## Lecture \#15

# Ray Tracing - Basics 

Computer Graphics
Winter Term 2020/21

Marc Stamminger / Roberto Grosso

## Introduction

- Up to now: Rasterization
- Scanline algorithm to enumerate hit pixels
- Local illumination using Phong lighting
- Occlusion using depth buffer
- Ray Tracing:

A different rendering paradigm

- More illumination effects $\rightarrow$ Global Illumination
- Physically motivated illumination computations


## Up to now: Rasterization

```
for each triangle t
    find pixels inside t
        shade pixel
```



## Ray Casting

- Ray Casting $\subset$ Ray Tracing



## Ray Casting $\rightarrow$ Ray Tracing

- Having a method at hand that intersects a ray with our scene, we use this to generate new lighting effects that are not directly possible with rasterization $\rightarrow$ reflections
$\rightarrow$ refractions
$\rightarrow$ shadows
$\rightarrow$ indirect illumination (later)


## Ray Tracing

- Ray Casting $\rightarrow$ Ray Tracing


## eye rays

## shadow rays

## reflection rays

## refraction rays

## Introduction

- 1968: Ray Casting: Arthur Appel
- 1979: Recursive ray tracing: Turner Whitted



## Introduction



Image by Henrik Wann Jensen. He writes: One of my first ray tracing images (1990-1991). Rendered first time on an Amiga in HAM mode (the good old days).

## Introduction

- Car with reflections, refractions, environment lighting



## Introduction

- Indirect Illumination $\rightarrow$ not possible with simple ray tracing



## Introduction

- Caustics: light patterns generated by reflections off specular surfaces $\rightarrow$ not possible with simple ray tracing

graphics.ucsd.edu

https://blenderartists.org/forum/showthread.php?116585-Realistic-underwater-lighting-and-caustics-added-tutorial-link
- more in the lecture "Global Illumination", next summer term


## Ray Tracing

- Today: Basics of Ray Tracing
- how to generate eye rays
- how to intersect a ray with scene geometry
- a first ray caster
- Next Lectures:
- how to generate secondary rays
- recursive ray tracing procedure
- accelerations structures for fast ray tracing
- special effects possible with ray tracing


## Rays

- Mathematical representation of an (eye) ray
- Parametric line from ray origin (eye) $e$ in direction $d$

$$
p(t)=e+t d
$$

- $p(0)=e$
- $0<t_{1}<t_{2} \Rightarrow p\left(t_{1}\right)$ closer to the eye than $p\left(t_{2}\right)$
- $t<0 \Rightarrow p(t)$ behind the eye
- Ray test:
- find intersection of ray with scene with smallest $t>0$


## Eye Ray Generation

- Every eye ray belongs to one pixel
- Starting point of eye ray: camera
- Eye ray goes through pixel on image plane

- For a particular pixel $p$, the eye ray is:
$e=$ camera position $d=(p-e) /||p-e||$
- Intersection with objects: gather t -values with $t>0$
- Smallest $t \Rightarrow$ first intersection $\Rightarrow$ visible object


## Eye Ray Generation

- Remember from Lecture \#07: Viewing and Perspective Given camera position, view direction and up-vector $\rightarrow$ compute camera basis vectors $u, v, w$

- Use this to generate an image plane:

$$
e+w+x u+y v
$$



## Eye Ray Generation

- Point $(x, y)$ on image plane:

$$
e+w+x u+y v
$$

- Using the field of view fovy and aspect ratio aspect, a window is defined on the image plane:

$$
\begin{aligned}
& \quad(x, y) \in\left[-x_{m}, x_{m}\right] \times\left[-y_{m}, y_{m}\right] \\
& \text { with } y_{m}=\tan \frac{\text { fovy }}{2}, x_{m}=\text { aspect } y_{m}
\end{aligned}
$$

- Finally, we map integer pixel coordinates $(i, j)$ to this window:

$$
x=\underbrace{\left(\frac{i+0.5}{n_{x}} \times 2-1\right) x_{m,} \quad y \text { analog }}_{\begin{array}{c}
\text { relative } \\
\text { coordinate }
\end{array}} \begin{gathered}
\text { number of } \\
\text { pixels in } x
\end{gathered}
$$

## Eye Ray Generation

- Eye ray computation:
- compute $(u, v, w)$ for camera frame
- for pixel $(i, j)$ : eye ray is $e+t d$ with
$e=$ camera position

$$
\begin{gathered}
x=\left(\frac{i+0.5}{n_{x}} \times 2-1\right) \times \text { aspect } \times \tan \frac{\text { fovy }}{2} \text { and } \\
\qquad \begin{array}{c}
y=\left(\frac{j+0.5}{n_{y}} \times 2-1\right) \times \tan \frac{f o v y}{2} \\
d=\frac{w+x u+y v}{\|w+x u+y v\|}
\end{array}
\end{gathered}
$$

- Corresponds to pinhole camera with planar projection plane


## Eye Rays: Other Camera Models

- In ray tracing, we can easily handle camera types other than projective pinhole cameras, e.g. panoramic cameras, fish eye lenses, or similar
- Example: panoramic camera
- eye rays through grid on surrounding cylinder


By Nickj (Own work) [CC BY-SA 3.0 (http://creativecommons.org/licenses/by-sa/3.0) or GFDL (http://www.gnu.org/copyleft/fdl.html)], via Wikimedia Commons

## Eye Rays: Other Camera Models

- Kolb et al.: "A Realistic Camera Model for Computer Graphics"
- Simulate entire lens system to mimic real cinematic lenses


200mm tele

## Eye Rays: Antialiasing

- Simple Anti-Aliasing: shoot multiple eye rays through pixel
- jitter eye rays over pixel
$\rightarrow$ smoothing of edges
$\rightarrow$ multiple texture samples
$\rightarrow$ high impact on performance


## Eye Rays: Antialiasing

- Sample patterns



## uniform:

- very uniform distribution
- good numerical properties
- only square numbers as sample numbers



## random:

- non-uniform distribution
- worse numerical properties
- arbitrary sample numbers, incrementable



## random stratified:

- uniform distribution
- good numerical properties
- only square numbers as sample numbers


## Eye Rays: Adaptive Sampling

- Adaptive Sampling: try to reduce number of eye rays $\rightarrow$ don't cast multiple rays per pixel, if color in pixel uniform
- cast eye rays through pixel grid
 corners
- if color for one pixel vary strongly (by more than a threshold), subdivide corresponding cells and cast additional rays
- stop for tiny subpixels
- average results for each pixel


## Eye Rays: Depth of Field

- Real-world lenses have a focus plane
- Objects out of this plane get blurry

[Jason Waltman / jasonwaltman.com]
- We can also simulate Depth of Field by jittering eye rays


## Eye Rays: Depth of Field

- For each pixel, cast multiple rays, jitter starting point of ray around optical center (= pinhole)
- all rays of one pixel must intersect at focus plane
- average over rays per pixel $\rightarrow$ depth of field effect

focus plane

[Jason Waltman / jasonwaltman.com]


## Ray - Object Intersection

- Does a given ray $e+t d$ intersect a scene object ? And if so, where and at which ray parameter $t$ ?
- Today we look at triangles, polygons, and spheres



## Ray - Object Intersection

- Planes and plane equations
- Normal of a plane through points $A, B, C$

$$
n=(B-A) \times(C-A) /\|(B-A) \times(C-A)\|
$$

- plane equation, point-normal form

$$
n \circ(x-A)=0
$$

- constant-normal form

$$
n \circ x=s \quad(=\mathrm{n} \circ A=n \circ B=n \circ C)
$$

- $s$ is the distance of the plane to the origin



## Ray - Plane intersection

- Parametric representation of a ray

$$
p(t)=e+t d,\|d\|=1
$$

- Plane equation

$$
n \circ x=s
$$

$\Rightarrow n \circ p(t)=s \rightarrow t=\frac{s-n \circ e}{n \circ d} \rightarrow q=e+t d$

- Notes:
- if $n \circ d=0$, the ray is parallel to the plane
- if $t<0$, the intersection is "behind" the starting point $e$



## Ray - Triangle intersection

- First intersect with plane supported by triangle, then check whether intersection point is inside triangle
- Many algorithms exist for this problem
- Simple approach: use barycentric coordinates to describe intersection point
- System of equations:

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

(barycentric coordinate $\alpha$ replaced by $1-\beta-\gamma$ )

- Unknowns: $t, \beta, \gamma$
- Intersection at $q=e+t d$ if
- $t>0$ ( $q$ on positive part of ray)
- $\beta \geq 0, \gamma \geq 0, \alpha \geq 0 \Leftrightarrow \beta+\gamma \leq 1$ ( $q$ within triangle)



## Ray - Triangle Intersection

- Solve $e+t d=a+\beta(b-a)+\gamma(c-a)$ :

$$
\underbrace{\left(\begin{array}{ccc}
\vdots & \vdots & \vdots \\
d & a-b & a-c \\
\vdots & \vdots & \vdots
\end{array}\right)}_{A}\left(\begin{array}{c}
t \\
\beta \\
\gamma
\end{array}\right)=\left(\begin{array}{c}
\vdots \\
a-e \\
\vdots
\end{array}\right)
$$

## Ray - Triangle intersection

- Solve using Cramer's rule:
- $t=\operatorname{det}\left(\begin{array}{ccc}\vdots & \vdots & \vdots \\ a-e & a-b & a-c \\ \vdots & \vdots & \vdots\end{array}\right) / \operatorname{det} A$
$\begin{aligned} & \text { - } \beta \beta \operatorname{det}\left(\begin{array}{ccc}\vdots & \vdots & \vdots \\ d & a-e & a-c \\ \vdots & \vdots & \vdots\end{array}\right) / \operatorname{det} A \\ & \text { - } \gamma=\operatorname{det}\left(\begin{array}{ccc}\vdots & \vdots & \vdots \\ d & a-b & a-e \\ \vdots & \vdots & \vdots\end{array}\right) / \operatorname{det} A\end{aligned}$
- If $\operatorname{det} A=0$, then
- The triangle is degenerate (a line or a point) or
- The ray is parallel to the triangle


## Ray - Triangle intersection

```
Boolean raytri (ray r, point3 a, point3 b, point3 c,
                        interval[t0,t1])
    compute t
    if (t < t0) or (t > t1) then
        return false
    compute det(A)
    if det(A) == 0 // border case. no intersection
        return false
    compute }
    if ( }\gamma<0\mathrm{ ) or ( }\gamma>1)\mathrm{ then
        return false
    compute }
    if ( 
        return false
    if ( }\beta+\gamma>1) the
        return false
    else
        return true
```


## Ray - Triangle intersection

- Optimized implementation:
- first, use $\operatorname{det}(a, b, c)=a \circ(b \times c)=b \circ(c \times a)=c \circ(a \times b)$
- so in our case:
- normal $n=(b-a) \times(c-a) \quad$ (not normalized)
- $\operatorname{det} A=d \circ n$
- $t=\frac{\operatorname{det}(a-e, a-b, a-c)}{\operatorname{det} A}=\frac{(a-e) \circ n}{\operatorname{det} A}$
- $\beta=\frac{\operatorname{det}(d, a-e, a-c)}{\operatorname{det} A}=\frac{(\mathrm{a}-\mathrm{c}) \circ(d \times(a-e))}{\operatorname{det} \mathrm{A}}$
- $\gamma=\frac{\operatorname{det}(d, a-b, a-e)}{\operatorname{det} A}=-\frac{(a-b) \circ(d \times(a-e))}{\operatorname{det} A}$


## Ray - Triangle intersection

- These optimizations are used in the following code
- Note: ray is parameterized by two points $(p, q)$, so $e=p$ and $d=q-p$

```
// Given ray pq and triangle abc, returns whether ray intersects
// triangle and if so, also returns the barycentric coordinates (u,v,w)
// of the intersection point
int IntersectSegmentTriangle(Point p, Point q, Point a, Point b, Point c,
                                    float &u, float &v, float &w, float &t)
{
    Vector ab = b - a;
    Vector ac = c - a;
    Vector qp = p - q;
    // Compute triangle normal. Can be precalculated or cached if
    // intersecting multiple segments against the same triangle
    Vector n = Cross(ab, ac);
    // Compute denominator d.
    float d = Dot(qp, n);
    // If d == 0, ray is parallel to triangle or triangle is degenerate
    if (fabs(d) < 1e-10) return 0;
```


## Ray - Triangle intersection

```
    // Compute intersection t value of pq with plane of triangle.
    // A ray intersects iff 0 <= t.
    // Delay dividing by d until intersection has been found
    Vector ap = p - a;
    t = Dot(ap, n);
    if (t < O.Of) return 0;
    // Compute barycentric coordinate components and test if within bounds
    Vector e = Cross(qp, ap);
    v = Dot(ac, e);
    if (v < O.Of || v > d) return 0;
    w = - Dot(ab, e);
    if (w < 0.0f || v + w > d) return 0;
    // Segment/ray intersects triangle. Perform delayed division and
    // compute the last barycentric coordinate component
    float ood = 1.0f / d;
    t *= ood;
    v *= ood;
    w *= ood;
    u = 1.0f - v - w;
    return 1;
}
```


## Ray - Triangle intersection

- Interpretation of ray-triangle intersection test [Möller,Trumbore]
- Transform to new coordinate system (described by matrix $A$ ) from two triangle edges and ray direction
- then $t=e_{z}, \beta=b_{x}, \gamma=c_{y}$



## Ray - Polygon Intersection

- Given a planar polygon with
- $m$ vertices $p_{1}$ through $p_{m}$
- All on a plane with normal $n$

1. compute intersection $p$ of ray with the plane containing the polygon
2. test if $p$ is within the polygon

## Ray - Polygon Intersection

1. Ray Polygon Intersection

- Ray $e+t d$
- Compute intersection with plane containing polygon

$$
p=e+\frac{\left(p_{1}-e\right) \circ n}{d \circ n} d
$$

2. Test if $p$ lies in polygon

- Simple solution:

Project polygon and $p$ onto the coordinate plane with the largest projection.

- Then do 2D inside-outside test


In this example, we project the triangle on to the $x y$-plane because the normal has maximal component in $z$

## Ray - Polygon Intersection

- Two different cases:
- Polygon is convex $\rightarrow$ simple edge tests possible
- Polygon is non-convex $\rightarrow$ more complicated in/out test needed


## Ray - Polygon Intersection

- Convex polytopes (polyhedra)
- A polyhedron (convex polygon in 2D) can be described as the intersection of a set of half spaces
- a point $x$ is inside the polyhedron, if it is within all the half spaces

$$
\begin{aligned}
& n_{1} \circ x-d_{1}>0 \\
& n_{2} \circ x-d_{2}>0 \\
& n_{3} \circ x-d_{3}>0
\end{aligned}
$$

- $\rightarrow$ Lecture "Rasterization"
- Inside - Outside test for convex polygons
- convert edges to half space representation
- check point for all half spaces
$\rightarrow$ as soon as one fails, point is outside
$\rightarrow$ if none fails, point is inside
- ideally: half space vectors stored with triangle
$\rightarrow$ fast, but consumes memory


## Ray - Polygon Intersection

- For non-convex polygons, the previous test is not correct $\rightarrow$ example ?
- General polygon inside-outside test
- Generate ray from point in arbitrary direction
- Count intersections with polygon boundary
- Even $\rightarrow$ outside
- Odd $\rightarrow$ inside



## Ray - Polygon Intersection

- Polygon inside-outside test
- Problem: ray hits one vertex
$\rightarrow$ should it count twice or once?
- in the example on the right, the upper ray should count two intersections, the lower one only one...
- Simple robust solution:

If such a boundary case is detected, use other ray with new direction


## Ray - Sphere Intersection

- Implicit surface equation $f(x)=0$
- Example: sphere with center $c$ and radius $r$ :

$$
(x-c) \circ(x-c)-r^{2}=0
$$

- Set the ray in the implicit equation and find $t$ and the intersection point $p$, if possible
$f(p(t))=0 \Rightarrow$ ray parameter $t$


## Ray - Sphere Intersection

- Intersection with ray $p(t)=e+t d$ :

$$
(e+t d-c) \circ(e+t d-c)-r^{2}=0
$$

- Results in quadratic equation

$$
(d \circ d) t^{2}+2 d \circ(e-c) t+(e-c) \circ(e-c)-r^{2}=0
$$

- Since $(d \circ d)=1$ :

$$
t=\frac{-b \pm \sqrt{b^{2}-4 c}}{2}
$$

with $b=2 d \circ(e-c)$ and $c=(e-c) \circ(e-c)-r^{2}$

## Ray - Sphere Intersection

- Meaning of the discriminant $b^{2}-4 c$
- If negative $\rightarrow$ no intersection
- If positive $\rightarrow$ two intersections
- where ray enters the sphere
- where ray leaves the sphere

- If zero $\rightarrow$ ray touches sphere at exactly one point
- Always check discriminant first!
- sphere-ray intersection very fast: discriminant alone tells us, whether there is an intersection
$\rightarrow$ use sphere as bounding object for more complex objects


## Other Intersection Tests

- Similar tests available for
- ellipsoids
- cylinders
- cones
- tori
- boxes


## Ray Casting

- Up to now, we have the following:
for each pixel p in image plane
generate eye ray ( $e, d$ ) through pixel $p$
tmin = infinity; omin = null;
for each scene object o
t = intersect ray (e,d) with object o
just learned
if ray intersects object and $t$ < tmin
tmin = t;
omin = o;
if omin ! = null
compute lighting at hit point on object omin


## just learned

 set $p$ to this colorelse
set $p$ to background color

## Ray Casting - Lighting

- How to do the lighting at a found hit point?
- $\rightarrow$ we need the hit point, its surface normal, maybe texture coordinates etc.
- For a triangle, these can be interpolated from the vertices
- Typically, this information is stored in a Hit-Object

```
struct Hit {
    float t; // ray parameter
    Obj *obj; // hit scene object
    float alpha,beta,gamma; // barycentric coordinates
    vec3 getPosition() { ... }
    vec3 getNormal() { ... }
    vec2 getTexCoord() { ... }
}
Hit Scene::intersect(Ray &ray) { ... }
```


## Ray Casting - Lighting

- new version

```
for each pixel p in image plane
    Ray ray = camera->getEyeRay(p);
    Hit closestHit = null;
    for each scene object o
        Hit hit = o.intersect(ray);
        if closestHit == null || hit.t < closestHit.t
        closestHit = hit;
    if closestHit != null
        c = closestHit.obj.shader.computeLighting(
                        closestHit.getPos(),
                closestHit.getNormal(),
                ...);
        setPixelColor(p,c);
    else
        setPixelColor(p,backgroundColor);
```


## Ray Casting

- Up to now, we generate exactly the same images as with a rasterizer...
- But with much more effort:
- $n$ : number of objects (millions)
- m: number of pixels (millions)
- Intersecting one ray with all objects is $O(n)$
- Generating the image is then $O(\mathrm{mn}) \rightarrow$ impractical
- compare: rasterizer is $O(m+n)$ (for constant depth complexity)
- Next lectures:
- how we can simulate new nice effects with ray tracing
- how we can compute ray intersections in $O(\log n)$ ?

