INTRODUCTION TO RENDERING TECHNIQUES

22 Mar. 2012

Yanir Kleiman

What is 3D Graphics?

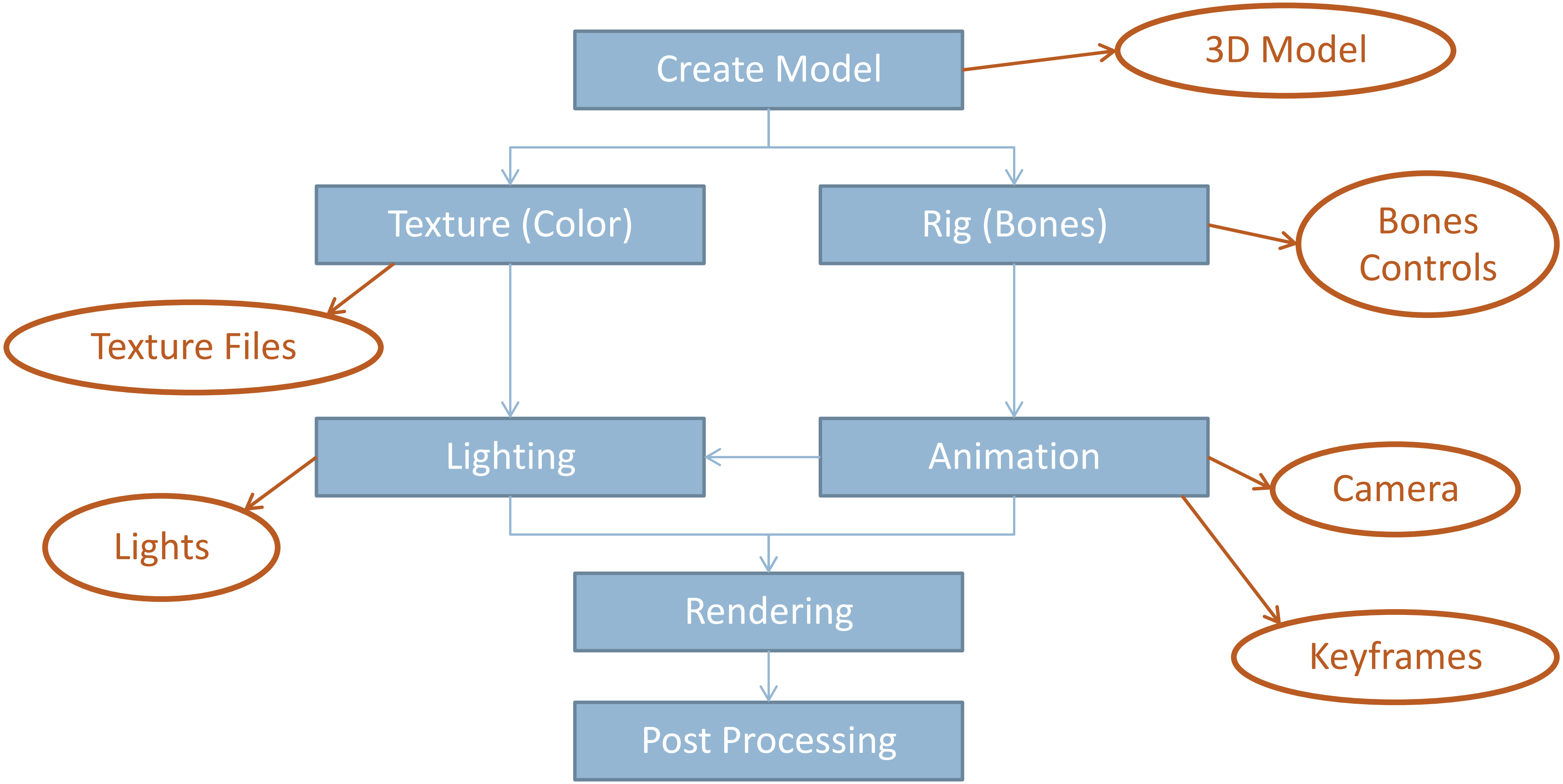
- Why 3D?

- Draw one frame at a time
- X 24 frames per second
- 150,000 frames for a feature film
- Realistic rendering is hard
- Camera movement is hard
- Interactive animation is hard

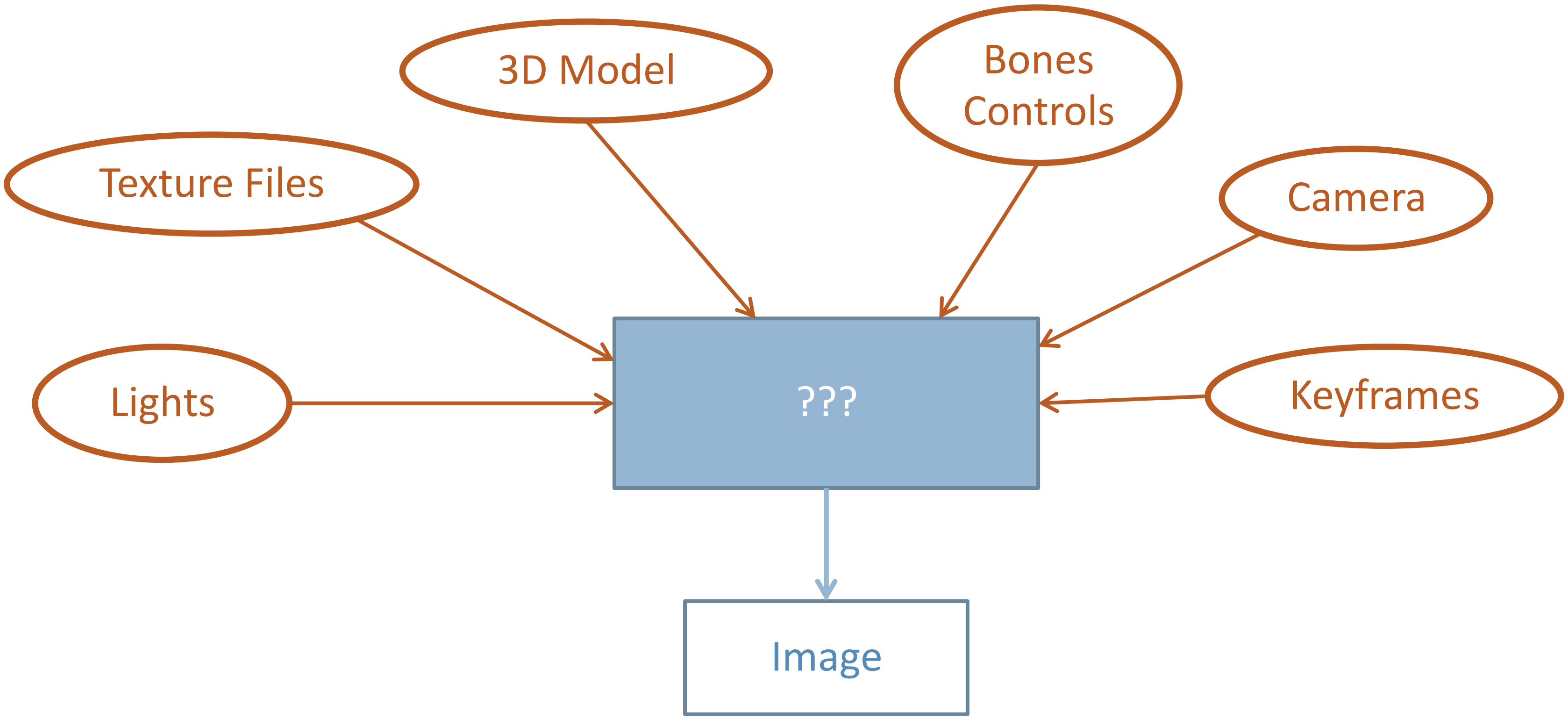
- Model only once
- Color / texture only once
- Realism / hyper realism
- A lot of reuse
- Computer time instead of artists time
- Can be interactive (games)

What is 3D Graphics?

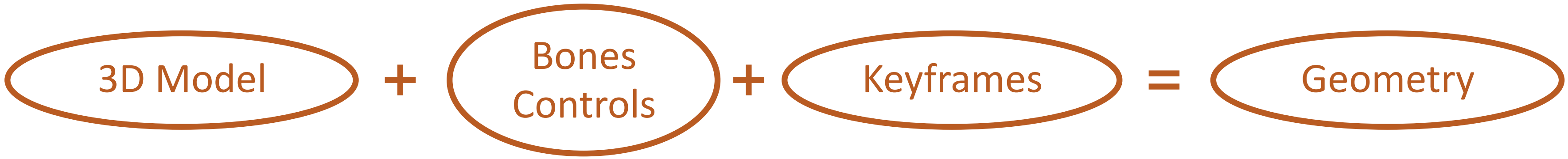
- Artists workflow – in a nutshell

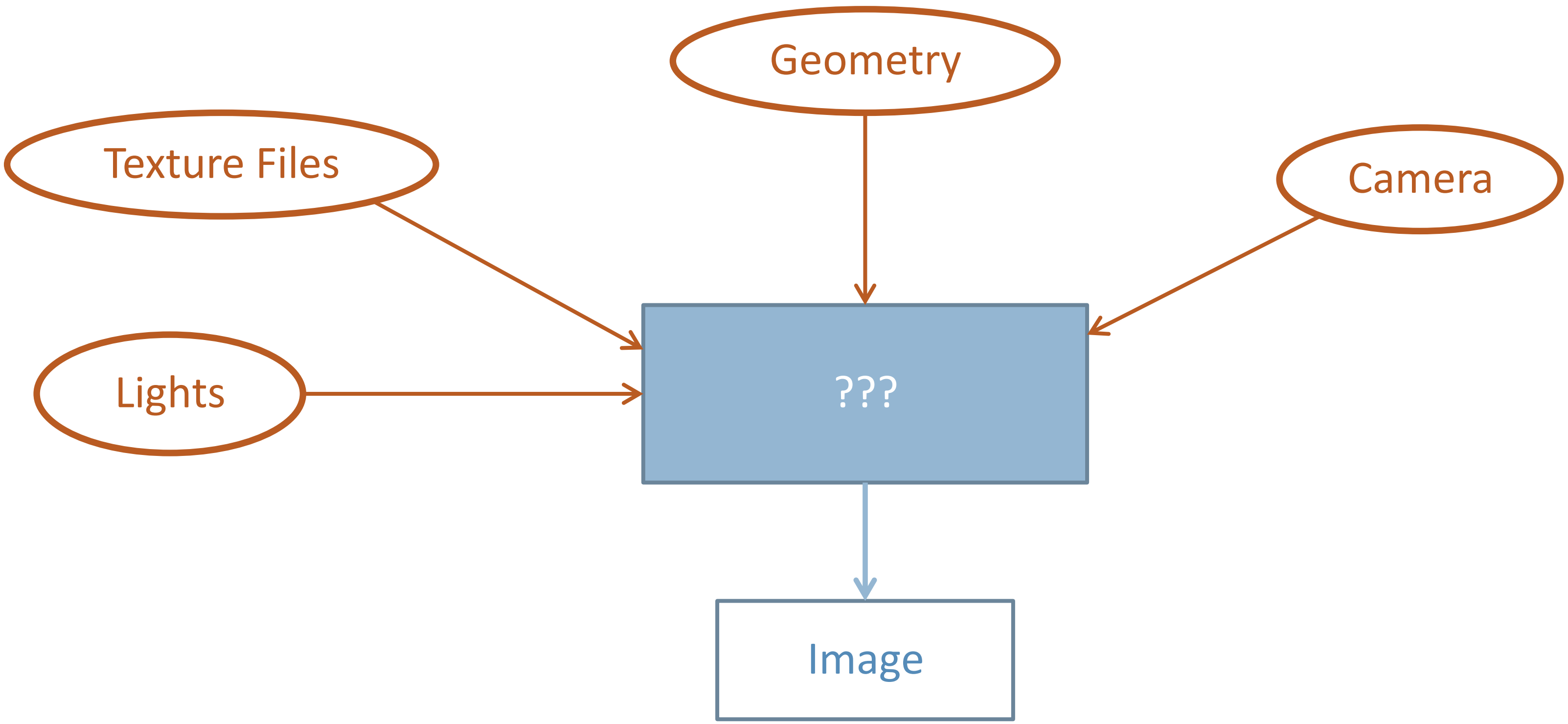


What is Rendering?



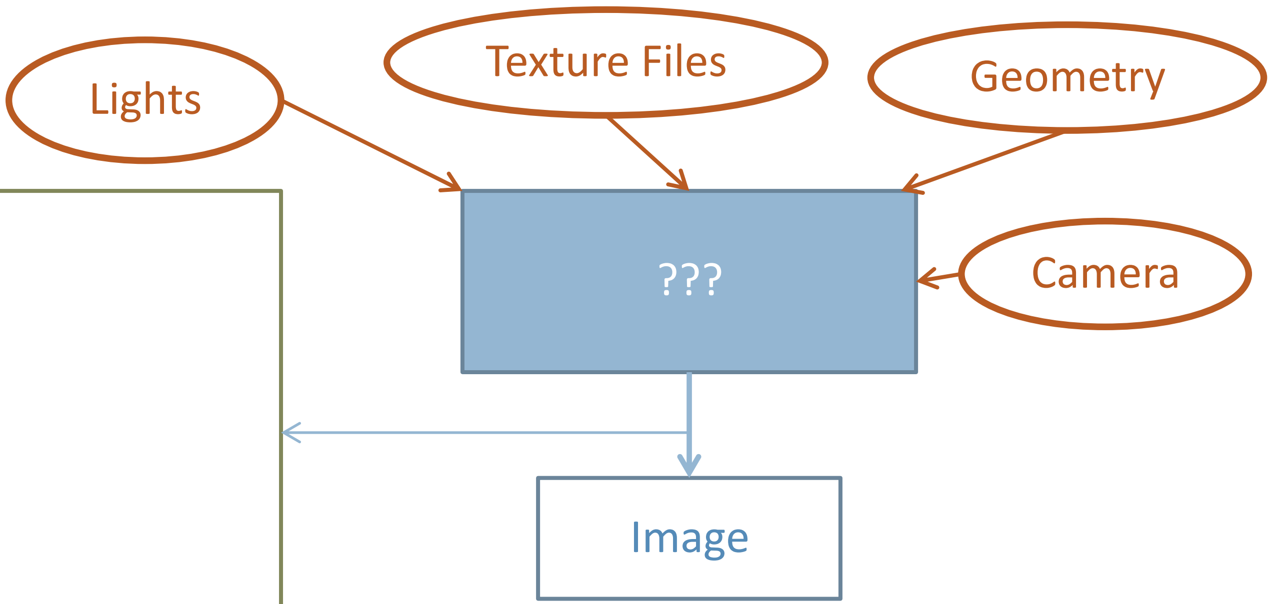
What is Rendering?





What is Rendering?

- Consider:
  - Perspective
  - Occlusion
  - Color / Texture
  - Lighting
  - Shadows
  - Reflections / Refractions
  - Indirect illumination
  - Sampling / Antialiasing



Two Approaches

- Start from **geometry**
  - For each polygon / triangle:
    - Is it visible?
    - Where is it?
    - What color is it?

Rasterization

- Start from **pixels**
  - For each pixel in the final image:
    - Which object is visible at this pixel?
    - What color is it?

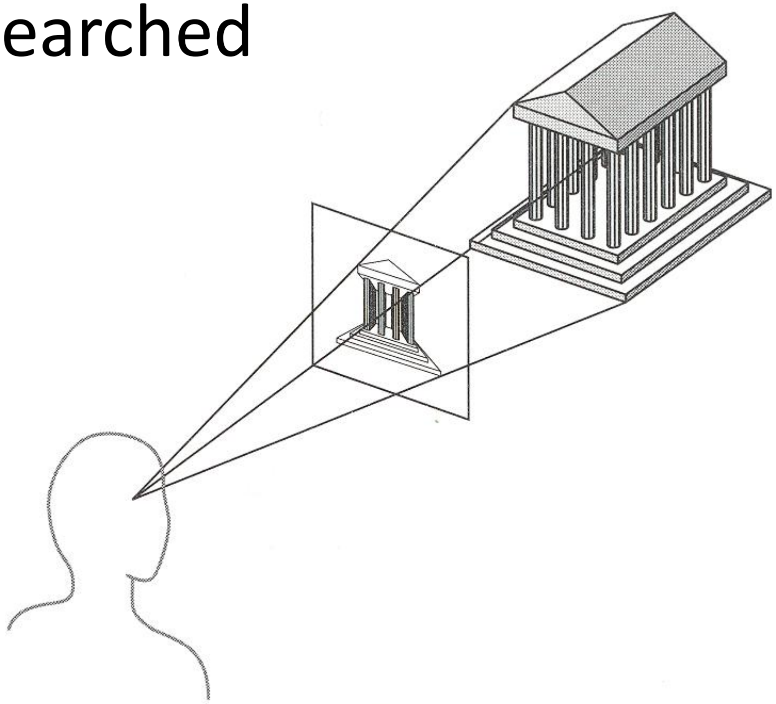
Ray Tracing

RASTERIZATION

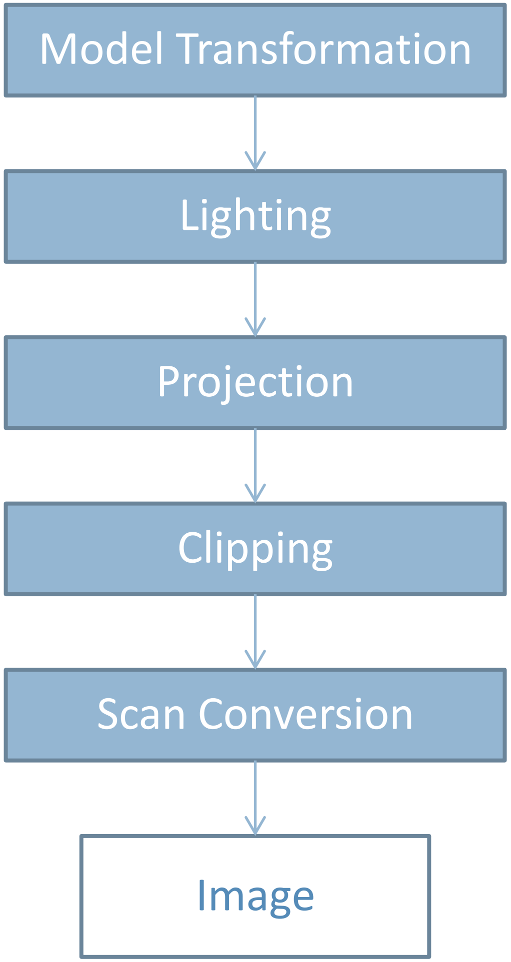
22 Mar. 2012

Rasterization

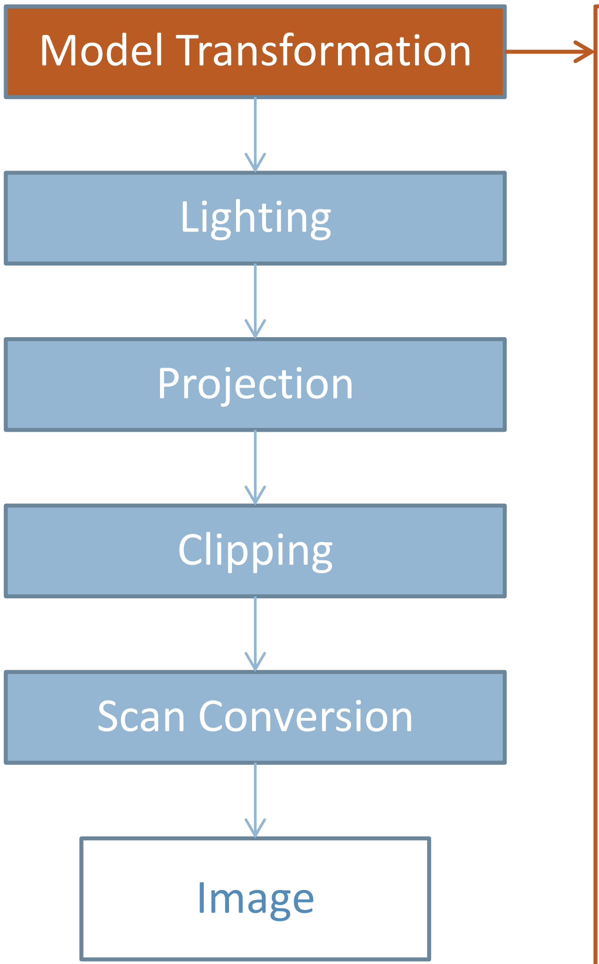
- Basic idea: Calculate projection of each triangle onto the 2D image space
- Extensively used and researched
- Optimized by GPU
- Strongly parallelized
- OpenGL
- DirectX



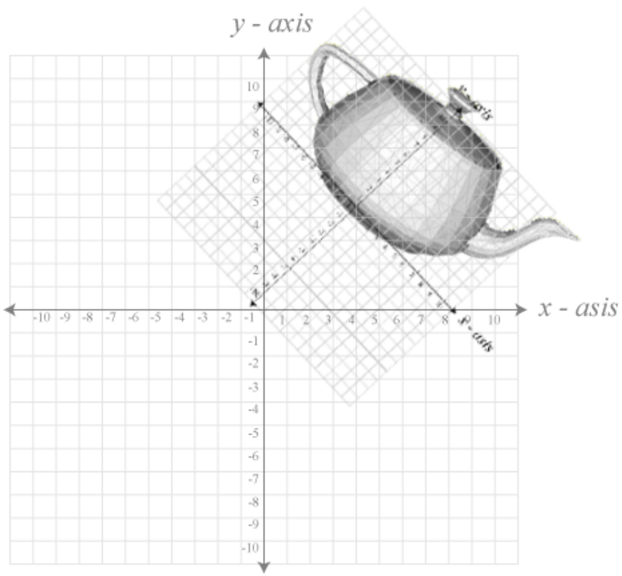
Rasterization – Graphics Pipeline



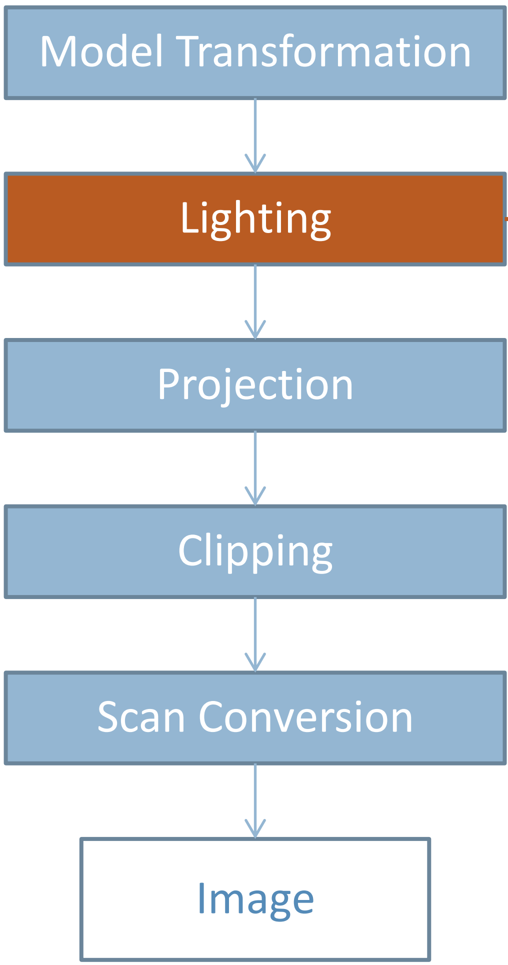
Rasterization – Graphics Pipeline



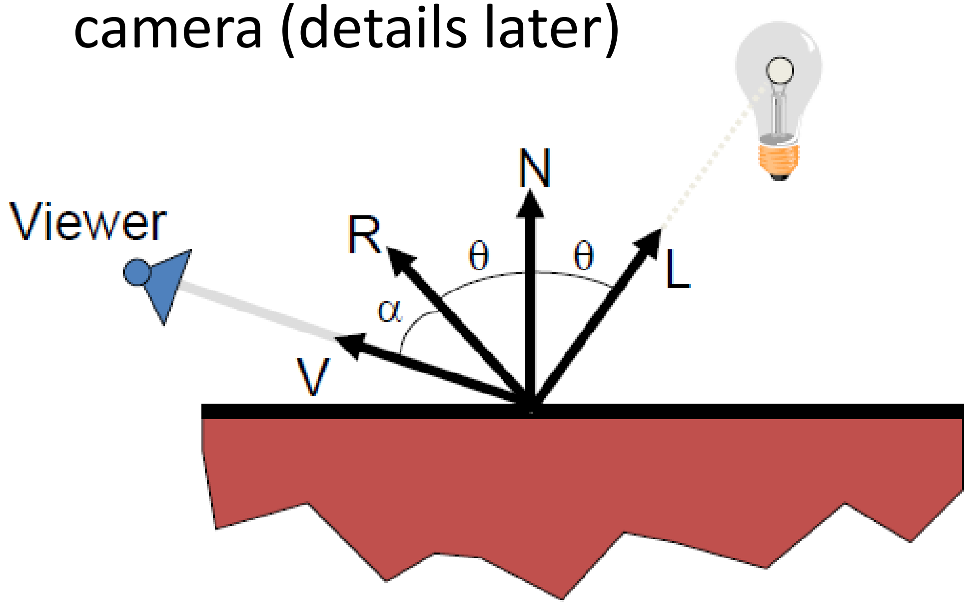
- Transform each triangle from **object space** to **world space**
- Local space -> Global space



Rasterization – Graphics Pipeline



- Computation is based on angles between light source, object and camera (details later)



- Backface culling

Rasterization – Graphics Pipeline

Model Transformation

Lighting

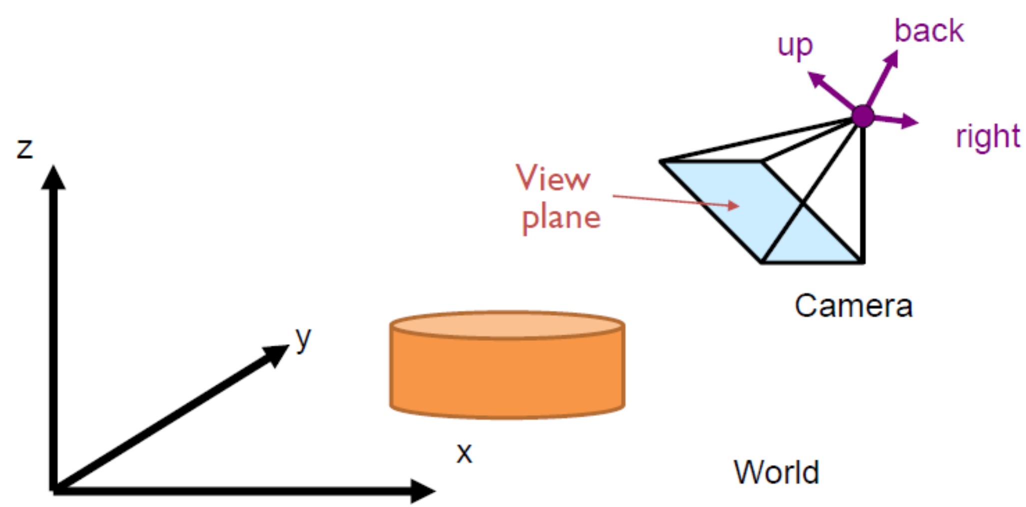
Projection

Clipping

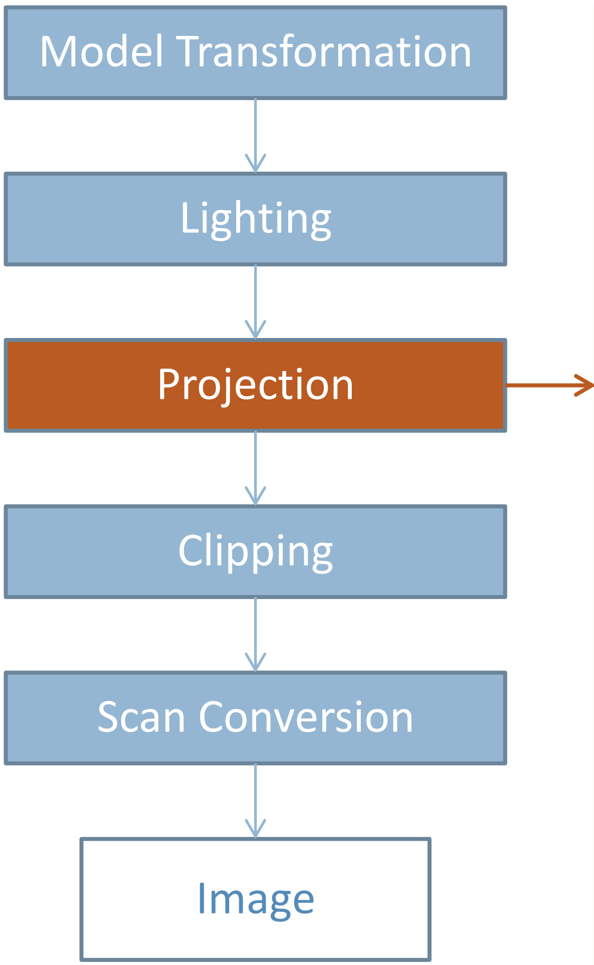
Scan Conversion

Image

- Step 1: Transform triangles from **world space** to **camera space** (orthogonal transformation)

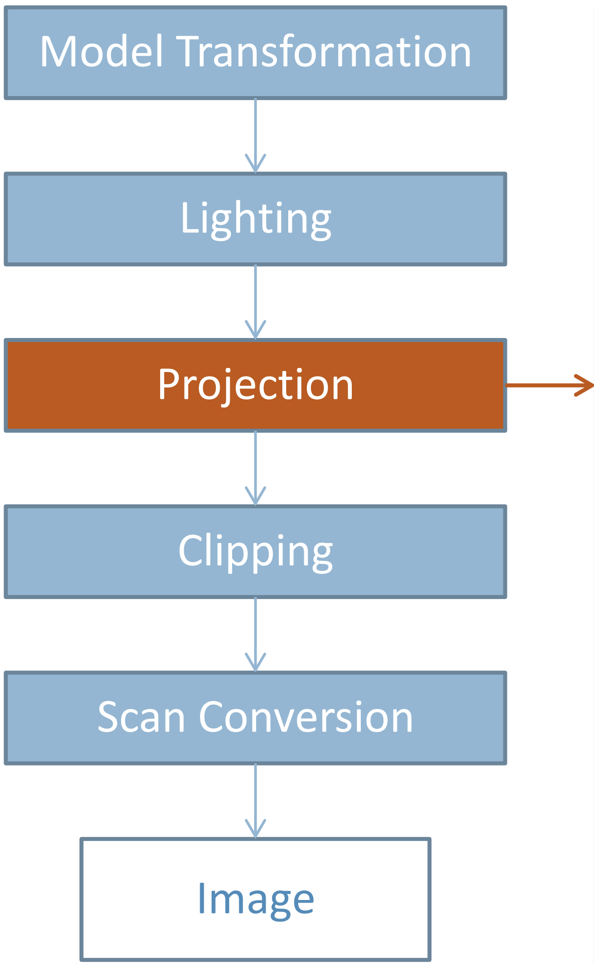


Rasterization – Graphics Pipeline

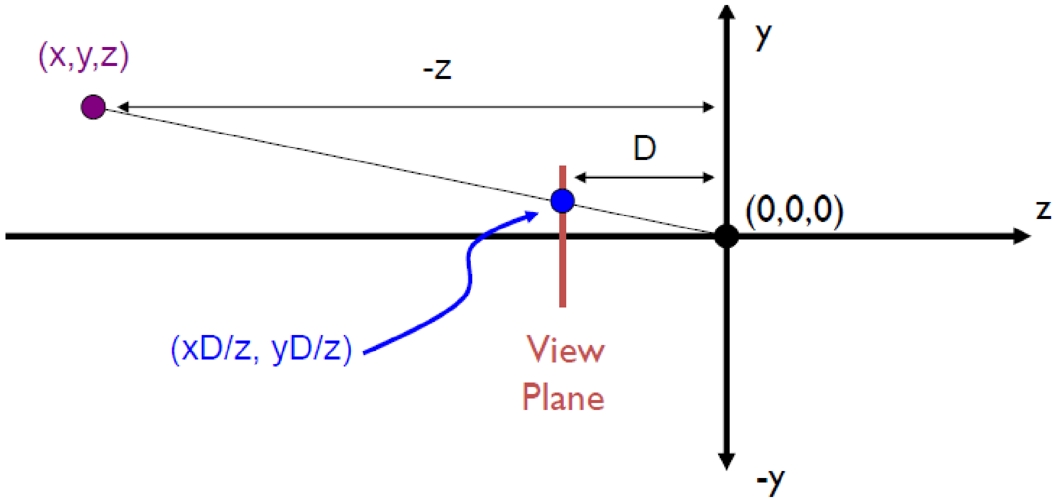


- Step 1: Transform triangles from **world space** to **camera space** (orthogonal transformation)
- Camera is at (0, 0, 0)
- X axis is right vector
- Y axis is up vector
- Z axis is “back vector” (away from camera)

Rasterization – Graphics Pipeline

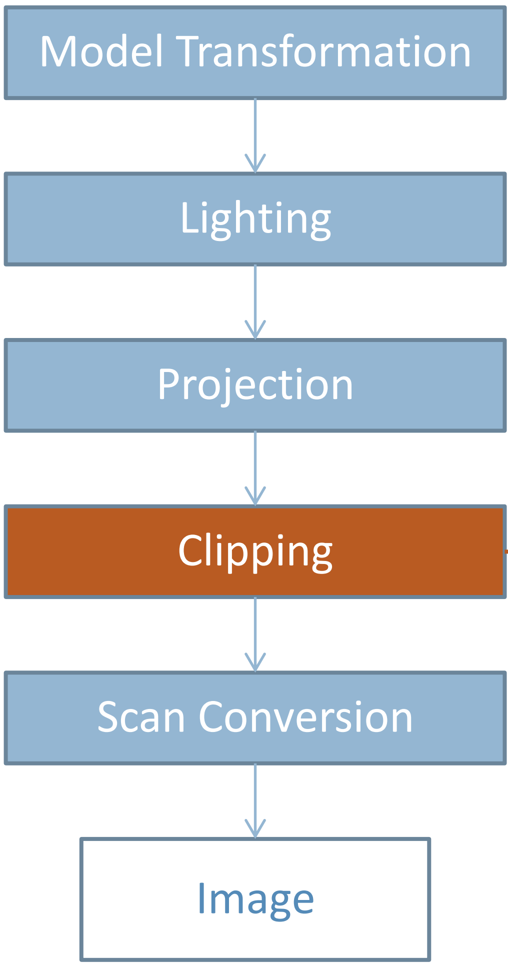


- Step 2: Perspective Projection
- Depends on focal length (D)



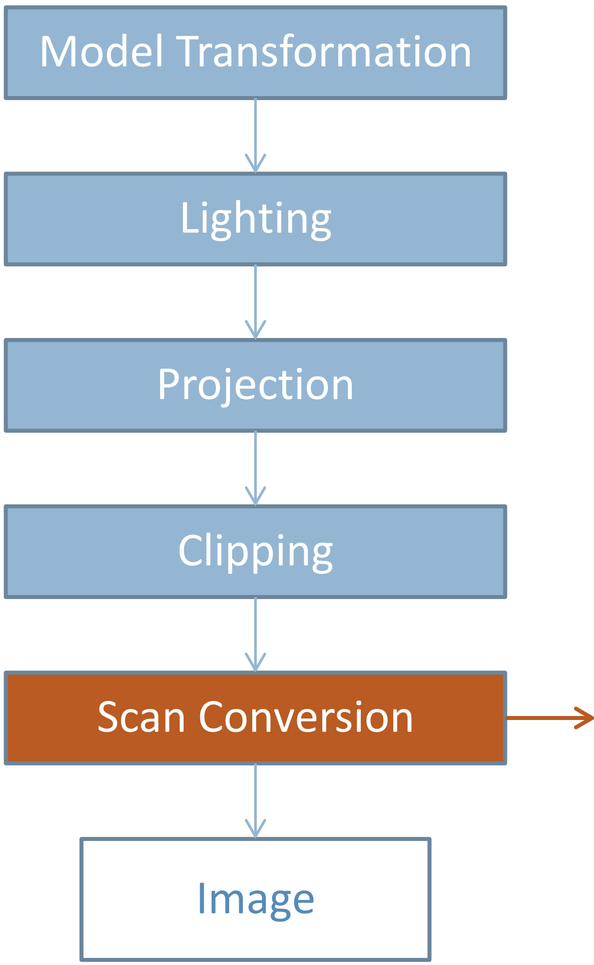
- Calculate Z-Buffer

Rasterization – Graphics Pipeline

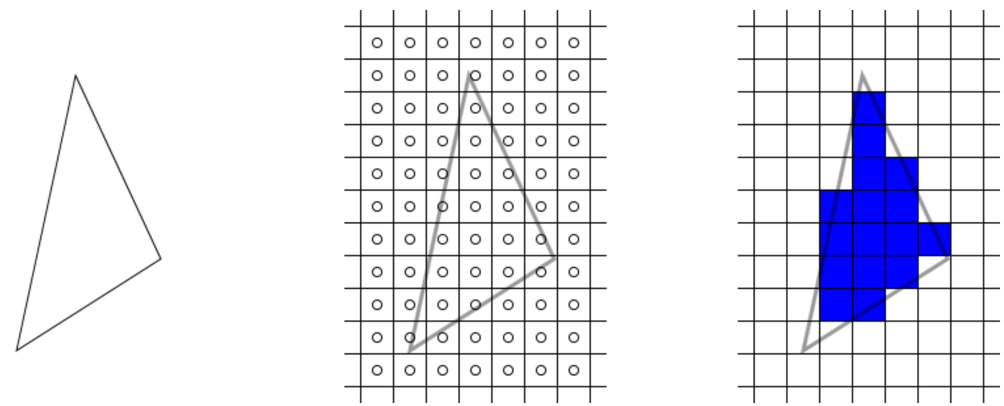


- Remove triangles that fall outside the clipping plane
- Determine boundaries of triangles partially within the clipping plane

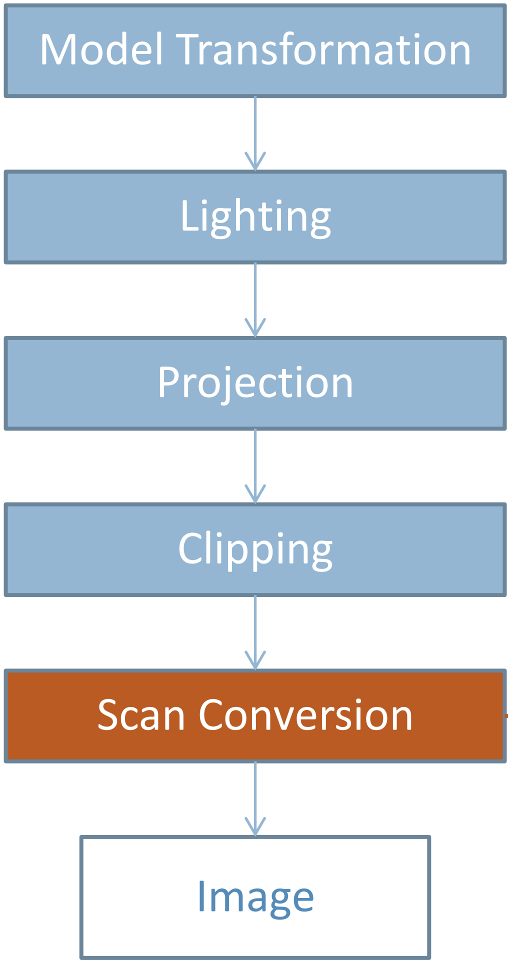
Rasterization – Graphics Pipeline



- Drawing the triangles in 2D
- Scanning horizontal scan lines for each triangle



Rasterization – Graphics Pipeline

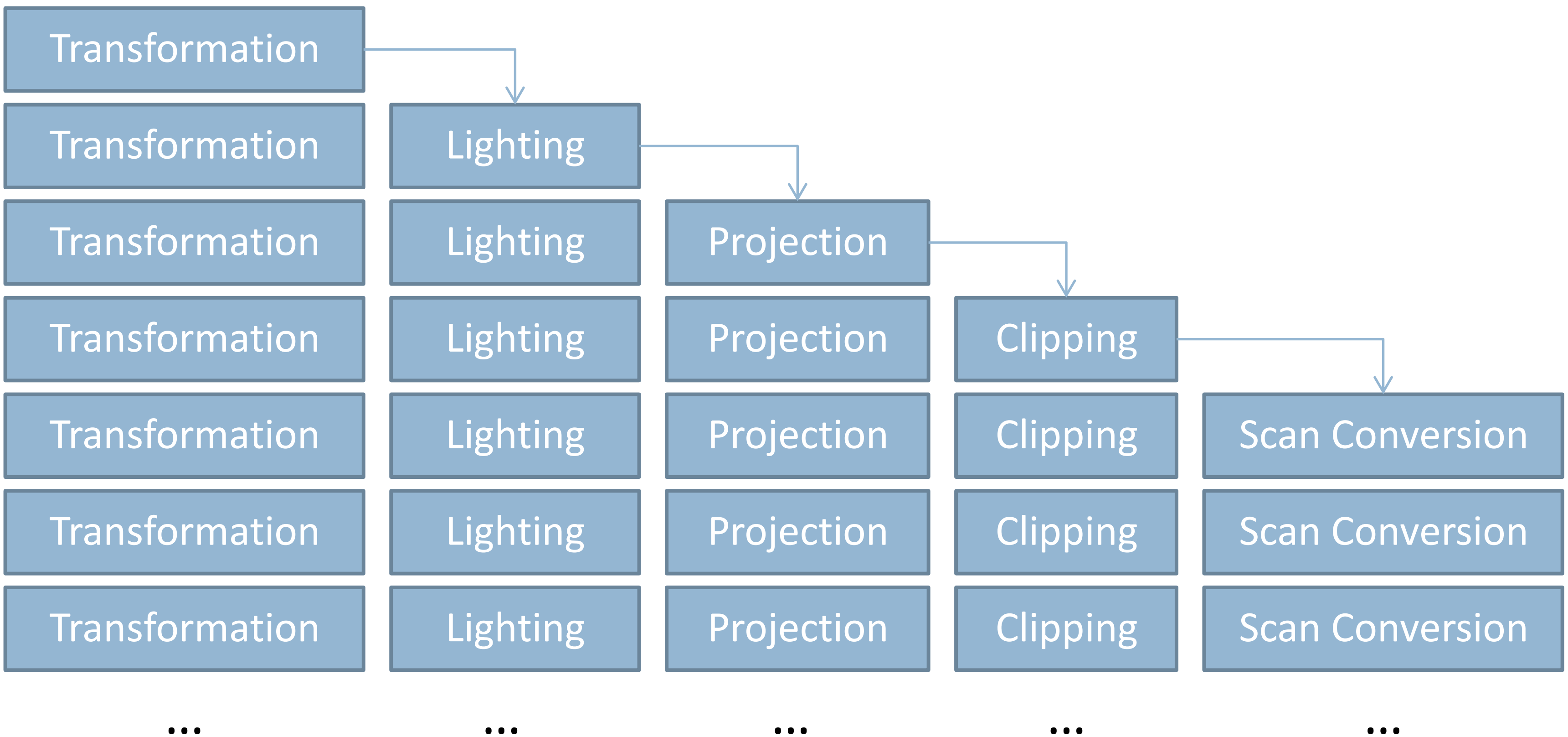


- Check z-buffer for intersections
- Use precalculated vertex lighting
- Interpolate lighting at each pixel (smooth shading)

- Texture: Every vertex has a texture coordinate (u, v)
- Interpolate texture coordinates to find pixel color

Rasterization – Parallel Processing

- Triangles are independent except for z-buffer
- Every step is calculated by a different part in the GPU

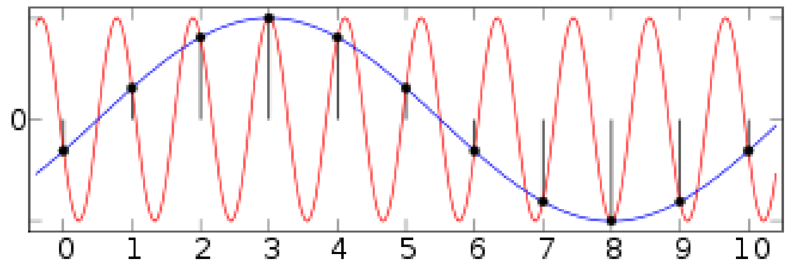


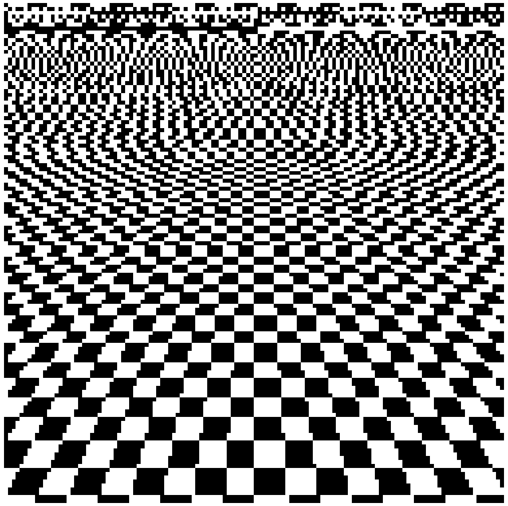
Rasterization – Parallel Processing

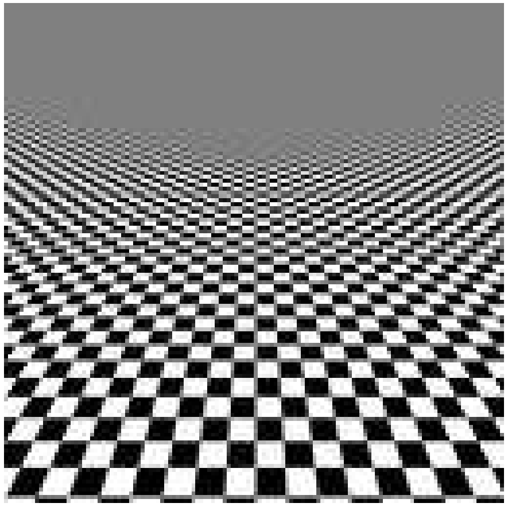
- Modern GPUs can draw **600M polygons** per second
- Suitable for real time applications (gaming, medical)
- But what about…
  - Shadows?
  - Reflections?
  - Refractions?
  - Antialiasing?
  - Indirect illumination?

Rasterization – Antialiasing

- Aliasing examples



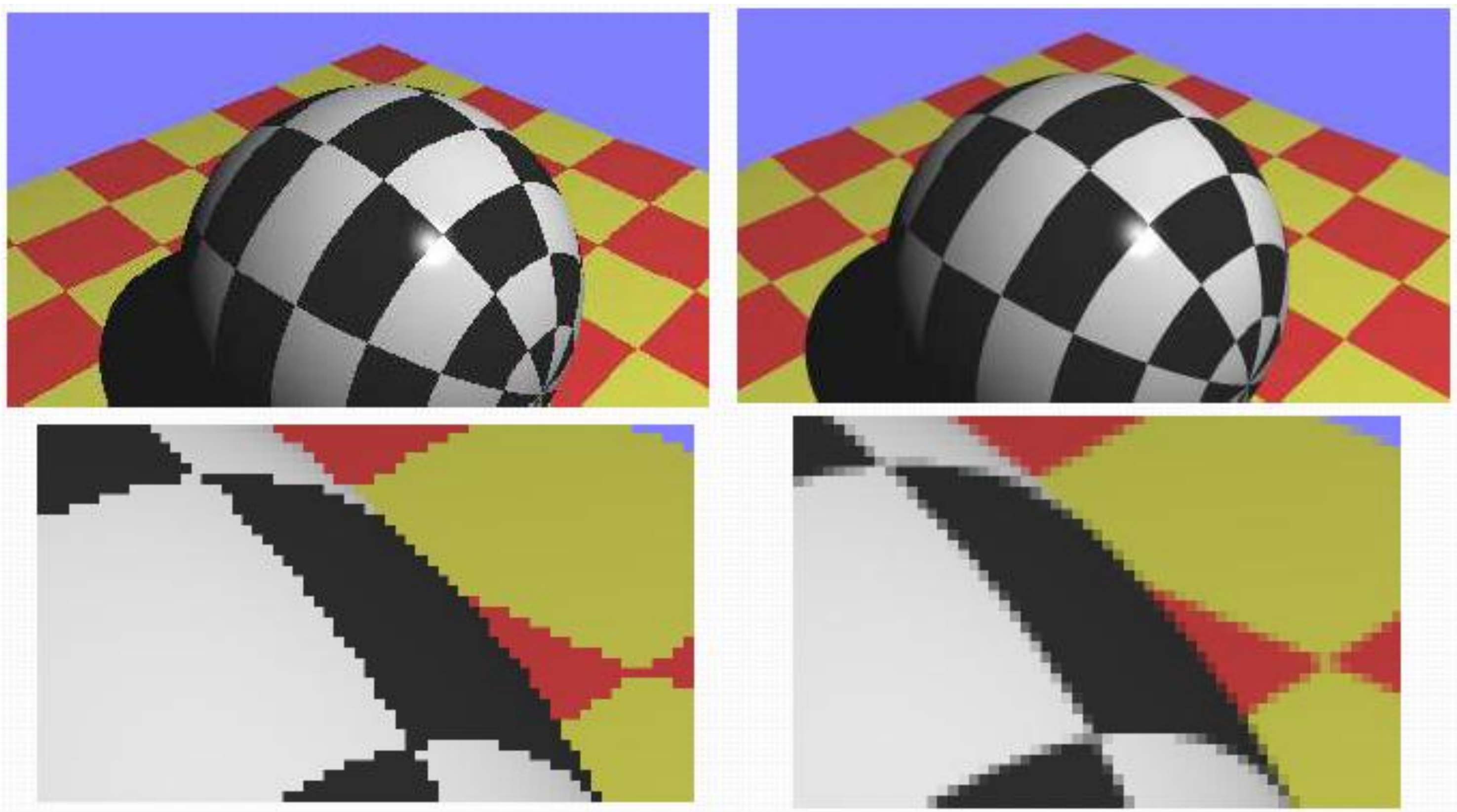




Rasterization – Antialiasing

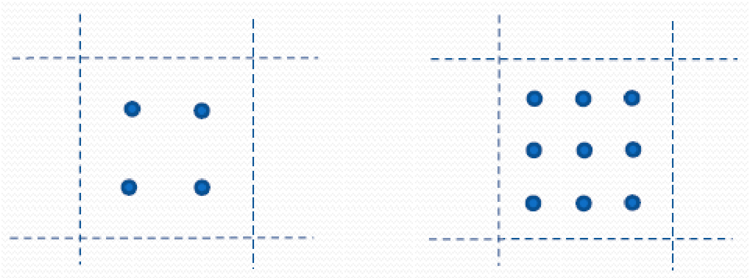
- Aliasing examples

Aliasing **Anti-aliased**



Rasterization – Antialiasing

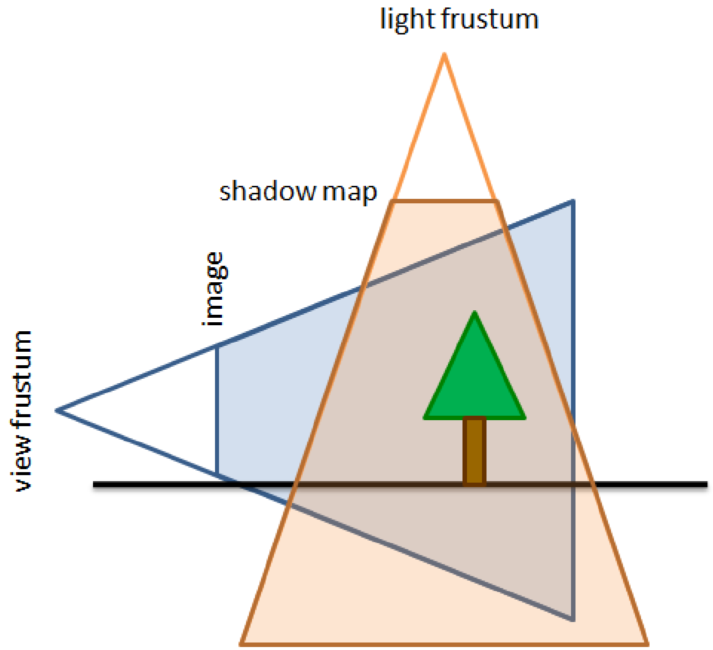
- Antialiasing: Trying to reduce aliasing effects
- Simple solution: Multisampling
- Only the last step changes!
- During scan conversion,
  sample subpixels and average



- This is equivalent to rendering a larger image
- Observation: Rendering twice larger resolution costs less then rendering twice – since scanline is efficient and the rest doesn't change!

Rasterization – Shadow Maps

- Render an image from the light's point of view (the light is the camera)] **Shadow map**
- Keep "depth" from light of every pixel in the map
- During image render:
  Calculate position and depth on the **shadow map** for each **pixel** in the **final image** (not vertex!)
- If **pixel depth** > **shadow map depth** the pixel will not receive light from this source

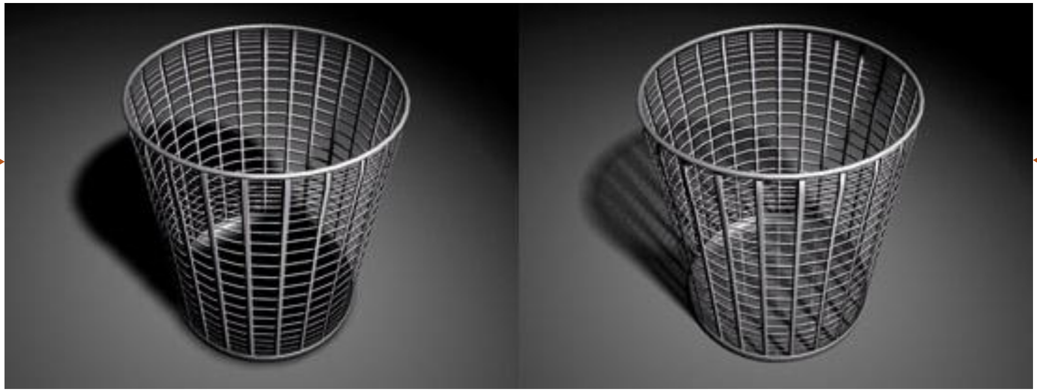


Rasterization – Shadow Maps

- This solution is not optimal
- Shadow map resolution is not correlated to render resolution – one shadow map pixel can span a lot of rendered pixels!
- Shadow aliasing
- Only allows sharp shadows
- Semi-transparent objects

Various hacks and complex solutions

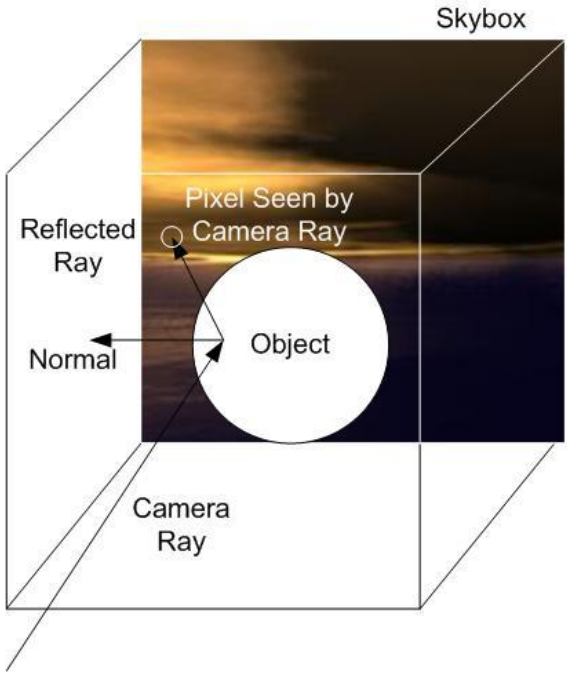
Blurred hard shadows (shadow map)



True soft shadows (ray tracing)

Rasterization – Reflection Maps

- Not a true reflection – a “cheat”
- **Precalculate** reflection map from a point in the center (can be replaced by an existing image)
- The reflection map is mapped to a sphere or cube surrounding the scene
- Each direction (vector) is mapped to a specific color according to where it hits the sphere / cube
- During render, find the reflection color according to the **reflection vector** of each pixel (not vertex!)



Rasterization – Reflection Maps

- Can produce fake reflections (no geometry needed)
- Works well for:
  - Environment reflection (landscape, outdoors, big halls)
  - Distorted reflections
  - Weak reflections (wood, plastic)
  - Static scenes
- Not so good for:
  - Reflections of near objects
  - Moving scenes
  - Mirror like objects
  - Optical effects

Rasterization – Reflection Maps

- Examples: Reflection maps

Used to create the map

Rasterization – Reflection Maps

- Examples: Ray traced reflections

Rasterization – Reflection Maps

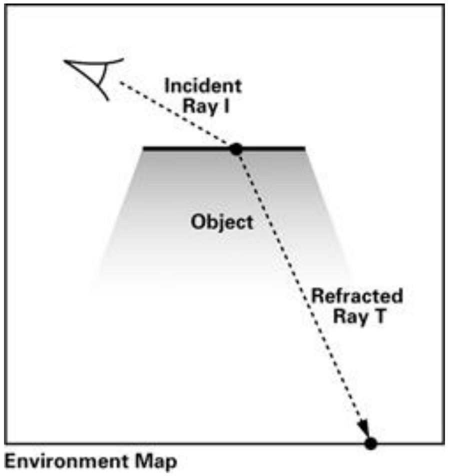
- Examples:

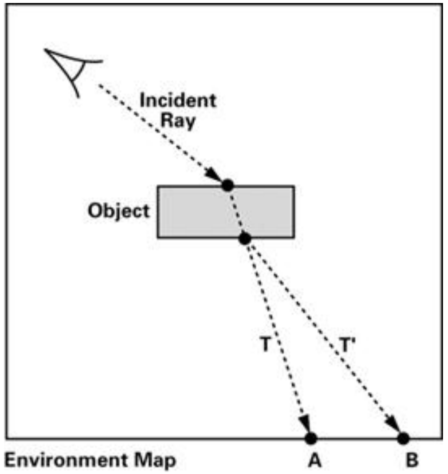
Reflection Map

Ray Traced Reflection

Rasterization – Refractions

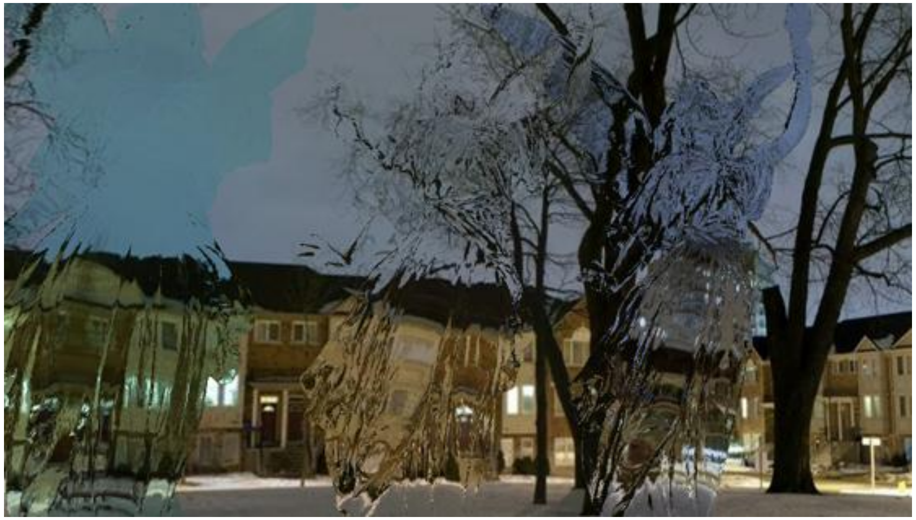
- There is no real solution
- Refraction maps: same as reflection maps but the angle is computed using **refractive index**
- Only simulates the first direction change, not the second (that would require ray tracing)
- Refraction is complex so fake refractions are hard to notice
- Doesn't consider near objects, only static background



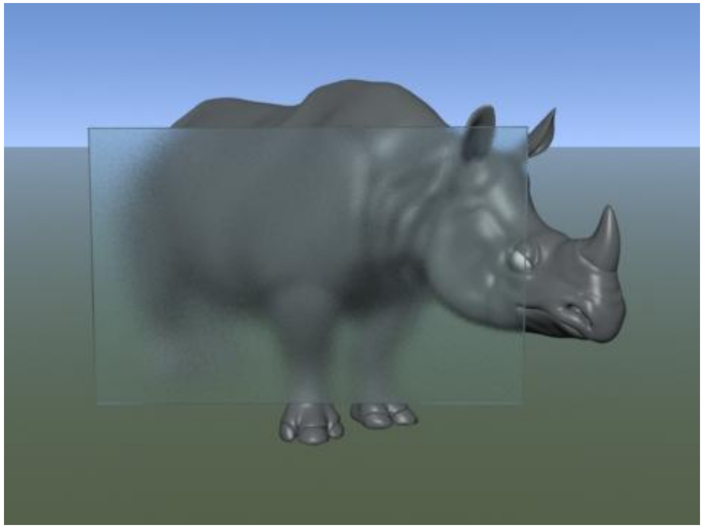


Rasterization – Refractions

- Other "fake" solutions:
- Distort the background according to a precomputed map
- "Bake" ray traced refractions into a texture file (for static scenes)



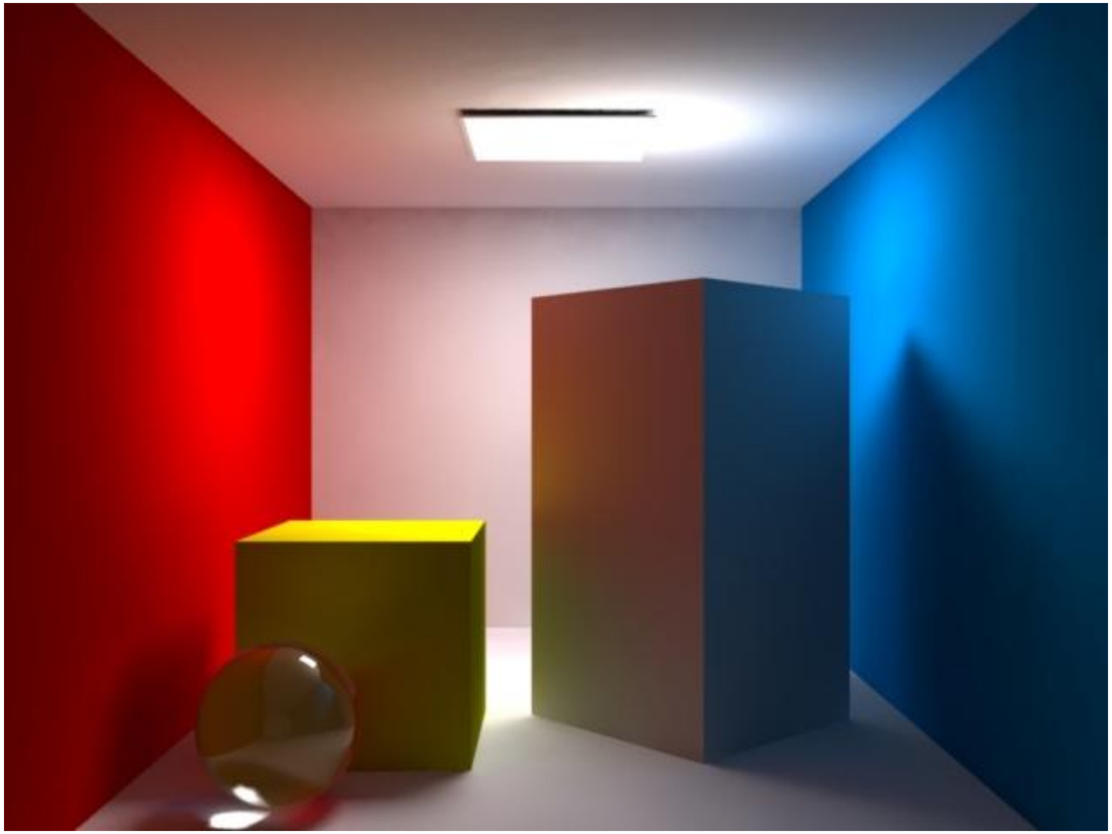
Refraction Map



Distort Background

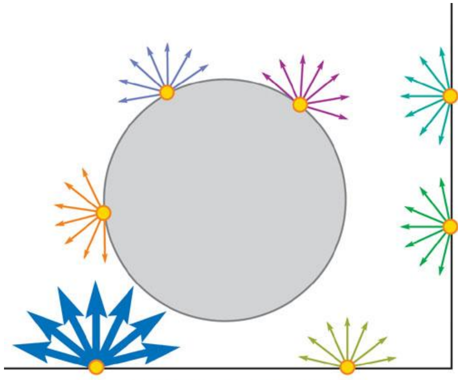
Rasterization – Indirect Illumination

- Indirect / global illumination means taking into account light bouncing off other objects in the scene



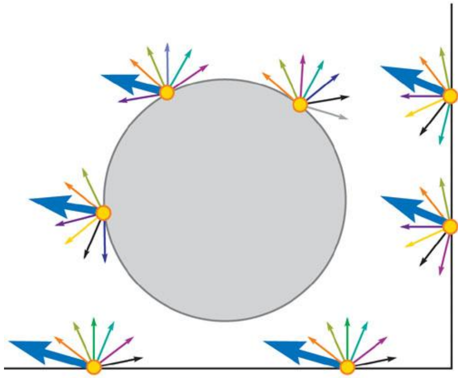
Rasterization – Indirect Illumination

- Surprisingly, there are methods to approximate global illumination using only rasterization, **without** ray tracing
- **"High-Quality Global Illumination Rendering Using Rasterization"**, *Toshiya Hachisuka, The University of Tokyo*
- Main idea: Use a lot of fast rasterized "renders" from different angles to compute indirect illumination at each point
- Rasterization is super quick on GPU



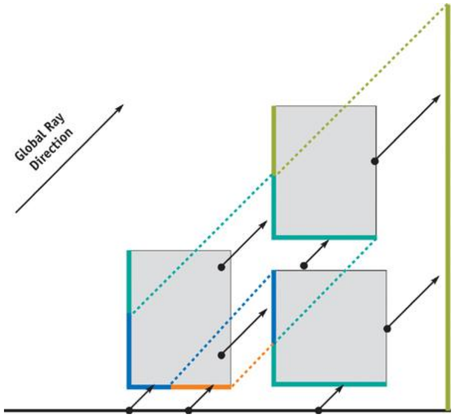
Set of Rays in Conventional Final Gathering

(a)



Set of Rays in Our Method

(b)



Rasterization – Indirect Illumination

- Results:

Results of equal render time

Photon mapping (ray tracing)

Rasterizer (GPU)

TRANSFORMATIONS

22 Mar. 2012

Introduction to Rendering Techniques

Transformations

- We saw 2 types of transformations
- **Viewing transformation**: Can move, rotate and scale the object but does not skew or distort objects
- **Perspective projection**: This special transformation projects the 3D space onto the image plane
- How do we represent such transformations?
- Homogeneous coordinates: Adding a 4th dimension to the 3D space

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Viewing Transformations

- Types of transformations

Scale

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Translate (move)

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & x_0 \\ 0 & 1 & 0 & y_0 \\ 0 & 0 & 1 & z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Rotations

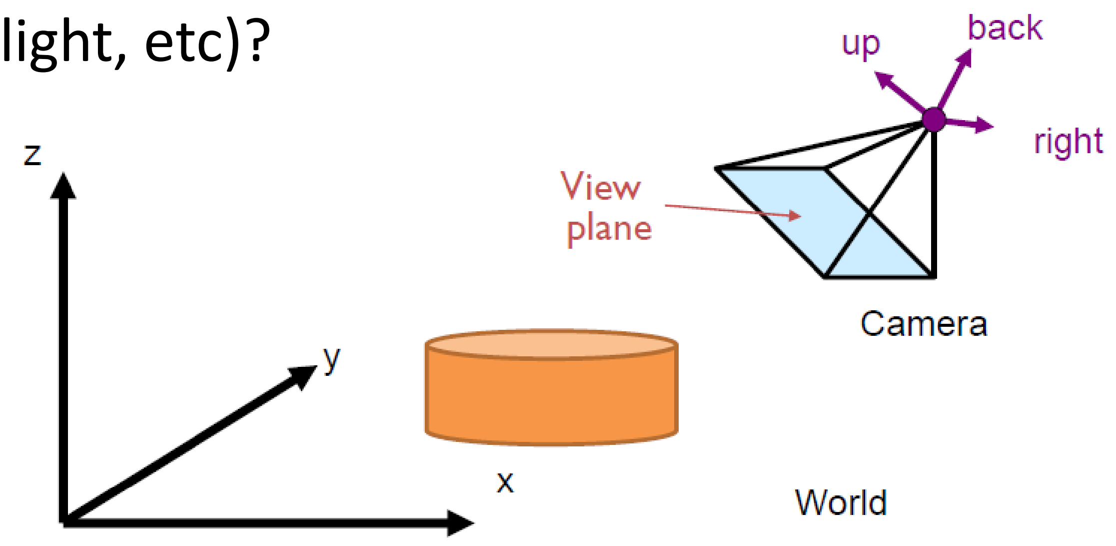
$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

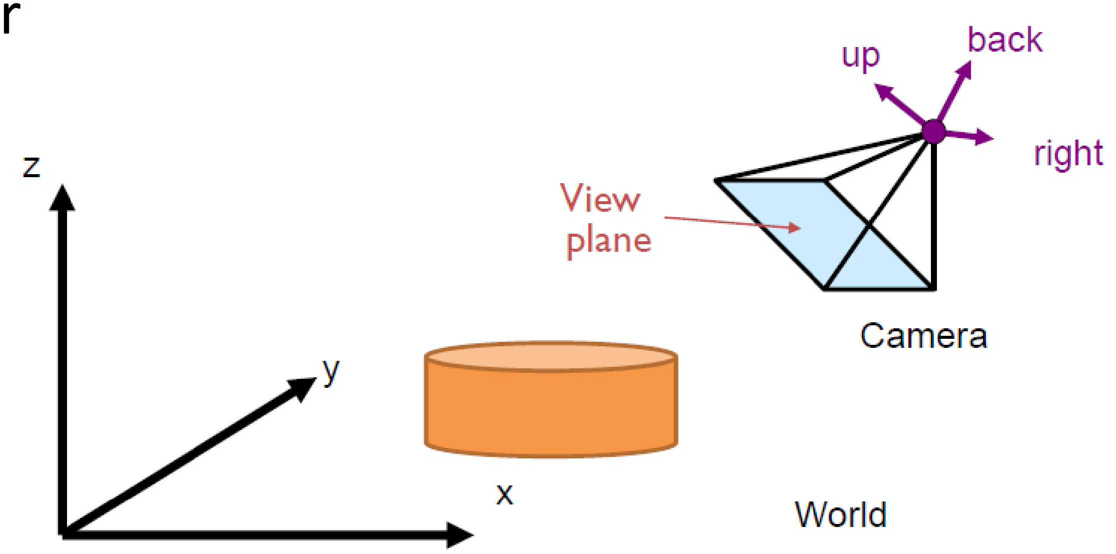
Viewing Transformations

- Any combination of these matrices is a viewing transformation matrix
- Last coordinate is only for moving the pivot, w' is always 1 and will not be used
- How to find the transformation to a certain view (could be camera, light, etc)?



Viewing Transformations

- After the transformation:
- Eye position should be at (0, 0, 0)
- X axis = right vector
- Y axis = up vector
- Z axis = back vector



Viewing Transformations

- It is easy to construct the invert transformation, from camera coordinates to world

Right Vector, Up Vector, Back Vector, Eye Position

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} R_x & U_x & B_x & E_x \\ R_y & U_x & B_y & E_y \\ R_z & U_z & B_z & E_z \\ R_w & U_w & B_w & E_w \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

Viewing Transformations

- Examples:

(0, 0, 0) -> Eye Position

$$\begin{bmatrix} E_x \\ E_y \\ E_z \\ E_w \end{bmatrix} = \begin{bmatrix} R_x & U_x & B_x & E_x \\ R_y & U_x & B_y & E_y \\ R_z & U_z & B_z & E_z \\ R_w & U_w & B_w & E_w \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Camera X Axis -> Origin + Right vector

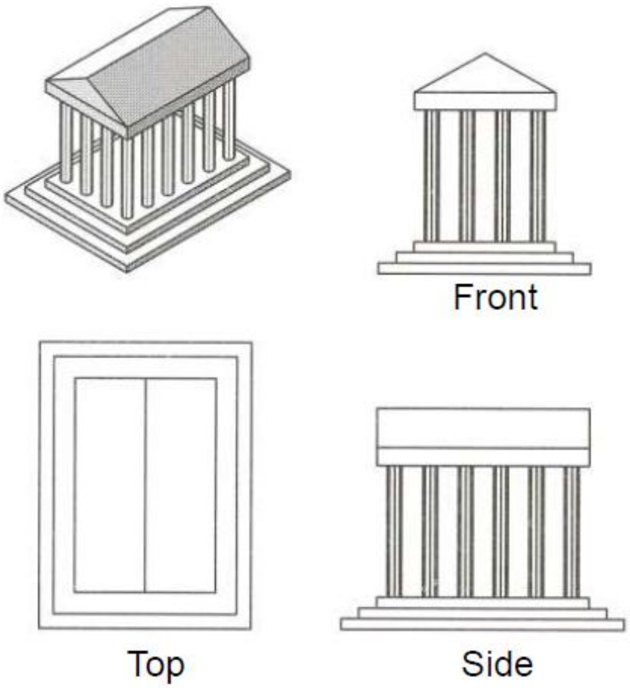
$$\begin{bmatrix} R_x + E_x \\ R_y + E_y \\ R_z + E_z \\ R_w + E_w \end{bmatrix} = \begin{bmatrix} R_x & U_x & B_x & E_x \\ R_y & U_x & B_y & E_y \\ R_z & U_z & B_z & E_z \\ R_w & U_w & B_w & E_w \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

- Now all we have to do is invert T (always invertible), and we have our view transformation

Projections

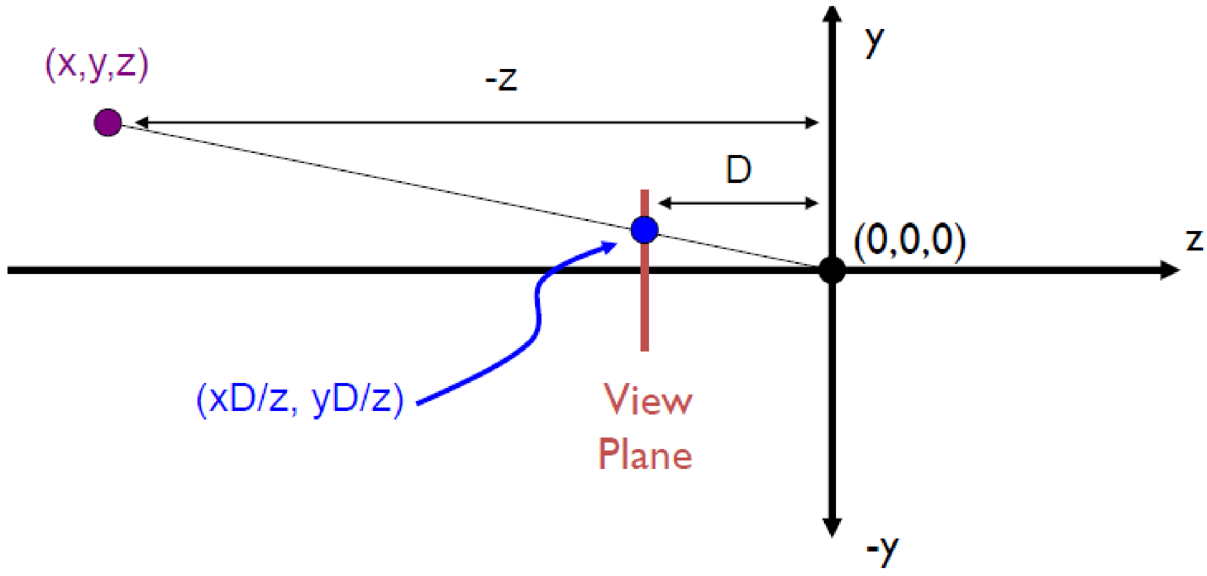
- A projection transform points from higher dimension to a lower dimension, in this case 3D -> 2D
- The most simple projection is **orthographic**
- Simply remove the Z axis after the viewing transformation

$$\begin{bmatrix} x_p \\ y_p \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} x_v \\ y_v \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix}$$



Perspective Projections

- **Perspective projections** map points onto the **view plane** toward the **center of projection** (the viewer)
- Since the viewer is at (0, 0, 0) the math is very simple
- D is called the **focal length**
- $x' = x*(D/z)$
- $y' = y*(D/z)$



Perspective Projections

- Matrix form of the perspective projection using homogeneous coordinates

$$\begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} dx & dy & dz & z \end{bmatrix} \Rightarrow \begin{bmatrix} \frac{d}{z}x & \frac{d}{z}y & \mathrm{d} \end{bmatrix}$$

Divide by 4th coordinate

(the “w” coordinate)

- Singular matrix – projection is many to one
- D = infinity gives an orthographic projection
- Points on the viewing plane z = D do not move
- Points at z = 0 are not allowed – usually by using a clipping plane at z = ε

LIGHTING

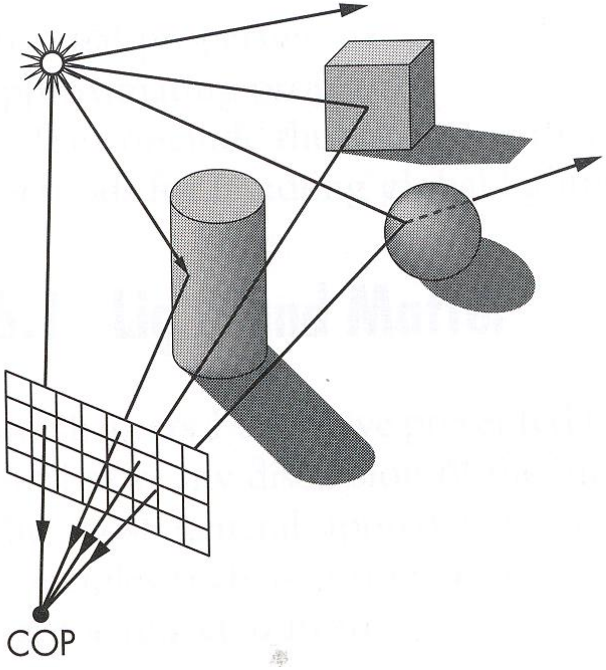
RAY TRACING

22 Mar. 2012

Introduction to Rendering Techniques

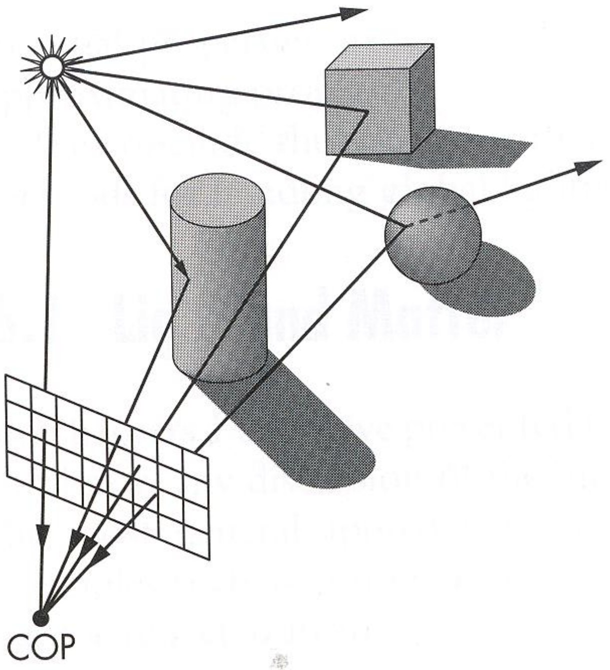
Ray Tracing

- Basic idea: Shoot a “visibility ray” from center of projection (camera) through each pixel in the image and find out where it hits
- This is actually backward tracing – instead of tracing rays **from** the light source, we trace the rays from the viewer back **to** the light source



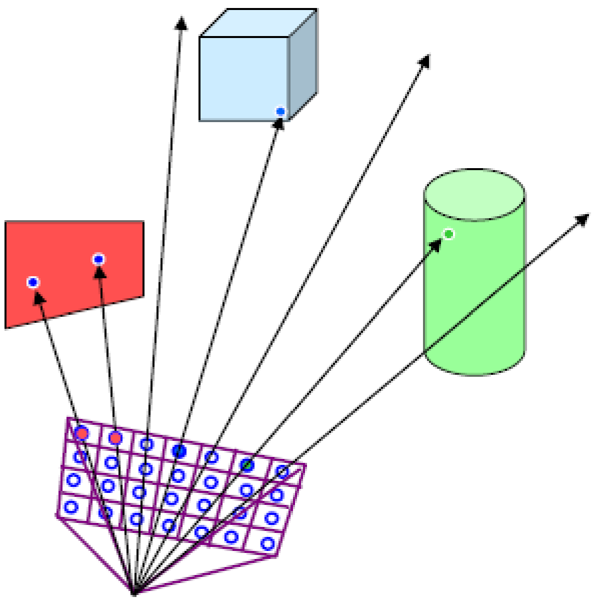
Ray Tracing

- Backward tracing is called **Ray Casting**
- Simple to implement
- For each ray find intersections with every polygon – slow…
- Easy to implement realistic lighting, shadows, reflections and refractions, and indirect illumination



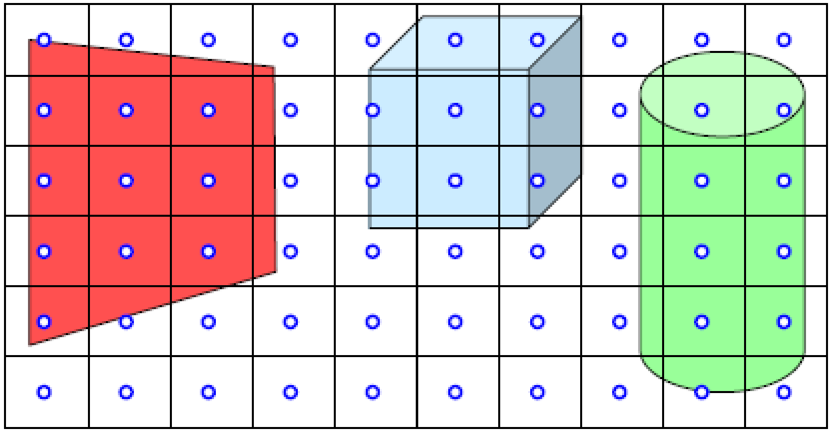
Ray Tracing

- For each sample (pixel or subpixel):
- Construct a **ray** from eye position through viewing plane



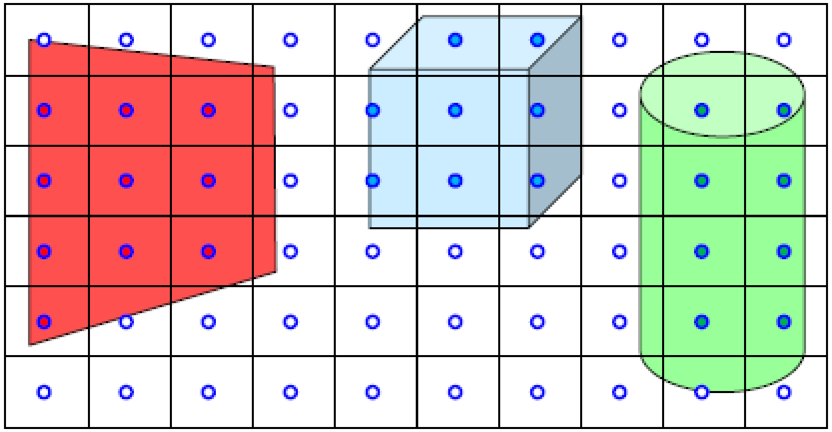
Ray Tracing

- For each sample (pixel or subpixel):
- Construct a **ray** from eye position through viewing plane
- Find first (closest) surface that intersects the ray



Ray Tracing

- For each sample (pixel or subpixel):
- Construct a **ray** from eye position through viewing plane
- Find first (closest) surface that intersects the ray
- Compute color based on surface **radiance**

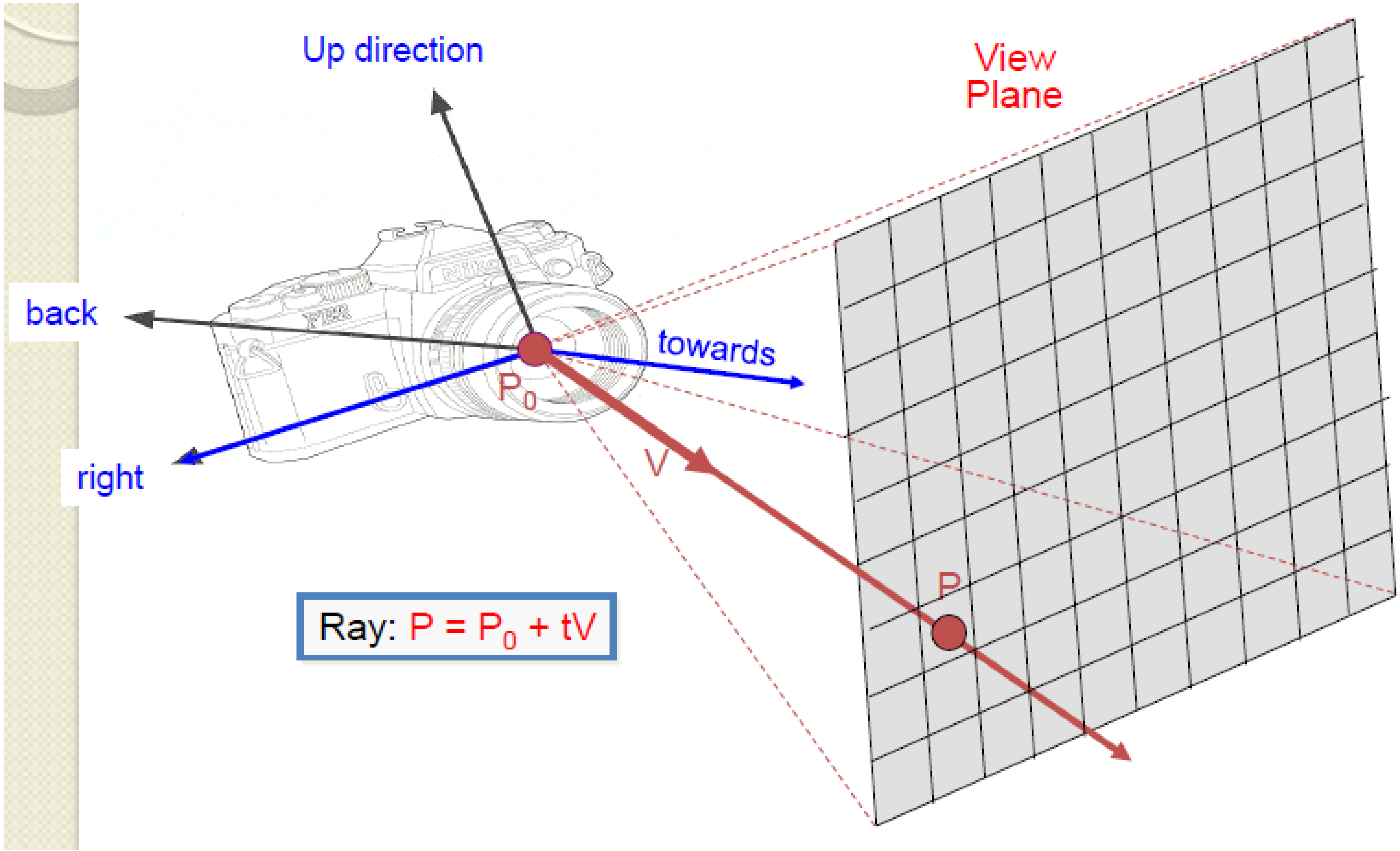


Ray Tracing

- For each sample (pixel or subpixel):
- Construct a **ray** from eye position through viewing plane
- Find first (closest) surface that intersects the ray
- Compute color based on surface **radiance**
- Computing radiance requires casting rays toward the light source, reflected and refracted objects and **recursive illumination rays** from reflected and refracted objects

Ray Tracing – Casting Rays

- Construct a ray through viewing plane:



Ray Tracing – Casting Rays

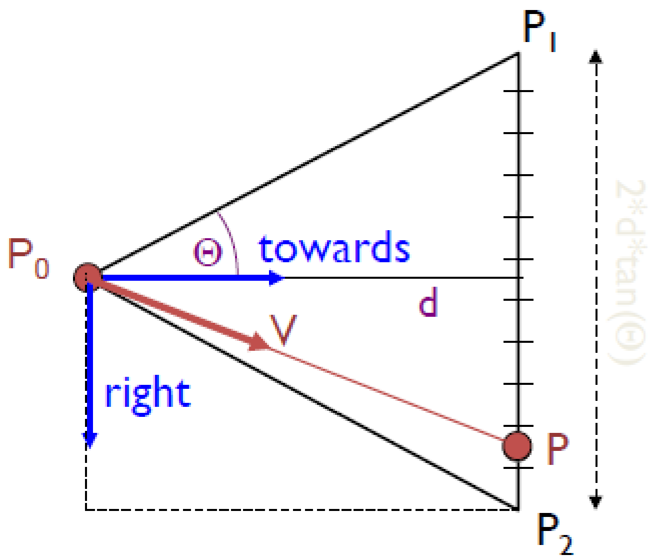
- Construct a ray through viewing plane:
- 2D Example:

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

$P_1 = P_0 + d*towards - d*tan(\Theta)*right$
$P_2 = P_0 + d*towards + d*tan(\Theta)*right$

$P = P_1 + (i/width - 0.5) * 2*d*tan(\Theta)*right$
$V = (P - P_0) / ||P - P_0||$

For every i between (–width/2) and (width/2)



Ray: $P = P_0 + tV$

Ray Tracing - Intersections

- Finding intersections
  - Intersecting spheres
  - Intersecting triangles (polygons)
  - Intersecting other primitives
  - Finding the closest intersection in a group of objects / all scene

Ray Tracing - Intersections

- Finding intersections with a sphere: Algebraic method

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

Substituting for P, we get:

$$|\mathbf{P_0} + \mathbf{tV} - O|^2 - r^2 = 0$$

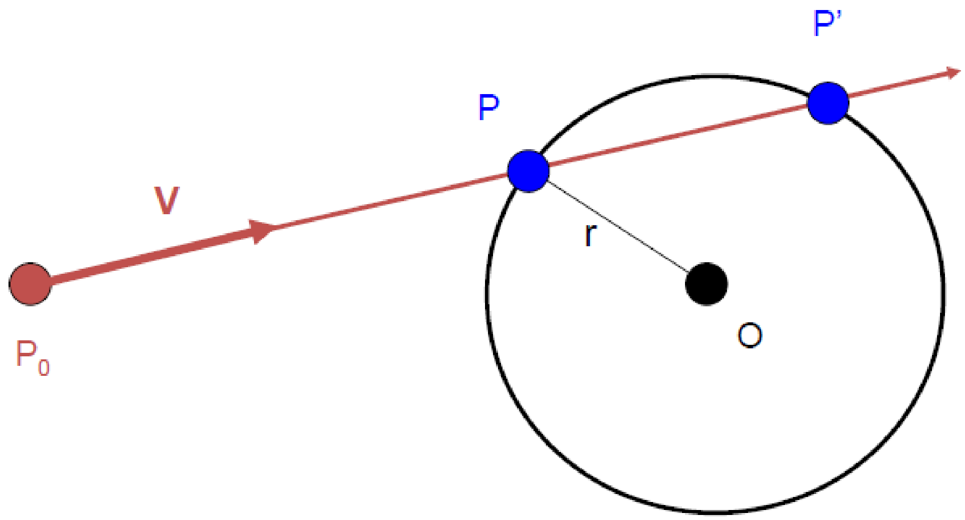
Solve quadratic equation:

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

where:

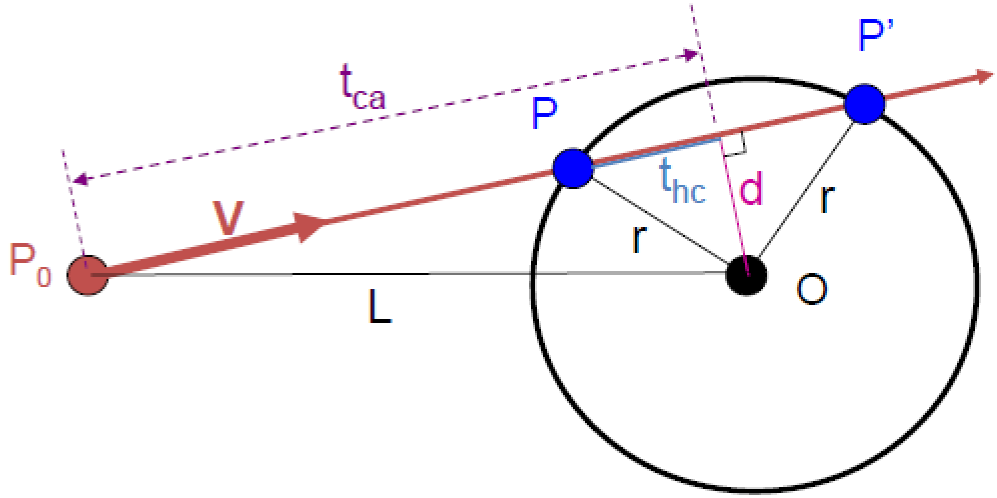
$$a = 1$$
$$b = 2\, V \cdot (P_0 - O)$$
$$c = |P_0 - O|^2 - r^2 = 0$$

Solve for t → $P = P_0 + tV$



Ray Tracing - Intersections

- Finding intersections with a sphere: Geometric method



Ray: $P = P_0 + tV$

Sphere: $|P - O|^2 - r^2 = 0$

$L = O - P_0$

$t_{ca} = L \cdot V$

if ($t_{ca} < 0$) return 0

$d^2 = L \cdot L - t_{ca}^2$

if ($d^2 > r^2$) return 0

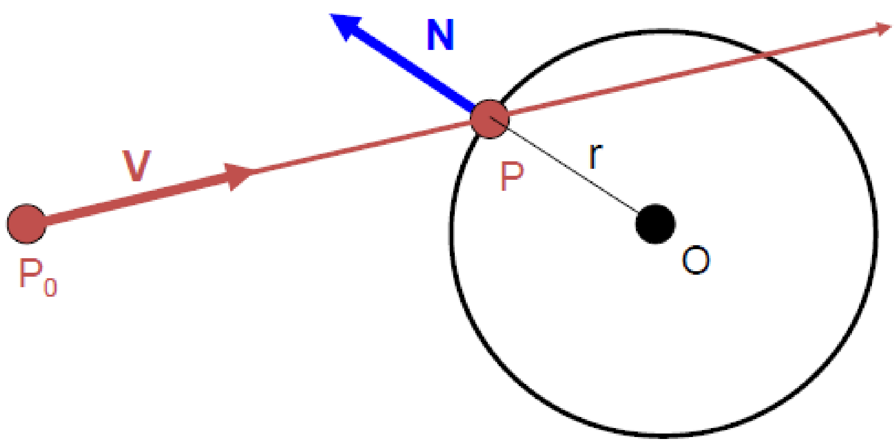
$t_{hc} = \text{sqrt}(r^2 - d^2)$

$t = t_{ca} - t_{hc}$ and $t_{ca} + t_{hc}$ → **Solve for t** → $P = P_0 + tV$

Ray Tracing - Intersections

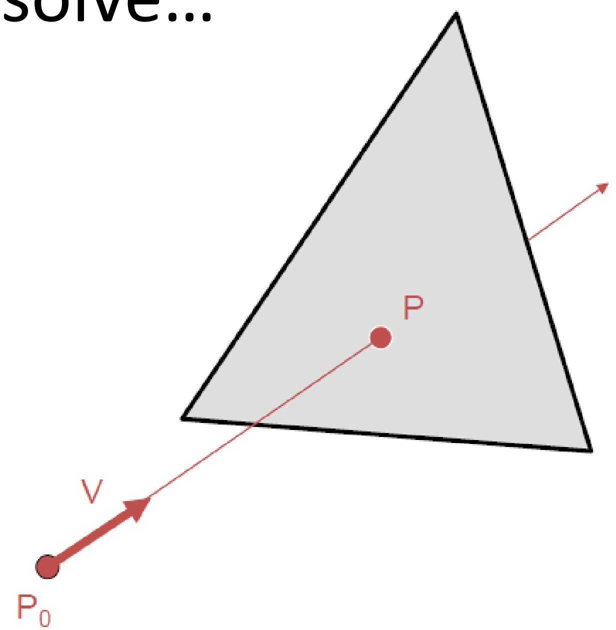
- Finding intersections with a sphere: Calculating normal
- We will need the **normal** to compute lighting, reflection and refractions

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



Ray Tracing - Intersections

- Finding intersections with a triangle:
- Step 1: find intersection with the plane
- Step 2: check if point on plane is inside triangle
- Many ways to solve…



Ray Tracing - Intersections

- Step 1: find intersection with the plane: Algebraic method

Ray: $P = P_0 + tV$

Plane: $N(P-P_0)=0 \rightarrow P \bullet N + c = 0$

Substituting for P, we get:

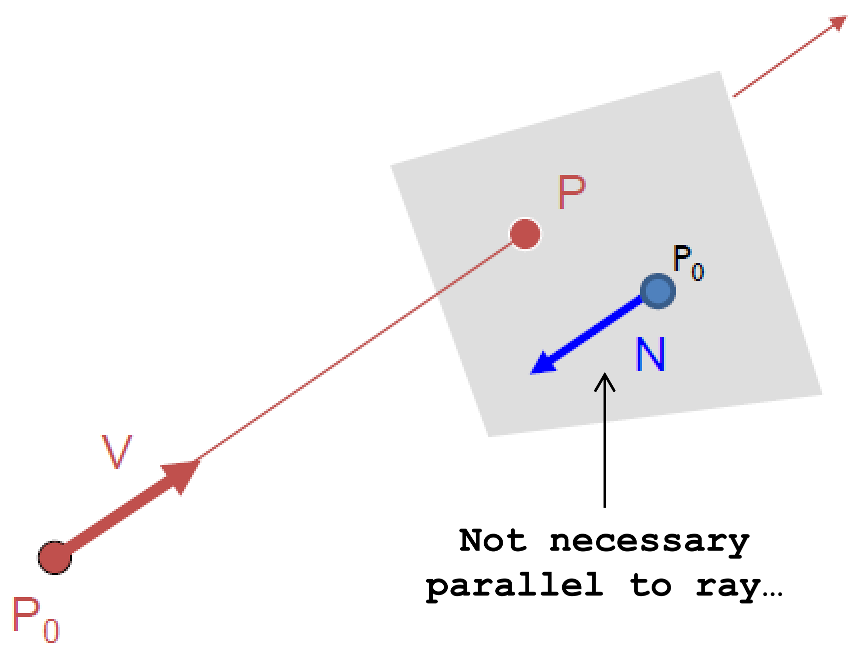
$$(\mathbf{P_0 + tV}) \bullet N + c = 0$$

Solution:

$$t = -(P_0 \bullet N + c) / (V \bullet N)$$

And the intersection at:

$P = P_0 + tV$



Ray Tracing - Intersections

- Step 2: Check if point is inside triangle
  Algebraic method

For each side of triangle

$V_1 = T_1 - P_0$

$V_2 = T_2 - P_0$

$N_1 = V_2 \times V_1$

Normalize N_1

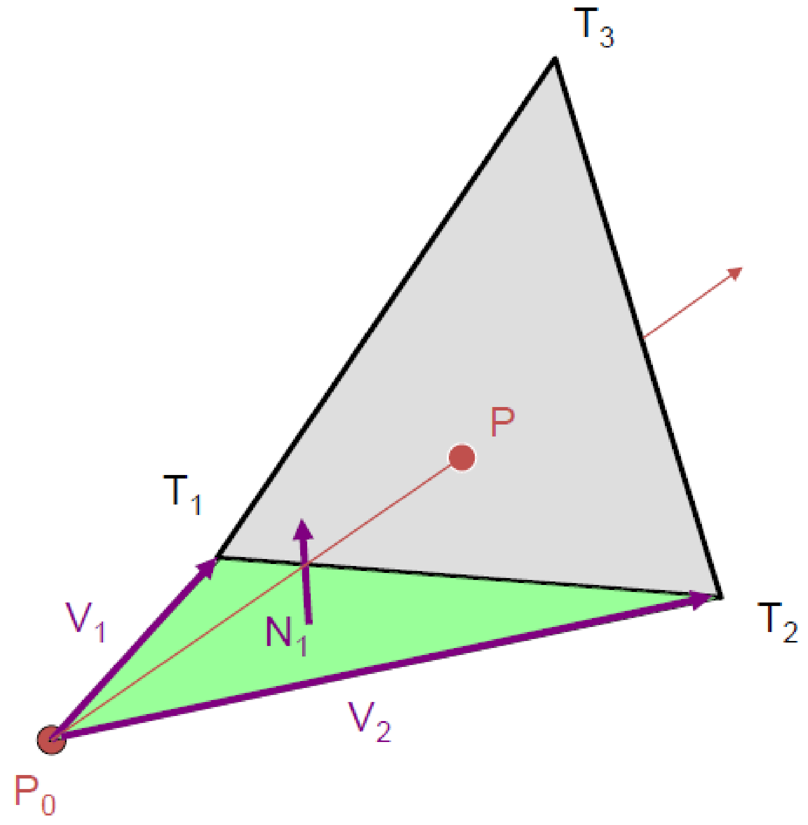
if $(P - P_0) \cdot N_1 < 0$

return FALSE;

end

↓

If all 3 succeed the point is inside the triangle



Ray Tracing - Intersections

- Step 2: Check if point is inside triangle
  Paramteric method

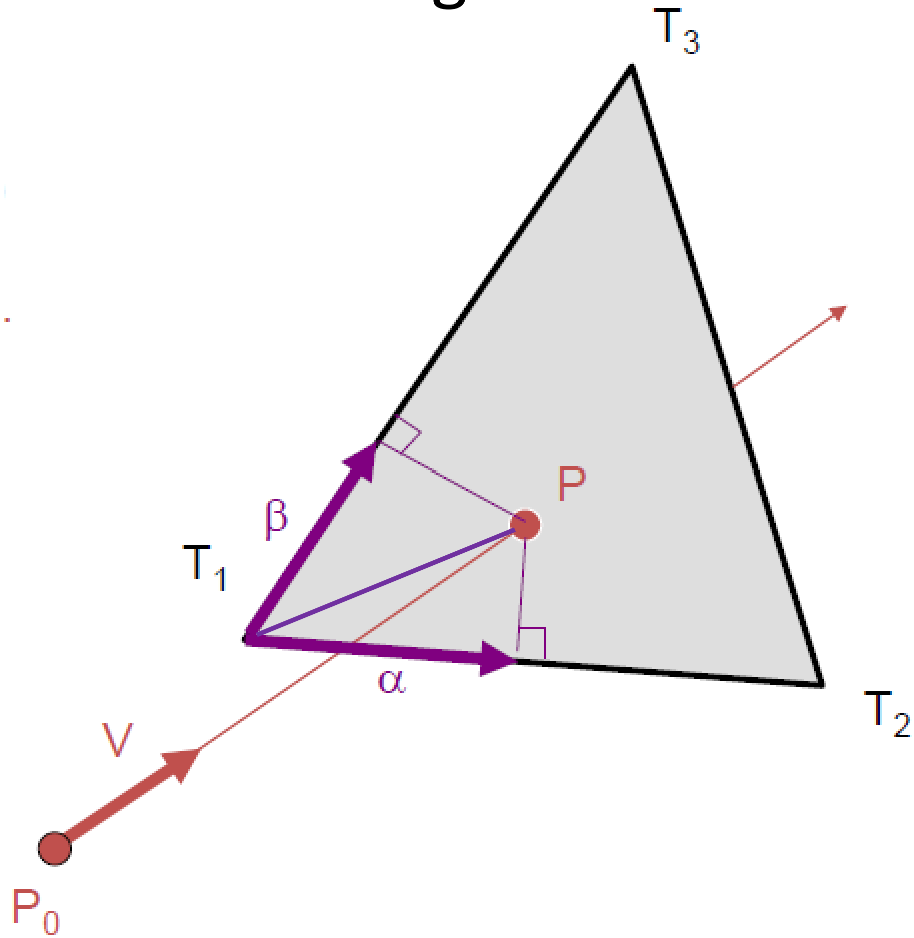
Compute α, β:

$$P = \alpha (T_2 - T_1) + \beta (T_3 - T_1)$$

Using dot products
$(P-T_1) \bullet (T_2-T_1)$ and $(P-T_1) \bullet (T_3-T_1)$

Check if point inside triangle.

$$0 \le \alpha \le 1 \text{ and } 0 \le \beta \le 1$$
$$\alpha + \beta \le 1$$



Ray Tracing - Intersections

- Ray tracing can support other primitives
  - Cone, Cylinder, Ellipsoid: similar to sphere
  - Convex Polygon:
    Point in Polygon is a basic problem in computational geometry and has algebraic solutions
  - Concave Polygon:
    Same plane intersection
    More complex point-in-polygon test
  - Alternatively, divide the polygon to triangles and check each triangle

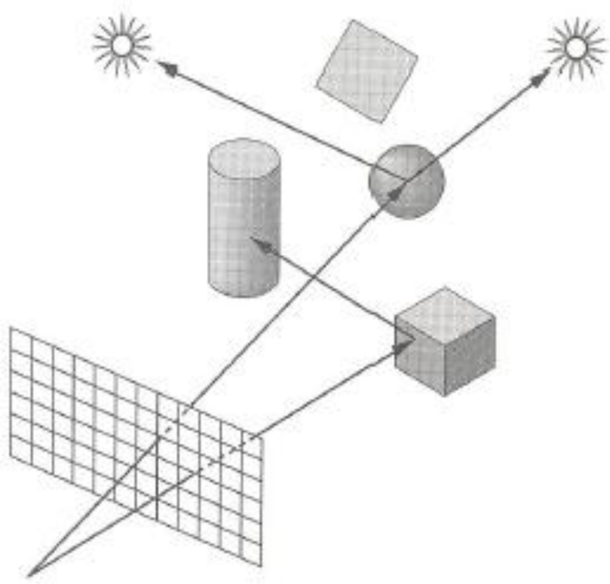
Ray Tracing - Intersections

- Find closest intersection:
- Simple solution is go over each polygon in the scene and test for intersections
- We will see optimizations for this later… (maybe)

- We have an intersection – what now?

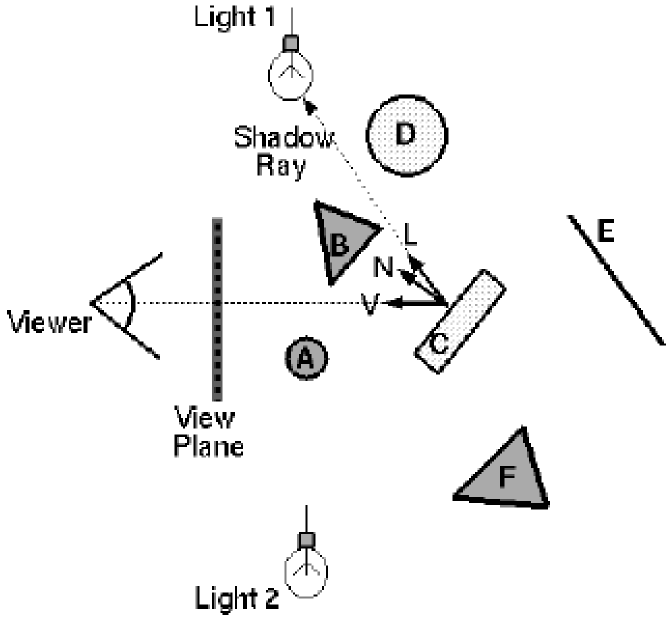
Ray Tracing – Computing Color

- Computing lighting can be similar to the process when rasterizing (using normals)
- This is not for a vertex but for the intersection point
- For better accuracy: **ray trace lighting**
  - At each intersection point cast a ray towards every light source
  - Provides lighting, shadows, reflections, refractions and indirect illumination
  - Easy to compute soft shadows, area lights



Ray Tracing – Shadows

- Shadow term tell which light source are blocked
- S_L = 0 if ray is blocked,
  S_L = 1 otherwise
- Direct illumination is only calculated for unblocked lights



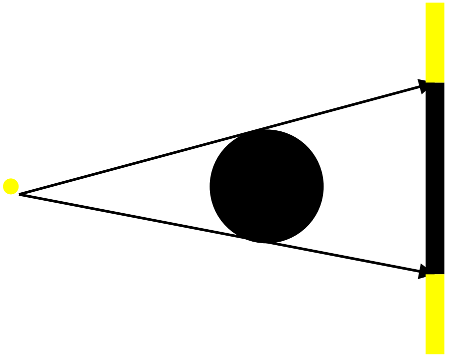
- Illumination formula:

$$I = I_E + K_A I_A + \sum_L (K_D (N \bullet L) + K_S (V \bullet R)^n) S_L I_L$$

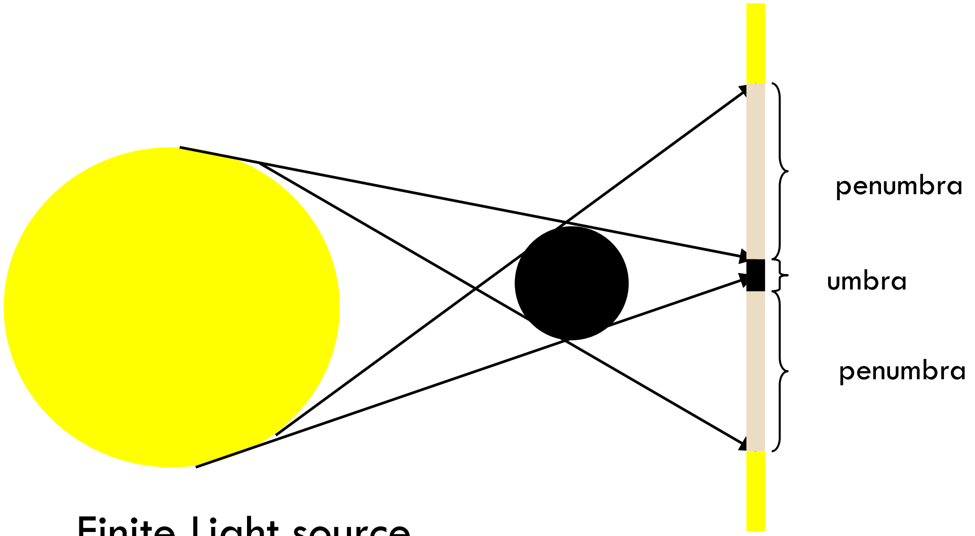
Shadow term

Ray Tracing – Soft Shadows

- Why are real life shadows soft?
- Because light source is not truly a point light



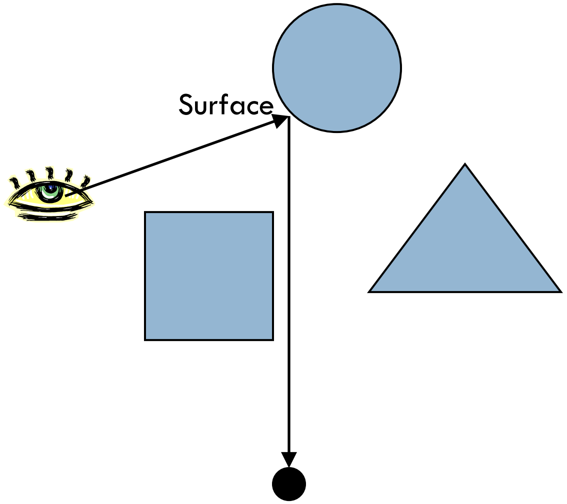
Point Light source



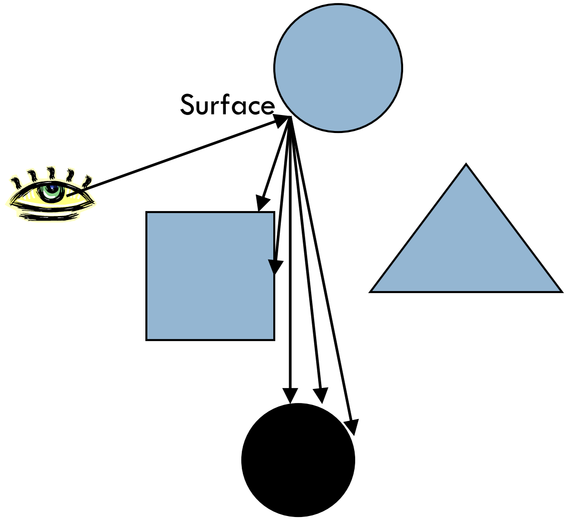
Finite Light source

Ray Tracing – Soft Shadows

- Simulate the area of a light source by casting several (random) rays from the surface to a small distance around the light source



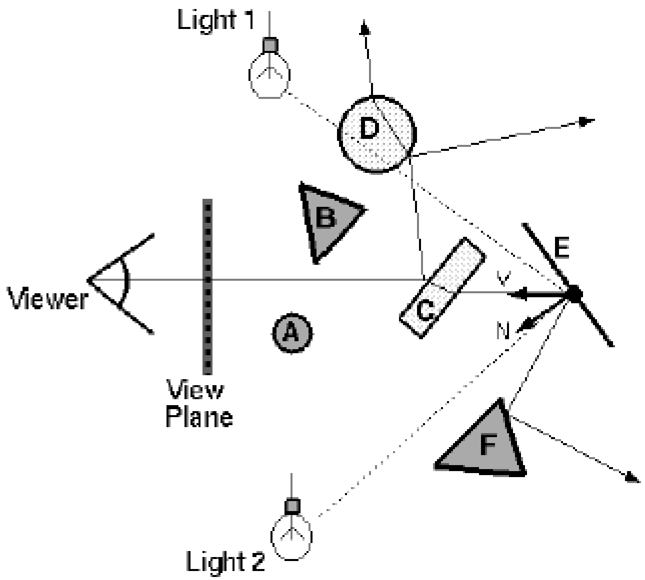
Point light source: The surface is completely lighted by the light source.



Finite light source: 3/5 of the rays reach the light source. The surface is partially lighted.

Ray Tracing – Reflection / Refraction

- Recursive ray tracing: Casting rays for reflections and refractions
- For every point there are **exact directions** to sample reflection and refraction (calculated from normal)

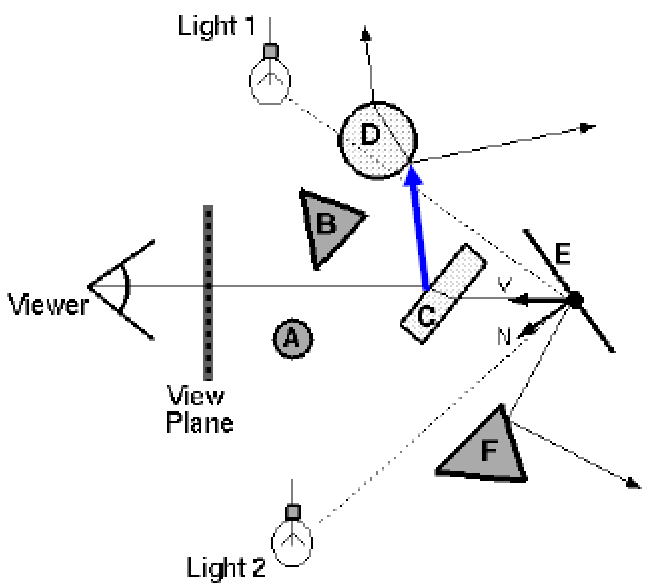


- Illumination formula:

$$I = I_E + K_A I_A + \sum_L (K_D(N \bullet L) + K_S(V \bullet R)^n) S_L I_L + K_S I_R + K_T I_T$$

Ray Tracing – Reflection / Refraction

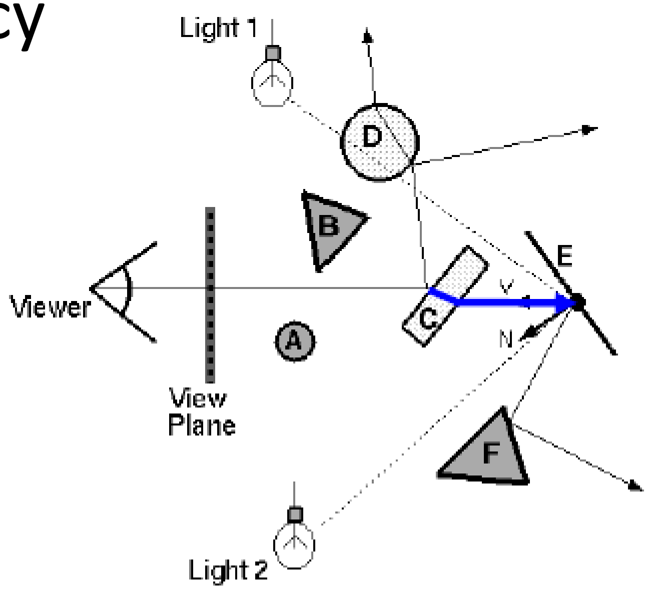
- Cast a reflection ray
- Compute color at the hit point (using ray tracing again!)
- Multiply by **reflection term** of the material
- To avoid aliasing sample **several** rays in the required direction and average



$$I = I_E + K_A I_A + \sum_L (K_D (N \bullet L) + K_S (V \bullet R)^n) S_L I_L + K_S I_R + K_T I_T$$

Ray Tracing – Reflection / Refraction

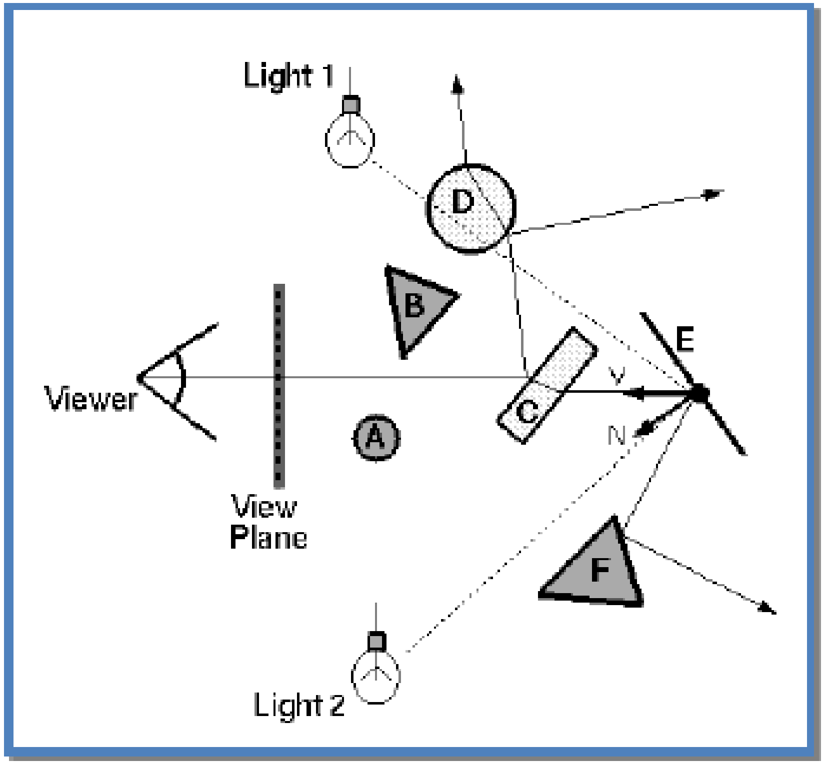
- … And the same for **refractions**
- Last coefficient is transparency
- $K_T = 1$ for **translucent** objects
  $K_T = 0$ for **opaque** objects
- Consider **refractive index** of object
- Again use several rays to avoid aliasing



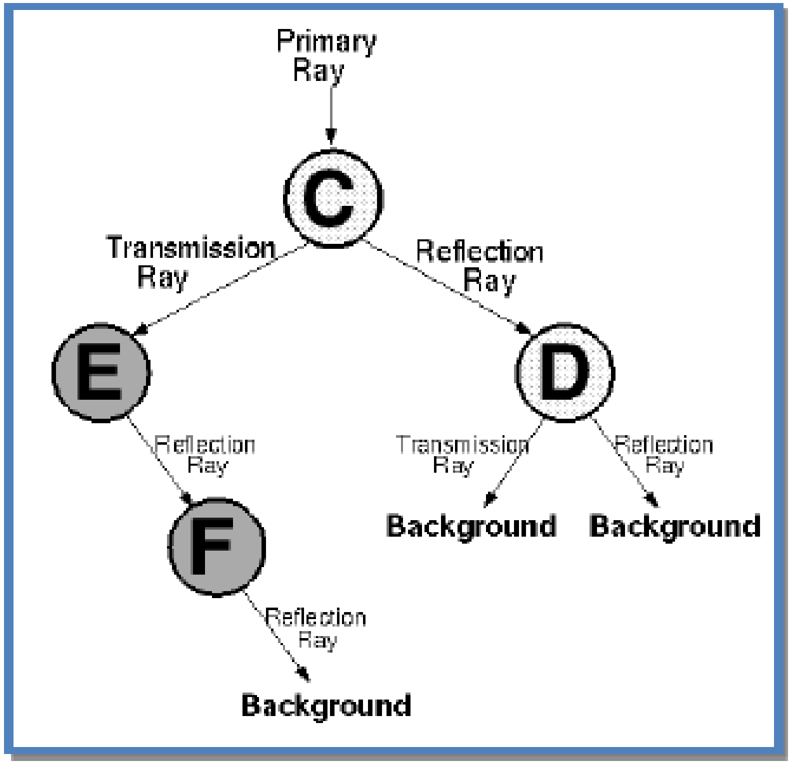
$$I = I_E + K_A I_A + \sum_L (K_D (N \bullet L) + K_S (V \bullet R)^n) S_L I_L + K_S I_R + K_T I_T$$

Ray Tracing – Reflection / Refraction

- Ray tree represents recursive illumination computation



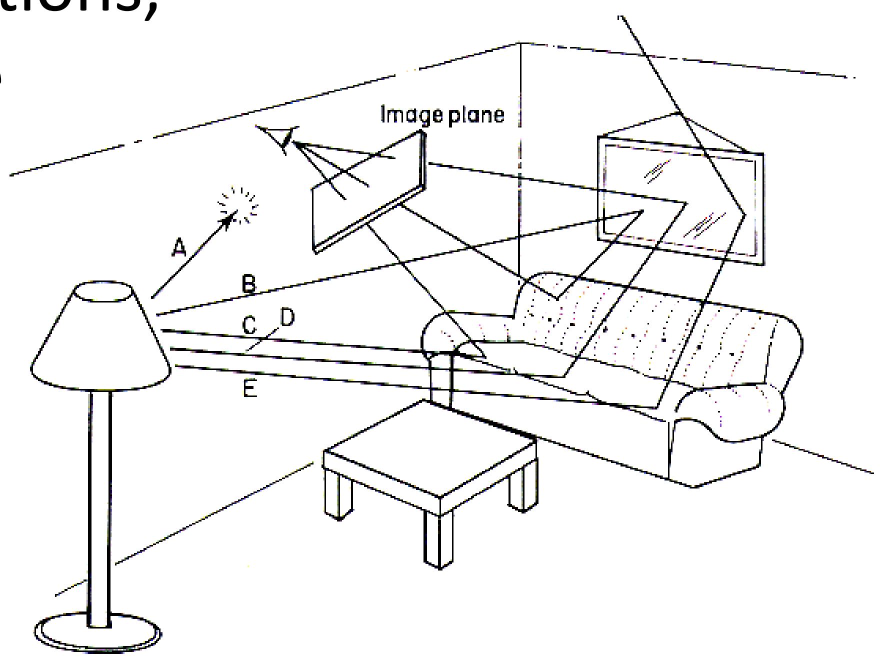
Ray traced through scene



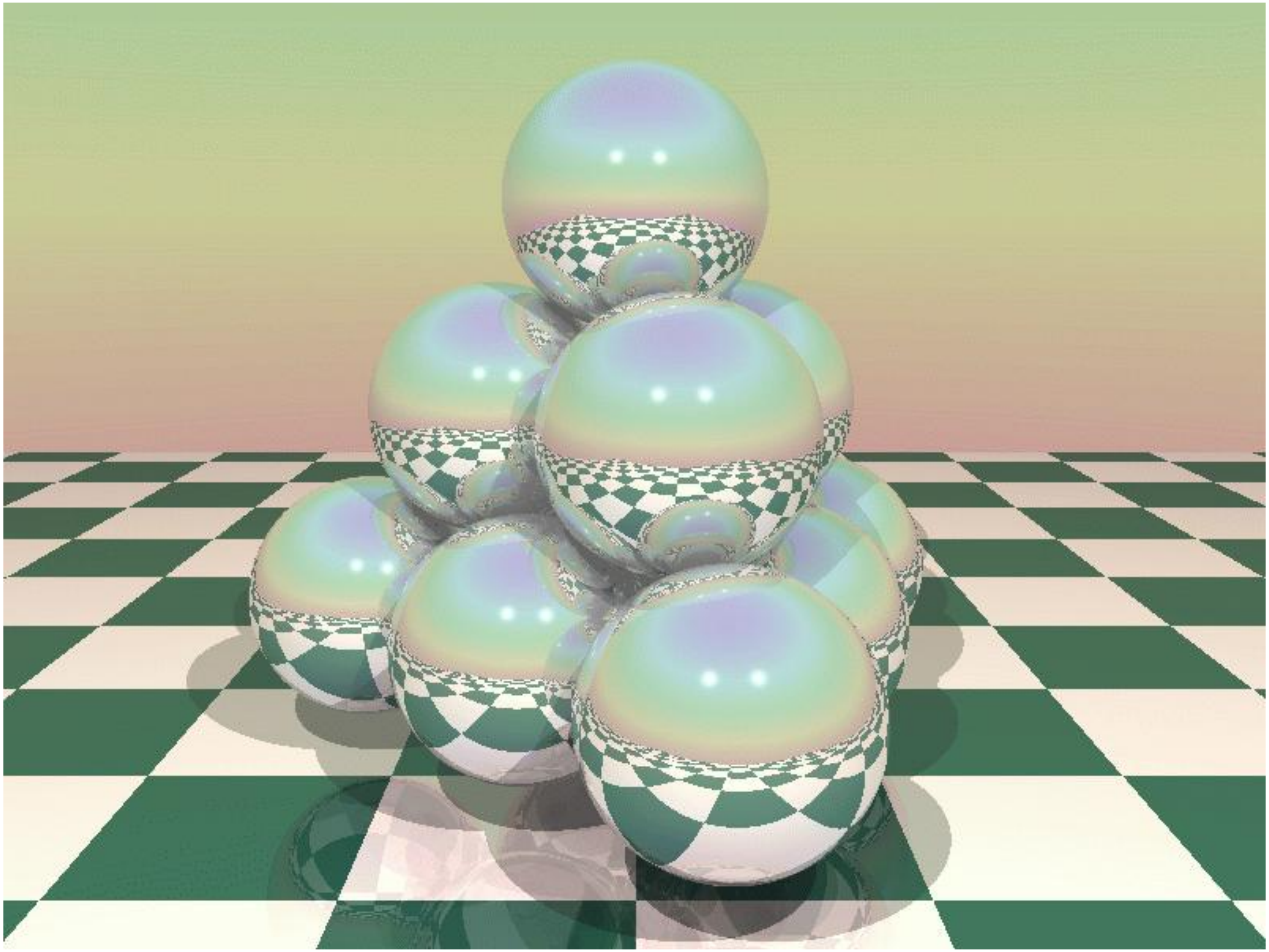
Ray tree

Ray Tracing – Reflection / Refraction

- Number of rays grows exponentially for each level!
- Common practice: limit maximum depth
- After 2-3 bouncing reflections, the cost is high and there is little benefit



Ray Tracing – Antialiasing

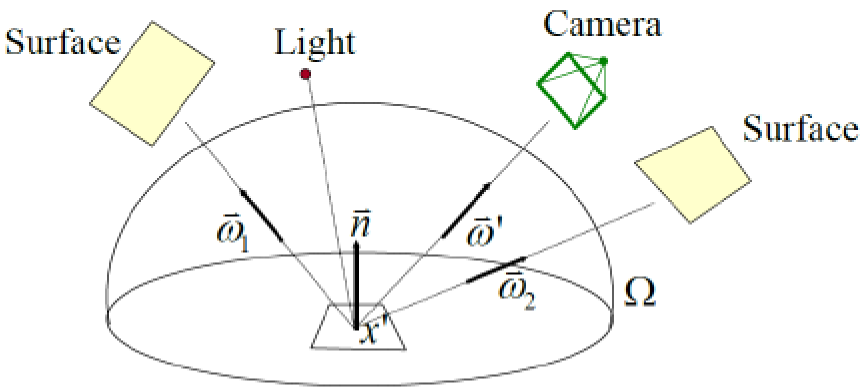


Ray Tracing – Antialiasing

- Aliasing in ray tracing can be severe, since only one ray is casted per pixel
- The computation is based on the size of the pixels, not on the size of the actual polygons which can be relatively small
- Supersampling: Instead of casting one ray per pixel, cast several per pixel
- Since this is done at the first step, it is as inefficient as possible (running the whole process again)

Ray Tracing – Indirect Illumination

- What we've seen so far is only an approximation of real lighting: The rays are only casted **directly** towards the light
- Use reflections, but not indirect lighting
- Global illumination: A method to approximate indirect lighting from every direction



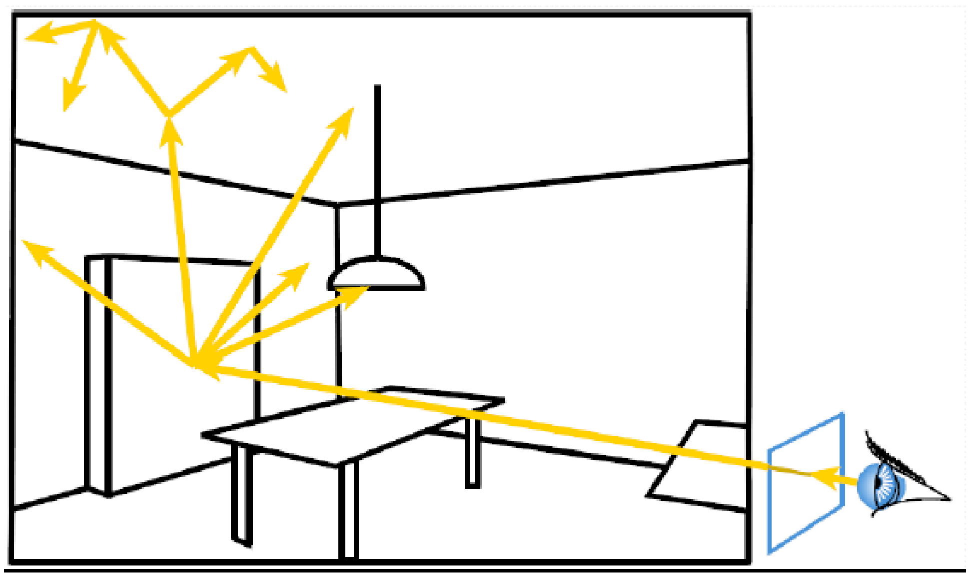
Ray Tracing – Indirect Illumination

- Example:
- Top image uses direct lighting only
- Bottom image uses indirect illumination
- Notice the ground is “reflected” naturally on the character
- Not because of reflective material but because of lighting contribution

Ray Tracing – Indirect Illumination

- Monte-Carlo path tracing
- Step 1: Cast regular rays through each pixel in viewing plane
- Step 2: Cast random rays from visible point
- Step 3: Recurse

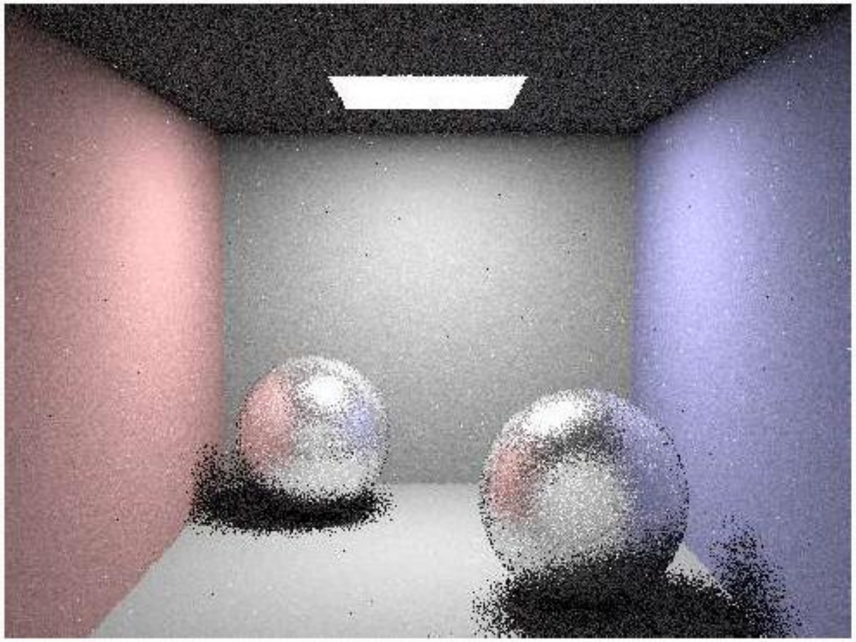
- Very expensive!



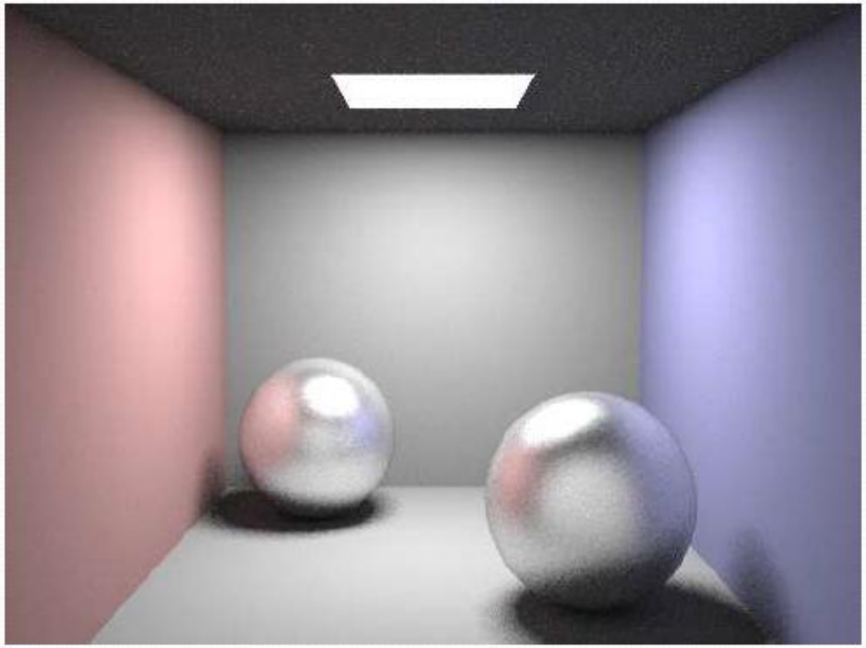
Ray Tracing – Indirect Illumination

- Monte-Carlo path tracing

1 random ray per pixel
no recursion



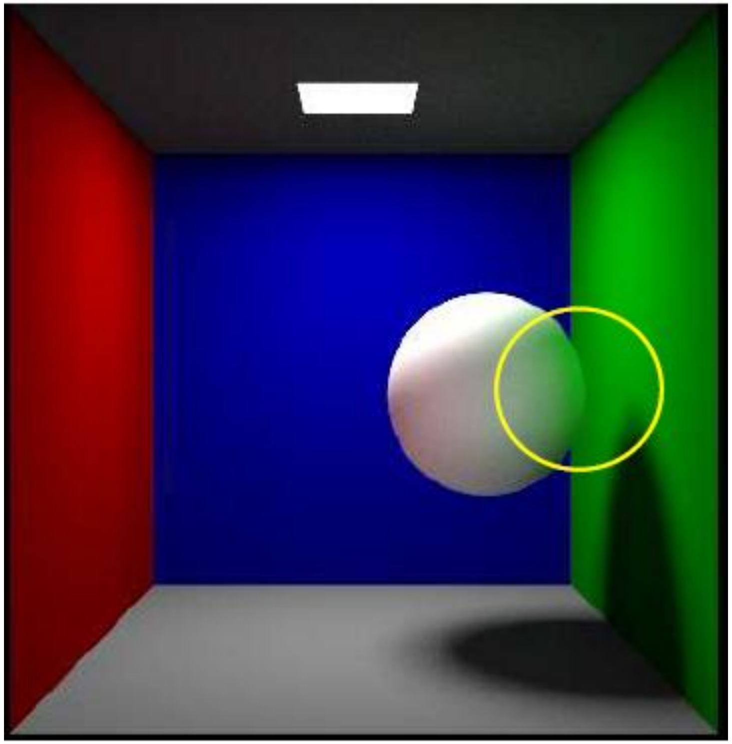
16 random rays per pixel
3 levels of recursion



Ray Tracing – Indirect Illumination

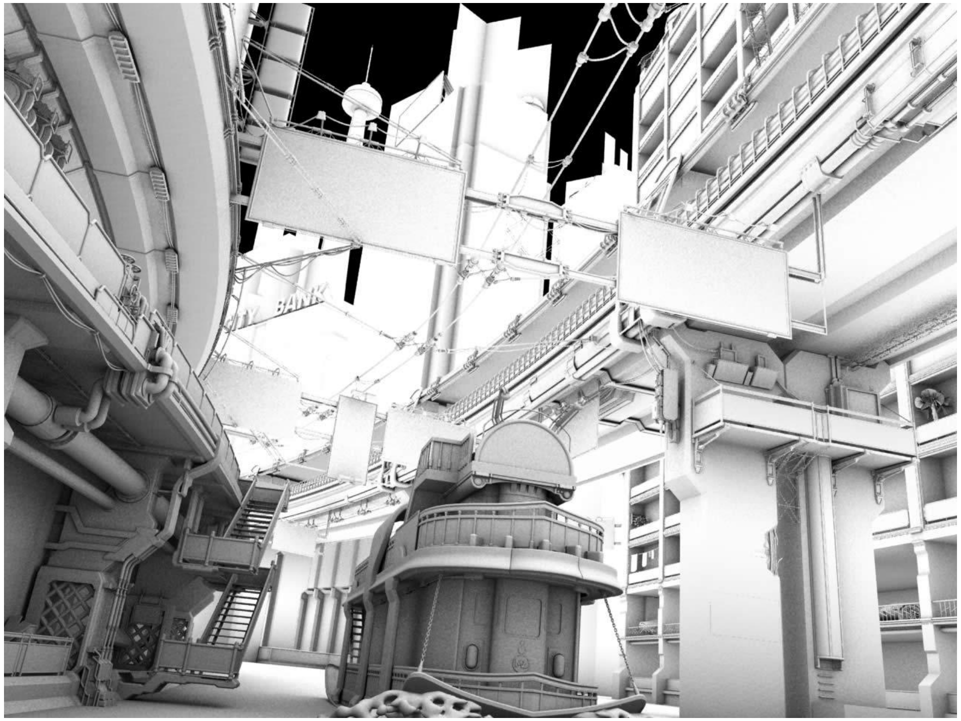
- Monte-Carlo path tracing
- Need a lot of rays and recursions to look good
- Random rays cause flickering problems
- Computation time measured in hours!
- Common practice:
  Bake global illumination map of one frame and use it for all frames

64 random rays per pixel
3 levels of recursion



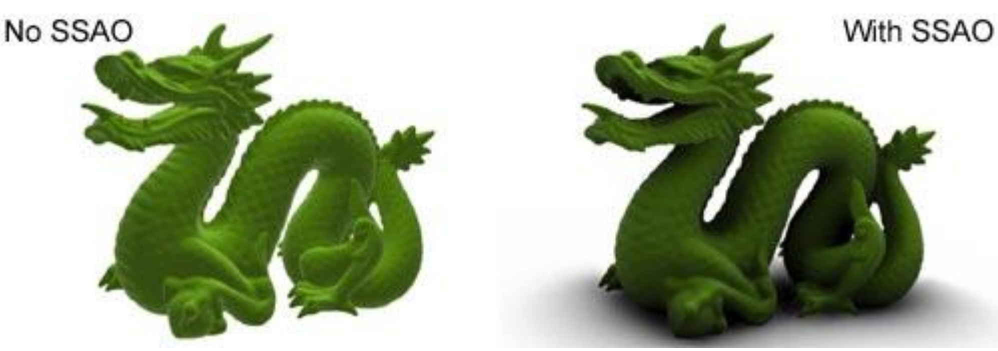
Ray Tracing - Ambient Occlusion

- Ambient Occlusion is a simpler form of global illumination
- Cast random rays from visible point and calculate distance to the nearest object
- The more rays hit near objects, the point is occluded and therefore darker
- A cheat - "make nice" button
- Everything looks **better** with ambient occlusion!



Ray Tracing - Ambient Occlusion

- Good for contact shadows
- Examples:



Summary

Rasterization	Ray Tracing
Fast renderer	Slow renderer - only today we see some real time ray tracing possible
Optimized for GPUs	Not optimized for GPUs
Antialiasing is easy and fast	Antialiasing is expensive
Scales well for larger images	Doesn’t scale so well
Parallel computing possible on GPU	Parallel computing is easy
Shadows are hard to compute and inaccurate	Shadows are easy including sofy shadows
Relections and refractions are a hack	Relections and refractions are easy
Indirect illumination complex but possible (rarely used in practice)	Indirect illumination complex but possible (rarely used in practice)

What Artists Do

- In practice: Both are used side by side
- Games:
  Real time, mostly rasterized except for special effects
- Movies / Animation:
  - Not real time, but time = money
    Usually a mix of rasterization and ray traced reflections / refractions.
  - Global illumination is sometimes used but usually faked using direct lights

What Artists Do

- Common practice: Use render layers and composite later using a video editing program (like After Effects)
- Render layers:
  - Color (radiance)
  - Reflections
  - Refractions
  - Depth map
  - Ambient Occlusion
- Makes it easy to make fast changes later without rendering again

THAT'S ALL, FOLKS!