CS380: Introduction to Computer Graphics

Ray tracing

Chapter 20

Min H. Kim

KAIST School of Computing



Color and Ray-Tracing

## SUMMARY



## Color Reproduction as Linear Algebra

- Goal of reproduction: visual response to $s$ and $s_a$ *is the same:*

$$M_{XYZ}\tilde{s} = M_{XYZ}\tilde{s}_a.$$

- Substituting in the expression for $s_a$,

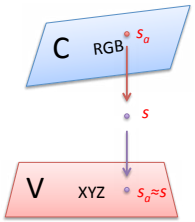
$$V = M_{XYZ}s. \quad s_a = M_{RGB}C.$$

$$M_{XYZ}\tilde{s} = M_{XYZ}M_{RGB}C.$$

$$C = (M_{XYZ}M_{RGB})^{-1}M_{XYZ}\tilde{s}.$$

Color reproduction model for display

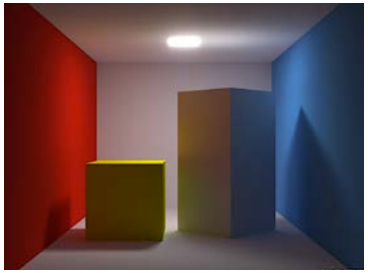
where $M_{XYZ} \in \mathbb{R}^{3\times n}$, $M_{RGB} \in \mathbb{R}^{n\times 3}$






Chapter 20

## RAY TRACING

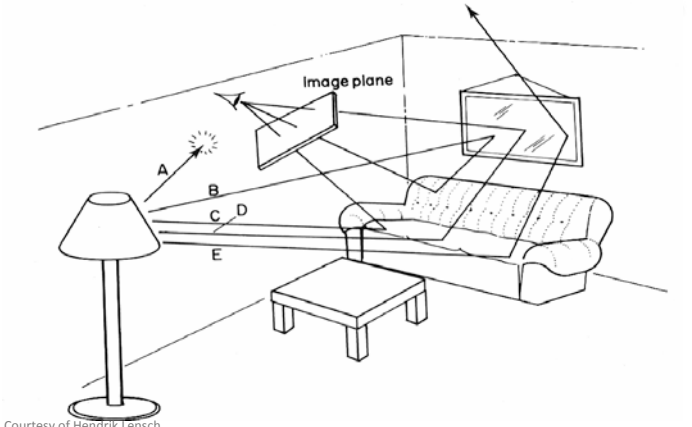


## Ray tracing

- Different approach to organizing the rendering process
- It is pixel and ray based, instead of triangle based.
- Historically, it has not been hardware accelerated, and is used for offline rendering.
- It is very flexible and so various optical effects can be put in very easily.
- It is at the heart of photorealistic methods of light simulation.
- We will cover it only briefly.



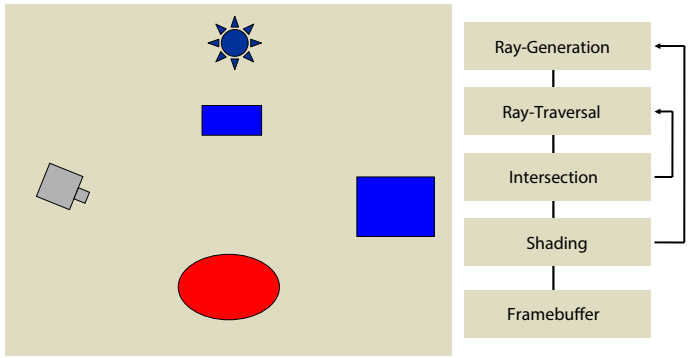
## Ray tracing pipeline




Courtesy of Hendrik Lensch



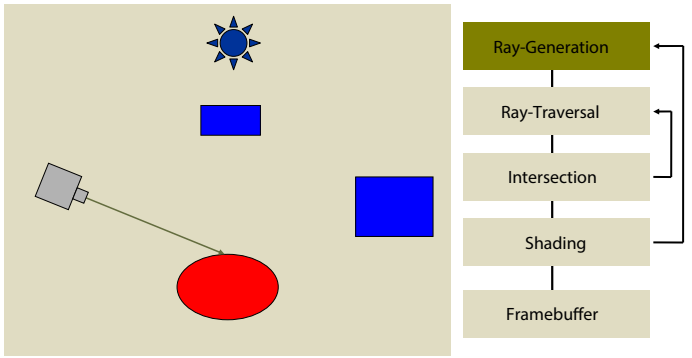
## Ray tracing pipeline




Courtesy of Hendrik Lensch

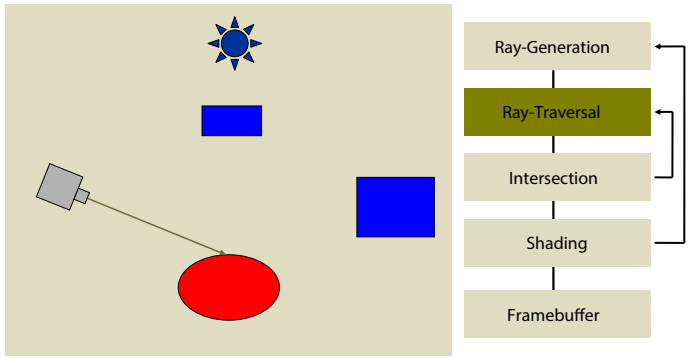


## Ray tracing pipeline



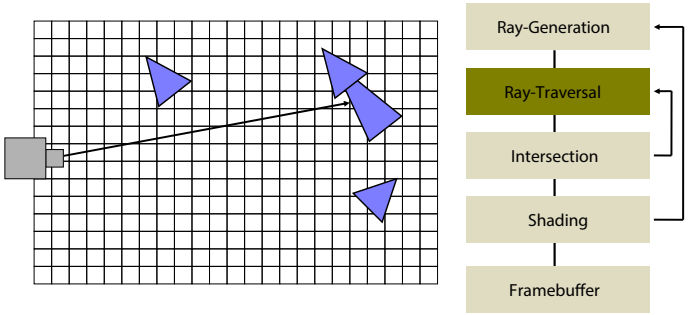

Courtesy of Hendrik Lensch

## Ray tracing pipeline



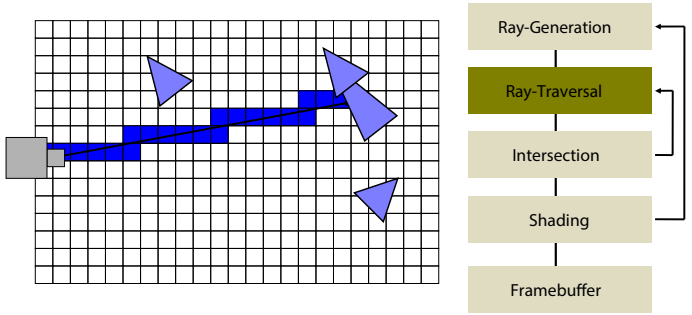

Courtesy of Hendrik Lensch
Min H. Kim (KAIST) *Foundations of 3D Computer Graphics, S. Gortler, MIT Press, 2012* 9

## Ray tracing pipeline



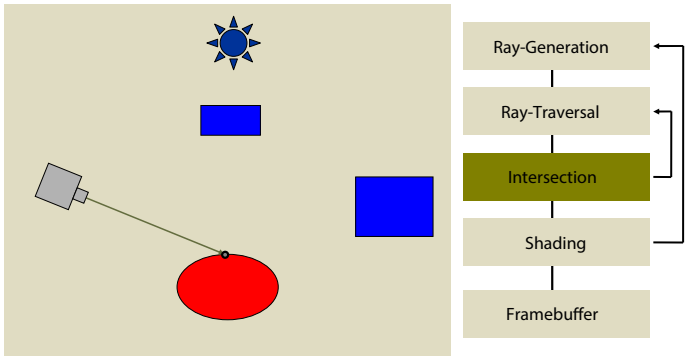

Courtesy of Hendrik Lensch
Min H. Kim (KAIST) *Foundations of 3D Computer Graphics, S. Gortler, MIT Press, 2012* 10

## Ray tracing pipeline



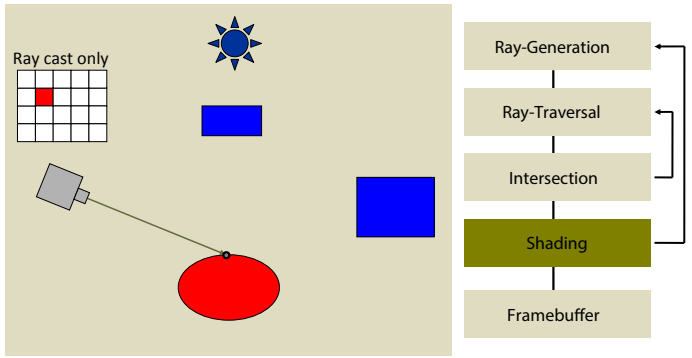

Courtesy of Hendrik Lensch
Min H. Kim (KAIST) *Foundations of 3D Computer Graphics, S. Gortler, MIT Press, 2012* 11

## Ray tracing pipeline




Courtesy of Hendrik Lensch
Min H. Kim (KAIST) *Foundations of 3D Computer Graphics, S. Gortler, MIT Press, 2012* 12

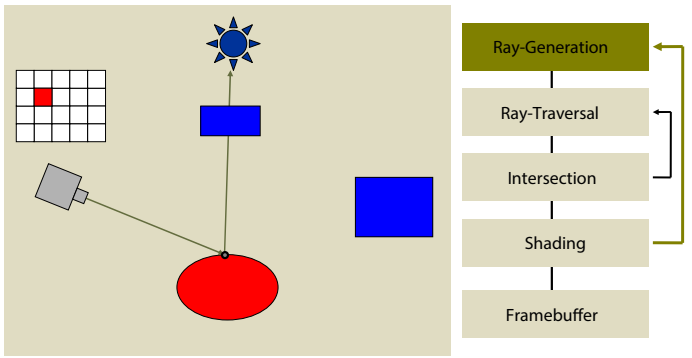
## Ray tracing pipeline




Courtesy of Hendrik Lensch



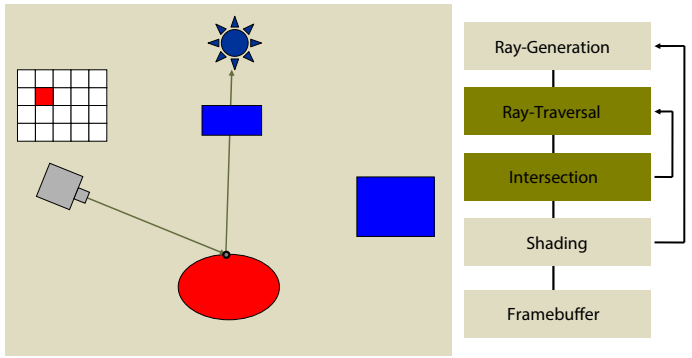
## Ray tracing pipeline




Courtesy of Hendrik Lensch



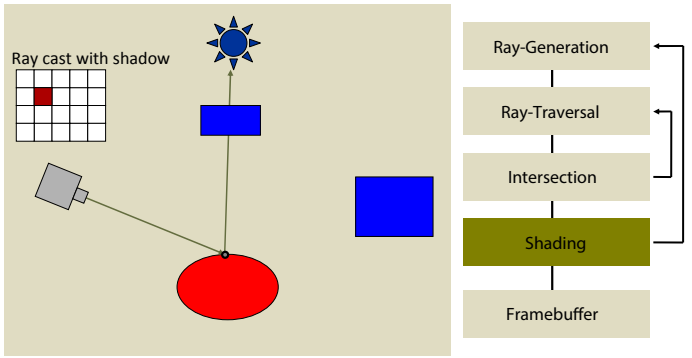
## Ray tracing pipeline




Courtesy of Hendrik Lensch

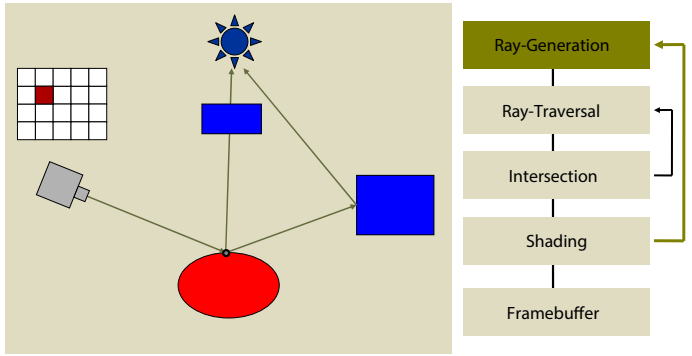


## Ray tracing pipeline




Courtesy of Hendrik Lensch

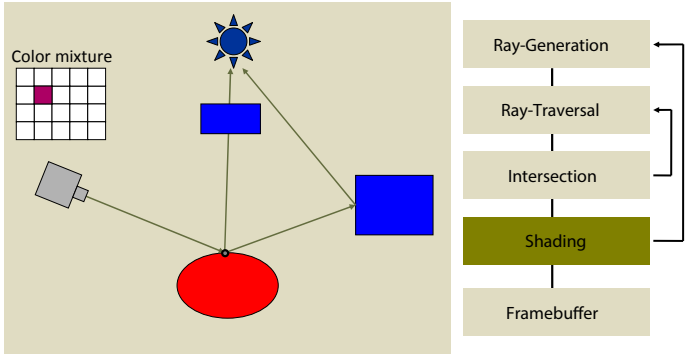
## Ray tracing pipeline




Courtesy of Hendrik Lensch



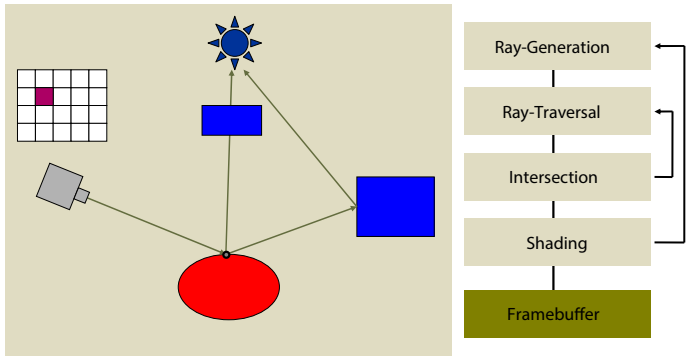
## Ray tracing pipeline




Courtesy of Hendrik Lensch



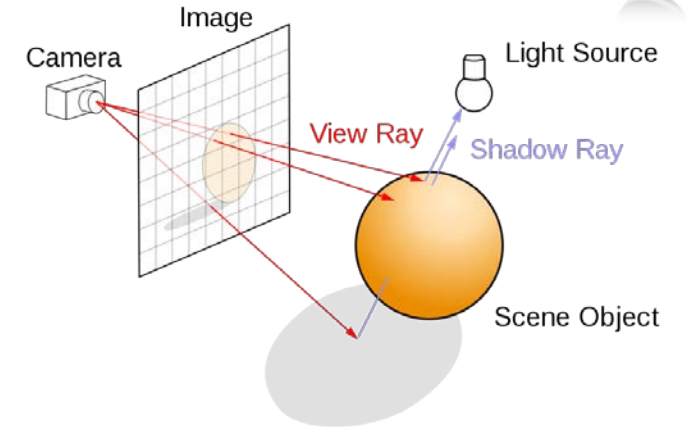
## Ray tracing pipeline




Courtesy of Hendrik Lensch



## Ray tracing



## Loop ordering: raster

- *Rasterization* is organized therefore:

```
initialize z-buffer
for all triangles
  for all pixels covered by the triangle
    compute color and z
    if z is closer than what is already in the z-buffer
      update the color and z of the pixel
```

- In basic *ray casting*, we reverse the loop orders to obtain:

```
for all pixels on the screen
  for all objects seen in this pixel
    if this is the closest object seen at the pixel
      compute color and z
      set the color of the pixel
```



## Benefits of rasterization

- This algorithm has the nice property that each triangle in the scene is touched only once, and in a predictable order.
  - Good for memory usage
  - Use in Pixar Renderman
- Setup is reduced over a triangle



## Benefits of ray casting

- Shading is automatically deferred, until after visibility is done.
- We can maintain depth order of fragments with minimal memory.
  - Letting us easily model non-refractive transparency
  - Letting us easily model constructive solid geometry (CSG).
- Using root finding, we can directly intersect rays with smooth geometries and do not need to triangulate.

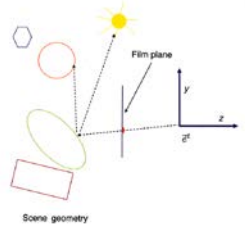


## Benefits of ray casting

- Easy to use ray infrastructure for tons of generalizations
  - Shadows reflection…
- Note: many of these things can be done to varying degrees in a rasterize, but with some extra effort expended.
- Also note that with programmable GPUs shading is the bigger cost, not the rasterization, and that the GPUs can be used to do ray calculations as well.

## Intersection

- The main computation needed in ray tracing is computing the intersection of a geometric ray $(\tilde{p},\vec{d})$ with an object in the scene.
  - Here $\tilde{p}$ is the start of the ray, which goes off in the direction $\vec{d}$





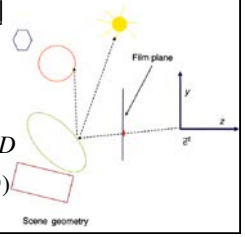

## Ray-plane

- Plane described by the equation $Ax + By + Cz + D = 0$
- We start by representing every point along the ray using a single parameter $\lambda$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} + \lambda \begin{bmatrix} d_x \\ d_y \\ d_z \end{bmatrix}$$

- Plugging this into the plane equation, we get

$$0 = A(p_x + \lambda d_x) + B(p_y + \lambda d_y) + C(p_z + \lambda d_z) + D$$
$$= \lambda(Ad_x + Bd_y + Cd_z) + (Ap_x + Bp_y + Cp_z + D)$$






## Ray-plane

- And we see that

$$\lambda = -\frac{(Ap_x + Bp_y + Cp_z + D)}{(Ad_x + Bd_y + Cd_z)}$$

- $\lambda$ tells us where along the ray the intersection point is
  - Negative valued $\lambda$ are backwards along the ray.
  - Comparisons between λ values can be used to determine which, among a set of planes, is the first one intersected along a ray.
  - *ABCD* per triangle, $d_{xyz}$ per pixel, $p_{xyz}$ per camera

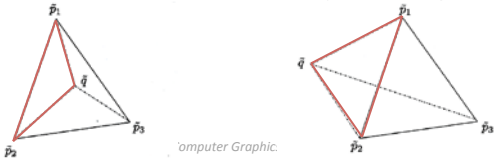


## Ray-Triangle

- In the first step, we compute the *A, B, C, D* of the plane supporting the triangle, and compute the ray-plane intersection as above.
- Next, we need a test to determine if the intersection point is inside or outside of the triangle.
- We can build such a test using the "counter clockwise" calculation seen earlier in culling

## Ray-triangle

- Suppose we wish to test if a point $\tilde{q}$ is inside or outside of a triangle $\Delta(\tilde{p}_1\tilde{p}_2\tilde{p}_3)$ in 2D
- Consider the three "sub" triangle $\Delta(\tilde{p}_1\tilde{p}_2\tilde{q}), \Delta(\tilde{p}_1\tilde{q}\tilde{p}_3)$ and $\Delta(\tilde{q}\tilde{p}_2\tilde{p}_3)$
- When $\tilde{q}$ is inside of $\Delta(\tilde{p}_1\tilde{p}_2\tilde{p}_3)$, then all three sub-triangles will agree on their clockwisedness. When $\tilde{q}$ is outside, then they all disagree.






## Ray-Sphere

- Suppose we have a sphere with radius *R* and center **c** modeled as the set of points $[x,y,z]^t$ that satisfy the equation

$$(x-c_x)^2+(y-c_y)^2+(z-c_z)^2-r^2=0$$

- Plugging the below into the above:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} + \lambda \begin{bmatrix} d_x \\ d_y \\ d_z \end{bmatrix}$$

- We get

$$0=(p_x+\lambda d_x-c_x)^2+(p_y+\lambda d_y-c_y)^2+(p_z+\lambda d_z-c_z)^2-r^2$$

$$0=(d_x^2+d_y^2+d_z^2)\lambda^2+(2d_x(p_x-c_x)+2d_y(p_y-c_y)+2d_z(p_z-c_z))\lambda+$$
$$(p_x-c_x)^2+(p_y-c_y)^2+(p_z-c_z)^2-r^2$$



## Ray-Sphere

- We can then use the quadratic formula to find the real roots λ of this equation.
  - If there are two real roots, these represent two intersections, as the ray enters and exits the sphere.
  - If there is one (doubled) real root, then the intersection is tangential.
  - If there are no real roots, then the ray misses the sphere. As above, any of these intersections may be backwards along the ray.
- At the intersection, the normal of the sphere at $[x,y,z]^t$ is in the direction $[x-c_x, y-c_y, z-c_z]^t$
- This fact may be useful for shading calculations.



## Early rejection

- When computing the intersection between a ray and the scene, instead of testing every scene object for intersection with the ray, we may use auxiliary data structures to quickly determine that some set of objects is entirely missed by the ray.
- For example, one can use a simple shape (say a large sphere or box) that encloses some set of objects.
- Given a ray, one first calculates if the ray intersects this volume.
- If it does not, then clearly this ray misses all of the objects in the bounded set, and no more ray intersection tests are needed.
- This idea can be further developed with hierarchies and spatial data structure.

## Secondary rays

- To determine if a scene point is in shadow, one follows a "shadow ray" from the observed point towards the light to see if there is any occluding geometry.
- Another easy calculation that can be done is mirror reflection (and similarly refraction). In this case, one calculates the bounce direction "bounce ray" off in that direction:

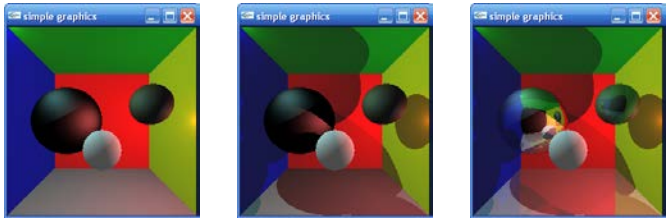
$$B(\vec{w}) = 2(\vec{w} \cdot \vec{n})\vec{n} - \vec{w}$$



## Secondary rays

- The color of the point hit by this ray is then calculated and used to determine the color of the original point on the mirror.
- This idea can be applied recursively some number of times to simulate multiple mirror.

Ray-casting only | Ray-casting + shadow | Rec. ray-tracing + shadow






## Even more rays

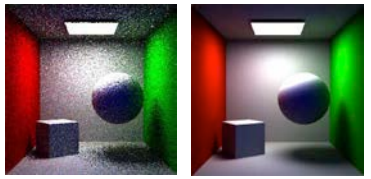
- More realistic optical simulation requires the computation of integrals, and these integrals can often be approximated by summing up contributions along a set of samples.
- Computing the values for these samples often involves tracing rays through the scene.
- For example, we may want to simulate a scene that is illuminated by a large light with finite area. This, among other things, results in soft shadow boundaries.



## Even more rays

  - Lighting due to such *area light sources* can be approximated by sending off a number of shadow rays towards the area light and determining how many of those rays hit the light source.
- Other similar effects such as focus effects of camera lenses and inter-reflection are discussed can be calculated by tracing many rays through the scene.

4 rays per pixel



1024 rays per pixel

## CS580: Computer Graphics

- If you want to learn more about advanced rendering for global illumination, there is a course (CS580) delivered by Prof. Kim.
- http://vclab.kaist.ac.kr/cs580