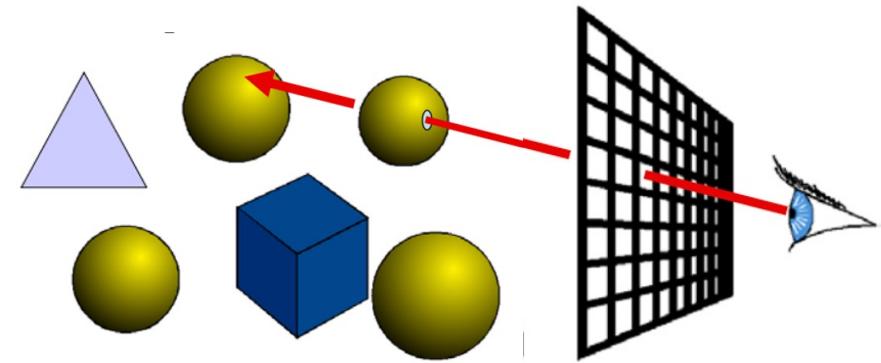
select an eye point and a screen plane

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      secondary ray
        color += local shading term
    If mirror
        color += colorrefl *
        trace reflected ray
    If transparent
        color += colortrans *
        trace transmitted ray
}
```

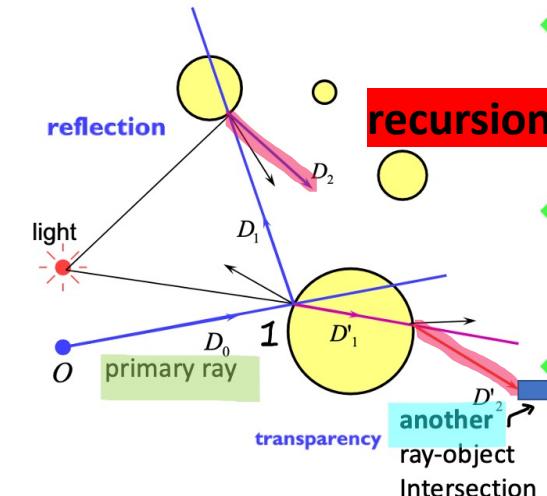
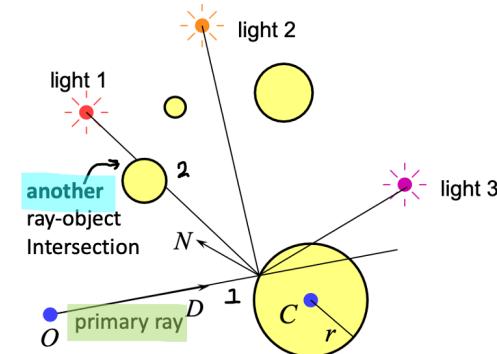
return pixel's colour to closest intersected object's color.



primary 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
        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 no other object is in the way, do shading
                if (!parser->getGroup()->intersectShadowRay(r2, h2, epsilon_shadow)) // epsilon self-shadowing
                {
                    pcolor += hit.getMaterial()->Shade(ray, hit, dirToLight, lcolor); // Diffuse Color + Specular Color
                    RayTree::AddShadowSegment(r2, 0, h2.getT());
                }
            }
        } continue...
    }
}

```

Shading

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

Ray Tracing Algorithm

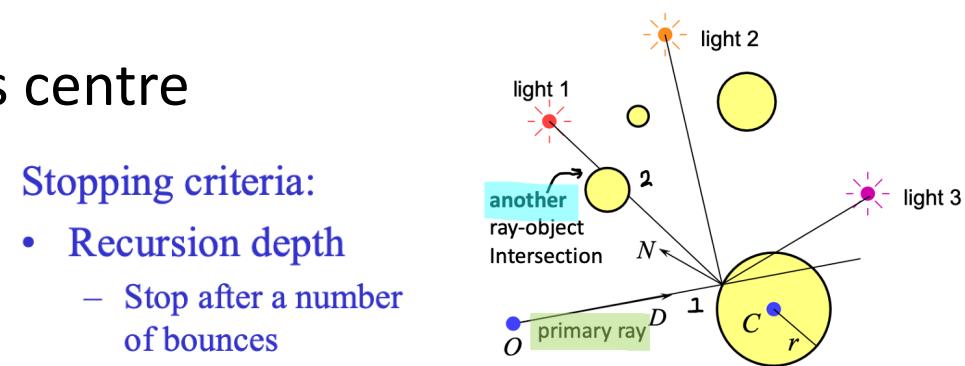
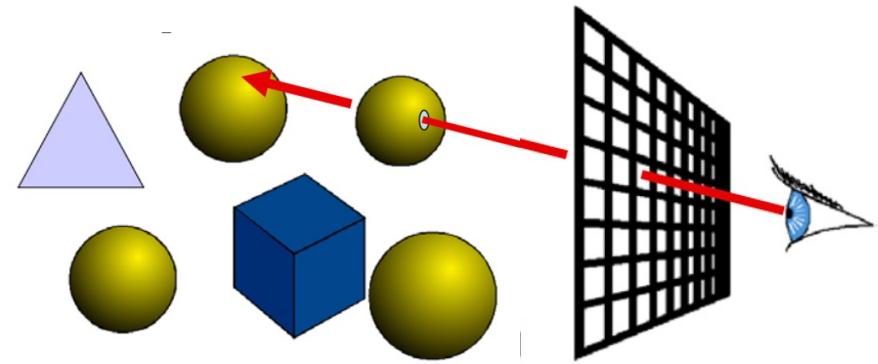
select an eye point and a screen plane

FOR every pixel in the screen plane:

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

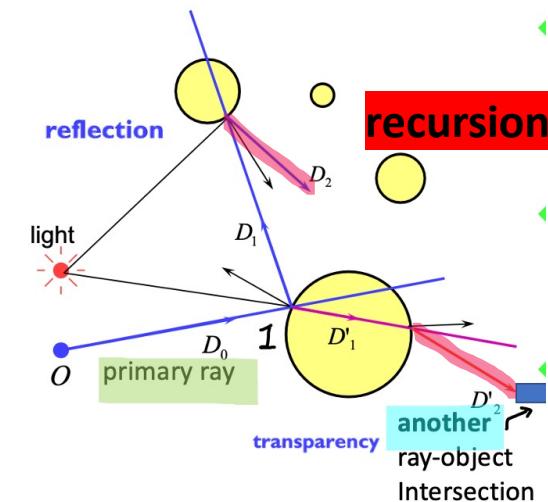
```
vec3color rayTracing ( ray ){
    Intersect all objects
    color = ambient term
    For every light
        cast shadow ray
        color += local shading term
    If mirror
        color += colorrefl *
        trace reflected ray
    If transparent
        color += colortrans *
        trace transmitted 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

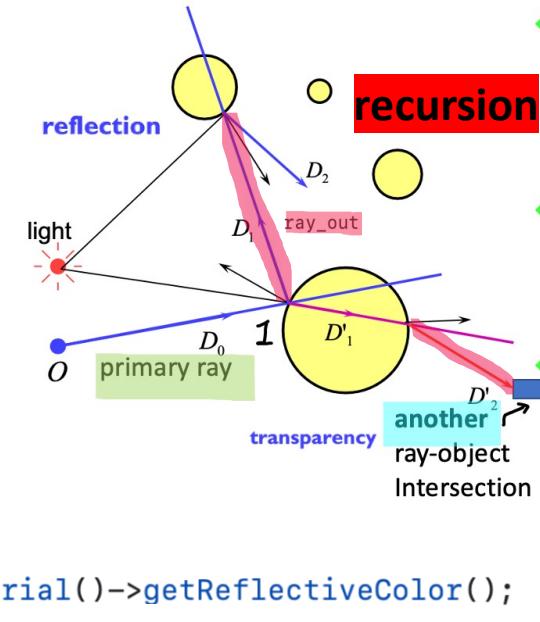


```

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; vec3color
        pcolor += traceRay(ray_out, epsilon, bounces + 1, weight, indexOfRefraction, h) * hit.getMaterial()->getReflectiveColor();
        RayTree::AddReflectedSegment(ray_out, 0, h.getT()); color_refl
    }
}

```



continue...

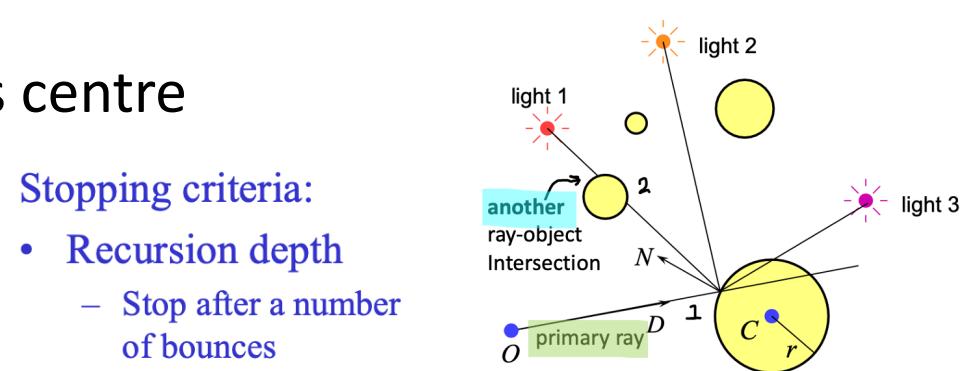
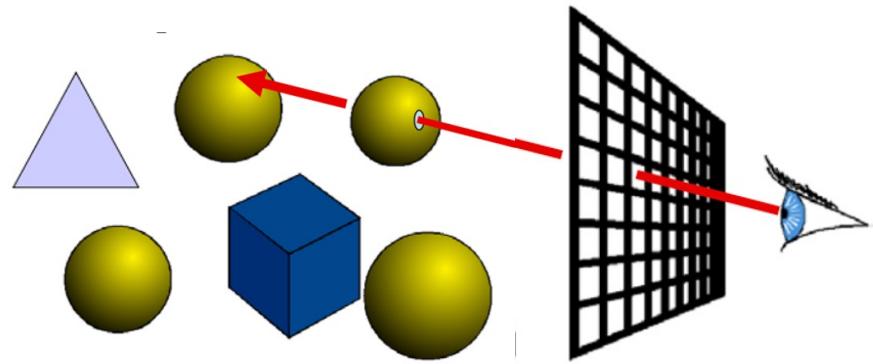
Ray Tracing Algorithm

select an eye point and a screen plane

FOR every pixel in the screen plane:

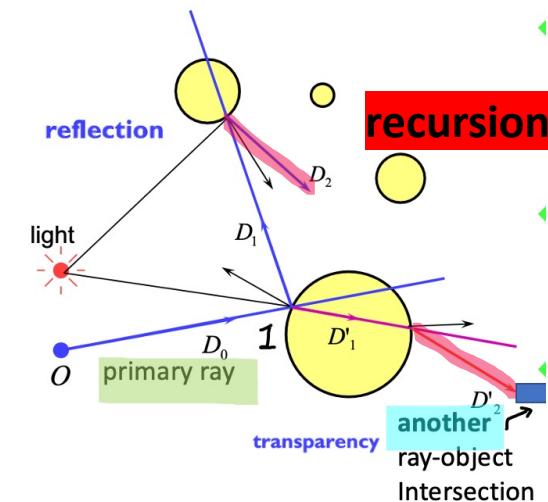
determine the ray from the eye through the pixel's centre

```
vec3color rayTracing ( ray ){  
    Intersect all objects  
    color = ambient term  
    For every light  
        cast shadow ray  
        color += local shading term  
    If mirror  
        color += colorrefl *  
            trace reflected ray  
    If transparent  
        color += colortrans *  
            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



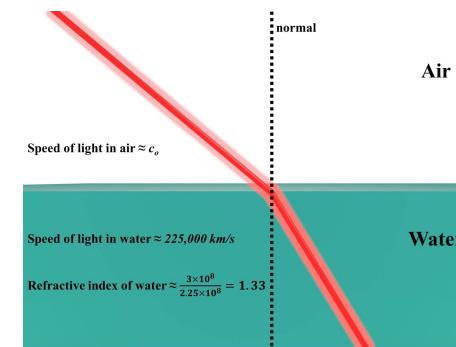
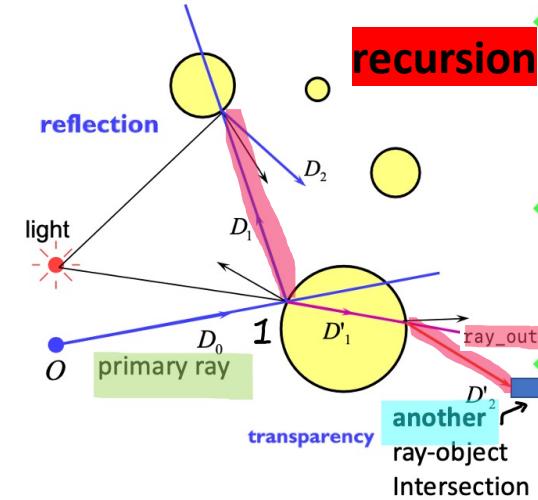
```

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; vec3color
                RefractionColor = traceRay(ray_out, epsilon, bounces + 1, weight, indexOfRefraction, h) * hit.getMaterial()>getTransparentColor();
                pcolor += RefractionColor;
                RayTree::AddTransmittedSegment(ray_out, 0, h.getT());
            }
        }
    }

    return pcolor;
}

```



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!

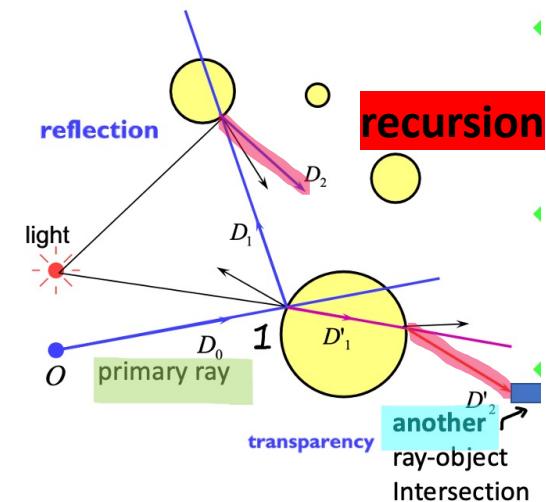
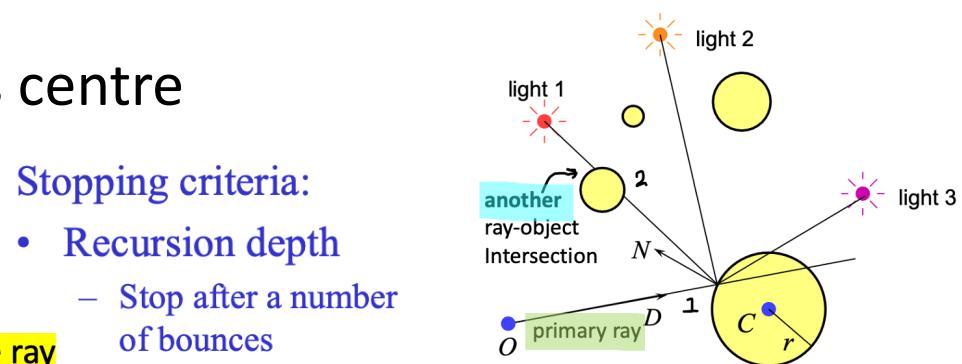
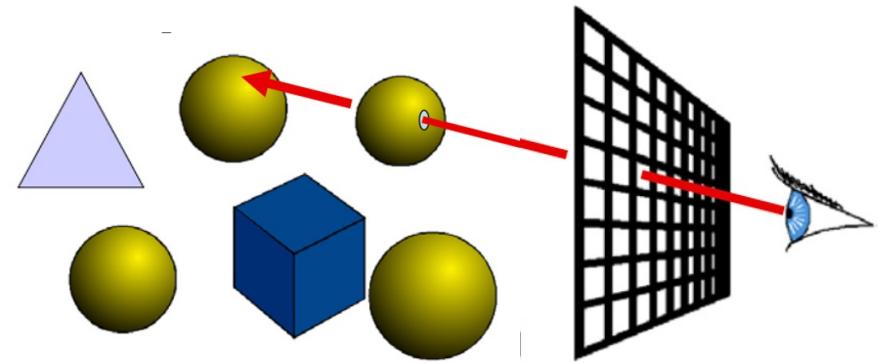
Ray Tracing Algorithm

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

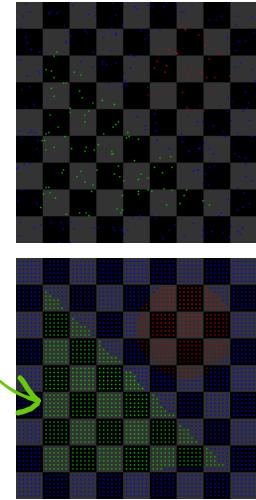
```
recursion  
of Ray  
Tracing  
Algo  
vec3color rayTracing ( ray ){  
    Intersect all objects  
    color = ambient term  
    For every light  
        cast shadow ray secondary ray IF the object is intersected by the ray  
        color += local shading term  
    If mirror  
        color +=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



Ray Tracing Algorithm Analysis

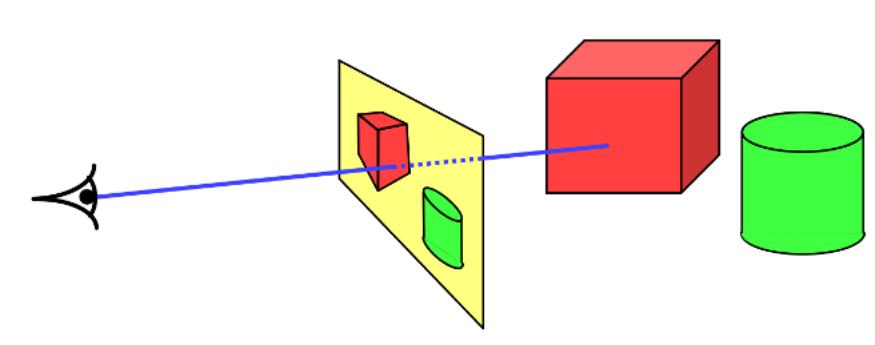
- Ray casting
- Lots of primitives
- Recursive
- Distributed Ray Tracing Effects
 - Soft shadows
 - Anti aliasing
 - Glossy reflection
 - Motion blur
 - Depth of field

$\text{cost} \approx \text{height} * \text{width} *$
num primitives *
intersection cost *
size of recursive ray tree *
num shadow rays *
num supersamples *
num glossy rays *
num temporal samples *
num focal samples *
...



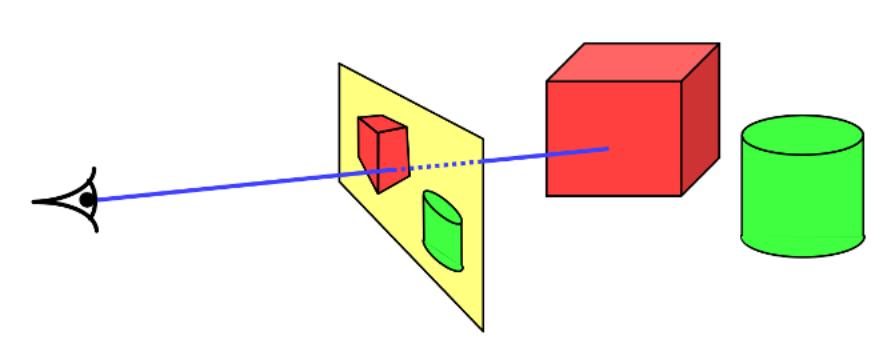


Q&A



Part 2: Ray-Object Intersection

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



IF the object is intersected by the ray

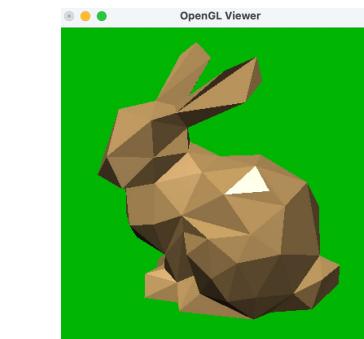
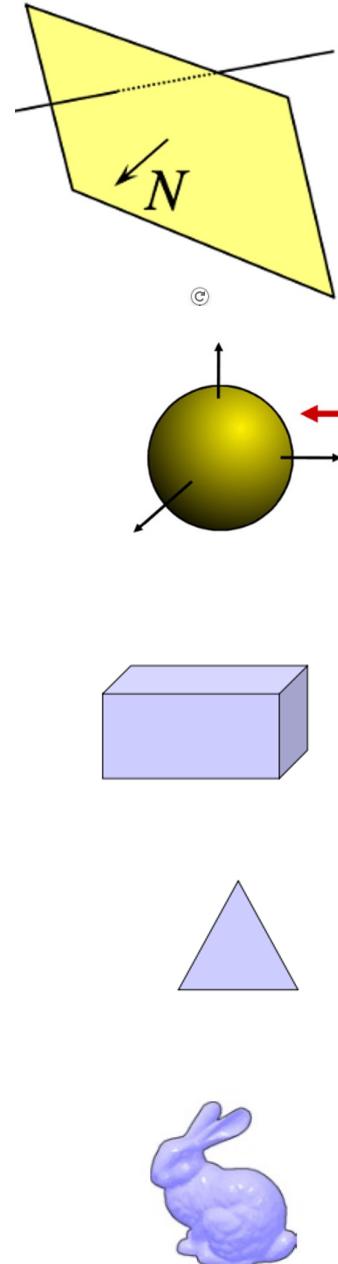
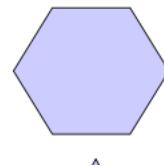
Object

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-Polygon Intersection



- Ray-Bunny Intersection & extra topics...



IF the object is intersected by the ray

Ray

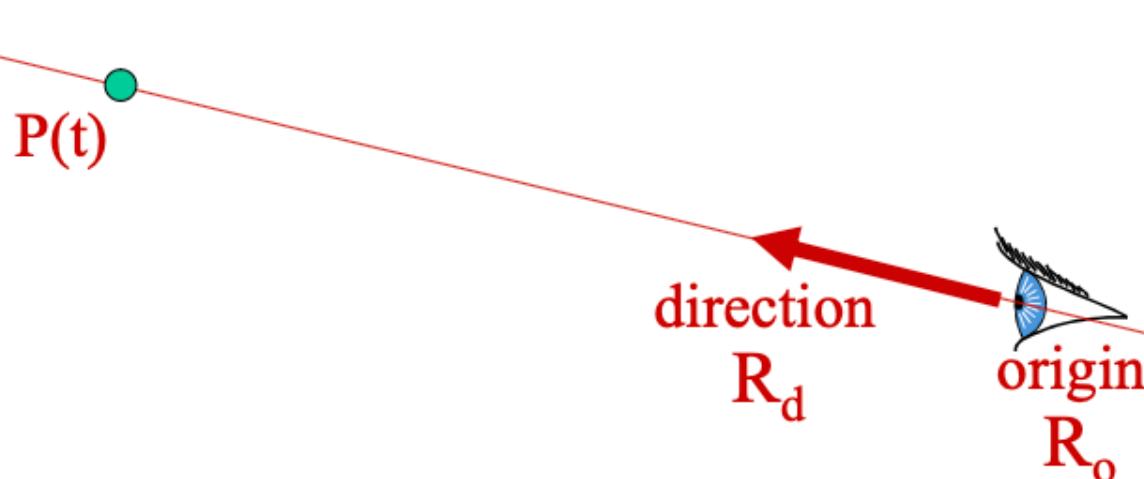
Recall: Ray Representation

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

Ray - Object Intersection

intersection point - $P(t)$

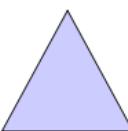
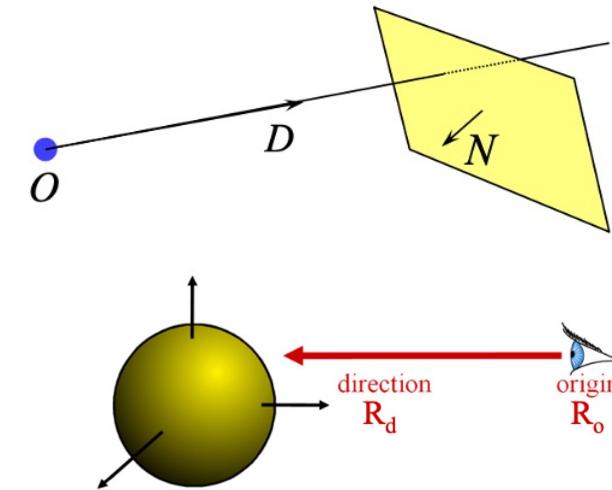
Math problem of Solving t



```
// 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;
};
```

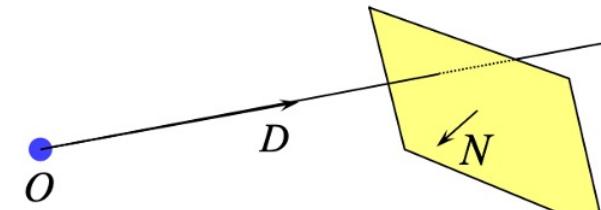
Object Representation

- Plane
 - Sphere
 - Cube / Axis-Aligned Bounding Boxes(AABBs)
 - Triangle (Frequently USED)
- Ray-Box Intersection
 - Ray-Triangle Intersection

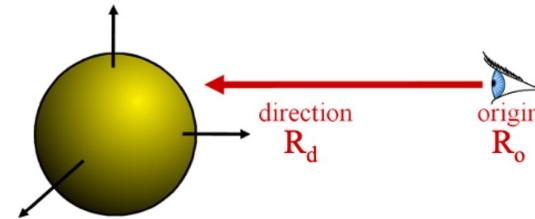


Plane Representation

- Plane



- Sphere



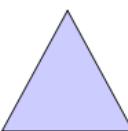
- Cube / Axis-Aligned Bounding Boxes(AABBS)

- Ray-Box Intersection



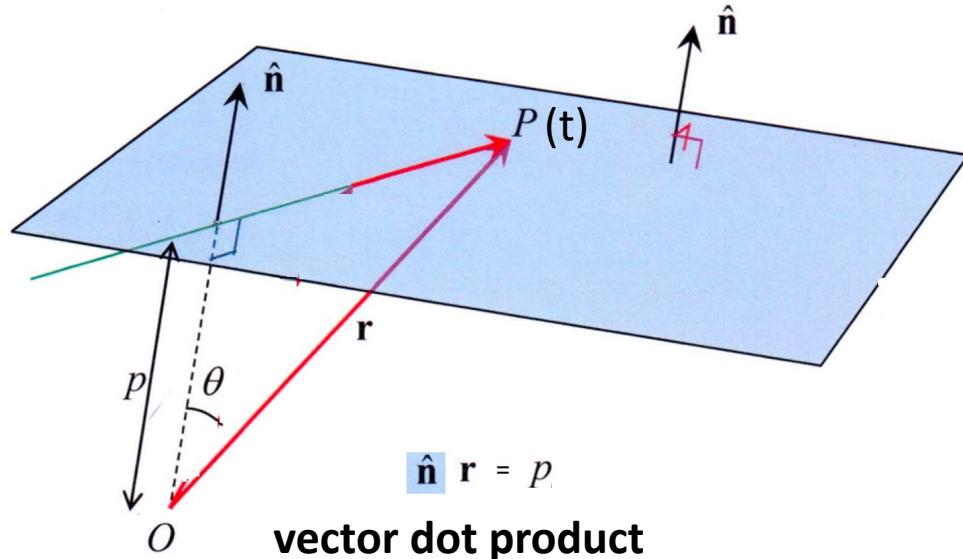
- Triangle (Frequently USED)

- Ray-Triangle Intersection



Plane Representation

Geometric meaning of the plane equation

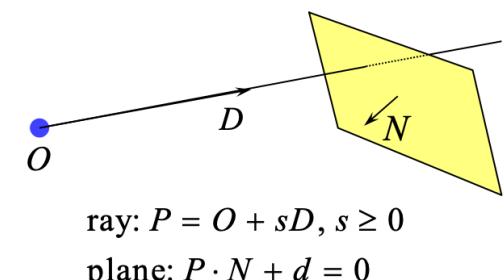


$$P(t) = R_o + t * R_d$$
$$H(P) = n \cdot P + D = 0$$

Distance of origin to plane

```
/* 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];  
};
```

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

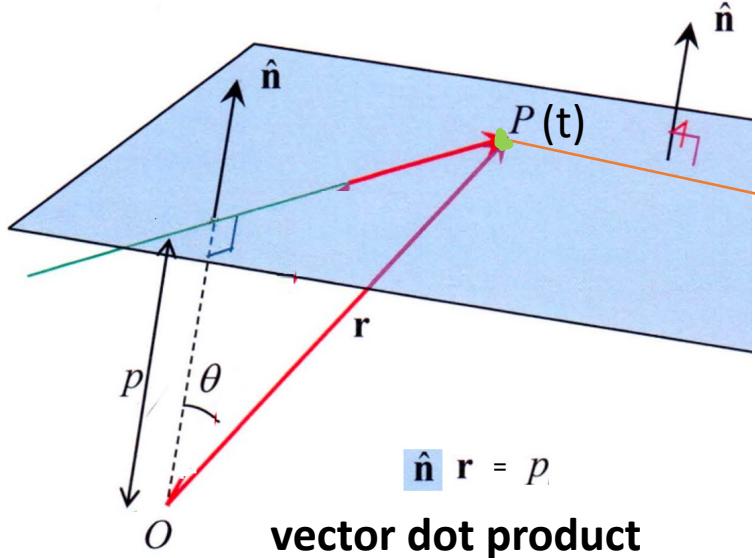


$$s = -\frac{d + N \cdot O}{N \cdot D}$$

Plane Representation

Recall: Ray Representation

Geometric meaning of the plane equation



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



MIT EECS 6.837, Cutler and Durand

32

$$P(t) = R_o + t * R_d$$
$$H(P) = n \cdot P + D = 0$$

IF the object is intersected by the ray

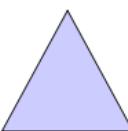
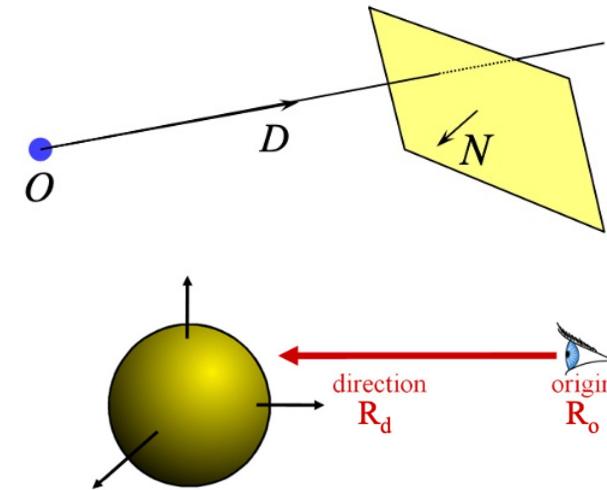
Ray - Object Intersection

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

Math problem of Solving t

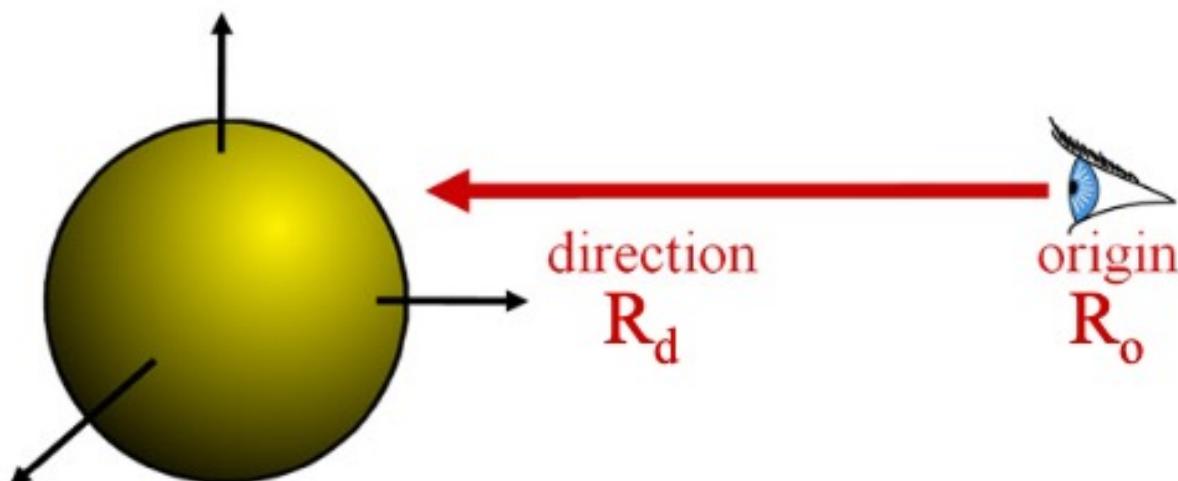
Sphere Representation

- Plane
 - **Sphere**
 - Cube / Axis-Aligned Bounding Boxes(AABBs)
 - Triangles (Frequently USED)
- Ray-Box Intersection
 - 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;
};
```

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

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

IF the object is intersected by the ray

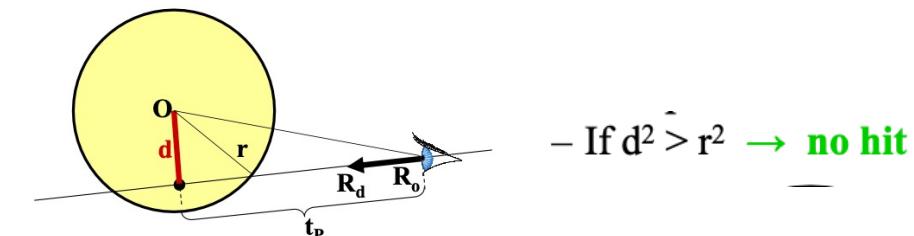
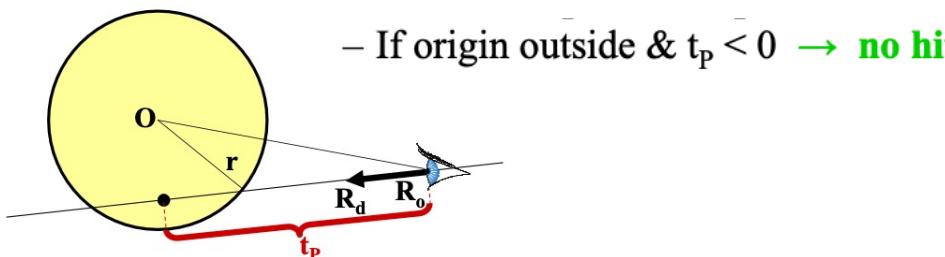
Ray - Object Intersection

Math problem of Solving t

Sphere Representation?

Geometric Ray-Sphere Intersection

- Is ray origin **inside/outside/on** sphere?
- Find closest point to sphere center, $t_p = -\mathbf{R}_o \cdot \mathbf{R}_d$.
- Find squared distance: $d^2 = \mathbf{R}_o \cdot \mathbf{R}_o - t_p^2$
- 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 $\rightarrow t = t_p + t'$



$$P(t) = \mathbf{R}_o + t * \mathbf{R}_d \quad H(P) = P \cdot P - r^2 = 0$$

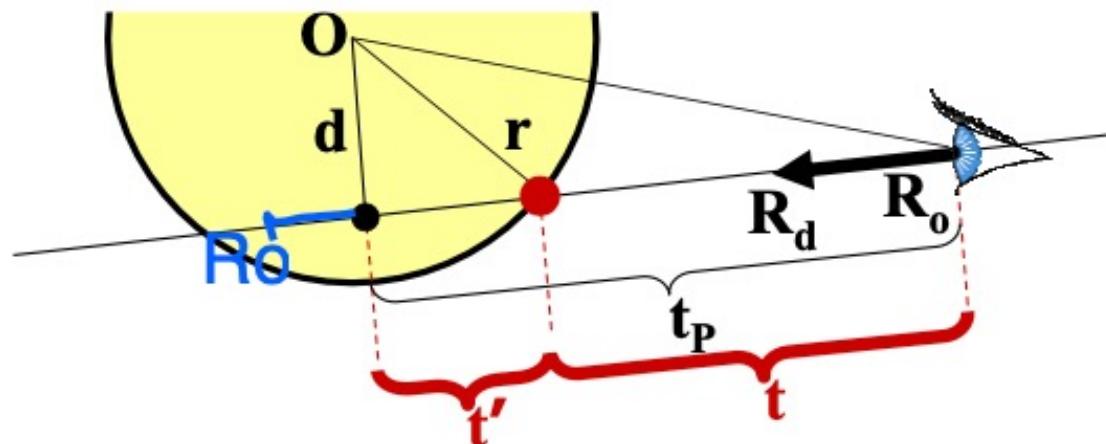
$$\mathbf{R}_d \cdot \mathbf{R}_d t^2 + 2\mathbf{R}_d \cdot \mathbf{R}_o t + \mathbf{R}_o \cdot \mathbf{R}_o - r^2 = 0$$

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

IF the object is intersected by the ray

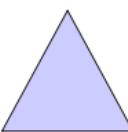
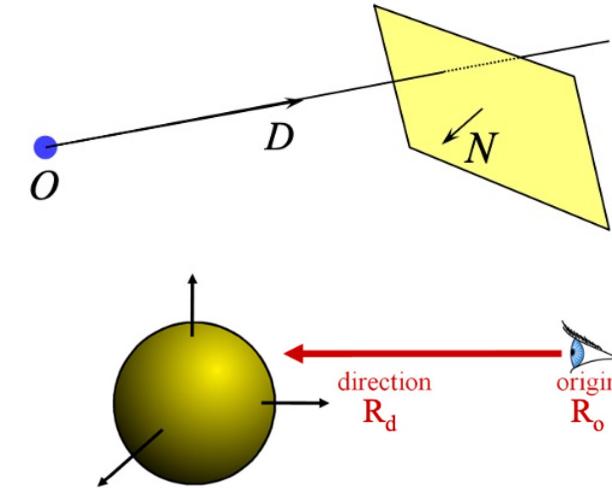
Ray - Object Intersection

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

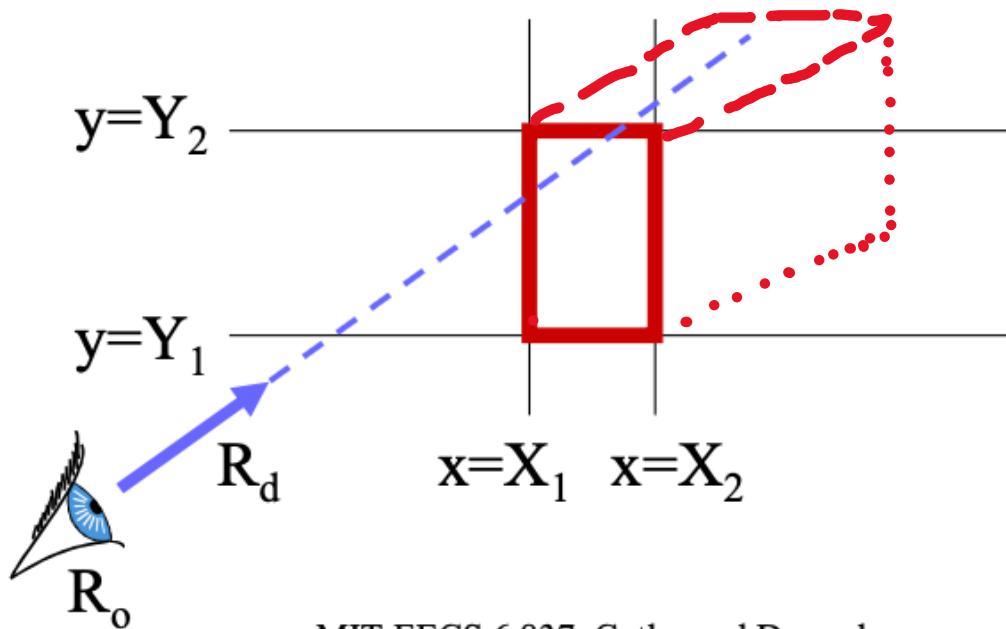
Cube or Axis-Aligned Bounding Boxes (AABBs)

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



(X_1, Y_1, Z_1) ·

(X_2, Y_2, Z_2)

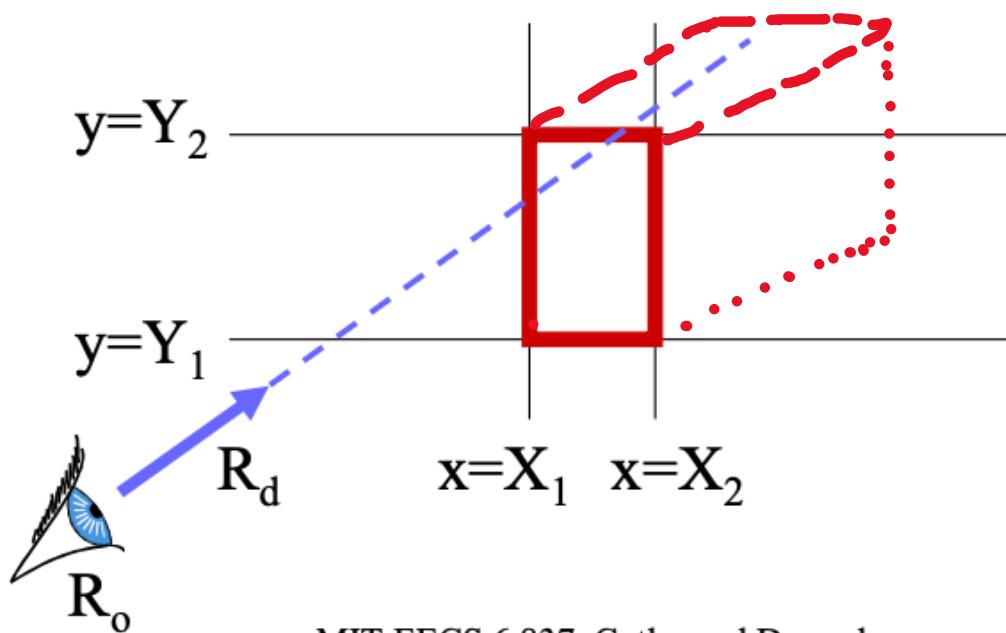


```
class Box{  
public:  
    // Bottom left point of the box : 3D coordinate  
    const float bmin[3];  
    // Top right point of the box: 3D coordinate  
    const float bmax[3];  
};
```

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: $P(t) = R_o + tR_d$



(X_2, Y_2, Z_2)

(X_1, Y_1, Z_1)

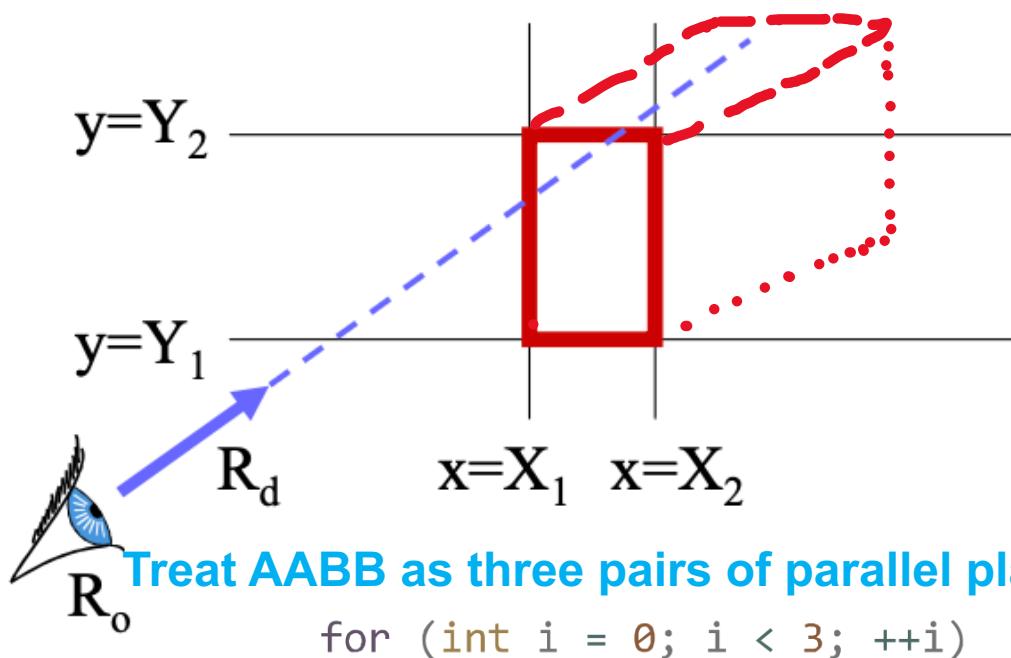
Naïve Ray-Box Intersection

- 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: $P(t) = R_o + tR_d$



(X_1, Y_1, Z_1)



(X_2, Y_2, Z_2)

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

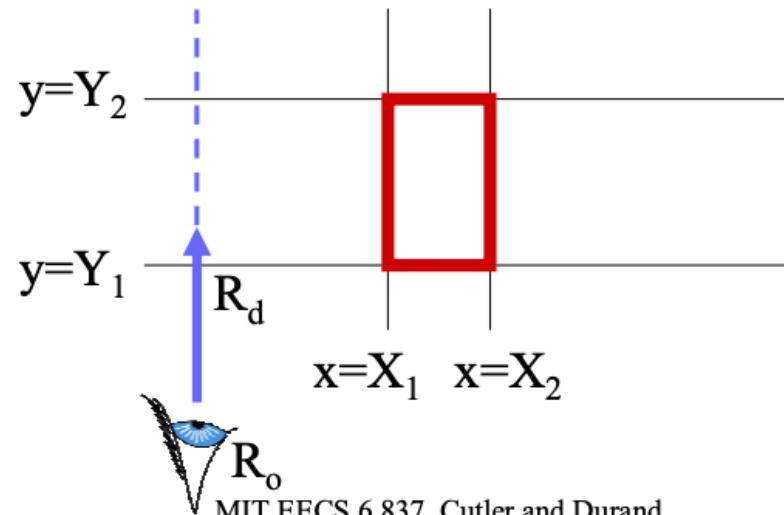
Cube or Axis-Aligned Bounding Boxes (AABBS)

(X_2, Y_2, Z_2)

Test if Parallel

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

(X_1, Y_1, Z_1)



Ray-Box Intersection

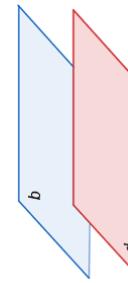
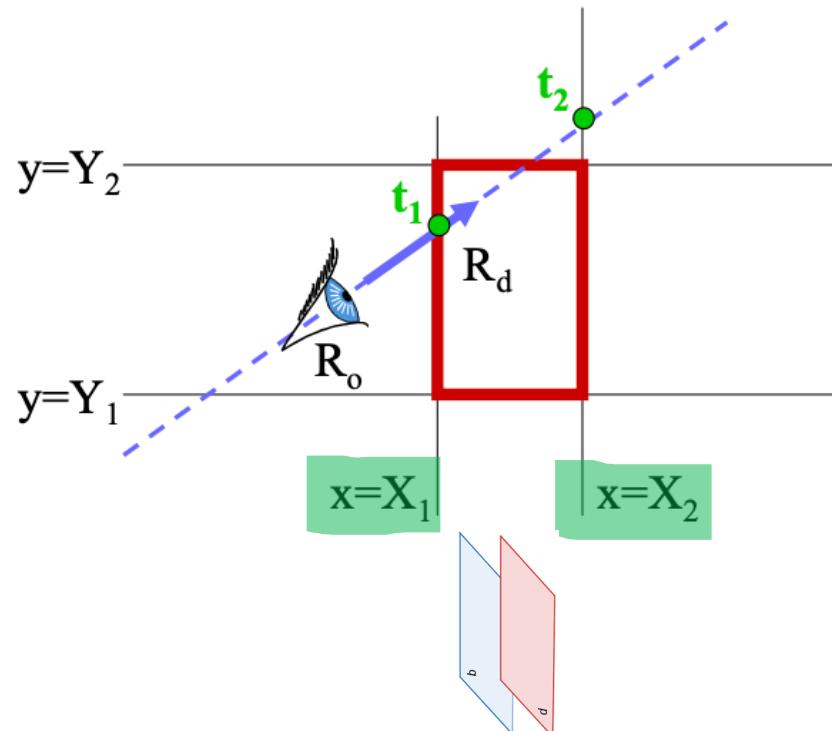
Cube or Axis-Aligned Bounding Boxes (AABBs)

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
 t_1, t_2 for x axis
Similarly,
 t_3, t_4 for y axis
 t_5, t_6 for z axis

Ray - Object Intersection

Math problem of Solving t

Cube or Axis-Aligned Bounding Boxes (AABBs)

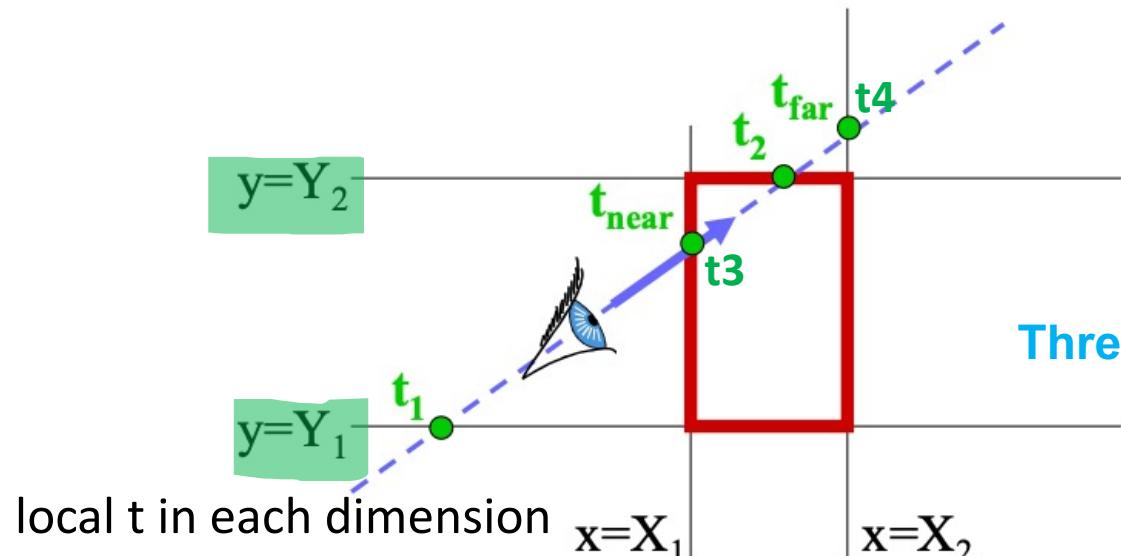
IF the object is intersected by the ray

Maintain t_{near} & t_{far}

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

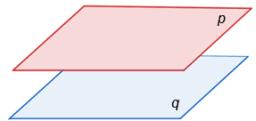
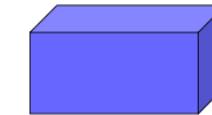
- Closest & farthest intersections *on the object*

- If $t_1 > t_{near}$, $t_{near} = t_1$
- If $t_2 < t_{far}$, $t_{far} = t_2$



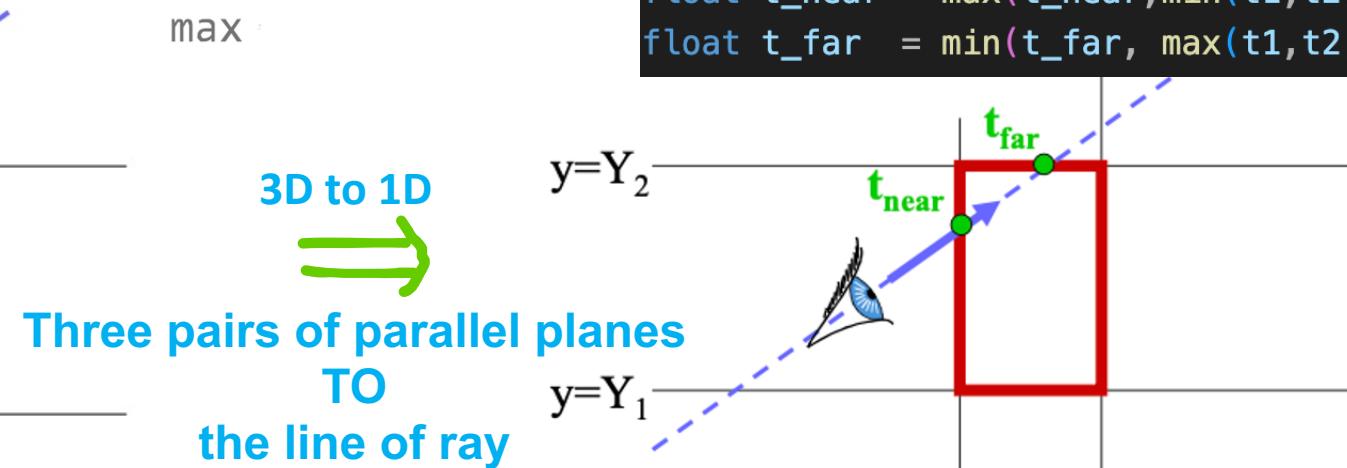
"first" intersection point
"exit" intersection point of the box

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



of each parallel plane

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



```
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));
```

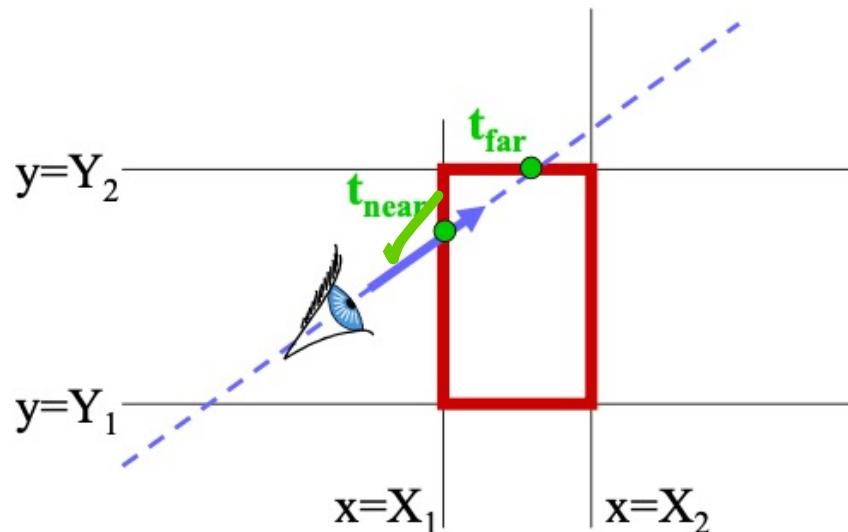
t₁, t₂ for x axis
t₃, t₄ for y axis
t₅, t₆ for z axis

Ray-Box Intersection

Cube or Axis-Aligned Bounding Boxes (AABBs)

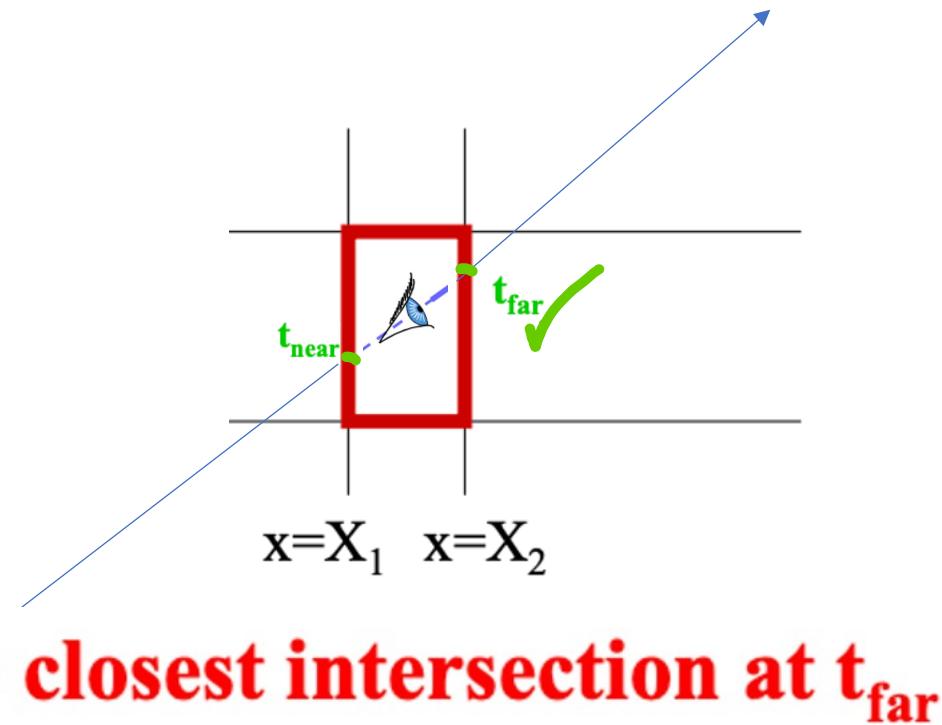
Return the Correct Intersection

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



closest intersection at t_{near}

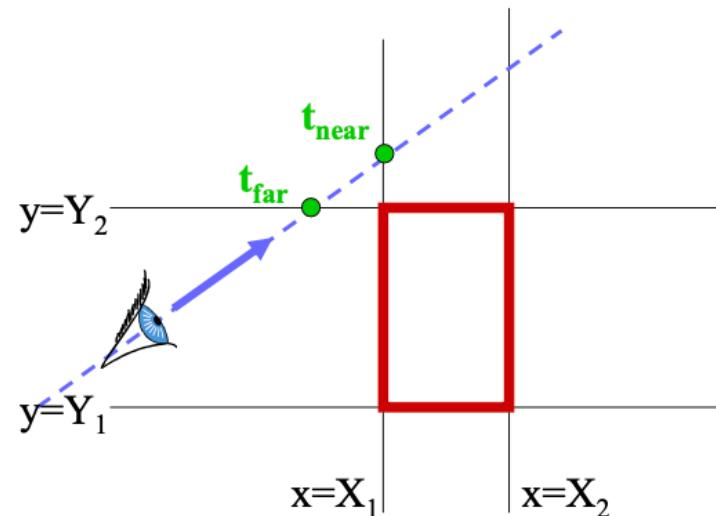
An intersection can only happen if the ray goes “into” the bounding region for all t values before it exits any of them.



closest intersection at t_{far}

Is there an Intersection?

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



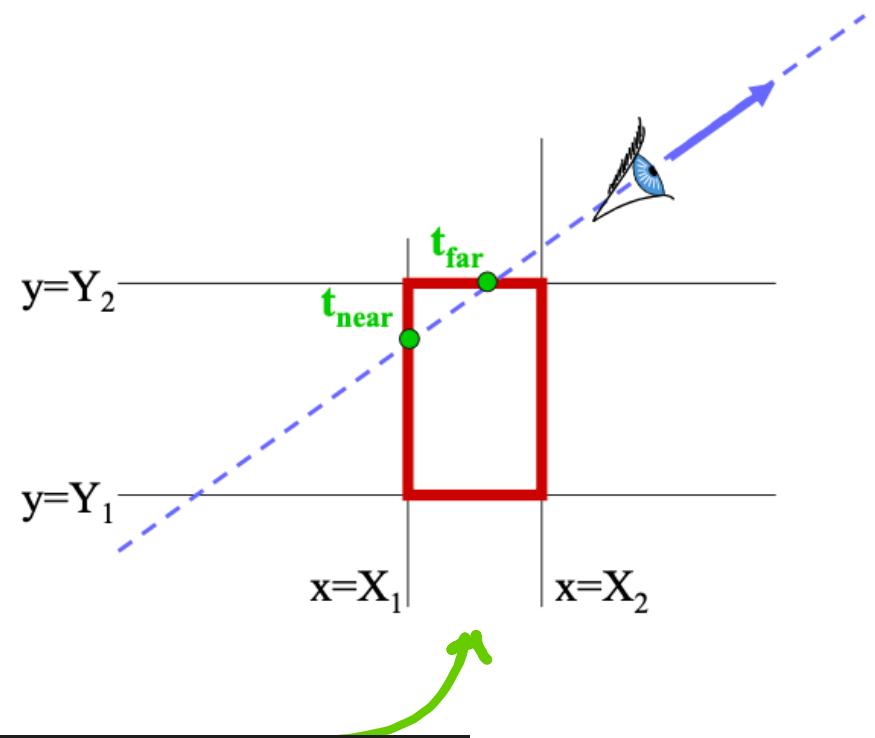
MIT EECS 6.837, Cutler and Durand

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

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

Is the Box Behind the Eyepoint?

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



Cube or Axis-Aligned Bounding Boxes (AABBs)

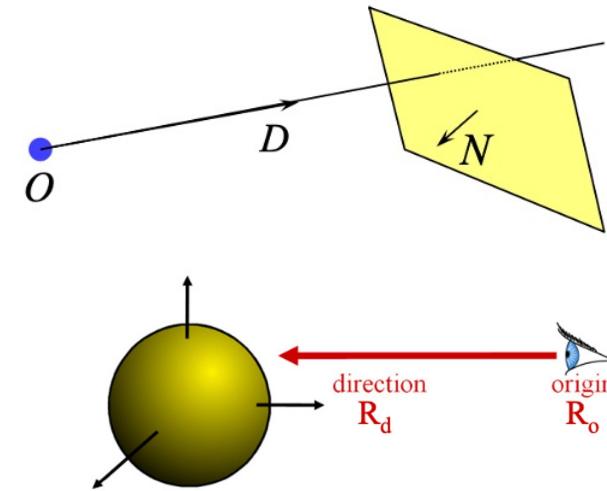
```
bool IntersectAABB( Ray & ray, const Box & b ) {
    float t_near, t_far;
    for (int i = 0; i < 3; ++i) {
        float t1 = (b.bmin[i] - ray.o[i]) / ray.D[i];
        float t2 = (b.bmax[i] - ray.o[i]) / ray.D[i];
        t_near = max(t_near,min(t1,t2));
        t_far = min(t_far,max(t1,t2));
    }
    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 $R_{ox} > X_2 \rightarrow \text{no intersection}$
- For each dimension, calculate intersection distances t_1 and t_2
 - $t_1 = (X_1 - R_{ox}) / R_{dx}$ $t_2 = (X_2 - R_{ox}) / R_{dx}$
 - If $t_1 > t_2$, swap
 - Maintain t_{near} and t_{far} (closest & farthest intersections so far)
 - If $t_1 > t_{\text{near}}$, $t_{\text{near}} = t_1$ If $t_2 < t_{\text{far}}$, $t_{\text{far}} = t_2$
- If $t_{\text{near}} > t_{\text{far}} \rightarrow \text{box is missed}$
- If $t_{\text{far}} < t_{\text{min}} \rightarrow \text{box is behind}$
- If $t_{\text{near}} > t_{\text{min}} \rightarrow \text{closest intersection at } t_{\text{near}}$
- Else $\rightarrow \text{closest intersection at } t_{\text{far}}$

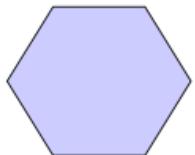
Triangle Representation

- Plane
 - Sphere
 - Cube / Axis-Aligned Bounding Boxes(AABBS)
 - Triangle (Frequently USED)
- Ray-Box Intersection
 - Ray-Triangle Intersection

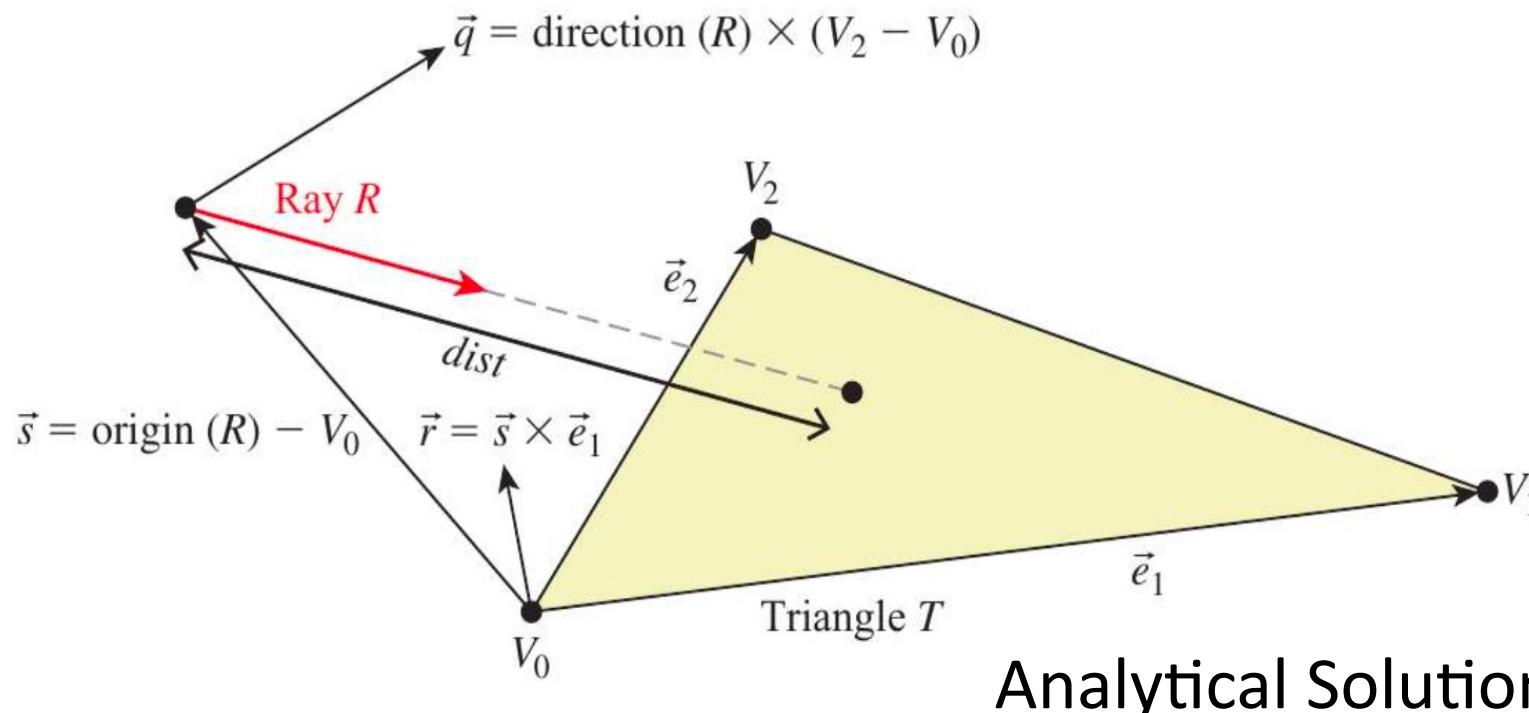


Ray-Triangle Intersection

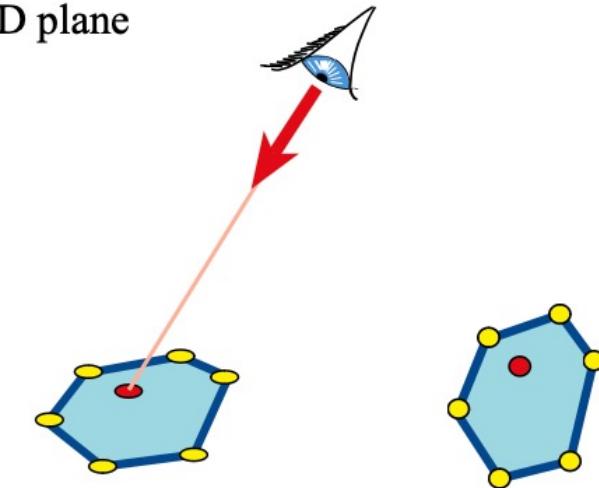
- Use ray-polygon
- Ray-Polygon Intersection



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

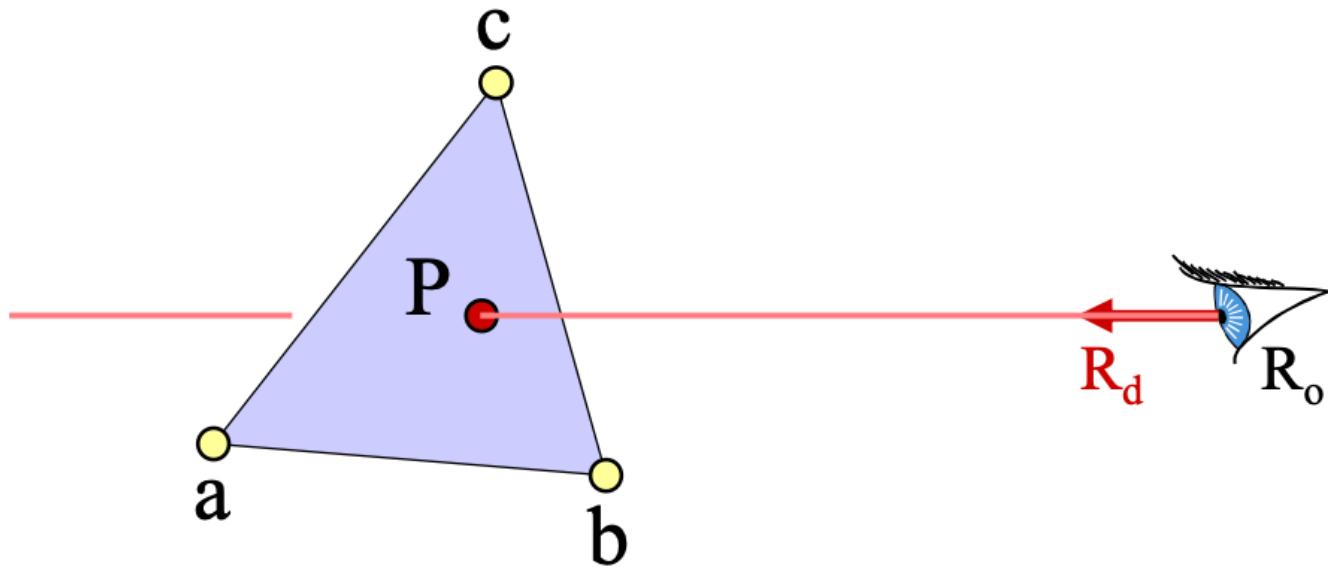


- Ray-plane intersection
- Test if intersection is in the polygon
 - Solve in the 2D plane



Ray-Triangle Intersection

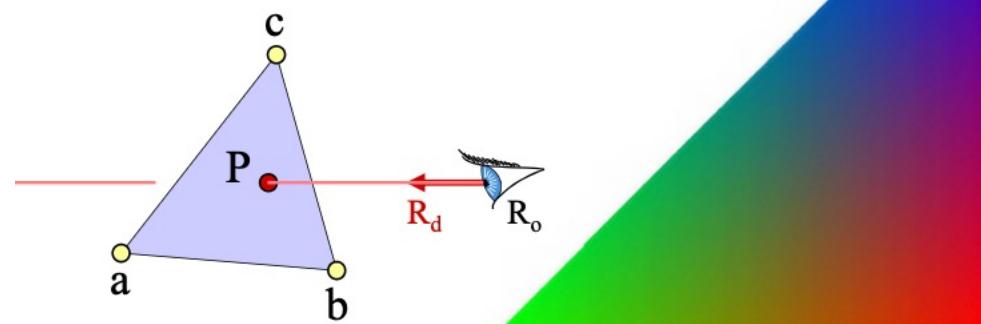
- Use ray-polygon
- Or try to be smarter
 - Use barycentric coordinates



MIT EECS 6.837, Cutler and Durand

Advantages of Barycentric Intersection

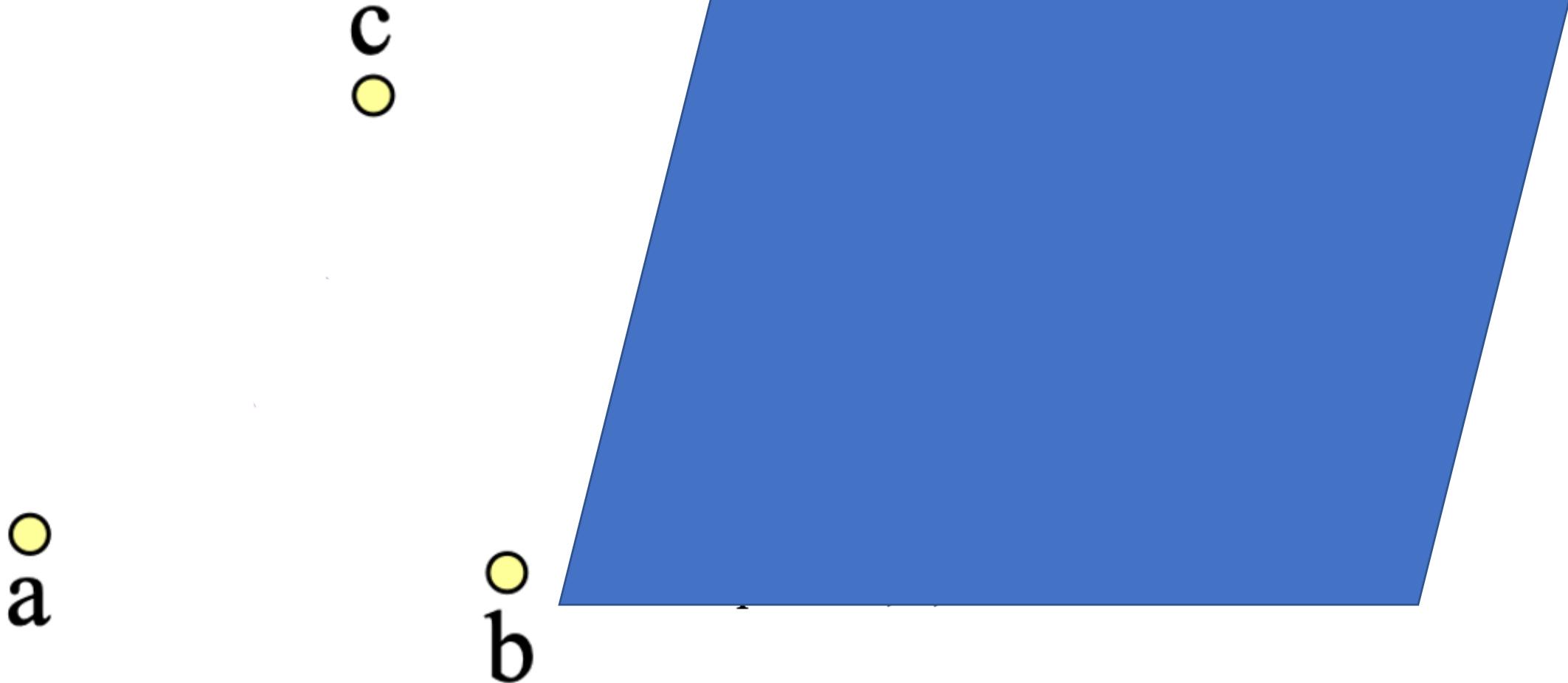
- Efficient
- Stores no plane equation
- Get the barycentric coordinates for free
 - Useful for interpolation, texture mapping



MIT EECS 6.837, Cutler and Durand

Barycentric Definition of a Plane

[Möbius, 1827]



Barycentric Definition of a Plane

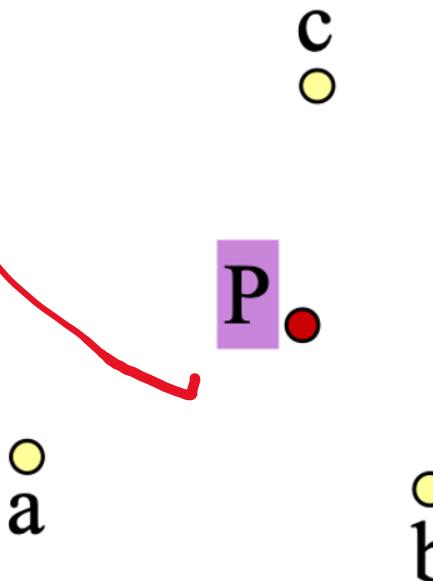
- $P(\alpha, \beta, \gamma) = \alpha a + \beta b + \gamma c$

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

- Is it explicit or implicit?

[Möbius, 1827]

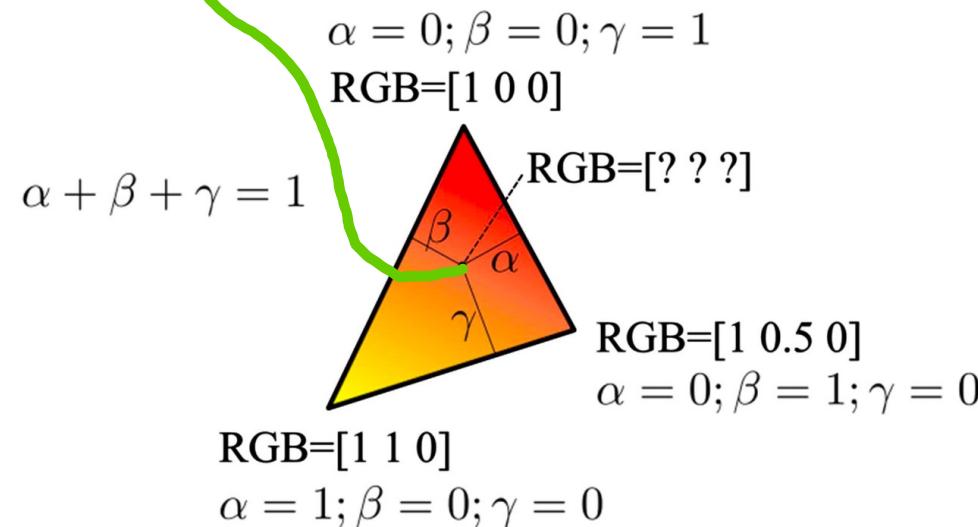
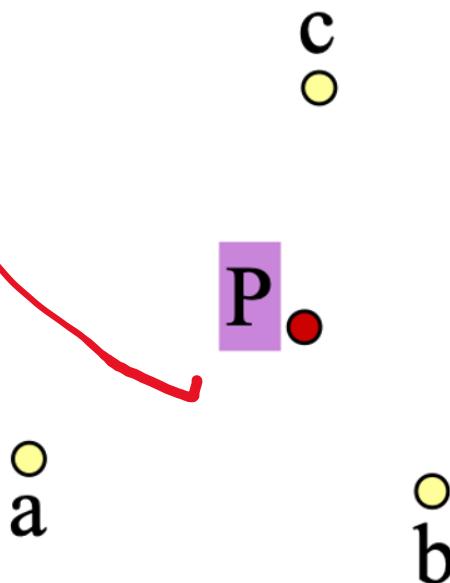
```
class Tri{  
public:  
    // Three points of the triangle: 3D coordinate  
    const float a[3];  
    const float b[3];  
    const float c[3];  
};
```



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 .

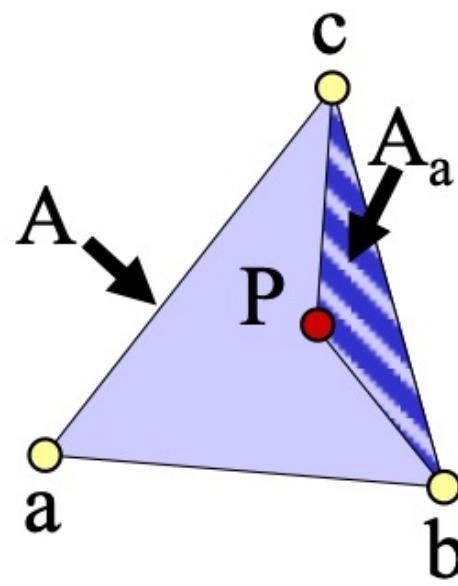
Barycentric Definition of a Plane

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



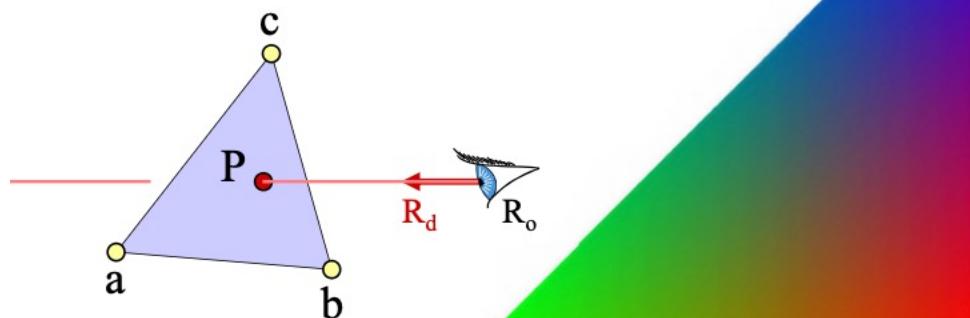
How Do We Compute α , β , γ ?

- Ratio of opposite sub-triangle area to total area
 - $\underline{\alpha} = A_a/A$ $\underline{\beta} = A_b/A$ $\underline{\gamma} = A_c/A$
- 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 $\alpha = 1 - \beta - \gamma$

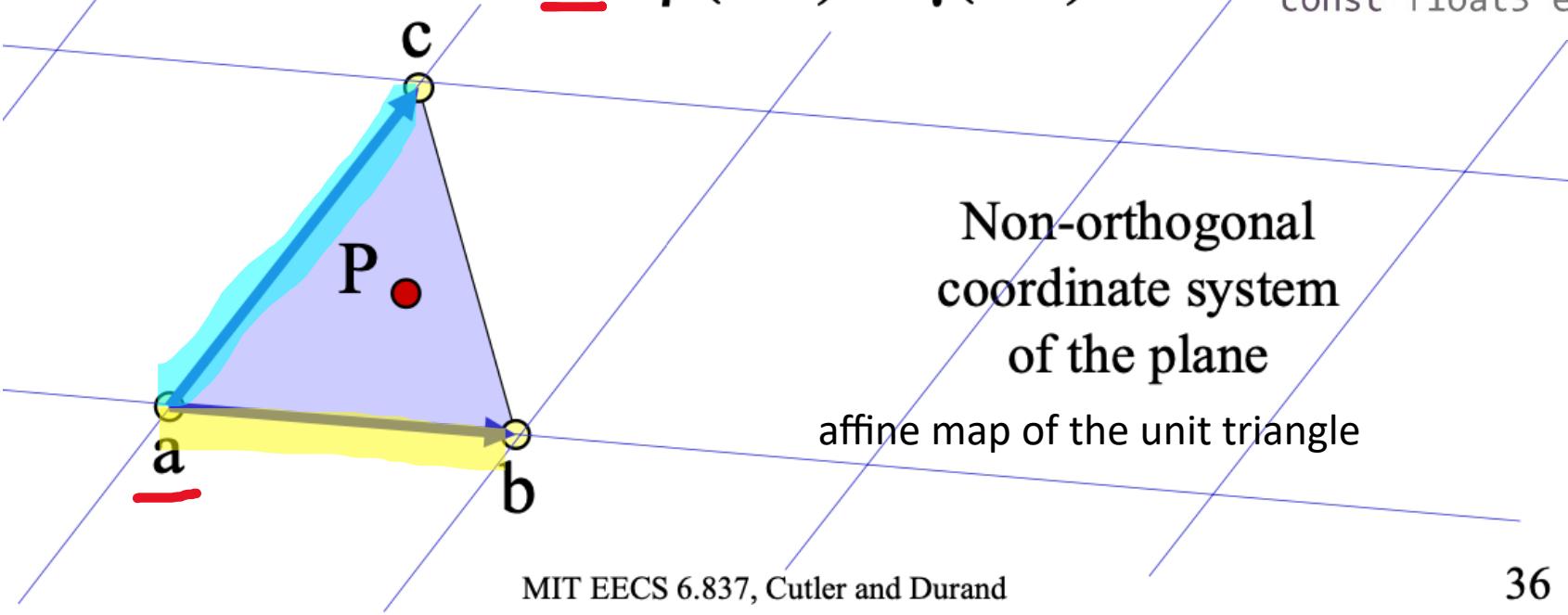
$$P(\alpha, \beta, \gamma) = \alpha a + \beta b + \gamma c \quad \text{rewrite}$$

$$\begin{aligned} P(\beta, \gamma) &= (1 - \beta - \gamma)a + \beta b + \gamma c \\ &= a + \beta(b-a) + \gamma(c-a) \end{aligned}$$

```
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



Math problem of Solving t

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

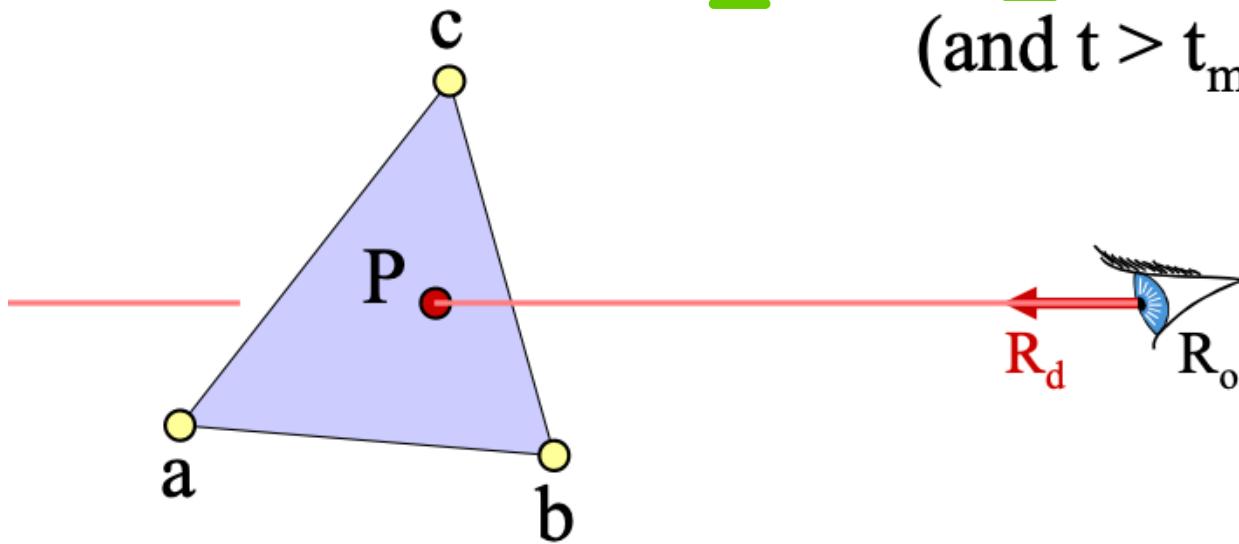
$$P(\alpha, \beta, \gamma) = \alpha a + \beta b + \gamma c \quad \text{rewrite}$$

$$\begin{aligned} P(\beta, \gamma) &= (1 - \beta - \gamma)a + \beta b + \gamma c \\ &= a + \underline{\beta(b-a)} + \underline{\gamma(c-a)} \end{aligned}$$

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

$$P(t) = R_o + \underline{t} * R_d$$

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



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

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

$$\left. \begin{array}{l} R_{ox} + tR_{dx} = a_x + \beta(b_x - a_x) + \gamma(c_x - a_x) \\ R_{oy} + tR_{dy} = a_y + \beta(b_y - a_y) + \gamma(c_y - a_y) \\ R_{oz} + tR_{dz} = a_z + \beta(b_z - a_z) + \gamma(c_z - a_z) \end{array} \right\} \begin{array}{l} \text{3D space } z \\ \text{3 equations,} \\ \text{3 unknowns} \end{array}$$

- Regroup & write in matrix form:

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

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

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

$$\left. \begin{array}{l} R_{ox} + tR_{dx} = a_x + \beta(b_x - a_x) + \gamma(c_x - a_x) \\ R_{oy} + tR_{dy} = a_y + \beta(b_y - a_y) + \gamma(c_y - a_y) \\ R_{oz} + tR_{dz} = a_z + \beta(b_z - a_z) + \gamma(c_z - a_z) \end{array} \right\} \begin{array}{l} \text{3D space } z \\ \text{3 equations,} \\ \text{3 unknowns} \end{array}$$

==

$$\left\{ \begin{array}{l} 3x + 2y + z = 7 \\ 2x - 3y + 2z = 3 \\ 4x - y - 6z = -9 \end{array} \right.$$

- Regroup & write in matrix form:

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

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

$$Ax = b$$

Intersection with Barycentric Triangle

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

$$\left. \begin{array}{l} R_{ox} + tR_{dx} = a_x + \beta(b_x - a_x) + \gamma(c_x - a_x) \\ R_{oy} + tR_{dy} = a_y + \beta(b_y - a_y) + \gamma(c_y - a_y) \\ R_{oz} + tR_{dz} = a_z + \beta(b_z - a_z) + \gamma(c_z - a_z) \end{array} \right\} \begin{array}{l} \text{3D space } z \\ \text{3 equations,} \\ \text{3 unknowns} \end{array}$$

==

$$\left\{ \begin{array}{l} 3x + 2y + z = 7 \\ 2x - 3y + 2z = 3 \\ 4x - y - 6z = -9 \end{array} \right.$$

- Regroup & write in matrix form:

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

MIT EECS 6.837, Cutler and Durand

38

$$x_i = \frac{\det A_i}{\det A},$$

where A_i 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

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

$$\left. \begin{array}{l} R_{ox} + tR_{dx} = a_x + \beta(b_x - a_x) + \gamma(c_x - a_x) \\ R_{oy} + tR_{dy} = a_y + \beta(b_y - a_y) + \gamma(c_y - a_y) \\ R_{oz} + tR_{dz} = a_z + \beta(b_z - a_z) + \gamma(c_z - a_z) \end{array} \right\} \begin{array}{l} \text{3D space } z \\ \text{3 equations,} \\ \text{3 unknowns} \end{array}$$

- Regroup & write in matrix form:

$$\begin{matrix} \text{edge1} & \text{edge2} \\ \begin{bmatrix} a_x - b_x & a_x - c_x & R_{dx} \\ a_y - b_y & a_y - c_y & R_{dy} \\ a_z - b_z & a_z - c_z & R_{dz} \end{bmatrix} & \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} \end{matrix} = \begin{bmatrix} a_x - R_{ox} \\ a_y - R_{oy} \\ a_z - R_{oz} \end{bmatrix}$$

MIT EECS 6.837, Cutler and Durand

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

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

38

MIT EECS 6.837, Cutler and Durand

| 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

39

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


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

Intersection with Barycentric Triangle

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

$$\left. \begin{array}{l} R_{ox} + tR_{dx} = a_x + \beta(b_x - a_x) + \gamma(c_x - a_x) \\ R_{oy} + tR_{dy} = a_y + \beta(b_y - a_y) + \gamma(c_y - a_y) \\ R_{oz} + tR_{dz} = a_z + \beta(b_z - a_z) + \gamma(c_z - a_z) \end{array} \right\} \begin{array}{l} \text{3D space } z \\ \text{3 equations,} \\ \text{3 unknowns} \end{array}$$

- Regroup & write in matrix form:

$$\begin{bmatrix} \text{edge1} & \text{edge2} \\ a_x - b_x & a_x - c_x & R_{dx} \\ a_y - b_y & a_y - c_y & R_{dy} \\ a_z - b_z & a_z - c_z & R_{dz} \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} a_x - R_{ox} \\ a_y - R_{oy} \\ a_z - R_{oz} \end{bmatrix}$$

Or

$$\begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} \text{edge1} & \text{edge2} \\ a_x - b_x & a_x - c_x & R_{dx} \\ a_y - b_y & a_y - c_y & R_{dy} \\ a_z - b_z & a_z - c_z & R_{dz} \end{bmatrix}^{-1} \begin{bmatrix} a_x - R_{ox} \\ a_y - R_{oy} \\ a_z - R_{oz} \end{bmatrix}$$

Cramer's Rule

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

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

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

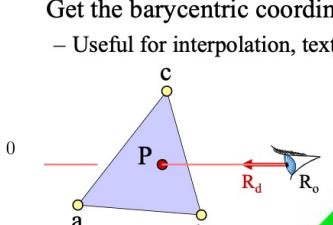
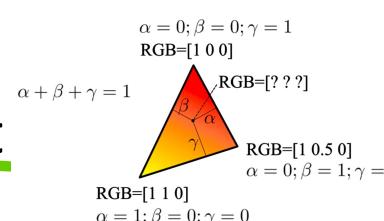
38

| denotes the determinant
matrix form
Can be copied mechanically into code

Advantages of Barycentric Intersection

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

Ray - Object Intersection Math problem of Solving t



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

Intersection with Barycentric Triangle

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

$$\left. \begin{array}{l} R_{ox} + tR_{dx} = a_x + \beta(b_x - a_x) + \gamma(c_x - a_x) \\ R_{oy} + tR_{dy} = a_y + \beta(b_y - a_y) + \gamma(c_y - a_y) \\ R_{oz} + tR_{dz} = a_z + \beta(b_z - a_z) + \gamma(c_z - a_z) \end{array} \right\} \begin{array}{l} \text{3D space } z \\ \text{3 equations,} \\ \text{3 unknowns} \end{array}$$

- Regroup & write in matrix form:

$$\begin{bmatrix} \text{edge1} & \text{edge2} \\ a_x - b_x & a_x - c_x & R_{dx} \\ a_y - b_y & a_y - c_y & R_{dy} \\ a_z - b_z & a_z - c_z & R_{dz} \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} a_x - R_{ox} \\ a_y - R_{oy} \\ a_z - R_{oz} \end{bmatrix}$$

Or

$$\begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} \text{edge1} & \text{edge2} \\ a_x - b_x & a_x - c_x & R_{dx} \\ a_y - b_y & a_y - c_y & R_{dy} \\ a_z - b_z & a_z - c_z & R_{dz} \end{bmatrix}^{-1} \begin{bmatrix} a_x - R_{ox} \\ a_y - R_{oy} \\ a_z - R_{oz} \end{bmatrix}$$

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

Cramer's Rule

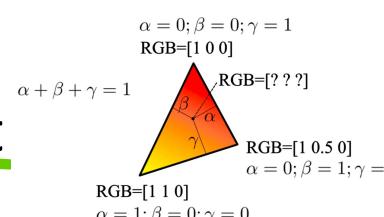
- Used to solve for one variable at a time in system of equations

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

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

38

MIT EECS 6.837, Cutler and Durand



| denotes the determinant
matrix form
Can be copied mechanically into code

Advantages of Barycentric Intersection

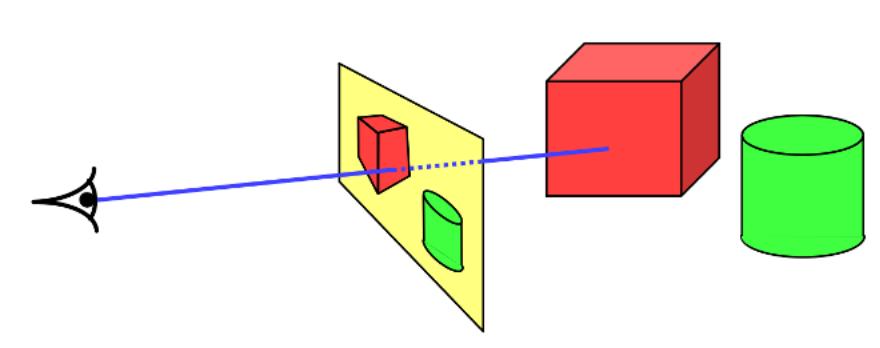
- Efficient
- Stores no plane equation
- Get the barycentric coordinates for free
 - Useful for interpolation, texture mapping

Ray - Object Intersection Math problem of Solving t

Coding: Step by step explanation

Möller–Trumbore intersection algorithm

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



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



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

Intersection with Barycentric Triangle

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

$$\left. \begin{array}{l} R_{ox} + tR_{dx} = a_x + \beta(b_x - a_x) + \gamma(c_x - a_x) \\ R_{oy} + tR_{dy} = a_y + \beta(b_y - a_y) + \gamma(c_y - a_y) \\ R_{oz} + tR_{dz} = a_z + \beta(b_z - a_z) + \gamma(c_z - a_z) \end{array} \right\} \begin{array}{l} \text{3D space} \\ \text{3 equations,} \\ \text{3 unknowns} \end{array}$$

- Regroup & write in matrix form:

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

MIT EECS 6.837, Cutler and Durand

38

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

MIT EECS 6.837, Cutler and Durand

| denotes the determinant
matrix form
Can be copied mechanically into code

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



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

Intersection with Barycentric Triangle

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

$$\left. \begin{array}{l} R_{ox} + tR_{dx} = a_x + \beta(b_x - a_x) + \gamma(c_x - a_x) \\ R_{oy} + tR_{dy} = a_y + \beta(b_y - a_y) + \gamma(c_y - a_y) \\ R_{oz} + tR_{dz} = a_z + \beta(b_z - a_z) + \gamma(c_z - a_z) \end{array} \right\} \begin{array}{l} \text{3D space} \\ \text{3 equations,} \\ \text{3 unknowns} \end{array}$$

- Regroup & write in matrix form:

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

Scalar Triple Product Formula



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



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

IF the object is intersected by the ray

How to solve the determinant $|A|$?

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



```
if (a > -0.0001f && a < 0.0001f) denominator !=0  
floating-point precision
```

return false; // ray parallel to triangle

```
const float f = 1 / a;
```



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

```
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 );
```

$$t = \frac{\text{edge1} \quad \text{edge2} \quad s = \text{ray.0} - \text{tri.vertex0}}{\left| \begin{array}{ccc} a_x - b_x & a_x - c_x & a_x - R_{ox} \\ a_y - b_y & a_y - c_y & a_y - R_{oy} \\ a_z - b_z & a_z - c_z & a_z - R_{oz} \end{array} \right|}$$

MIT EECS 6.837, Cutler and Durand

$|$ denotes the determinant
matrix form
Can be copied mechanically into code

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



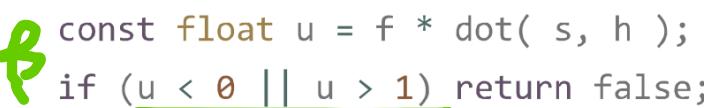
if ($a > -0.0001f \&\& a < 0.0001f$) denominator !=0
floating-point precision

 return false; // ray parallel to triangle

const float f = 1 / a; 

const float3 s = ray.0 - 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 $\beta + \gamma < 1$ & $\beta > 0$ & $\gamma > 0$

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

$s = \text{ray.0} - \text{tri.vertex0}$ 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 $s = \text{ray.0} - \text{tri.vertex0}$

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

edge1 edge2 $s = \text{ray.0} - \text{tri.vertex0}$

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

| denotes the determinant
matrix form
Can be copied mechanically into code

MIT EECS 6.837, Cutler and Durand

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



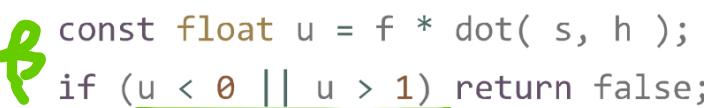
if ($a > -0.0001f \&\& a < 0.0001f$) denominator !=0
floating-point precision

 return false; // ray parallel to triangle

const float f = 1 / a; 

const float3 s = ray.0 - 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 $\beta + \gamma < 1$ & $\beta > 0$ & $\gamma > 0$

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

$s = \text{ray.0} - \text{tri.vertex0}$ 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 $s = \text{ray.0} - \text{tri.vertex0}$

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

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

| denotes the determinant
matrix form
Can be copied mechanically into code

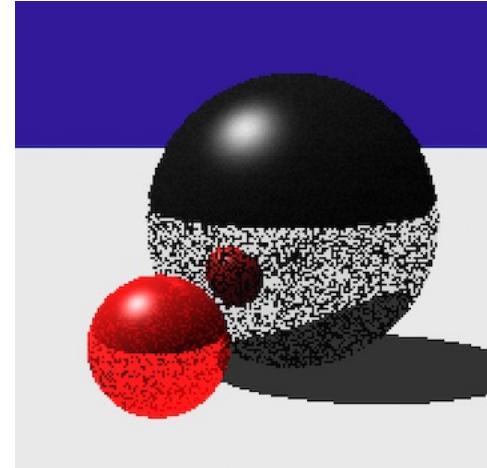
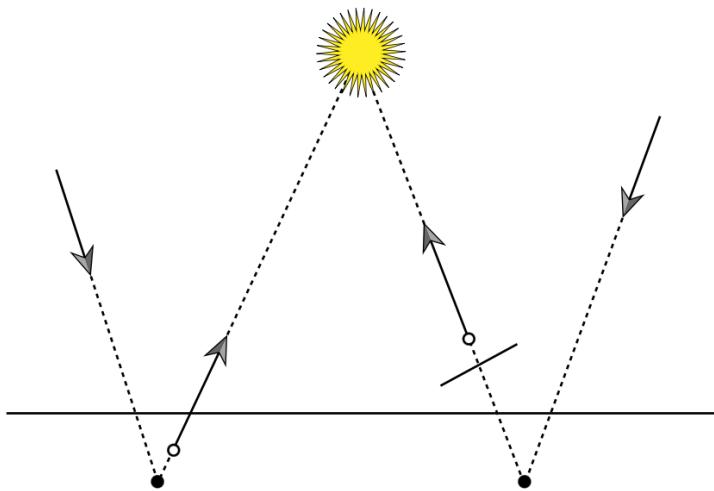
MIT EECS 6.837, Cutler and Durand

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);
 ϵ



The evil ϵ

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

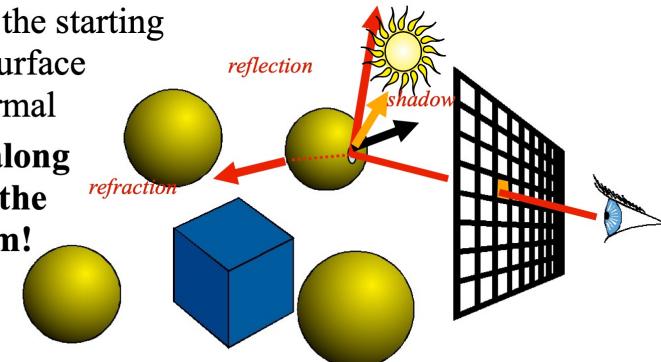


Figure 3.39: Geometric Settings for Rounding-Error Issues That Can Cause Visible Errors in Images. The incident ray on the left intersects the surface. On the left, the computed intersection point (black circle) is slightly below the surface and a too-low “epsilon” offsetting the origin of the shadow 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

Ray-triangle intersection—a different way

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

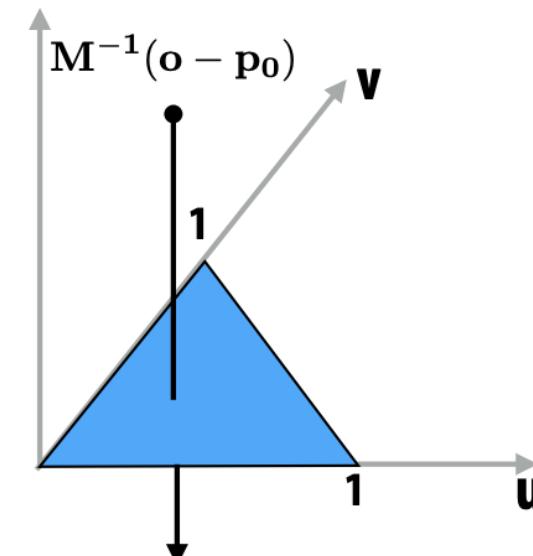
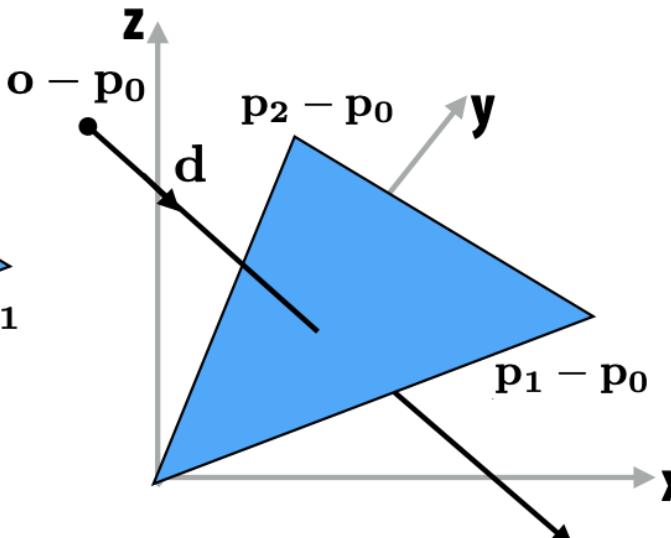
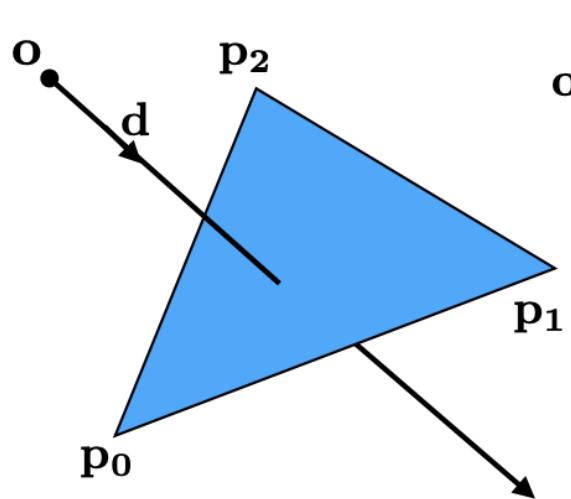
$$\mathbf{p}_0 + u(\mathbf{p}_1 - \mathbf{p}_0) + v(\mathbf{p}_2 - \mathbf{p}_0) = \mathbf{o} + t\mathbf{d}$$

Solve for u, v, t :

$$\begin{bmatrix} \mathbf{p}_1 - \mathbf{p}_0 & \mathbf{p}_2 - \mathbf{p}_0 & -\mathbf{d} \end{bmatrix} \begin{bmatrix} u \\ v \\ t \end{bmatrix} = \mathbf{o} - \mathbf{p}_0$$

$\overbrace{\qquad\qquad\qquad}^{\mathbf{M}}$

\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

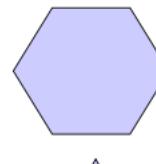
- Other primitives

- Boxes



- Polygons

- Ray-Polygon Intersection



- Triangles



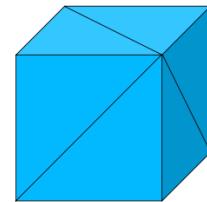
- IFS?

- Ray-Bunny Intersection
& extra topics...



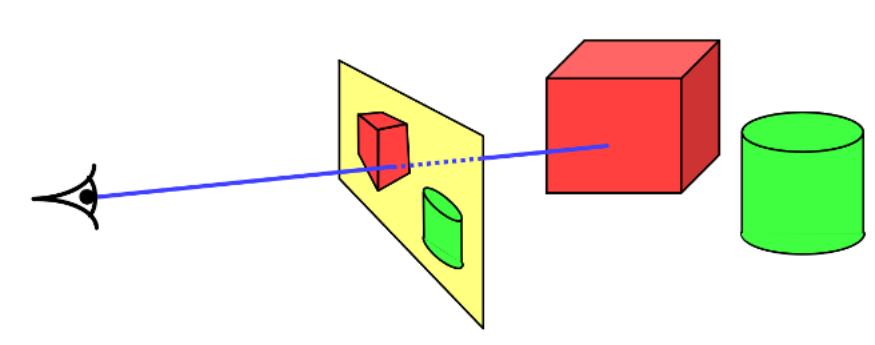
Triangle Meshes (.obj)

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
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





Q&A



RTX
ON



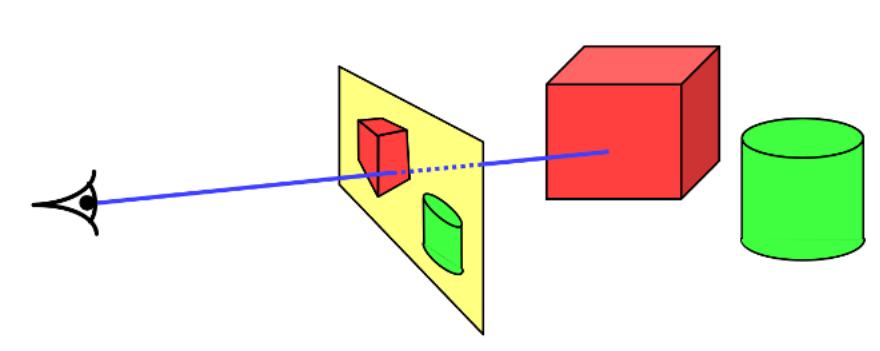
Peter Hu
1 Nov 2023

*Speeding up
real-time
Ray Tracing*

RTX
OFF

Part 3: Improvements of RT

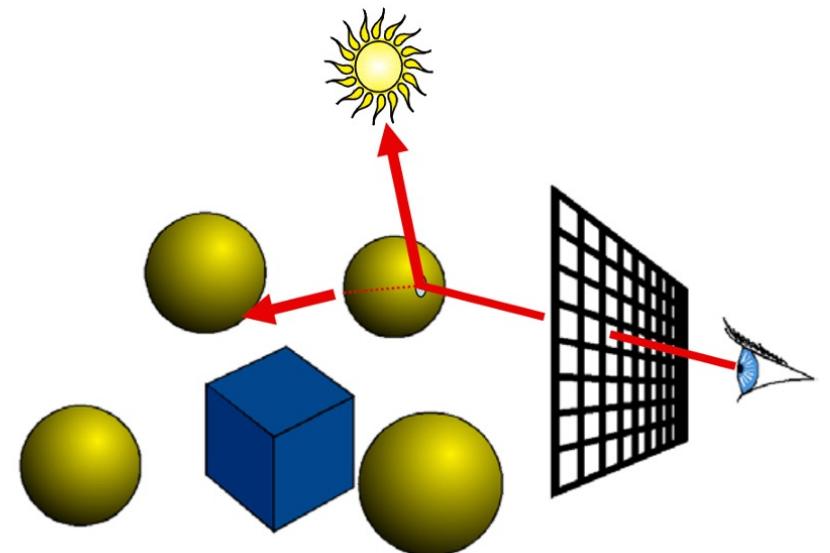
Acceleration Data Structure



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

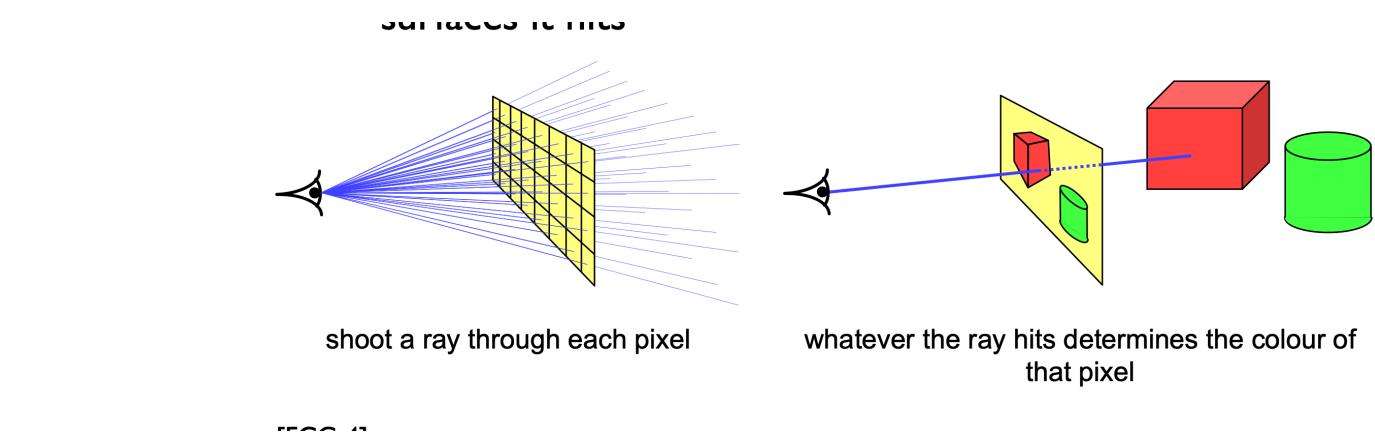
- Ray casting
- Lots of primitives
- Recursive
- Distributed Ray Tracing Effects
 - Soft shadows
 - Anti aliasing
 - Glossy reflection
 - Motion blur
 - Depth of field

$\text{cost} \approx \text{height} * \text{width} *$
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 Cost is tremendous!

- width, height = $1024 * 768$!!!
- num primitives $279k$ (Sponza) !!!
- each ray-geometry intersection cost
- size of recursive depth
- num shadow rays
- num super samples (anti-aliasing)
- num glossy rays (Glossy reflection)
- num temporal samples / focal samples. (Motion Blur)



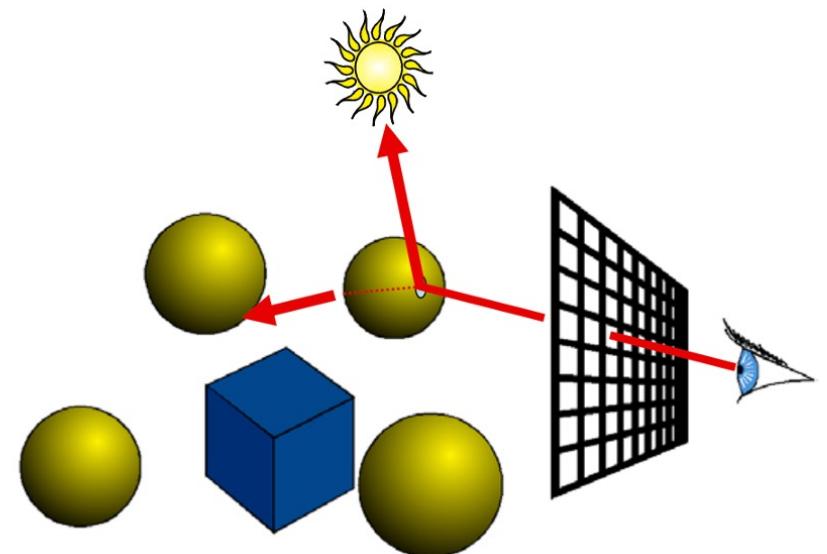
Processor Speed: 100M/s
1000s for one frame
0.001fps



Sponza scene

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

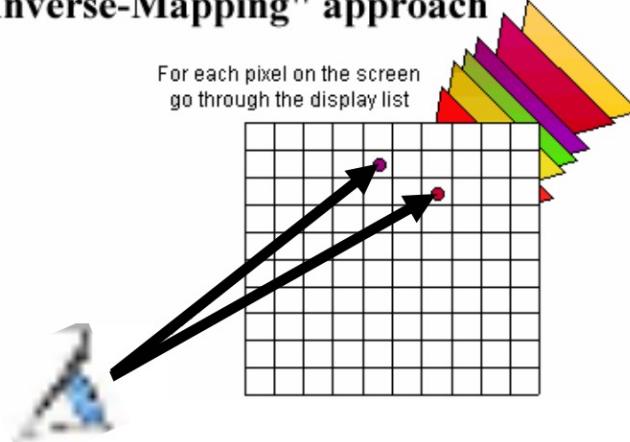
- Whole scene must be in memory

Send pixels to the scene

Discretize first

"Inverse-Mapping" approach

For each pixel on the screen
go through the display list



Rendering Pipeline

For each triangle

For each pixel

- Primitives processed **one at a time**

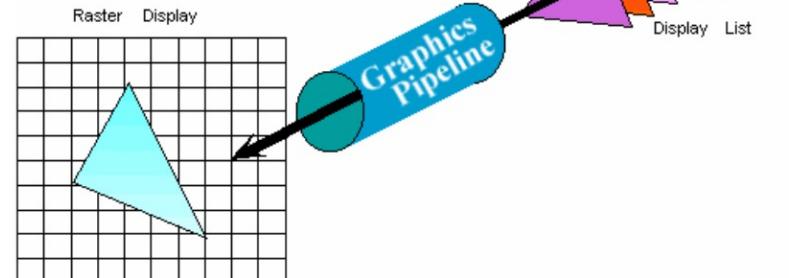
Project scene to the pixels

Discretize last

MVP transformation

Object -> World -> Screen coordinate

"Forward-Mapping" approach to Computer Graphics



RAY TRACING Cost is tremendous!

- width, height = 1024 * 768
- num primitives 279k (Sponza)

Idea1: Rasterization First + Ray Tracing for secondary ray

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

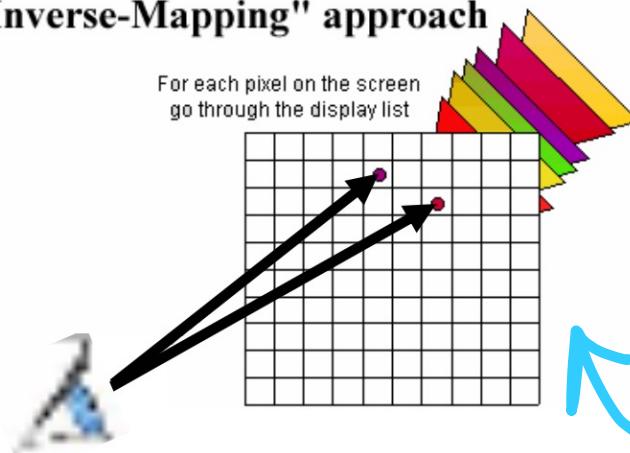
- Primitives processed one at a time

MVP transformation Projection

Object -> World -> Screen coordinate

"Inverse-Mapping" approach

For each pixel on the screen
go through the display list



"Forward-Mapping" approach to Computer Graphics

Raster Display



Acceleration Data Structures *for* *Ray Tracing*

Space subdivision

- Grids
- Bounding Volume

Triangle lists subdivision

- Bounding Volume Hierarchy

Rays spatial subdivision

- Ray space Hierarchy

Motivation

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

"Find the first primitive the ray hits"

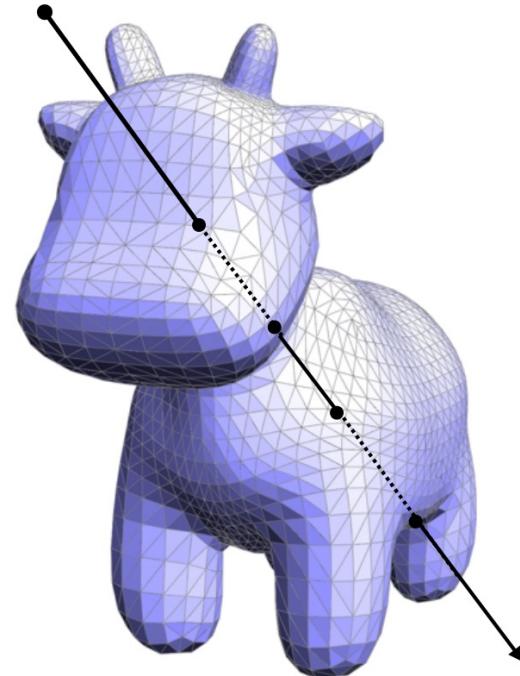
Naïve algorithm?

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

Complexity? $O(N)$

Can we do better?

Solution



CMU 15-46

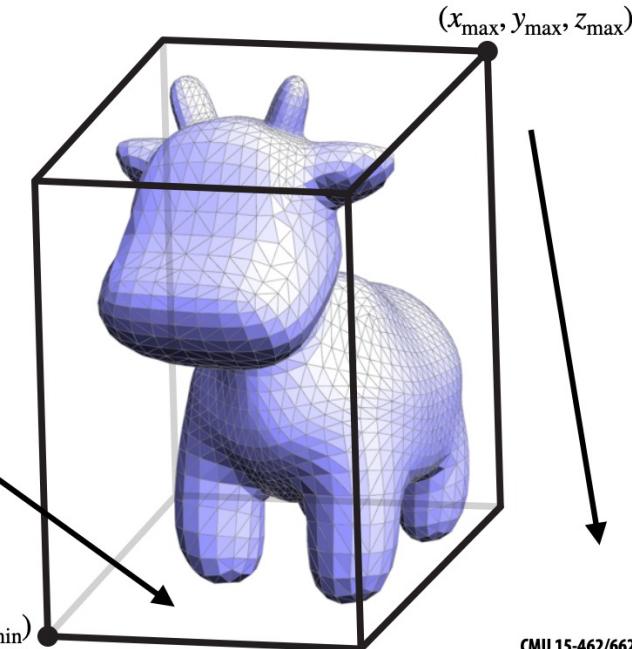
Bounding Box

- 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?)



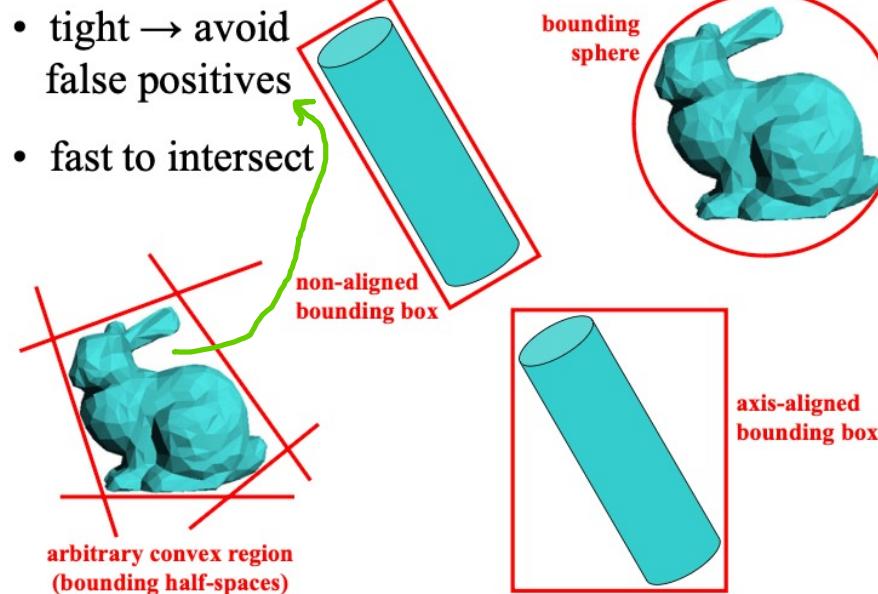
CMU 15-462/662

Ray-axis-aligned-box intersection

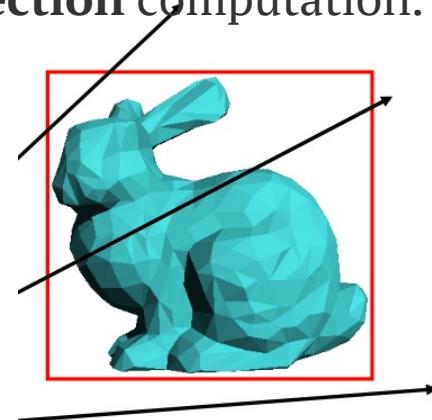
Bounding Volume

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

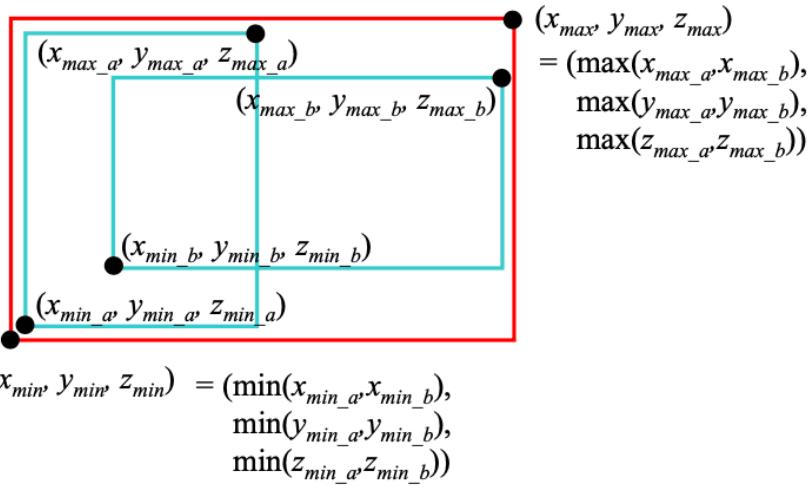


Axis Aligned Bounding Box (AABB) is commonly used due to its **efficiency** of the ray-AABB **intersection** computation.



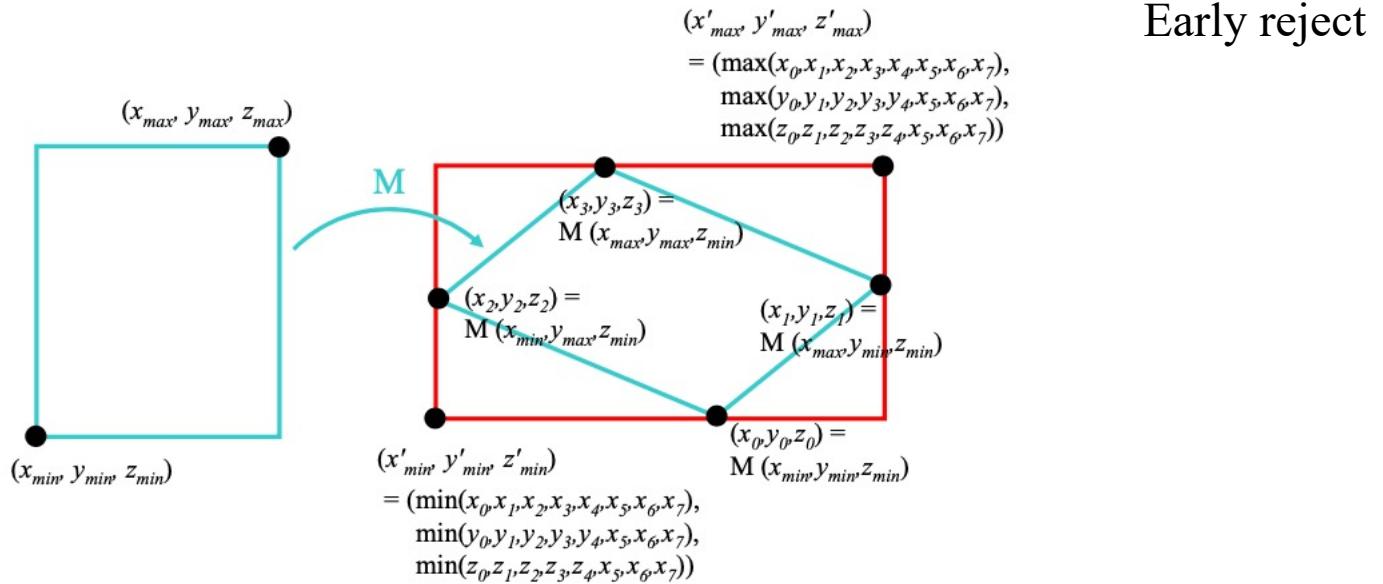
Others: MIT EECS 6.837 Computer Graphics (solid Math intro)

<https://groups.csail.mit.edu/graphics/classes/6.837/F04/calendar.html>

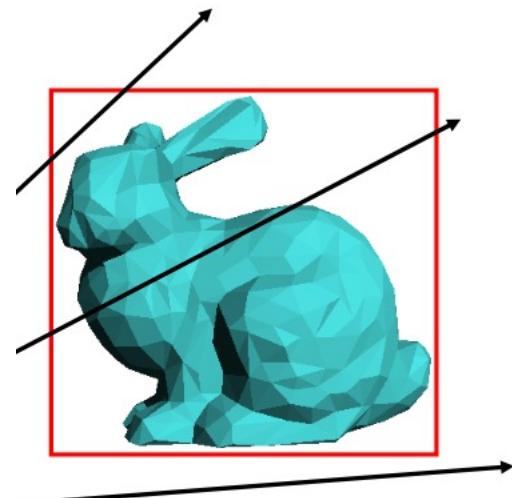


Bounding Boxes

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

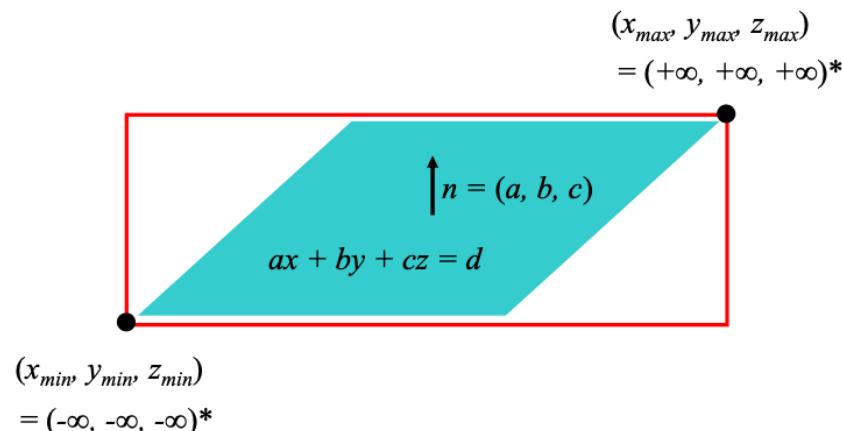


Early reject

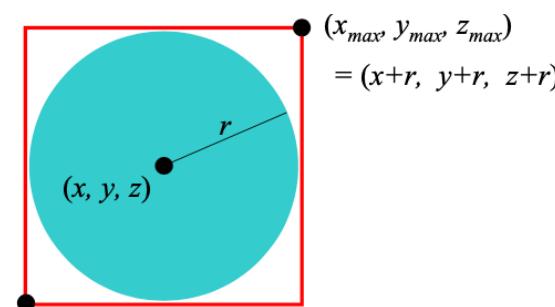


Object Representation

- **Plane**



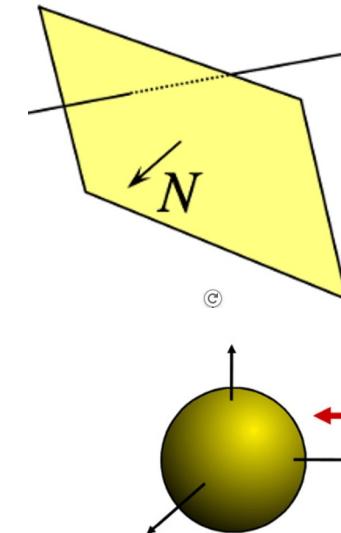
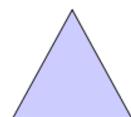
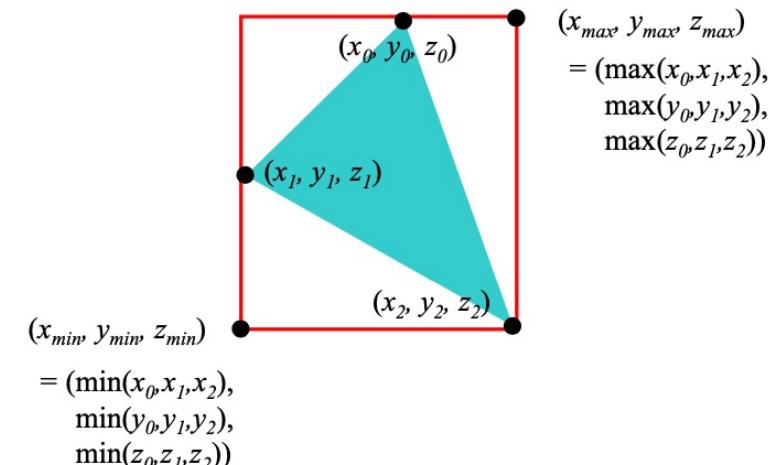
- **Sphere**

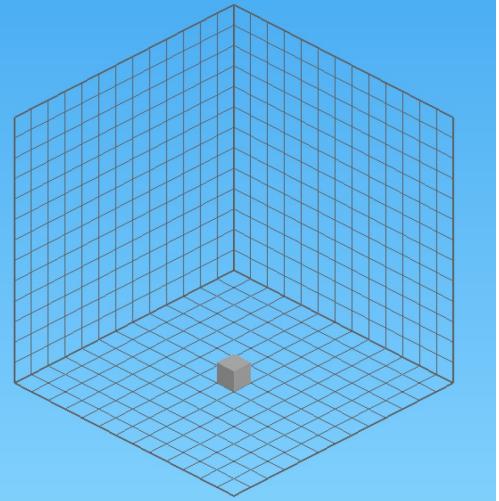


- ~~Cube / AABBs~~



- **Triangle (Frequently USED)**





Normal Grids

space subdivision

Ray Marching

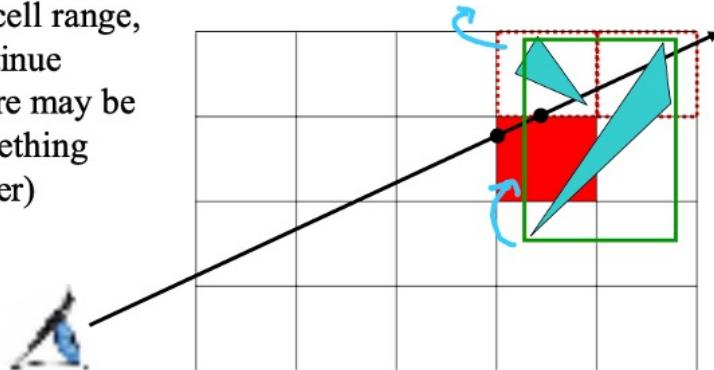
only intersect cells along the ray

Reduce # intersection!
from 279k primitives (Sponza)

resolution: $nx * ny * nz$

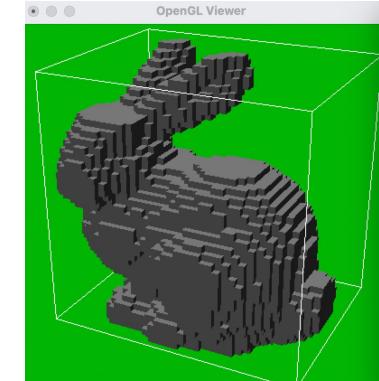
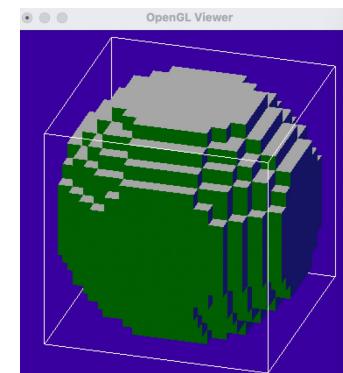
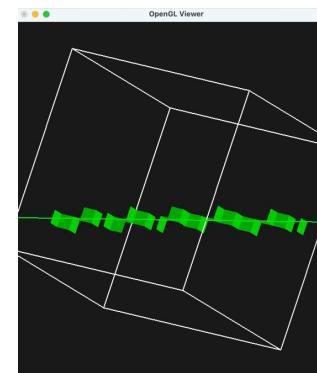
Using pointers to store the objects in each cell

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



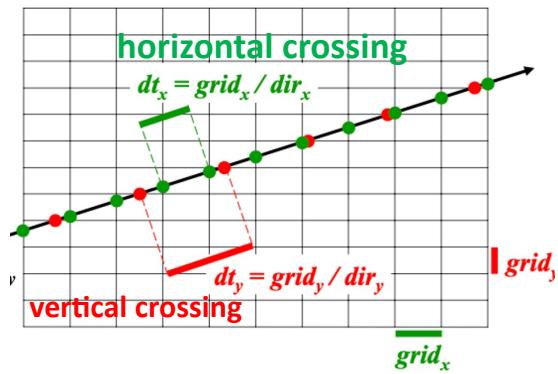
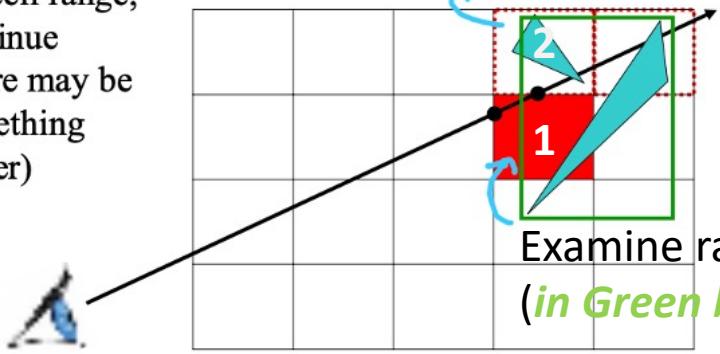
*Ray Marching
Visualization*

Minecraft style



Disadvantage: Sampling density
“Teapot in a Stadium” problem

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

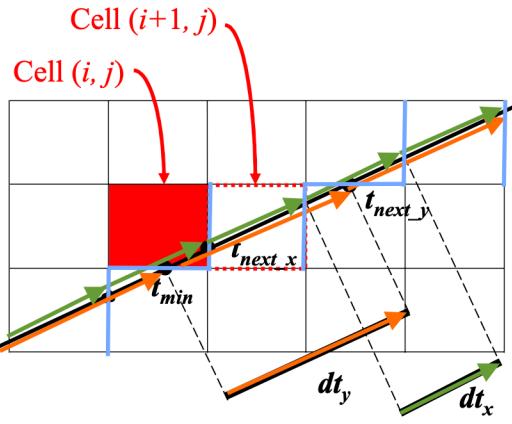


What's the Next Cell?

```

if ( $t_{next\_x} < t_{next\_y}$ )
     $i += sign_x$ 
     $t_{min} = t_{next\_x}$ 
     $t_{next\_x} += dt_x$ 
else
     $j += sign_y$ 
     $t_{min} = t_{next\_y}$ 
     $t_{next\_y} += dt_y$ 

```



$if (dir_x > 0) sign_x = 1 \text{ else } sign_x = -1$
 $if (dir_y > 0) sign_y = 1 \text{ else } sign_y = -1$

3D Digital differential analyser

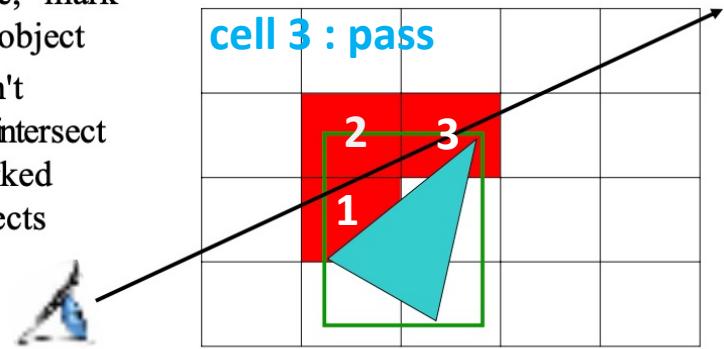
[https://en.wikipedia.org/wiki/Digital_differential_analyzer_\(graphics_algorithm\)](https://en.wikipedia.org/wiki/Digital_differential_analyzer_(graphics_algorithm))

similar: Bresenham's line algorithm

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

cache the result Preventing Repeated Computation

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



compare t_{next_x} and t_{next_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 Reduce # primitives!
            intersect r with p
            if intersection in range found
                return
        c = find next cell

```

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

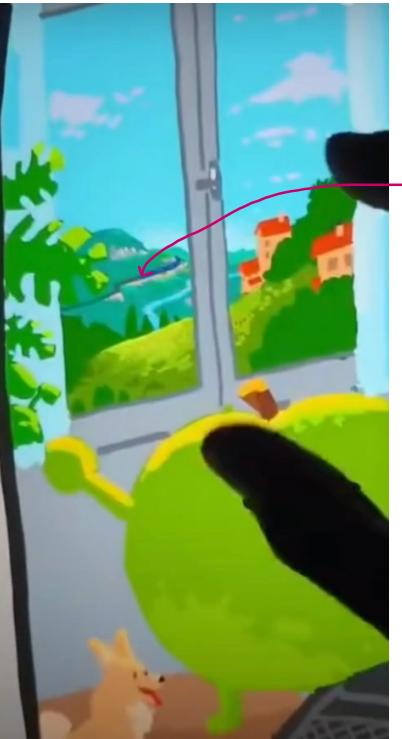
The difficulty to render small-scale details within a very large scene

Adaptive Grids

spatial subdivision



large scene



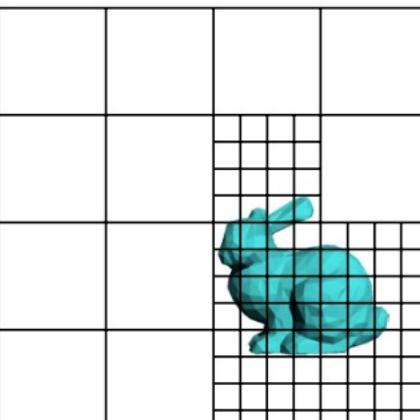
small-scale



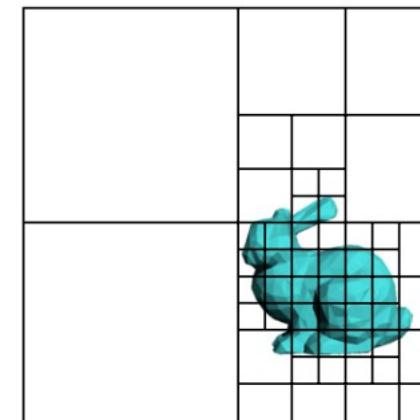
To solve “Teapot in a Stadium” problem Adaptive Grids

spatial subdivision

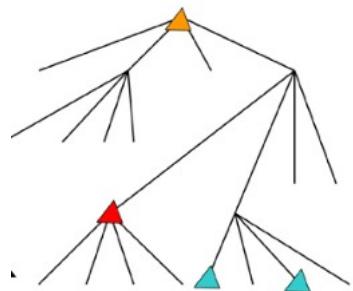
- Subdivide until each cell contains no more than n elements, or maximum depth d is reached



Nested Grids



Octree/(Quadtree)



Sampling density matches
Geometric density
triangle meshes

Disadvantage:
More expensive to traverse
GPU lack of local stack

space-partitioning

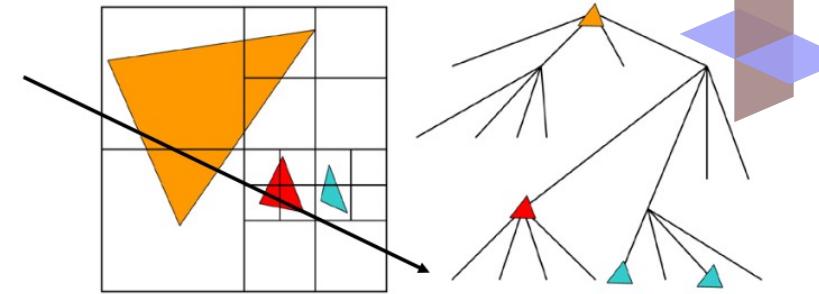
The ray traversal of k-dimensional tree implemented efficiently: a local **stack**.
Unfortunately, graphics hw doesn't have such on each element (vertex or pixel).
Solution: stack-less kD-tree traversal methods kD-restart and kD-backtrack.
by Foley et al. proposed.

- Advantages?

- grid complexity matches geometric density

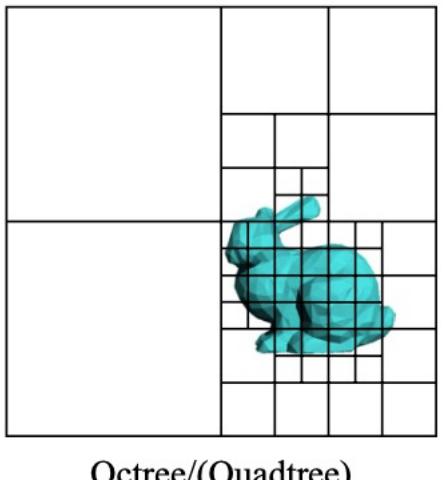
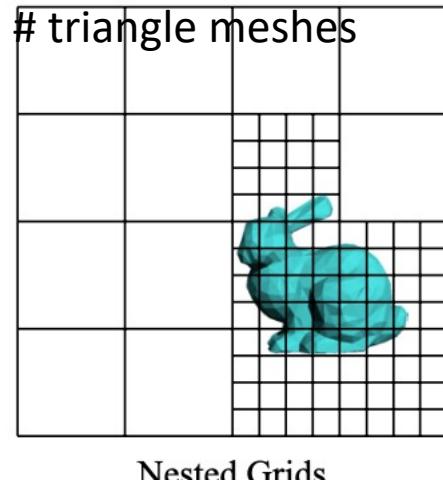
- Disadvantages?

- more expensive to traverse (especially octree)



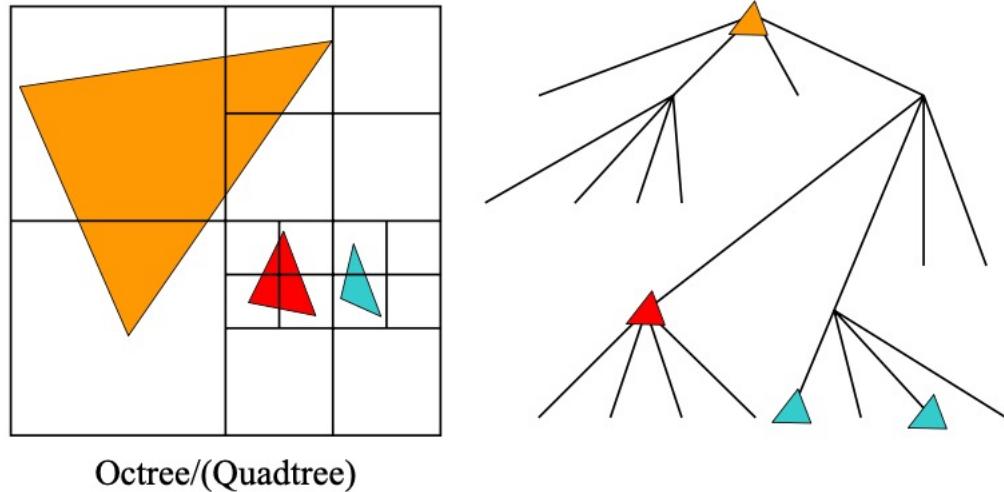
Adaptive Grids

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



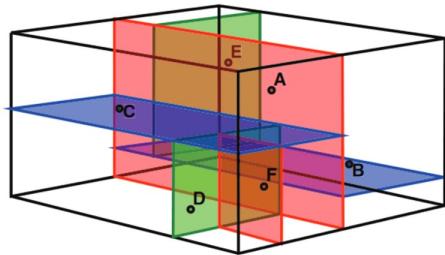
Primitives in an Adaptive Grid

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



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.

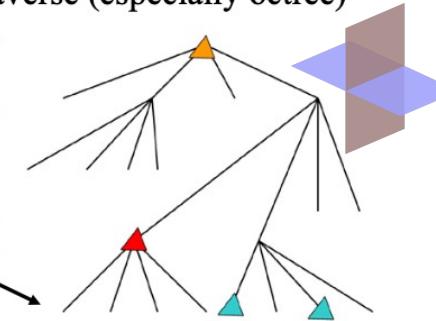
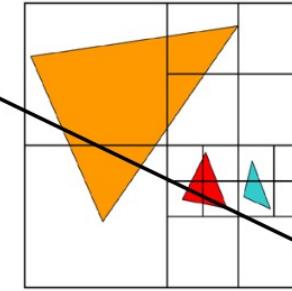


- Advantages?

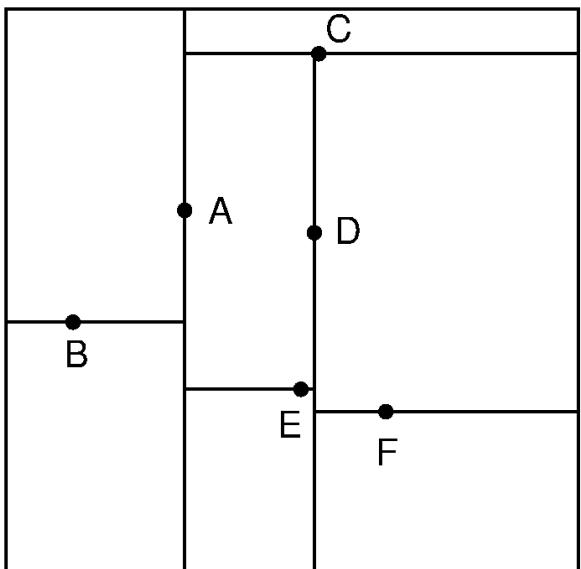
- grid complexity matches geometric density

- Disadvantages?

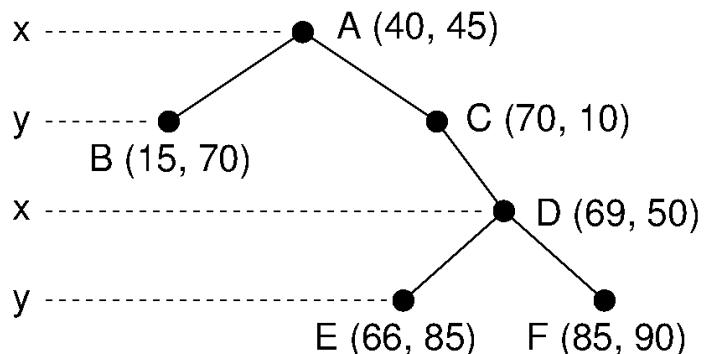
- more expensive to traverse (especially octree)



Adaptive Grids



(a)



(b)

linear-time radix sort

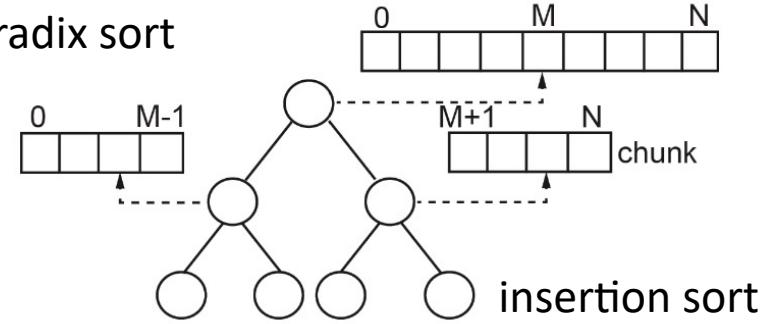
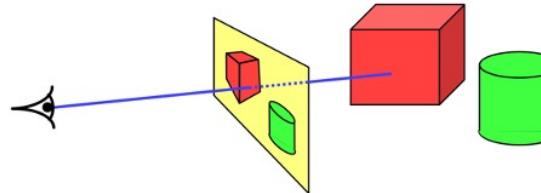


Fig. 2. Level 1 handles the entire array, and splits on the median between 0 and N . Nodes in level 2 each handle corresponding sub-arrays ("chunks") from their parent. This pattern continues until each node only handles a single element.

RAY TRACING Cost is tremendous!

- width, height = 1024 * 768
- num primitives 279k (Sponza)

Idea: Get Rid of triangles, use a bounding cube instead



whatever the ray hits determines the colour of that pixel

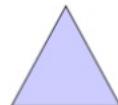
Ray-Object Intersection

IF the object is intersected by the ray

```
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;  
}
```

fixed-function
triangle
intersection

```
>>>  
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;  
}
```



Takeaway:
Ray-Triangle much more expensive than AABB



> much cheaper

[Skip details](#)

IF the object is intersected by the ray

```
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;  
}
```

fixed-function
triangle
intersection

>>>

```
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;  
}
```

Takeaway:

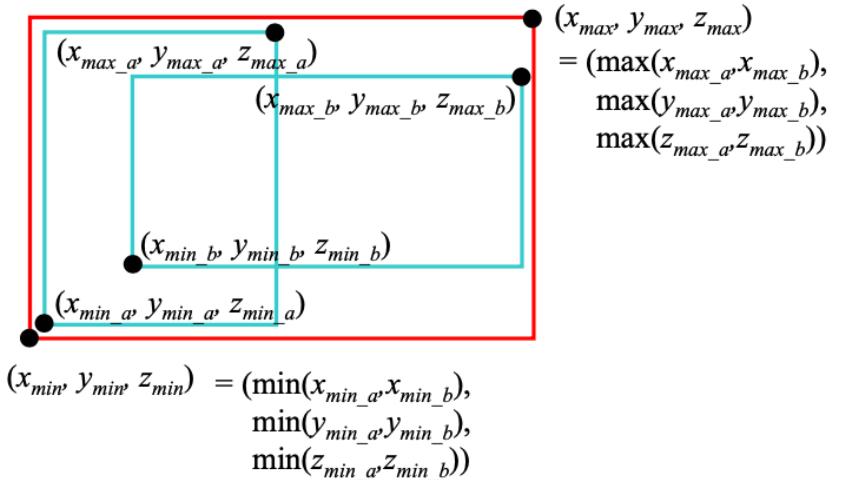
Ray-Triangle **much** more expensive than AABB

Idea3: Get Rid of triangles, use a bounding cube instead

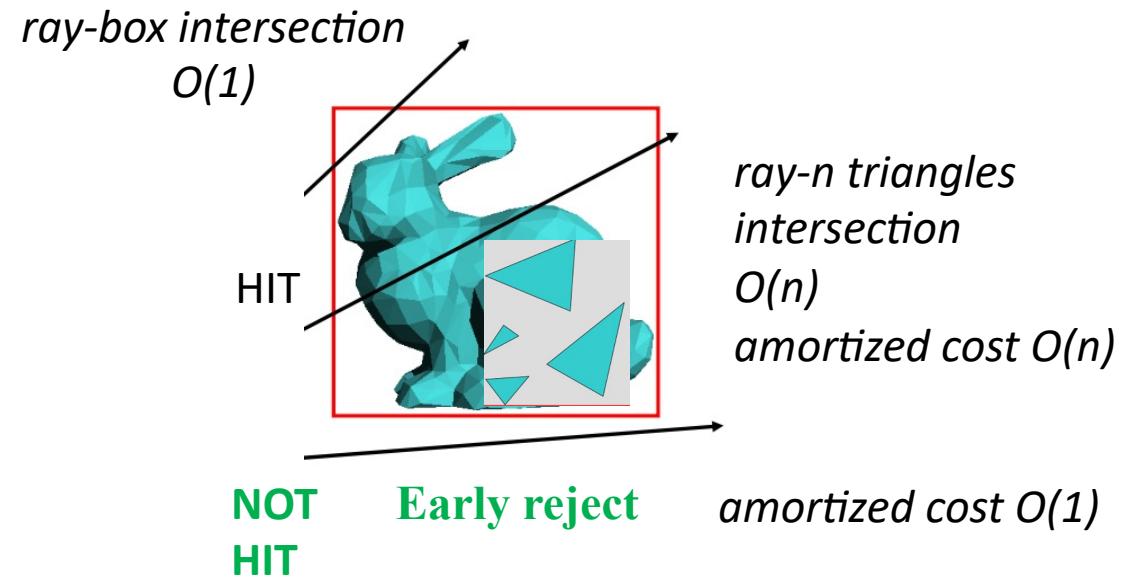
Bounding Volume

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

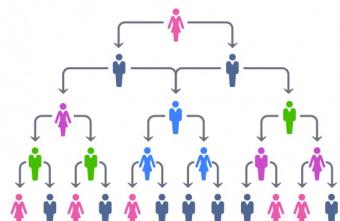
Pre-processing $O(n)$



Traversal: First check for the intersection with a conservative bounding volume (like AABB)



HIERARCHICAL ORGANIZATIONS



Q: How can we do better?

Grids

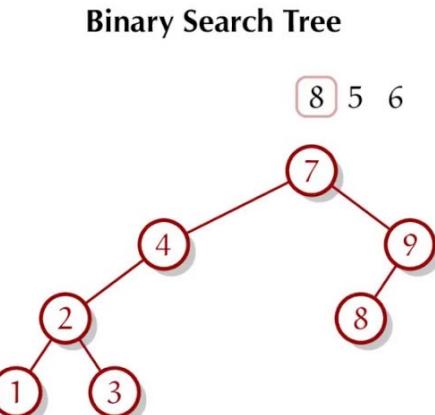
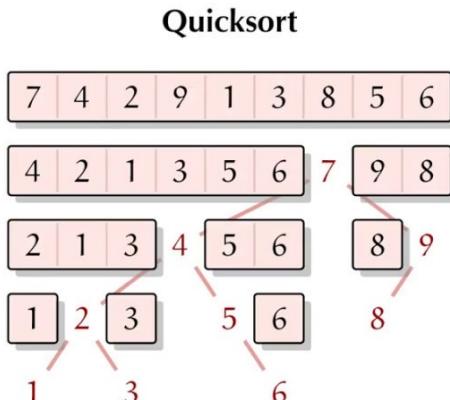
*subdivide **space** into smaller sub-spaces*

Bounding Volume Hierarchy

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

find the k-th largest in a list of integers : early reject

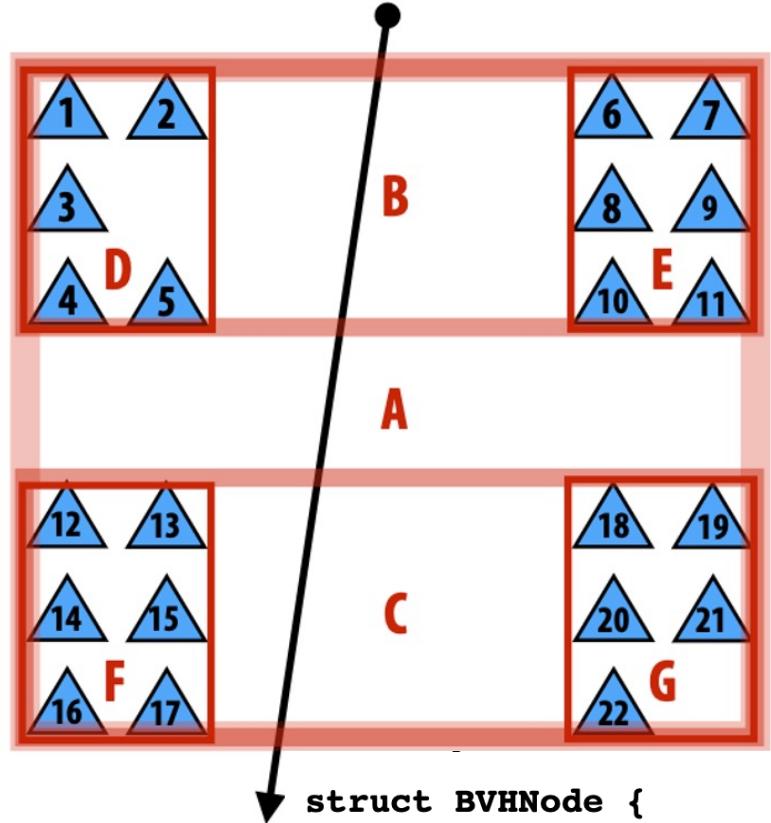
Meaning: building it recursively,
i.e. in a hierarchy way



A tree structure on a set of geometric objects

Bounding Volume Hierarchy

Construction: split to partition and build up a tree structure

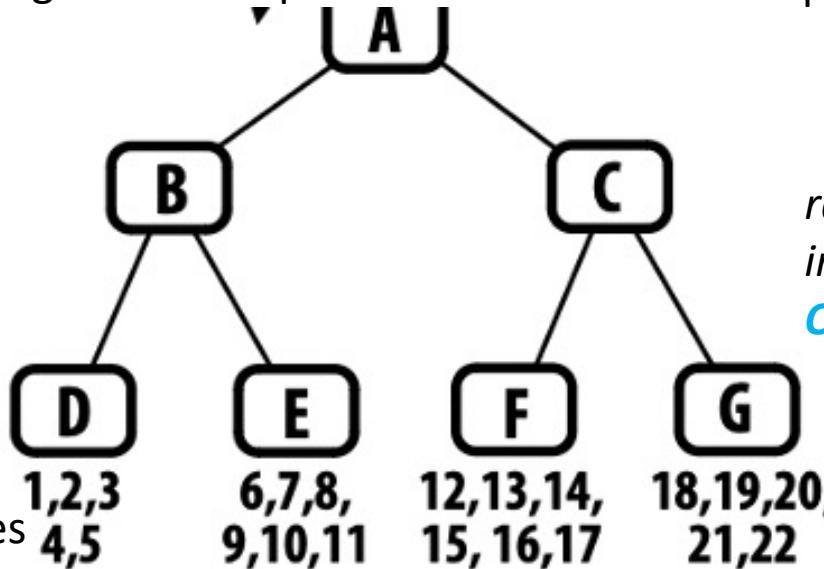


```
struct BVHNode {  
    bool leaf; // am I a leaf node?  
    BBox bbox; // min/max coords of enclosed primitives  
    BVHNode* child1; // "left" child (could be NULL)  
    BVHNode* child2; // "right" child (could be NULL)  
    Primitive* primList; // for leaves, stores primitives  
};
```

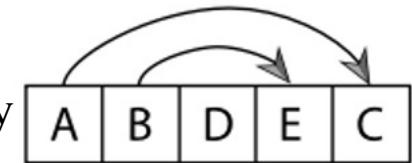
```
find_closest_hit(ray, node->child1, closest);  
find_closest_hit(ray, node->child2, closest);
```

Traversal: Check for the intersection with interior conservative bounding volumes iteratively

Interior nodes:
proxy for a large subset of primitives



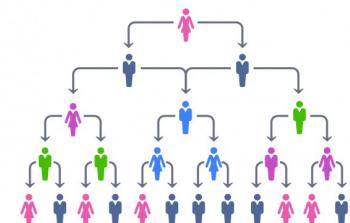
Leaf nodes:
small list of primitives



GPU memory layout
Depth First Search

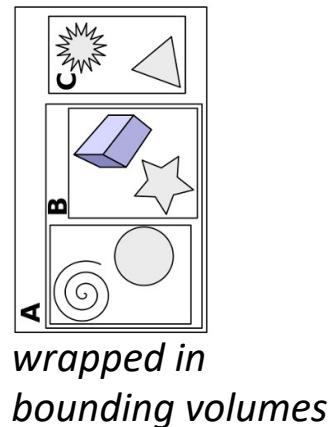
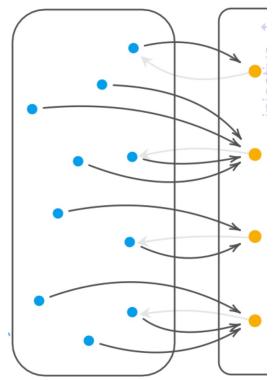
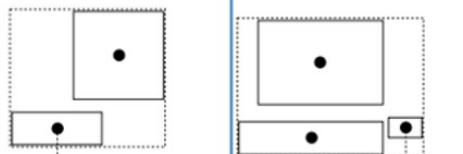
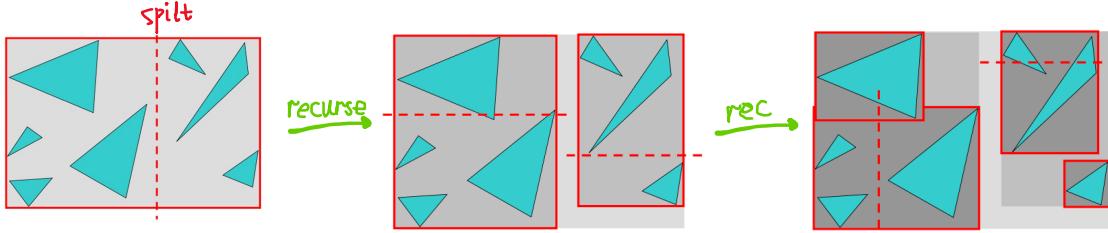
ray-n triangles
intersection
 $O(\log n)$

HIERARCHICAL ORGANIZATIONS

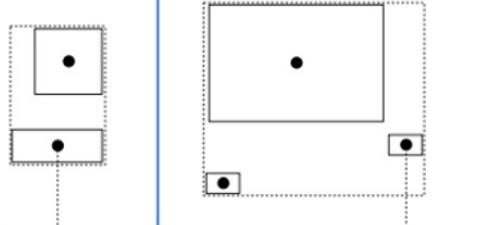


Bounding Volume Hierarchy

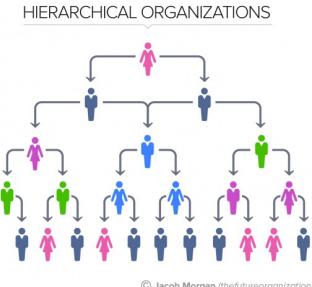
Construction: split to partition and build up a tree structure



Object split may have overlapping



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

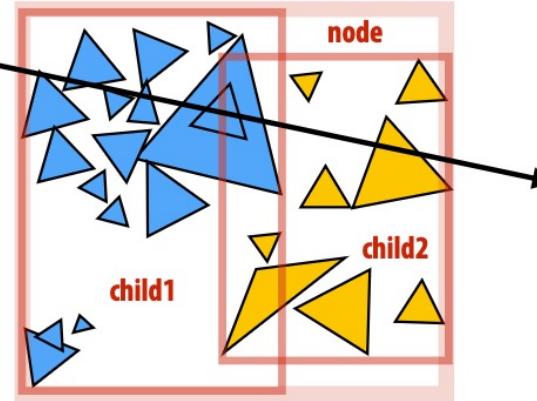


Ray-scene intersection using a BVH

```
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

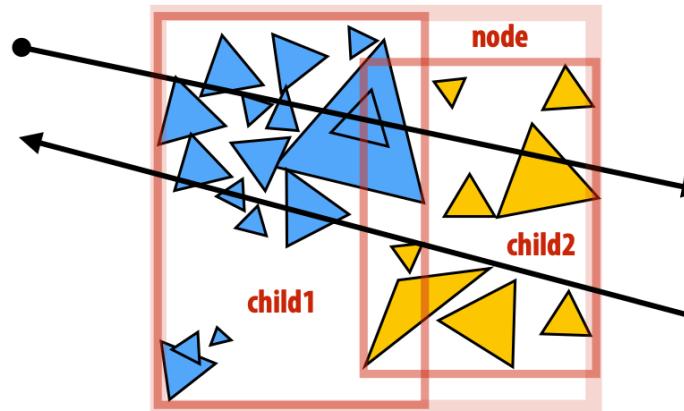
If NOT
Do the binary partition

Which child first?

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

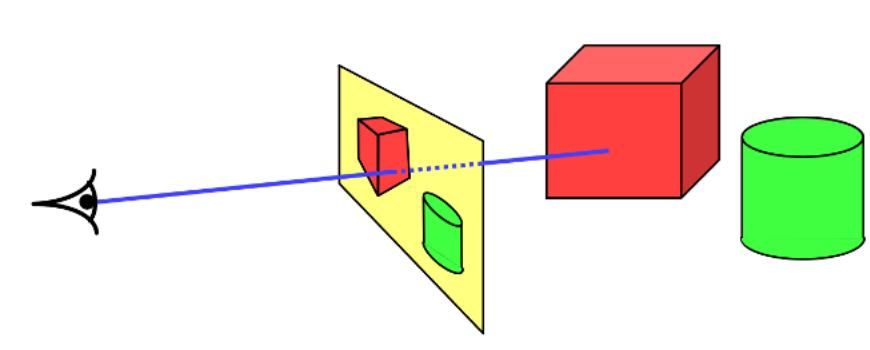
First Hit
Though not every time,
but is the usual case that
front will be the First Hit

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

Due to scene locality,
AI solution could have a go

Linear BVH (LBVH)

Sort the Morton Code for the center of the AABB



BVH Construction: Sort the Morton Code for the center of the triangle AABB

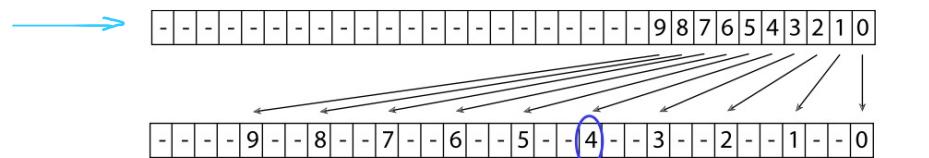
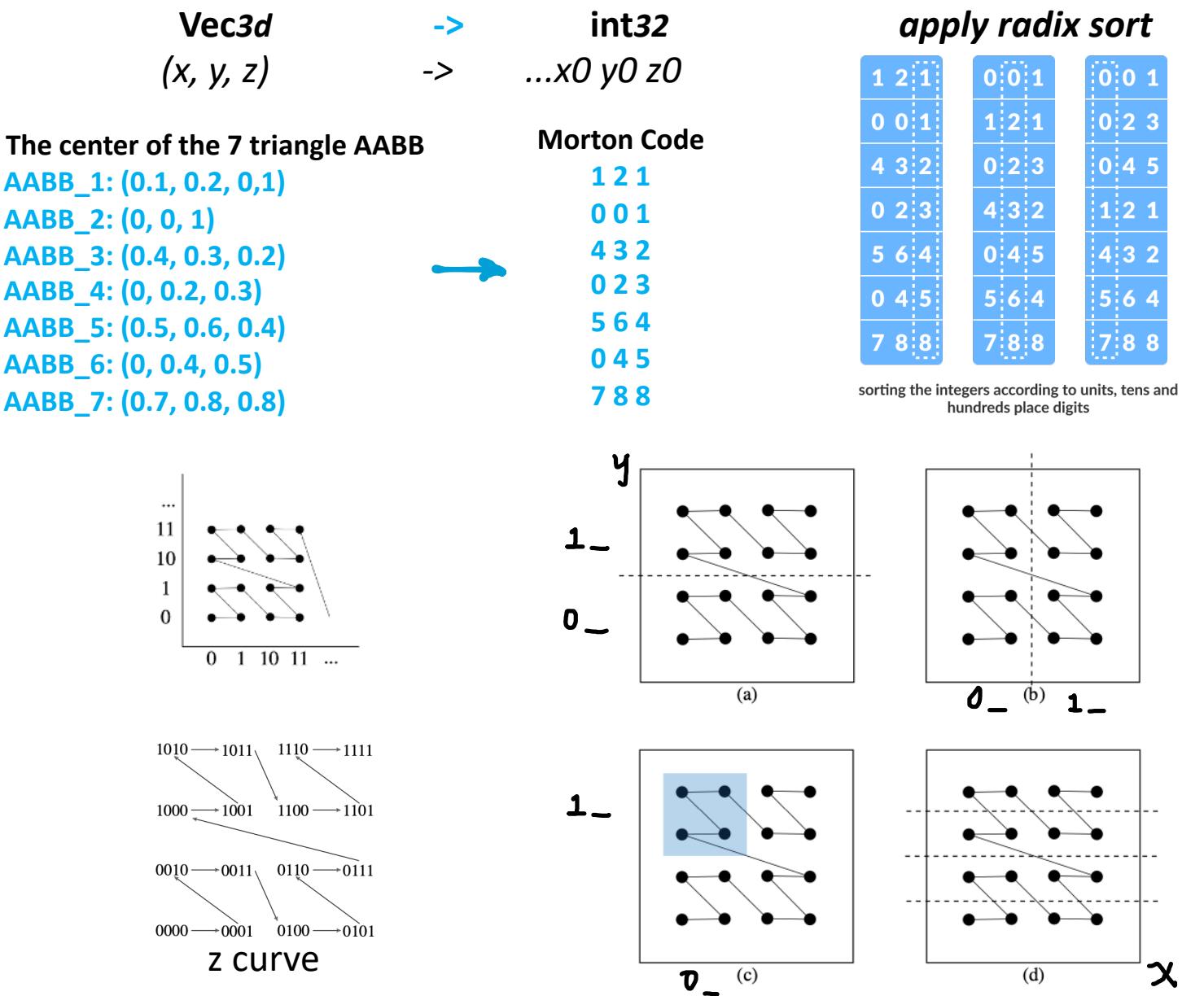


Figure 4.9: Bit Shifts to Compute 3D Morton Codes. The `LeftShift3()` function takes a 32-bit integer value and for the bottom 10 bits, shifts the i th bit to be in position $3i$ —in other words, shifts it $2i$ places to the left. All other bits are set to zero.

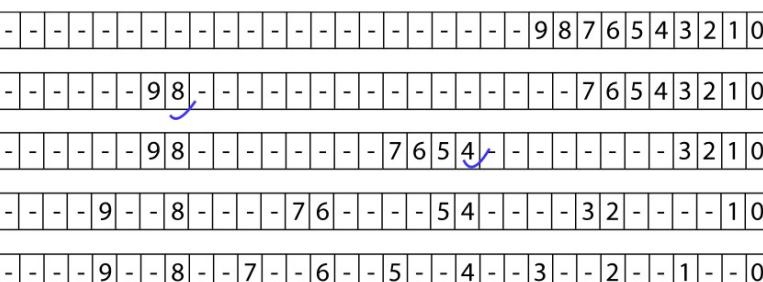


Figure 4.10: Power-of-Two Decomposition of Morton Bit Shifts. The bit shifts to compute the Morton code for each 3D coordinate are performed in a series of shifts of power-of-two size. First, bits 8 and 9 are shifted 16 places to the left. This places bit 8 in its final position. Next bits 4 through 7 are shifted 8 places. After shifts of 4 and 2 places (with appropriate masking so that each bit is shifted the right number of places in the end), all bits are in the proper position. This computation is implemented by the `LeftShift3()` function.

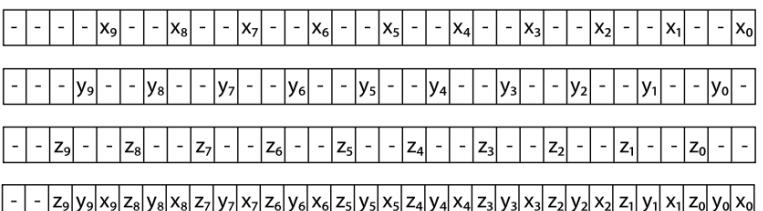
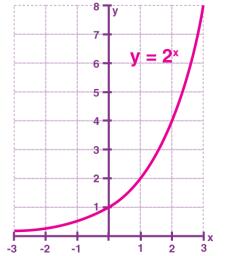


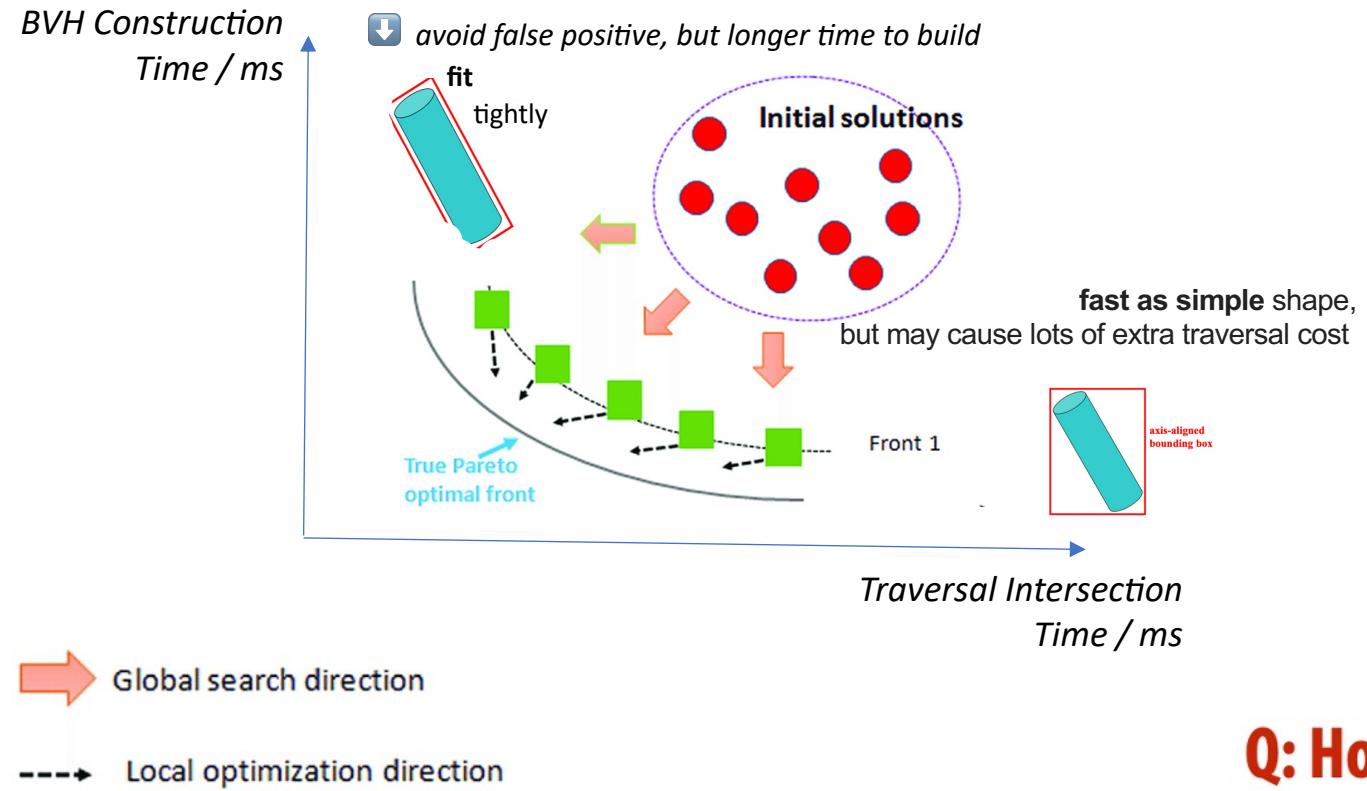
Figure 4.11: Final Interleaving of Coordinate Values. Given interleaved values for x , y , and z computed by `LeftShift3()`, the final Morton-encoded value is computed by shifting y and z one and two places, respectively, and then ORing together the results.

Bounding Volume Hierarchy Design Tradeoff

$2^{N/2}$ ways to partition N primitives into two groups
Exponential is horrible!



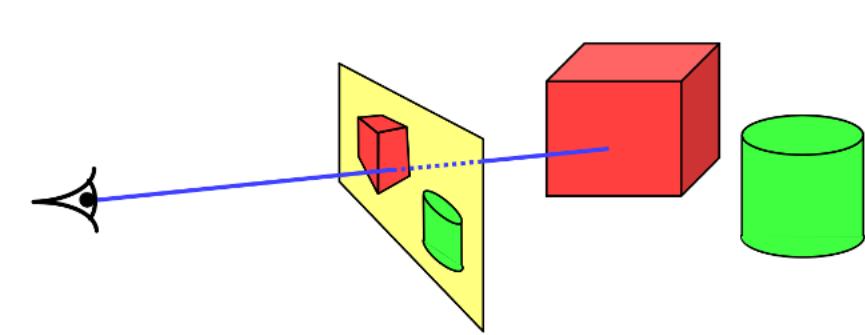
Multi-objective / Pareto optimization
more than one objective function



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

BVH based on Surface Area Heuristic (SAH)

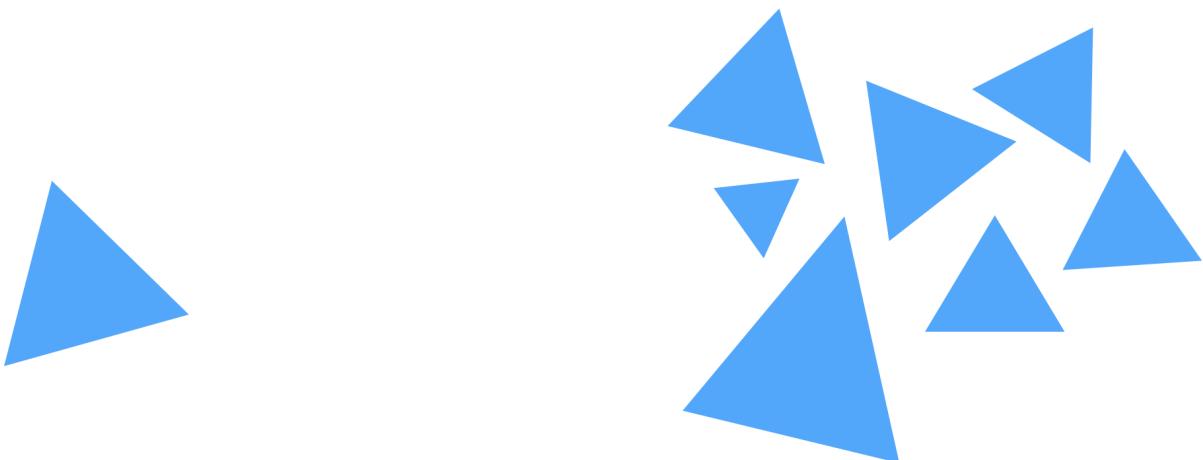
Good Partition Principle, SAH, other advanced ideas



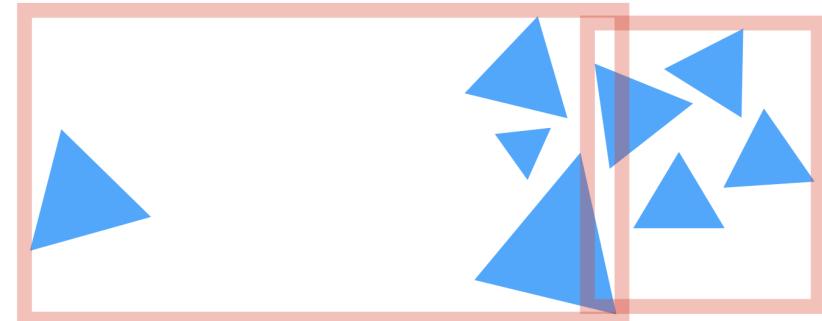
Heuristic

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.

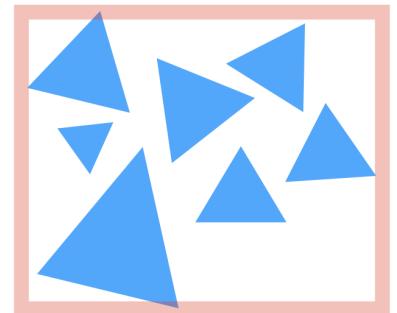
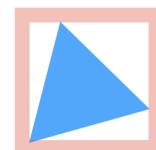
Good Partition Principle for exponential possibilities



False positive ✓



Partition into child nodes with equal numbers of primitives

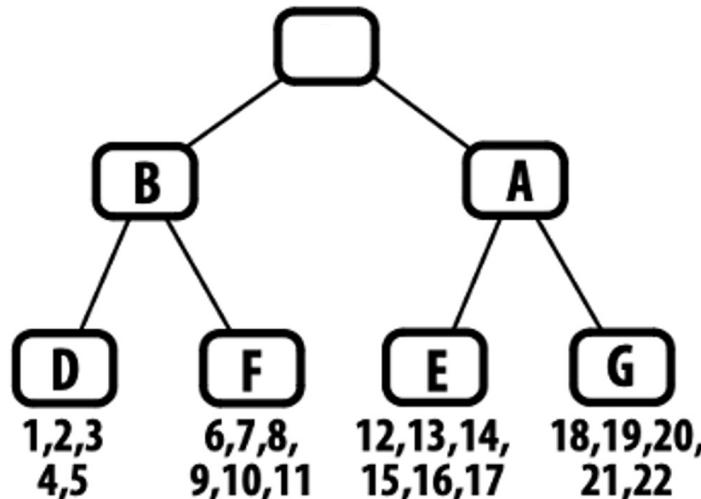


Better partition

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

Good Partition Principle

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.

Primitive count is common heuristic for child node costs:

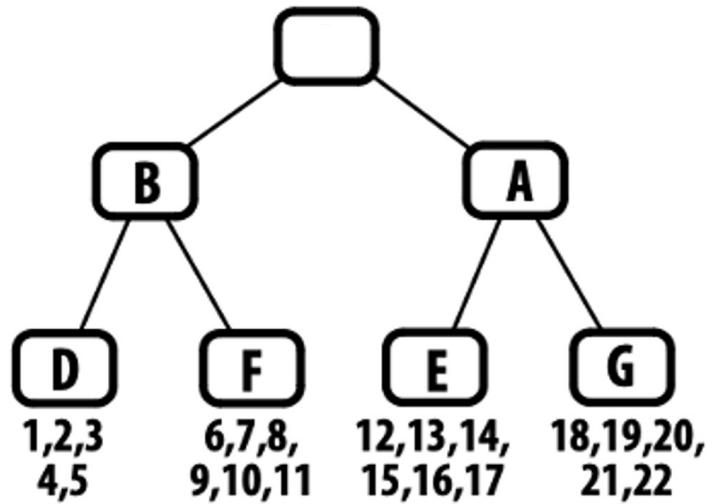
$$= NC_{\text{isect}}$$

(Common to assume all primitives have the same cost)

Good Partition Principle

for exponential possibilities

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



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

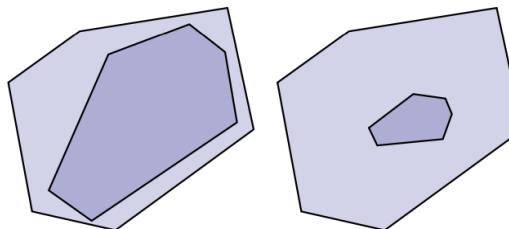
$$C = C_{\text{trav}} + \underline{p_A N_A C_{\text{isect}}} + \underline{p_B N_B C_{\text{isect}}}$$

Good Partition Principle for exponential possibilities

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)

Good Partition Principle for exponential possibilities

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...

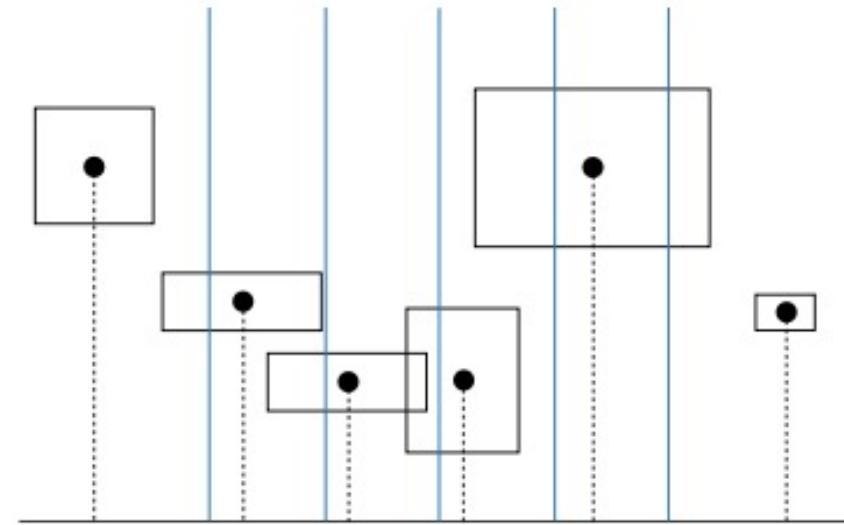
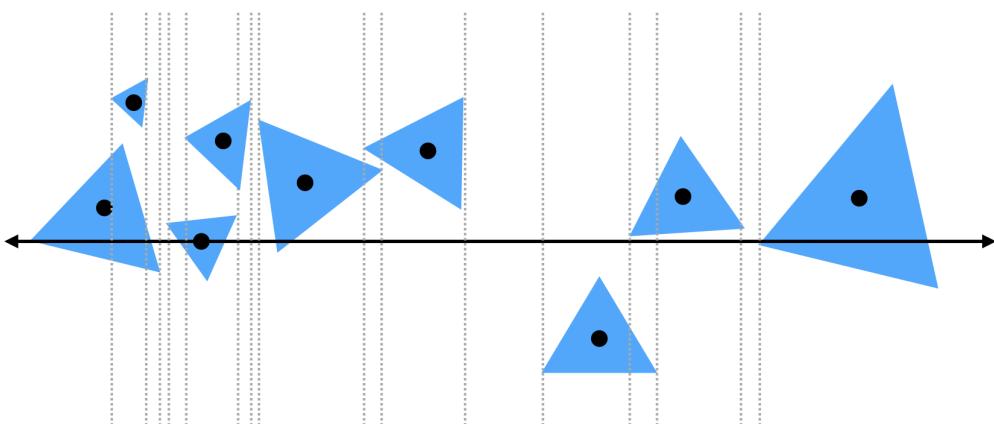


Figure 4.6: Choosing a Splitting Plane with the Surface Area Heuristic for BVHs. The projected extent of primitive bounds centroids is projected onto the chosen split axis. Each primitive is placed in a bucket along the axis based on the centroid of its bounds. The implementation then estimates the cost for splitting the primitives along the planes along each of the bucket boundaries (solid blue lines); whichever one gives the minimum cost per the surface area heuristic is selected.

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.

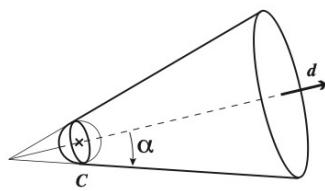


Figure 1: We use a cone-sphere structure for our ray-space hierarchy. Each node is defined by a sphere and a cone.

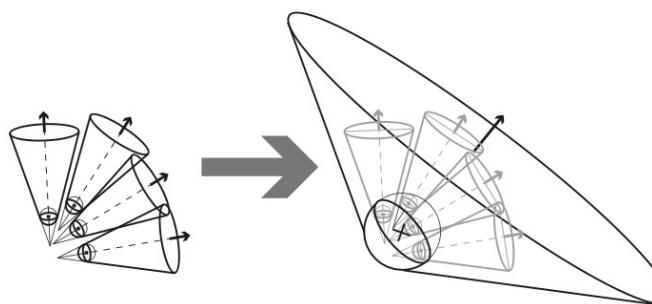


Figure 2: The parent node is constructed as the enclosing cone-sphere for the four children.

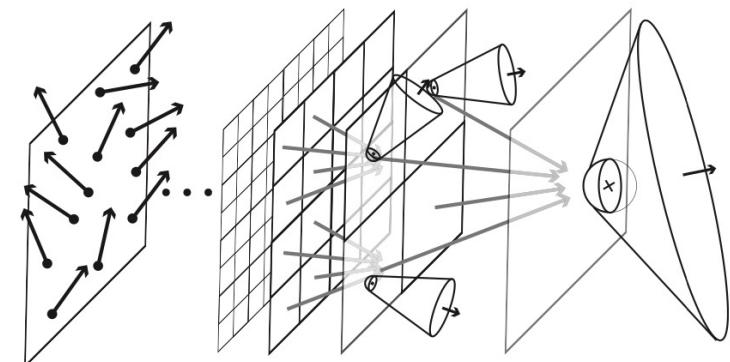


Figure 3: We start with the set of secondary rays, and recursively build the enclosing cone-sphere for each hierarchical level.

Ray-Space Hierarchy

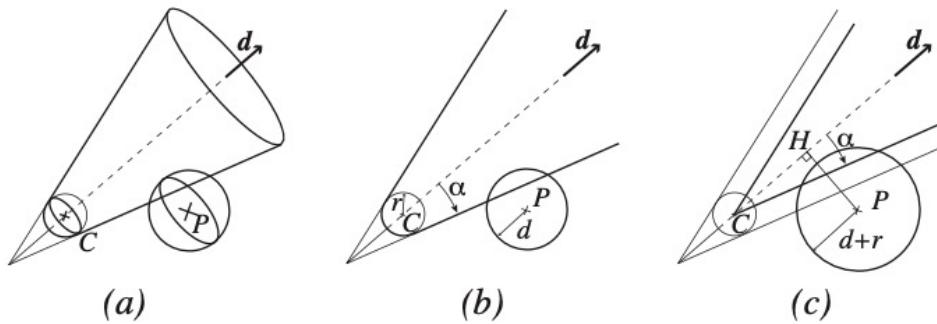


Figure 5: Testing the intersection between a node and the bounding sphere of a triangle (a) reduces to a 2D problem (b). It is equivalent to testing the intersection between a reduced cone and an enlarged sphere (c).

Future Direction

- *Hardware Ray Tracing*

Special-purpose computer hardware started from Princeton University 1996

[Imagination Technologies](#), after acquiring [Caustic Graphics](#), produced the Caustic Professional's R2500 and R2100 plug in cards containing RT2 ray trace units (RTUs). Each RTU was capable of calculating up to 50 million incoherent rays per second.^[21]

[Nvidia](#), partnering with Microsoft [DirectX](#), announced the Nvidia RTX developer library^[22] in 2018, which promised fast real-time ray tracing solutions powered by hardware accelerated ray tracing (ASIC tensor cores) found in the [Volta](#)-generation GPUs.^[23]

In October 2020, [AMD](#) announced further information regarding the "refresh" of the [RDNA](#) micro-architecture. According to the company, the [RDNA 2](#) micro-architecture supports real-time hardware accelerated ray tracing.^{[24][25]}

Intel released [Arc Alchemist](#) GPU in 2022,^{[26][27][28]} in which the GPU featured ray tracing acceleration core that are performing comparatively with [RTX 3000](#) series mid-range GPU.^[29]

On 4 November, 2021, [Imagination Technologies](#) announced their IMG CXT GPU with hardware-accelerated ray tracing.^{[30][31]}

On January 18, 2022, Samsung announced their [Exynos 2200](#) AP SoC with hardware-accelerated ray tracing.^[32]

On June 28, 2022, [Arm](#) announced their [Immortalis-G715](#) with hardware-accelerated ray tracing.^[33]

On November 16, 2022, [Qualcomm](#) announced their [Snapdragon 8 Gen 2](#) with hardware-accelerated ray tracing.^{[34][35]}

On September 12, 2023, Apple announced their [Apple A17](#) with hardware-accelerated ray tracing.^[36]

Future Direction

- BVH traversal path prediction

Predict the starting point to save memory access and time

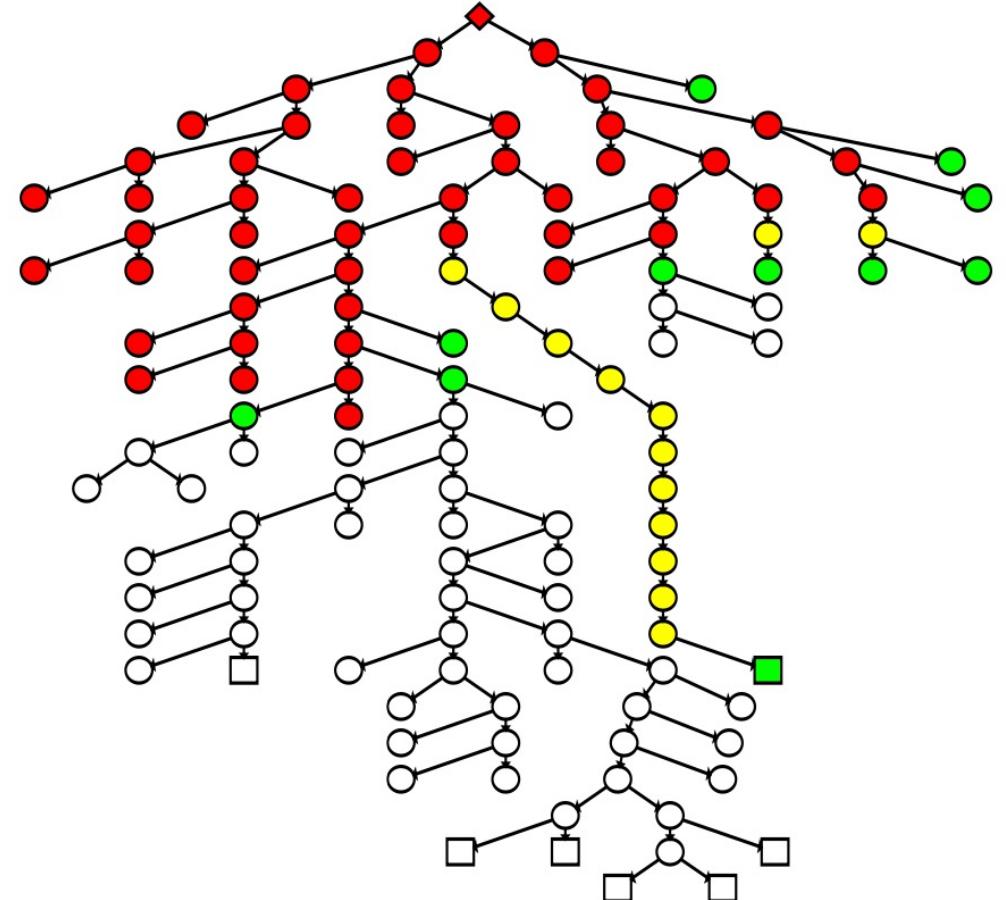
The **red** nodes are visited only by the standard traversal from the root.

The **green** nodes are entry points of our traversal algorithm.

The **yellow** nodes are visited by neither traversal method and only denote the path to candidate list elements.

The **white** nodes are visited by both types of traversal.

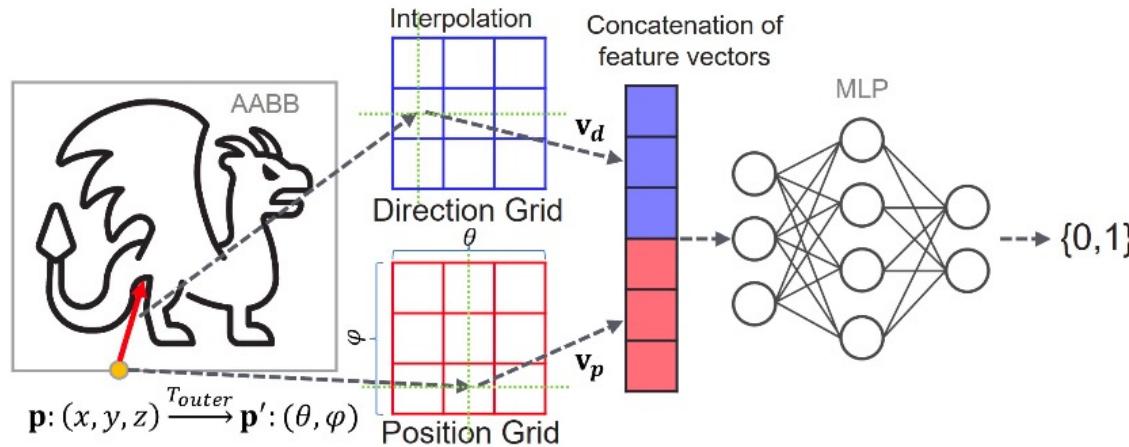
In this particular case, our method **skips 44% of nodes**, both on the path from the BVH root and in lateral branches.



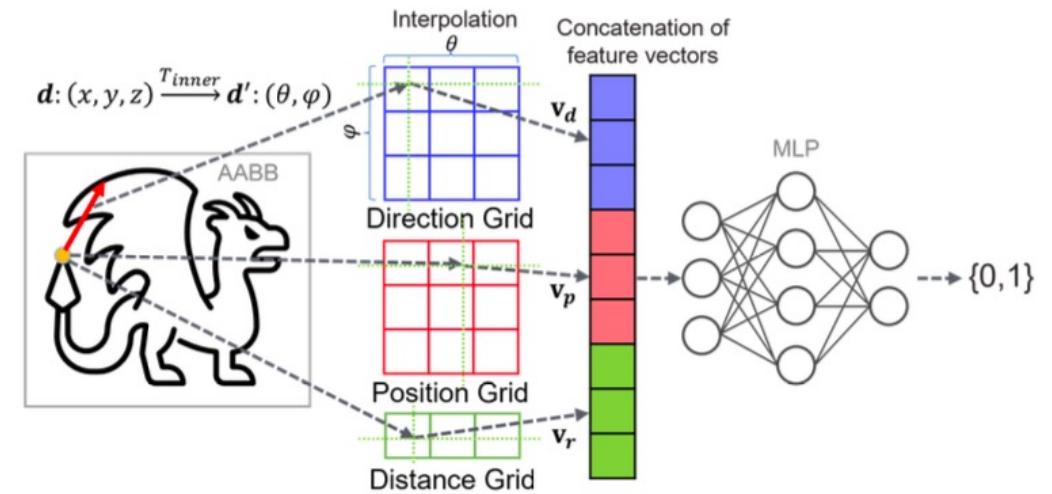
Future Direction

- *BVH Neural Intersection Function*

$$f : \{(\mathbf{p}, \mathbf{d}) | \mathbf{p}, \mathbf{d} \in \mathbb{R}^3\} \rightarrow \{0, 1\} \quad (1)$$



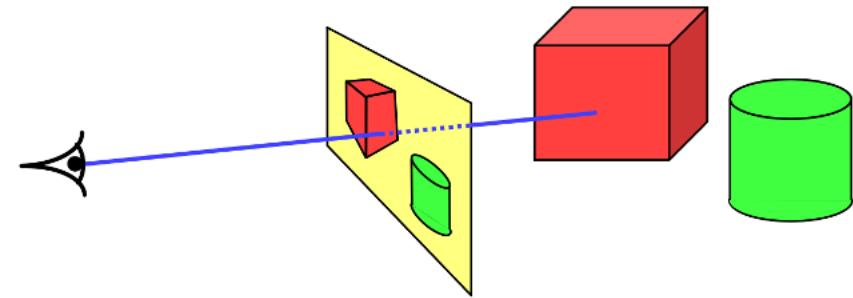
(a) NIF outer network



(b) NIF inner network

Future Direction

- *Higher branching factor for parallelism and better cache hit*
- *Moving down calculation to lower precision*
 - save energy for mobile system
- ...



With Sincere Thanks to

Lai Wei's Ray Tracing Wiki

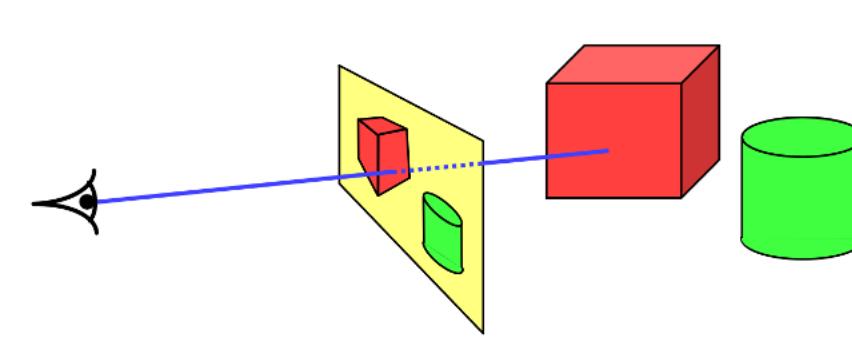
Other Useful Reference for Ray Tracing (and broader CG):

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>

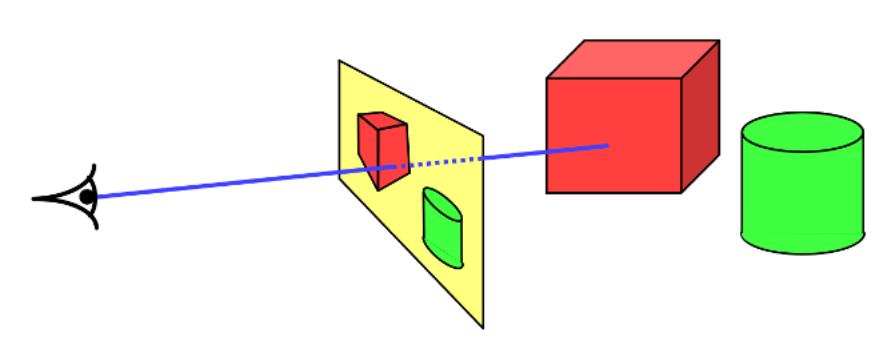
Cambridge IA Intro to Graphics <https://www.cl.cam.ac.uk/teaching/2223/Graphics/>

Vulkan_Ray_Tracing_Overview_Apr21 [Khronos Template 2015](#) (slides)





Q&A





RTX
ON

Q&A

Thank you for Listening!

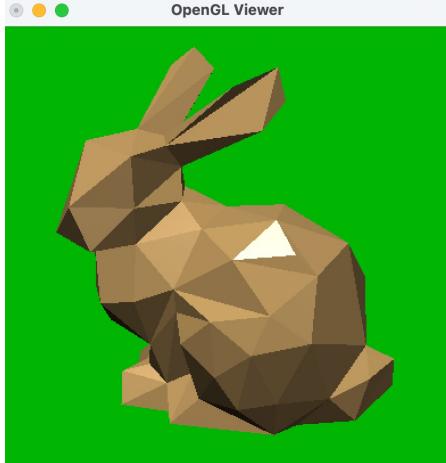
Wish you enjoy this journey on RAY TRACING!

Best of luck!

*Speeding up
real-time
Ray Tracing*



RTX
OFF



Thank you for Listening!

Wish you enjoy this journey on RAY TRACING!

Best of luck!

