Lecture 11: Rendering overview

DHBW, Computer Graphics

Lovro Bosnar

8.3.2023.

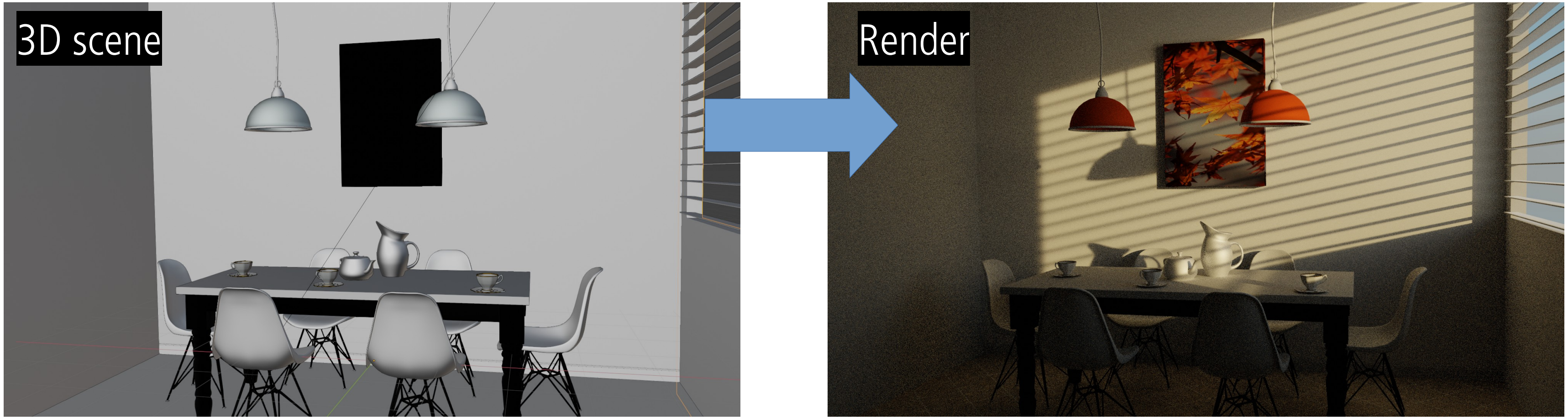
Syllabus

- 3D scene
  - Object
  - Light
  - Camera
- Rendering
- Image and display

- Rendering overview
  -

3D scene and rendering

- We have discussed how to model elements of 3D scene:
  - Objects: shape and material
  - Lights
  - Cameras
- Rendering creates an image from 3D scene
  - Information on 3D scene is used to simulate interaction of light with objects



Rendering

- The goal of rendering is to create viewable 2D image from 3D scene which can be displayed on raster display devices
  - Images are 2D array of pixels
- The task of rendering is to compute color for each pixel of virtual image plane placed in 3D scene
  - Find which objects are visible from pixels or what is covered by pixels
  - Calculate color of visible objects

3d scene, virtual camera, virtual image plane

Rendering: intuition

- Idea of computing pixel colors of virtual image plane is similar to how real camera works
  - Virtual image plane simulates digital camera sensor
- Digital camera sensor is made of array of photo-sensitive cells which convert incoming light into colors
- Light falling on camera is reflected from objects and emitted from light sources
- Therefore, we are only interested into light falling on camera sensor

Camera sensor and incoming light from scene

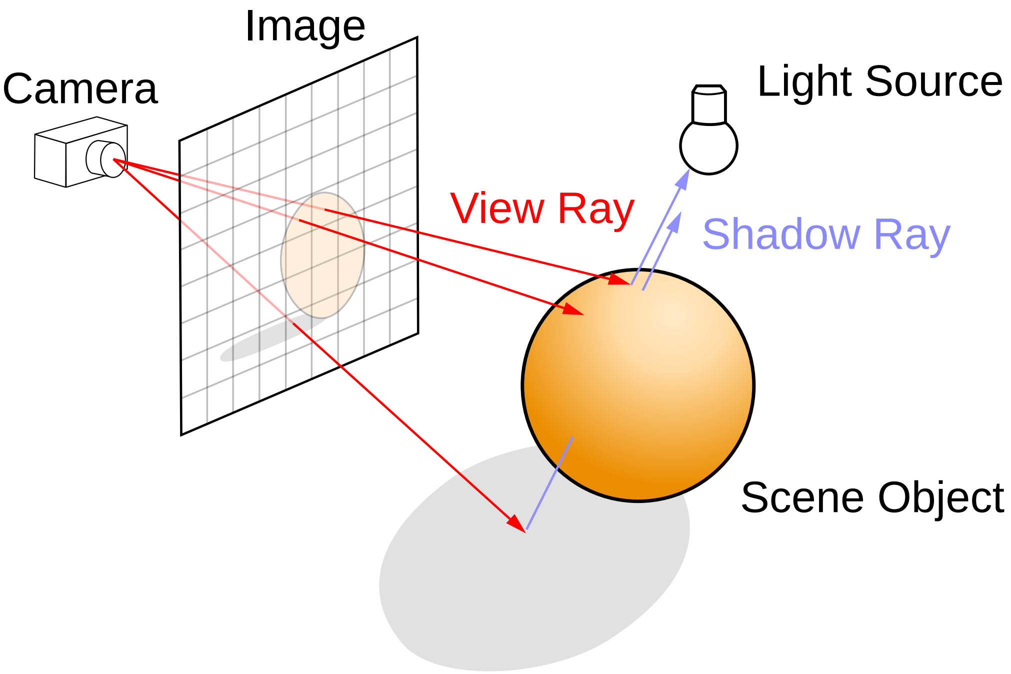
Rendering: physical and perceptual foundations

- Light emission, travel and interaction with objects and camera sensor is well described in physics (wave and geometrical optics)
- Light and image formation can not be only simulated on physical level; we need to **take in account human visual system**
  - Size and shape of objects gets smaller if are more further → **foreshortening effect**
    - Cameras produce images on this principle
    - In rendering we project objects on flat plane (image plane) using **perspective projection**
  - Which objects can we see → **visibility problem**
  - How objects appear → **color** visible objects

Rendering: main steps

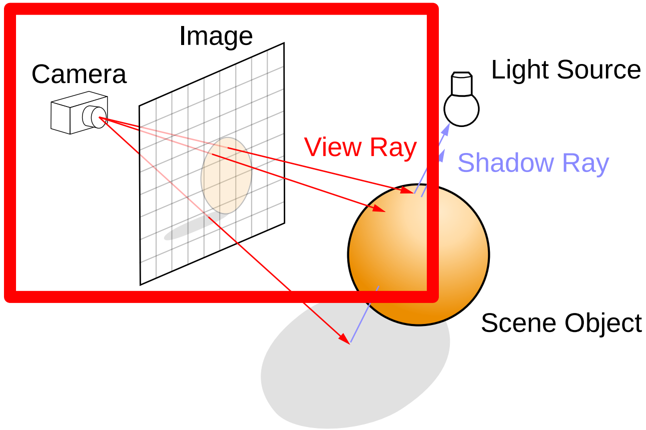
Rendering is solving:

- **Visibility** problem – which objects and surfaces are visible to each other or from camera
  - Rasterization
  - Ray-tracing
- **Shading** problem – how does visible objects and surfaces look like → color



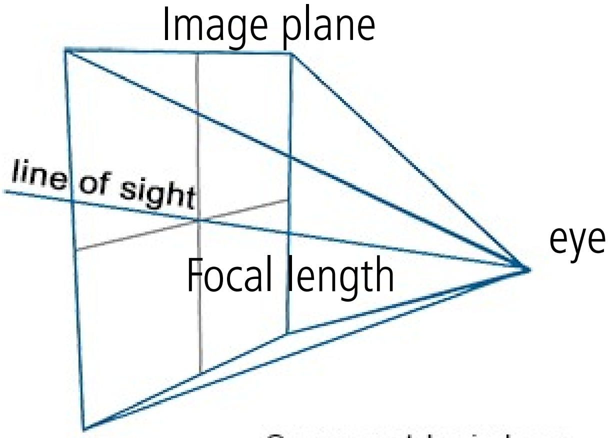
Visibility

- **Visibility determines if two points are visible one to another**
- Often, used to determine which objects are visible to camera
  - Human visual system is simulated → **perspective projection**



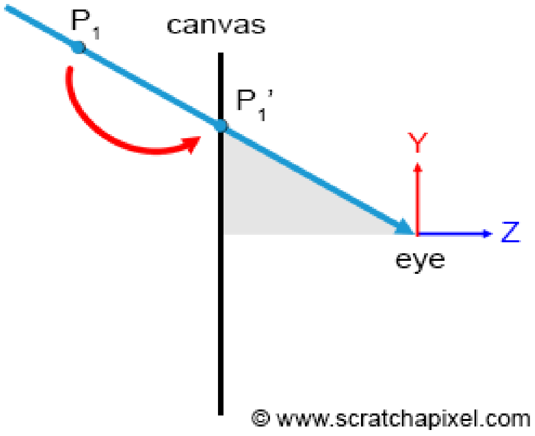
Perspective projection

- **Image is representation of 3D scene on a flat surface**, e.g., image plane
- **Perspective projection** simulates how human visual system forms images
  - **Objects are projected on image plane**
  - **Foreshortening effect**: more distant objects appear smaller
- Projection defines **view frustum**
  - **Objects outside of frustum are not visible**
  - **Shape of frustum depends on image planes size and focal length**
- Different projections are possible, e.g., **orthographic projectio**

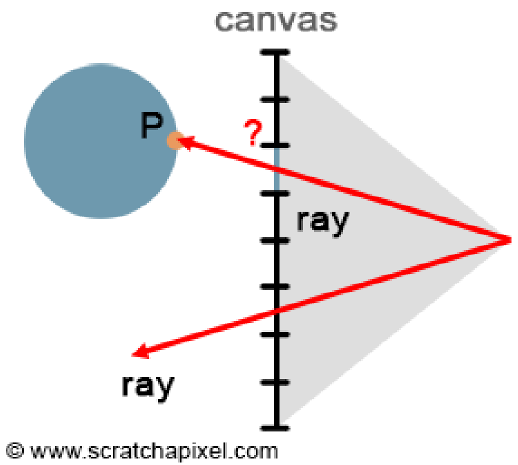


© www.scratchapixel.com

Visibility and projection



© www.scratchapixel.com

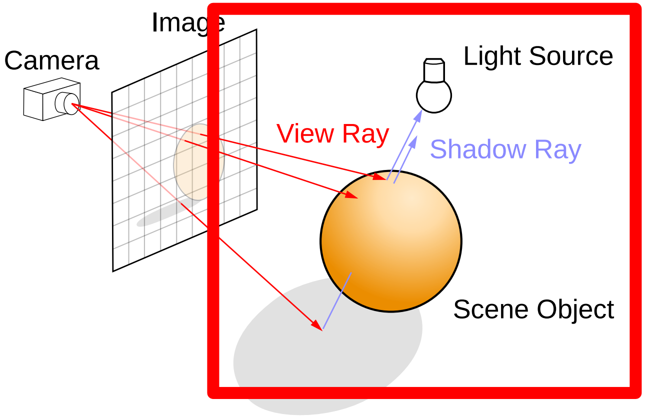


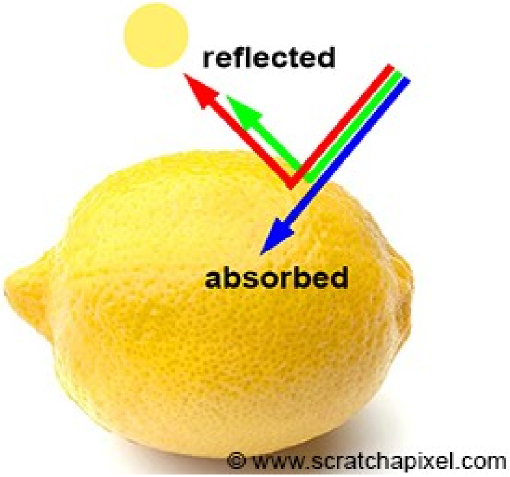
© www.scratchapixel.com

- **Rasterization-based** rendering projects object vertices on screen to solve visibility
  - Perspective projection matrix is used
- **Ray-tracing based** rendering generates rays from camera to solve visibility
  - Perspective projection will be inherently present

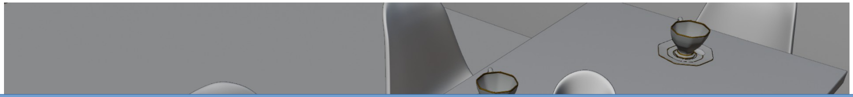
Shading

- **Shading calculates appearance (color) of objects visible to camera**
- Color of object is determined by:
  - Amount of light falling on it
  - Material which defines how light reflects and absorbs

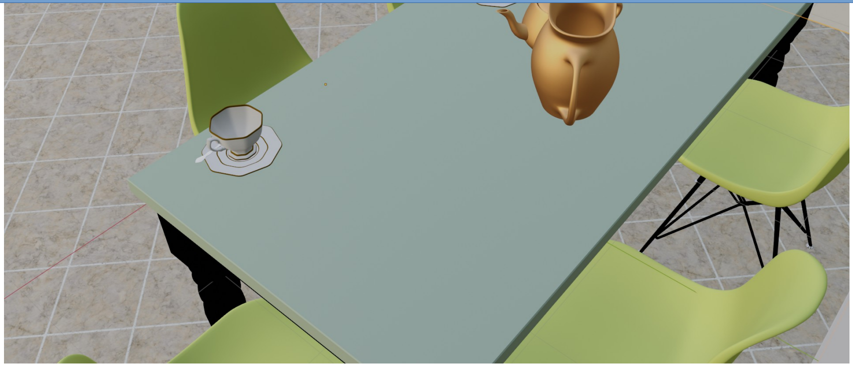




Shading



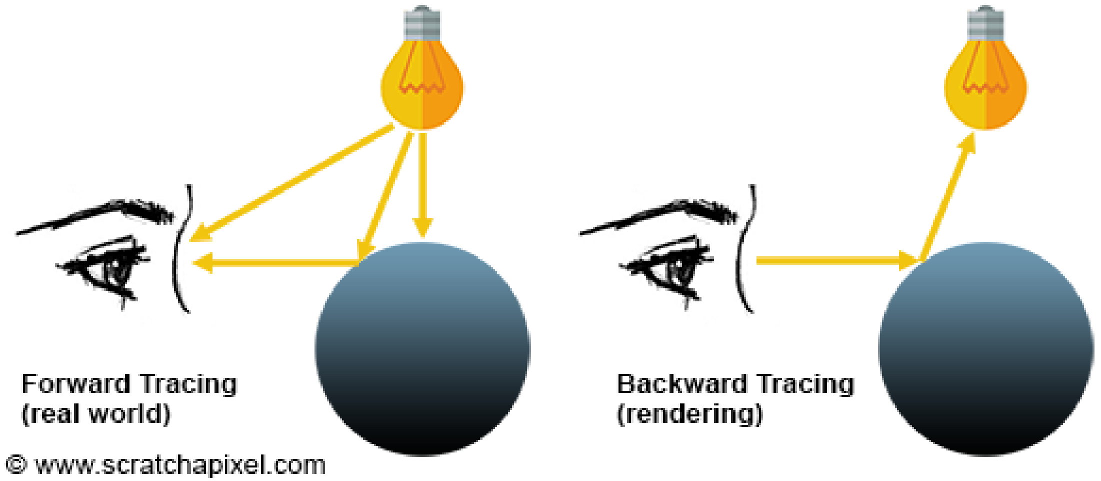
add: https://www.scratchapixel.com/lessons/3d-basic-rendering/rendering-3d-scene-overview/introdu



  - Form of visibility problem: which light sources and surfaces reflecting light are visible from shaded surface
  - This computation is called **light transport**

Light transport

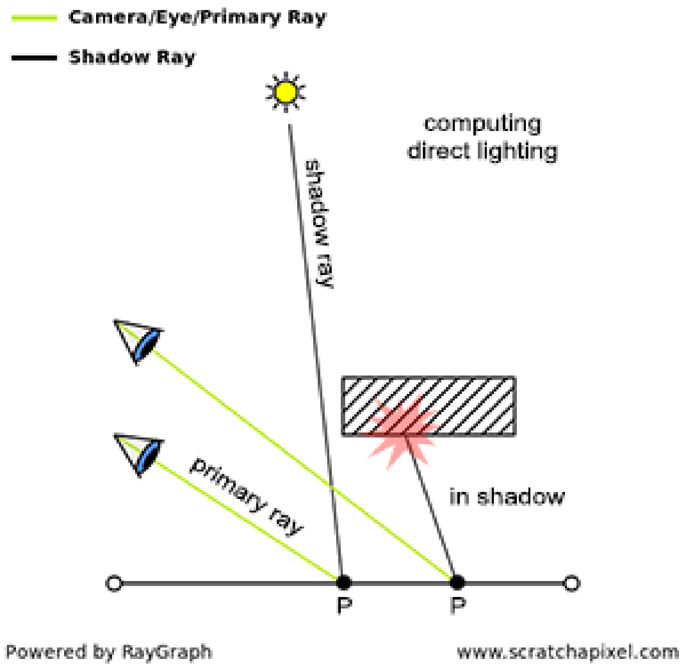
- **Information on light is crucial for shading**
- **Forward tracing** starts from light, bounces around the scene and enters camera
- **Backward tracing** starts from camera, finds a way to light source
- Light transport is used to find amount of light falling on objects visible from camera



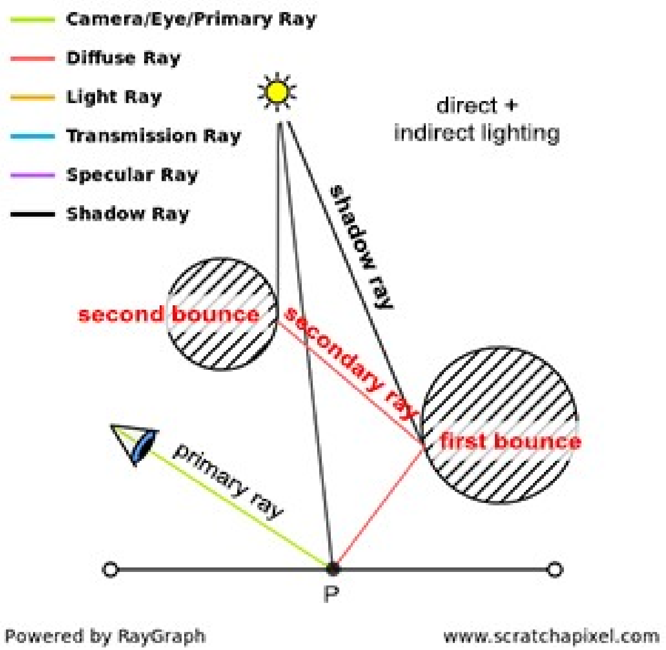
© www.scratchapixel.com

Light transport

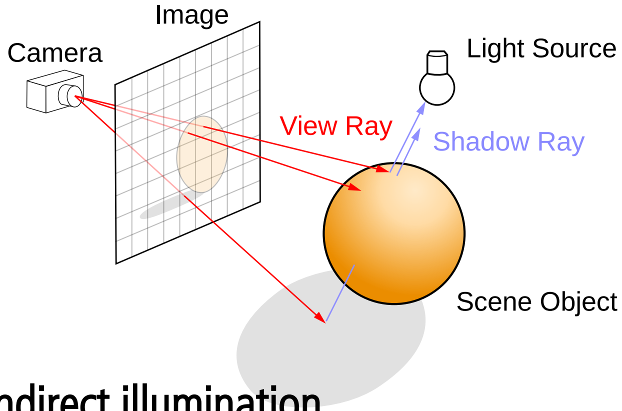
- Backward tracing: direct illumination
  - Only takes in account direct contribution from light source (shadow ray)



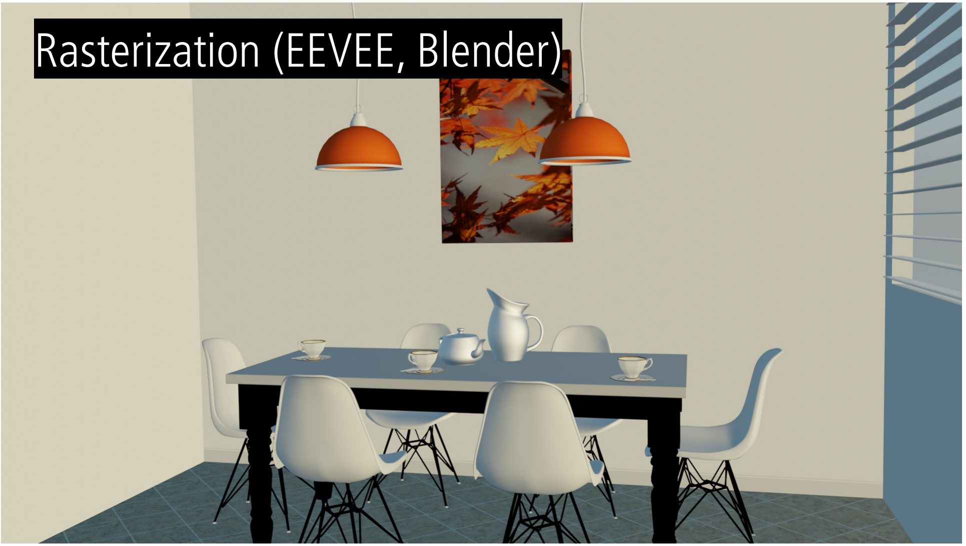
- Backward tracing: direct and indirect illumination
  - Takes in account both direct and indirect light contribution (shadow and secondary rays)

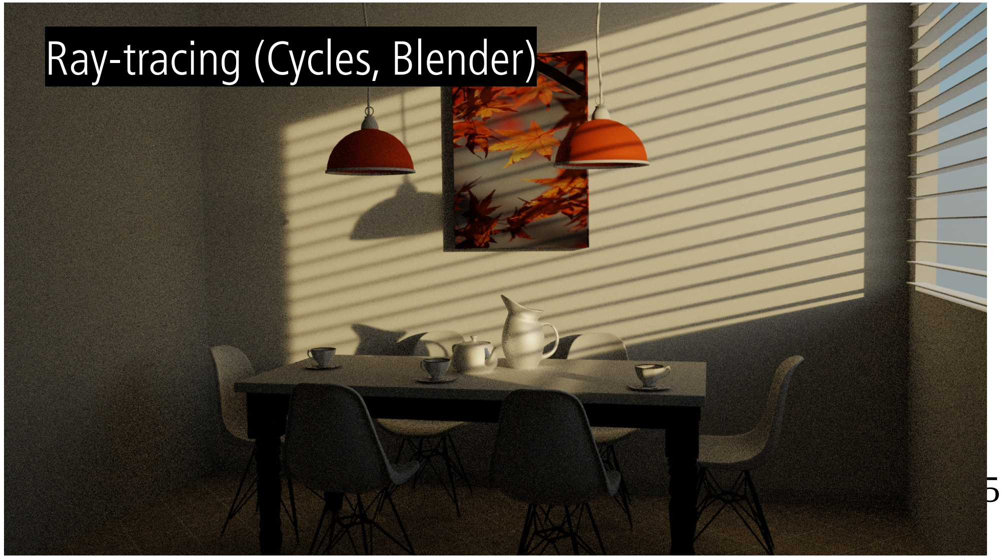


Light transport



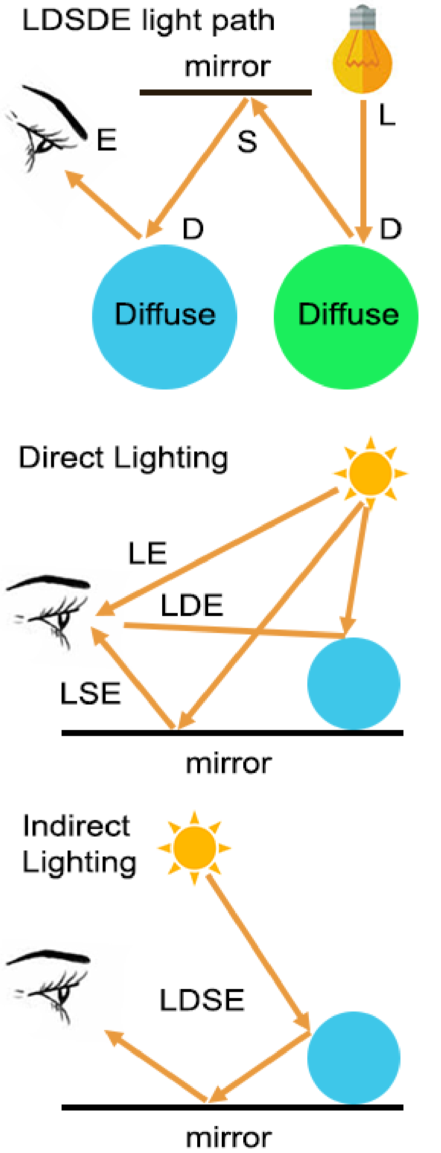
- Relies on concept of visibility
  - Which light sources are visible to shaded surface → **direct illumination**
  - Which surfaces are visible to shaded surface that may reflect light → **indirect illumination**
- Light transport significantly determines the resulting image realism


Rasterization (EEVEE, Blender)


Ray-tracing (Cycles, Blender)

Light transport

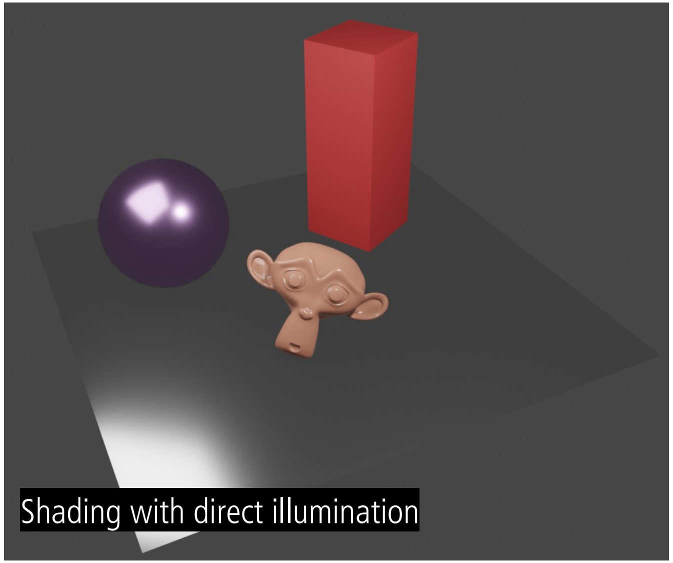
- Light is emitted from light source (**L**), reflects on objects, small amount enters the eye (**E**)
- Direction of light reflection depends on object material:
  - Diffuse surface (**D**)
  - Specular surface (**S**)
- Light paths solved by light transport can be characterized using **path labeling**: L, E, D, S
  - Paul Heckbert: "Adaptive Radiosity Textures for Bidirectional Ray Tracing"



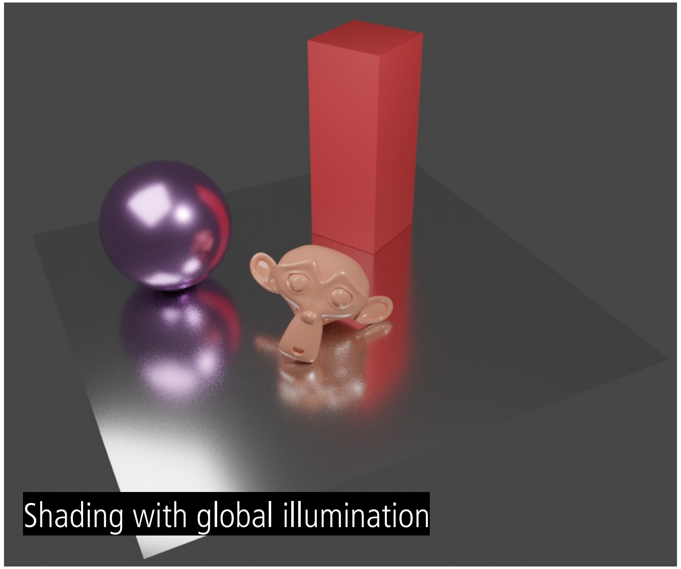
© www.scratchapixel.com

Object appearance

- Object appearance (color) depends on:
  - How light interacts with objects
  - How light travels between objects
- **Shading**: light-object interaction
  - Reflection, refraction, transparency, diffuse, specular, glossy, etc.
- **Light transport**: how much light falls on surface
  - **Direct illumination**: light from light sources
  - **Indirect illumination**: inter-reflections (indirect diffuse, indirect specular), soft shadows, transmission
  - **Global illumination: direct + indirect illumination**



Shading with direct illumination



Shading with global illumination

Decoupling rendering steps

- **Visibility & projection**
  - Perspective projection
  - Rays and camera
  - Ray-triangle intersection
  - Ray-mesh intersection
  - Ray-shape intersection
  - Object transformations
  - Rasterization
  - Ray-tracing
  - Etc.

- **Shading & light transport**
  - Rendering equation
  - Light transport algorithms: path-tracing
  - Non-physical lights (e.g., point lights)
  - Physical (area) lights
  - Material
  - Texture
  - Scattering functions; BRDF
  - Diffuse, glossy, specular
  - Etc.

Rendering equation

- Shading and light transport are described by rendering equation:

$$L_o(p,\omega_o) = L_e(p,\omega_o) + L_s(p,\omega_o)$$

Emission Scattering → reflection

$$L_o(p,\omega_o) = \boxed{L_e(p,\omega_o)} + \boxed{\int_\Omega f(p,\omega_o,\omega_i)L_i(p,\omega_i)(\omega_i \cdot n)d\omega_i}$$

Emission

BRDF

Incoming light

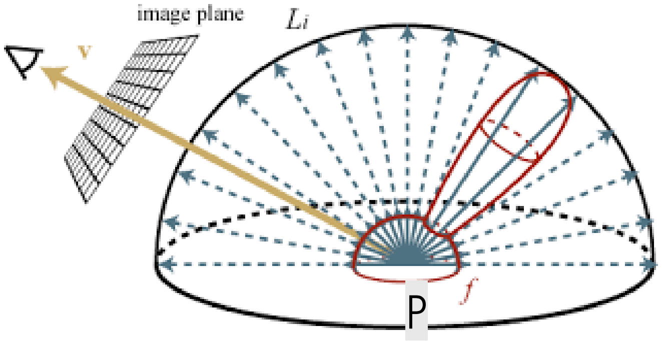
Attenuation due to surface orientation

- Rendering equation is foundation of physically-based rendering
- All rendering approaches are solves rendering equation to some extent or just some of its parts.

Kajiya - rendering equation (1986):
https://dl.acm.org/doi/10.1145/15886.15902

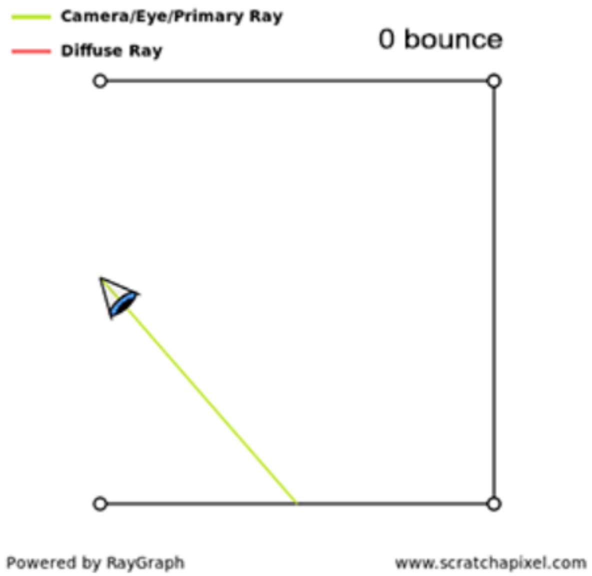
Rendering equation: incoming light directions

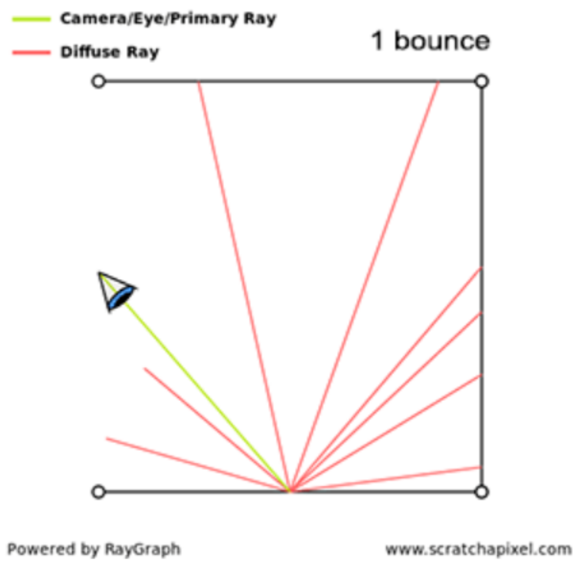
- Light can come from any direction on point P
- There are infinite number of possible directions (thus integral sign)
- This can not be solved analytically, solutions:
  - Simplify possible incoming directions: direct illumination – only look for light sources
  - Approximate by sampling smaller number of directions: indirect illumination – other surfaces may contribute

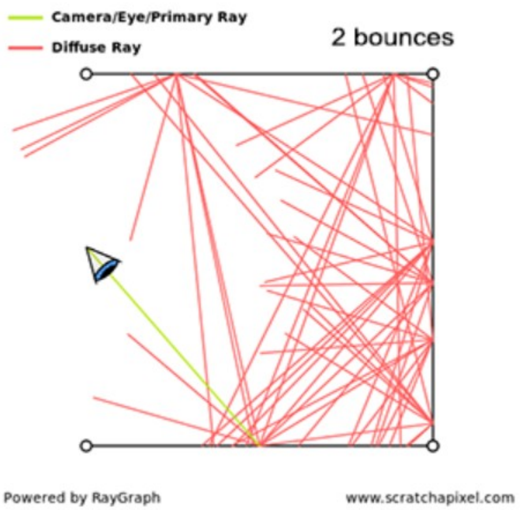


Rendering equation: recursive nature

- For each light direction incoming from another surface, the rendering equation must be evaluated again







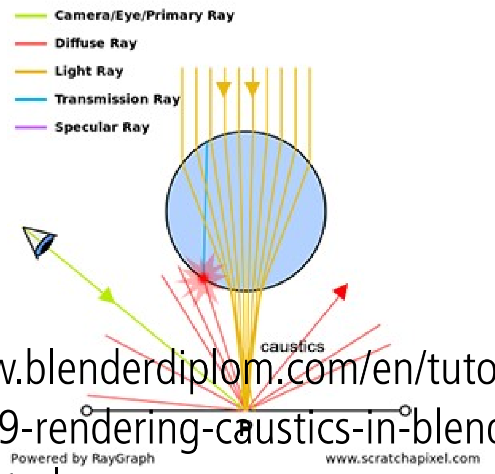
Rendering equation solvers

- Rendering equation is very general: describes wide range of light transport phenomena
  - Not possible to solve analytically
- Different methods compute incoming light with simplifications or approximations:
  - **Direct illumination: usually rasterization-based rendering**
  - **Consider only specularly reflective and transmissive surfaces: Whitted ray-tracing**
  - **Direct and indirect illumination approximations:**
    - **Stochastic approximation based on Monte-Carlo ray-tracing → path tracing, bidirectional path tracing, metropolis light transport, etc.**
    - **Approximation using finite element method → radiosity**

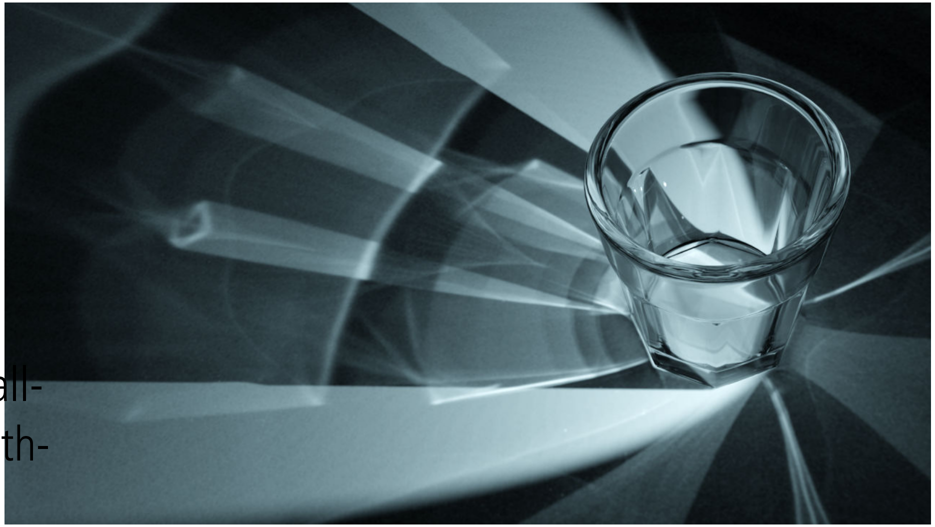
Rendering equation solvers

re-read

- Different light transport methods were developed for rendering different m
- Light paths labeling can be used to categorize different approaches
- Example: hard problem are specular surfaces (S) illuminating diffuse surfaces (D): **LSDE**
  - Light is emitted, refracted through glass object and due to refraction, light is concentrated towards few points → **caustics**
  - In rendering, we start from eye and diffuse surface and try to find incoming direction

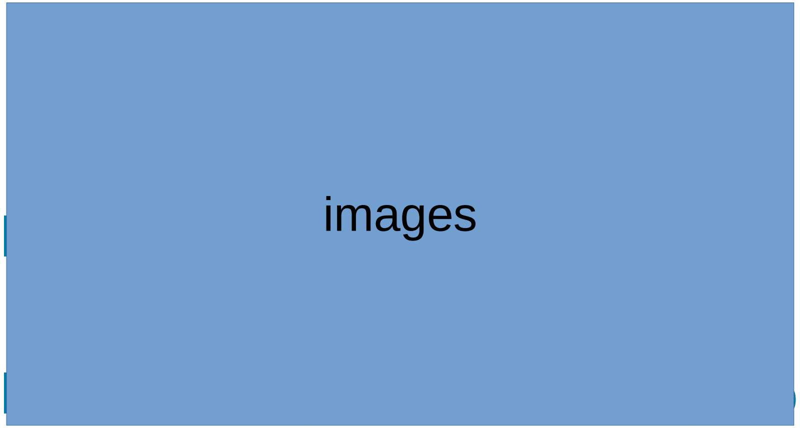


https://www.blenderdiplom.com/en/tutorials/all-tutorials/649-rendering-caustics-in-blender-with-appleseed.html



Rendering equation extensions

- Rendering equation describes light emission and surface reflection in vacuum.
- Limitations:
  - Transmission
  - Sub-surface scattering:
    https://www.pbr-book.org/3ed-2018/Vol
  - Volumetric scattering:
    https://www.pbr-book.org/3ed-2018/Vol cesses
  - Wave optics effects: polarization, diffraction, interference, etc.



Practical rendering

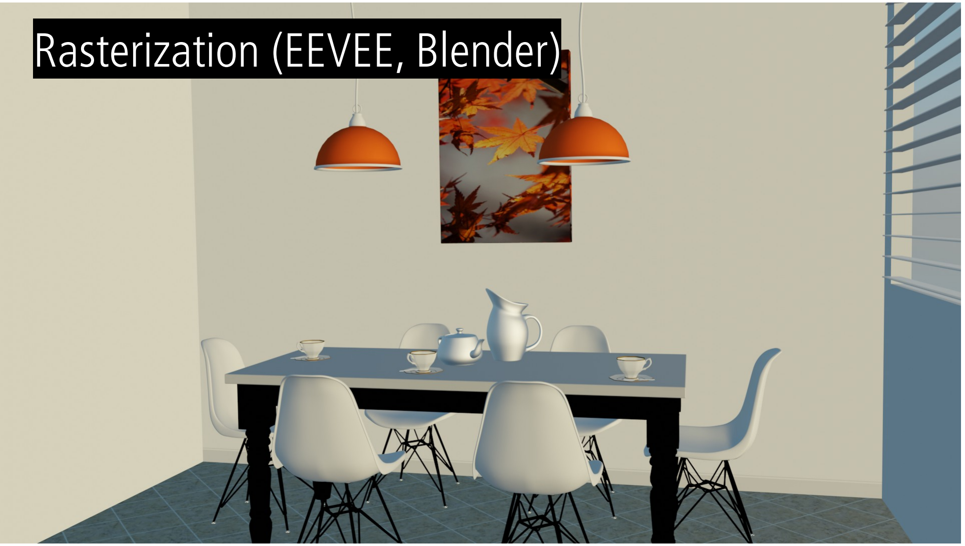
Practical rendering approaches

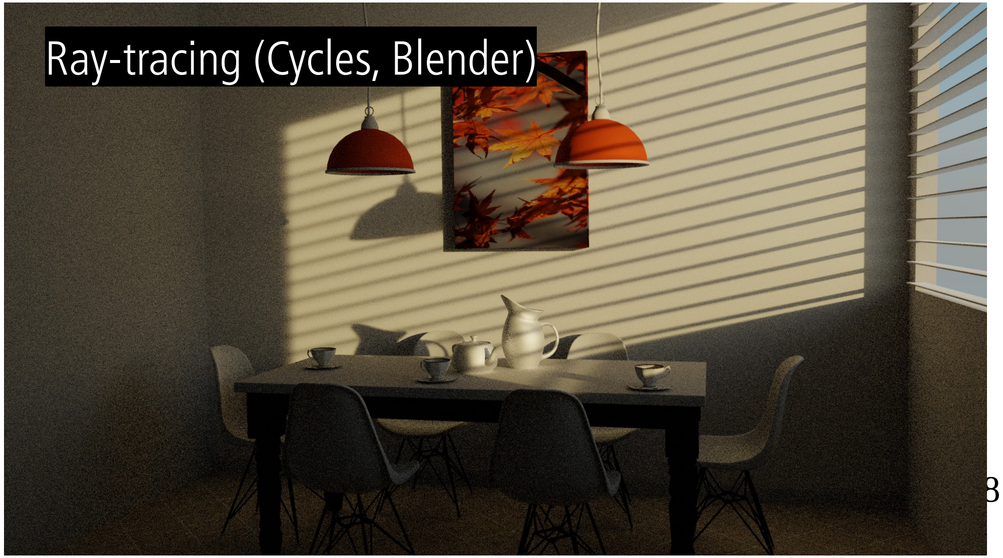
Two main rendering approaches are based on:

- Ray-tracing
- Rasterization

Ray-tracing vs Rasterization

- Compared to rasterization, ray-tracing is more directly inspired by the physics of light
  - As such it can generate substantially more realistic images: shadows, multiple reflections, indirect illumination, etc.


Rasterization (EEVEE, Blender)


Ray-tracing (Cycles, Blender)

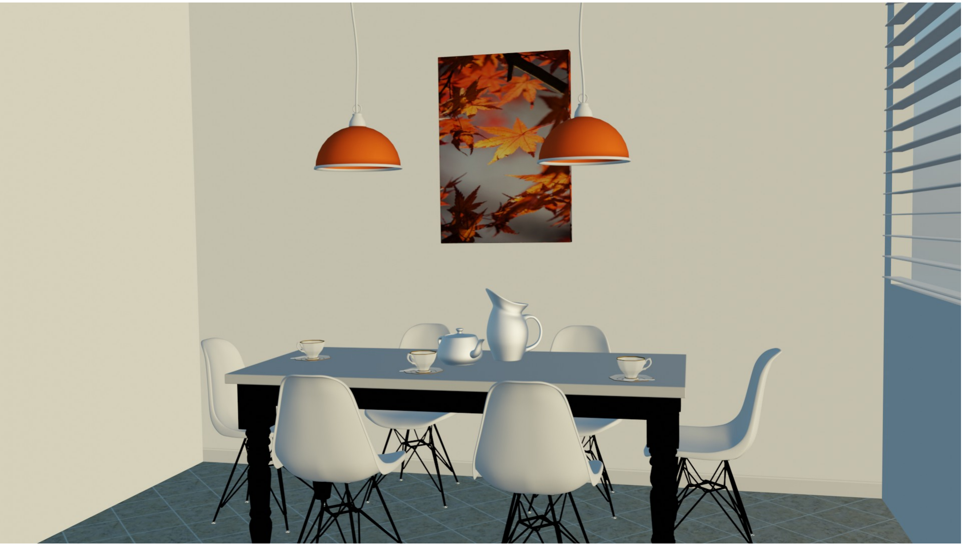
Visibility solvers

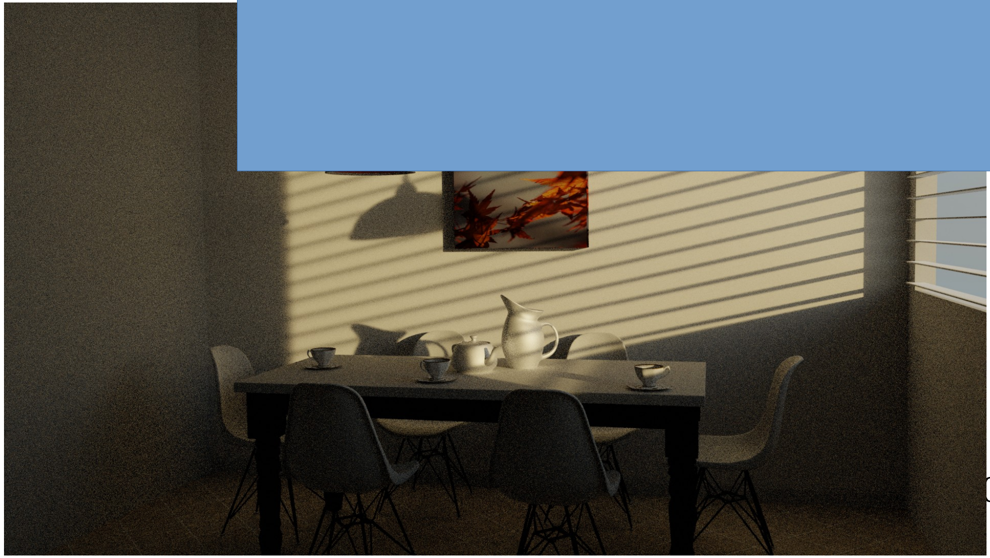
- **Rasterization and ray-tracing are methods for solving visibility**
- Both methods solve **visibility from camera**
- **Ray-tracing can further be used to solve visibility between any points in 3D scene**
  - particularly useful for shading and light transport

Practical tip: Ray-tracing vs Rasterization

- Important difference is that **ray-tracing** can shoot rays in any direction, not only from eye or light source
  - As we will see, this makes it possible to render **reflections, refractions ar**
  - On other words, images just look better
- Content creation is simpler for ray-tracing than rasterization since
  - Additional artist work is required when rasterization-based rendering is

Merge with prev slide



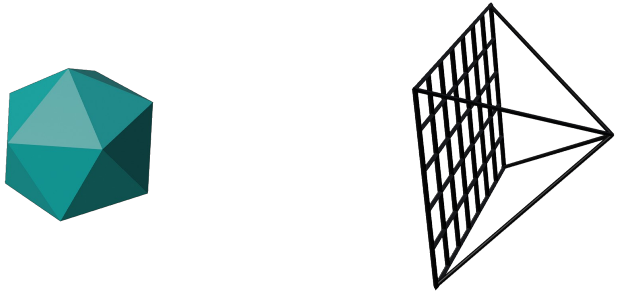


Practical tip: Ray-tracing vs Rasterization

- In production where final images are generated using ray-tracing, rasterization is used for preview and modeling, e.g., animation films
- In production where final images are generated using rasteriz
  Sometimes ray-tracing can be used to precalculate some effect
- <image: animation film: rasterization for preview and modelir
- <image: games: rasterization as final image with hacks and h

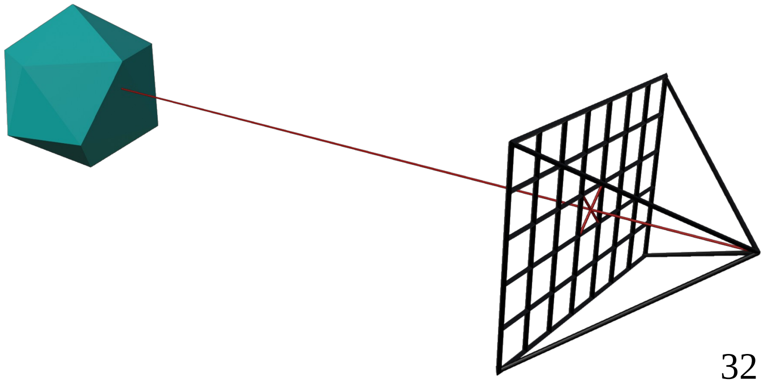
Merge with prev slide

Visibility: ray-tracing

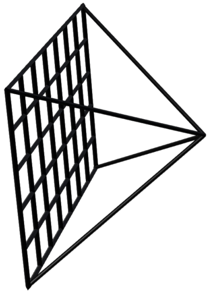


```
for P do in pixels
    for T do in triangles
        determine if ray through P hits T
    end for
end for
```

- Ray-tracing generates rays for each pixel of virtual image plane
- **Rays are constructed using camera and are tested for intersection with objects in 3D scene**
- Closest intersection determines objects visible from camera

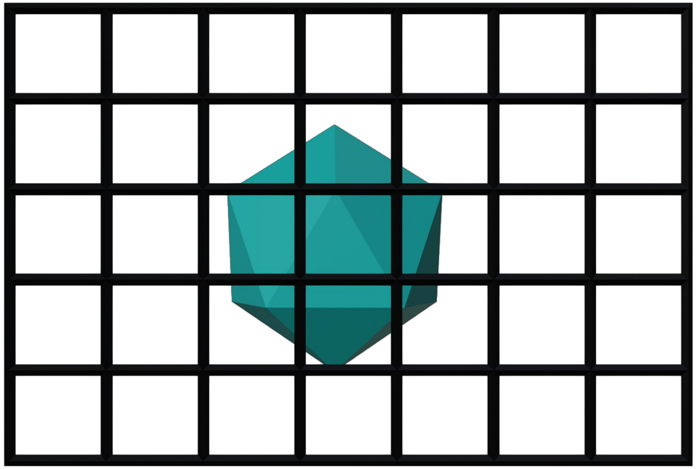


Visibility: rasterization



```
for T do in triangles
    for P do in pixels
        determine if P inside T
    end for
end for
```

- Rasterization projects objects on virtual image plane
- Camera information is used to construct (perspective) projection matrix
- For each pixel of virtual image plane find which objects are covered by pixel and take closest

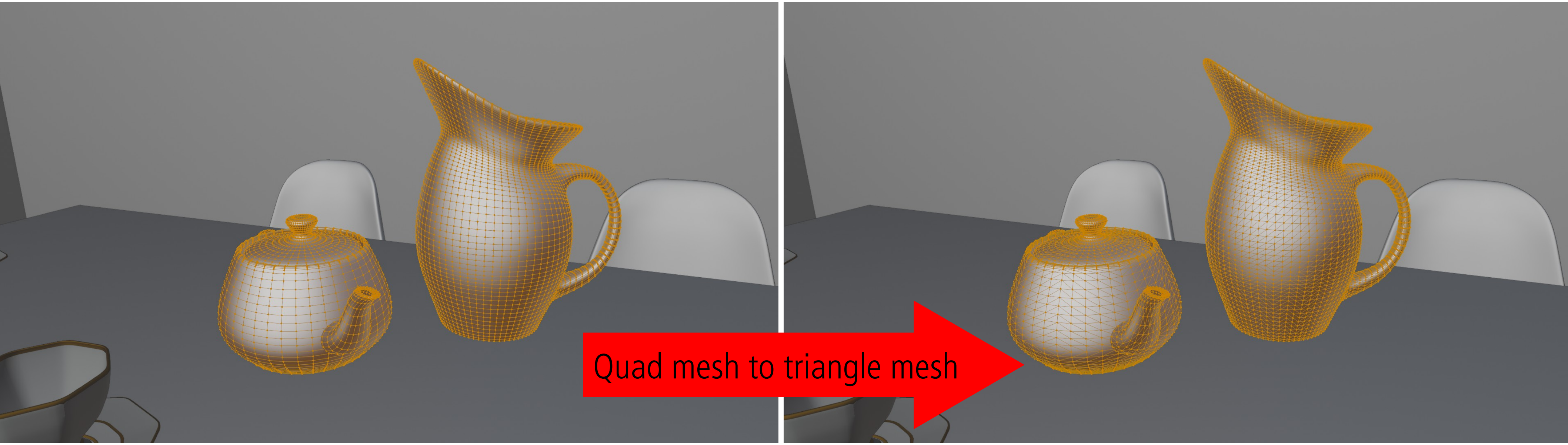


Raytracing vs rasterization

- **Raytracing** iterates through all pixels of image plane, generates rays and then iterates over all objects (triangles) in the scene.
  - This approach is called **image centric.**
- **Rasterization** loops over all geometric primitives in the scene (triangles), projects these primitives on image plane and then loops over all pixels of the image plane.
  - This approach is called **object centric**.

Reminder: triangulated mesh

- Although various shape representations exists for modeling purposes, when it comes to rendering, often all object shapes are transformed to triangles using tessellation process.



Quad mesh to triangle mesh

https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/why-are-triangles-useful

Rendering speed: Ray-tracing and Rasterization

- 3D scenes are complex and often large
- 24 million unique triangles
  - Object reuse via instancing results in 3.1 billion triangles

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

Rendering speed

- Number of rendered images for display per second is called **frames per second** (FPS):
  - Real-time graphics > 60 FPS
  - Interactive graphics 1 < x < 20 FPS
  - Offline graphics < 1 FPS

Speed: Ray-tracing vs Rasterization

- Both algorithms are conceptually simple
- To make them fast and usable in practice for n objects:
  - Ray-tracing is utilizing acceleration spatial data structures for achieving `O(log(n))`
    - Acceleration data structures: bounding volume hierarchy (BVH) or Kd tree
  - Rasterization relies on culling and hardware support ensuring better running time than `O(n)`
    - Culling methods for avoiding full scene processing: occlusion culling, frustum culling, backface culling
    - Deferred shading
    - GPUs are adapted for rasterization-based rendering

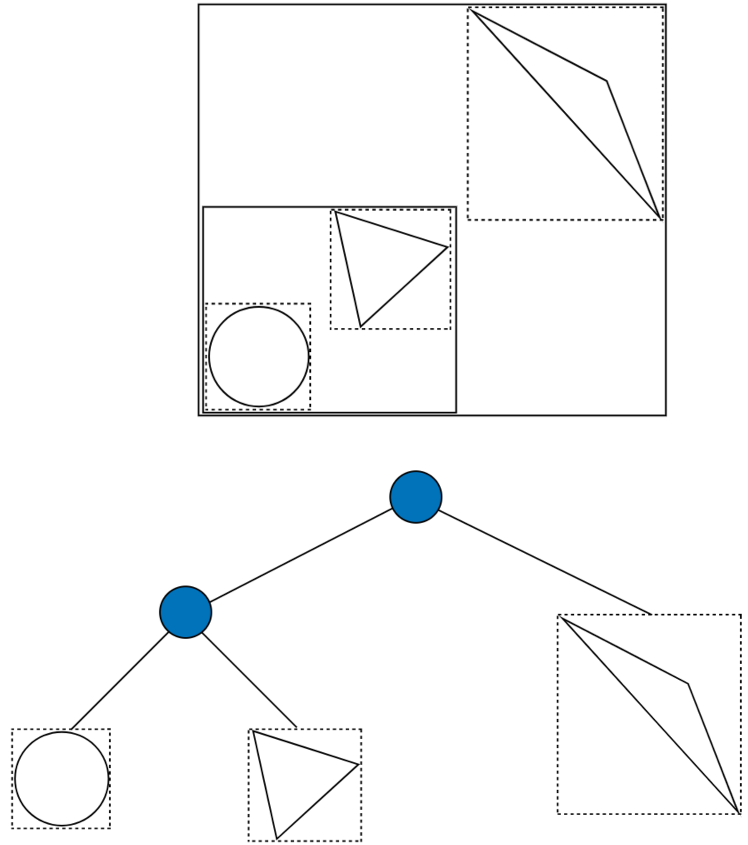
Re-read, see what can be remove
And is mentioned in next slides

Ray-tracing acceleration

- Tracing single ray through the scene naively would require testing intersection with all objects in 3D scene → linear in the number of primitives in the scene → slow!
- This is not optimal since rays might pass nowhere near to vast majority of primitives
- To reduce number of ray-object intersection tests, acceleration structures are used
- Two main categories of acceleration structures:
  - Object subdivisions
  - Spatial subdivisions

Object subdivisions: BVH

- Progressively breaking down objects in the scene into smaller parts of objects
  - Example: room can be separated into walls, floor, ceiling, table, chairs, etc.
- Resulting parts are in a hierarchical, tree like structure
  - For each ray, instead of looping through objects, traversal through the tree is performed
- If ray is not intersecting the parent object then its parts, children objects, are not tested for intersection
  - Example: if room is not intersected, then table can not be intersected as well



- Root node holds the bounds of the entire scene
- Objects are stored in the leaves, and each node stores a bounding box of the objects in the nodes beneath it

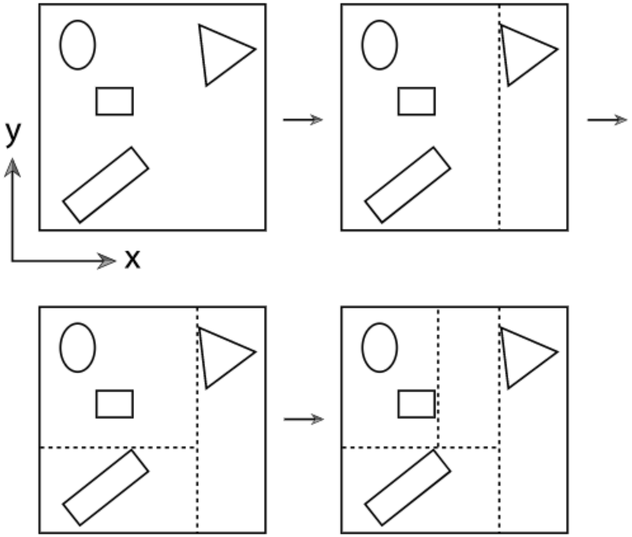
https://www.pbr-book.org/3ed-2018/Primitives_and_Intersection_Acceleration/Bounding_Volume_Hierarchies

Spatial subdivisions: BSP trees

- Binary space partitioning trees adaptively subdivide scene into regions using planes and record which primitives are overlapping regions
  - Process starts by bounding box encompassing whole scene
  - If the number of objects is larger then some defined threshold, box is split
  - Repeated until maximum depth or tree contains small enough regions
- Two BSP variants:
  - Kd trees
  - Octree
- When tracing rays, only objects in regions through which ray passes are tested for intersection

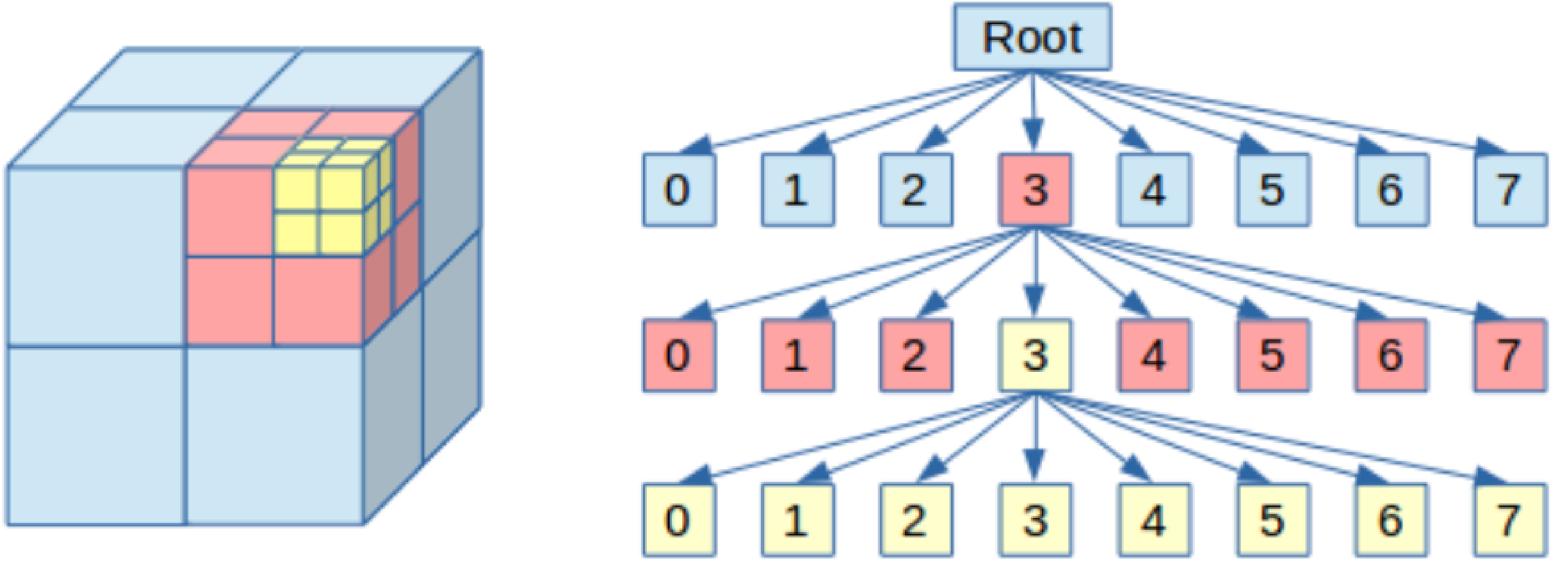
Spatial subdivisions: kd-trees

- Subdivision rule: planes must be perpendicular to on of coordiante axes



Spatial subdivisions: octrees

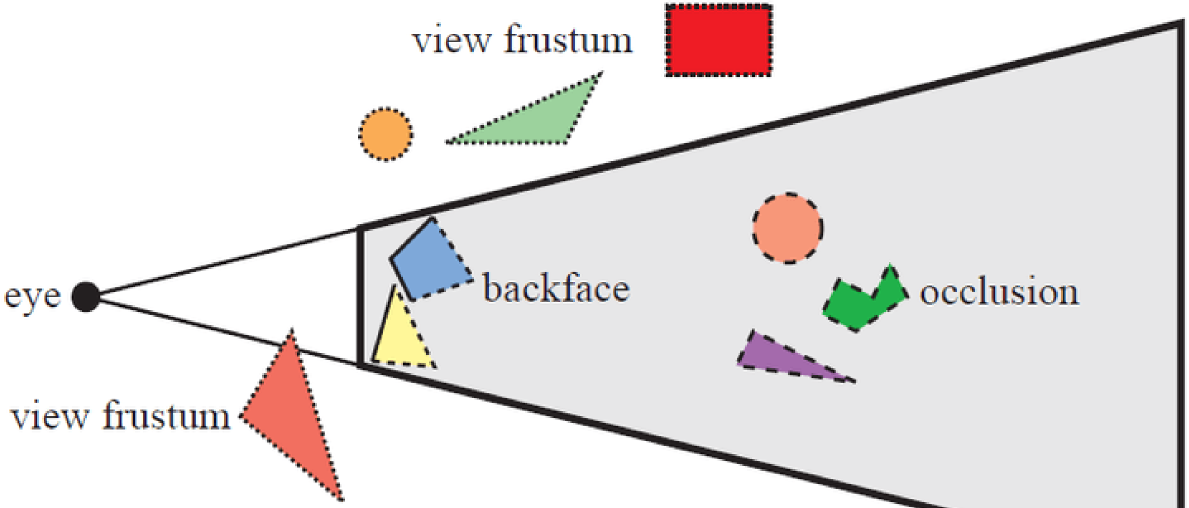
- Subdivision is done by three axis-perpendicular planes which splits box into eight smaller boxes



https://geidav.wordpress.com/2014/07/18/advanced-octrees-1-preliminaries-insertion-strategies-and-max-tree-depth/

Rasterization acceleration

- Whole scene processing is avoided using:
  - Frustum culling
  - Occlusion culling
  - Backface culling



https://bruop.github.io/frustum_culling/

Frustum culling

- https://learnopengl.com/Guest-Articles/2021/Scene/Frustum-Culling
- Horizon example

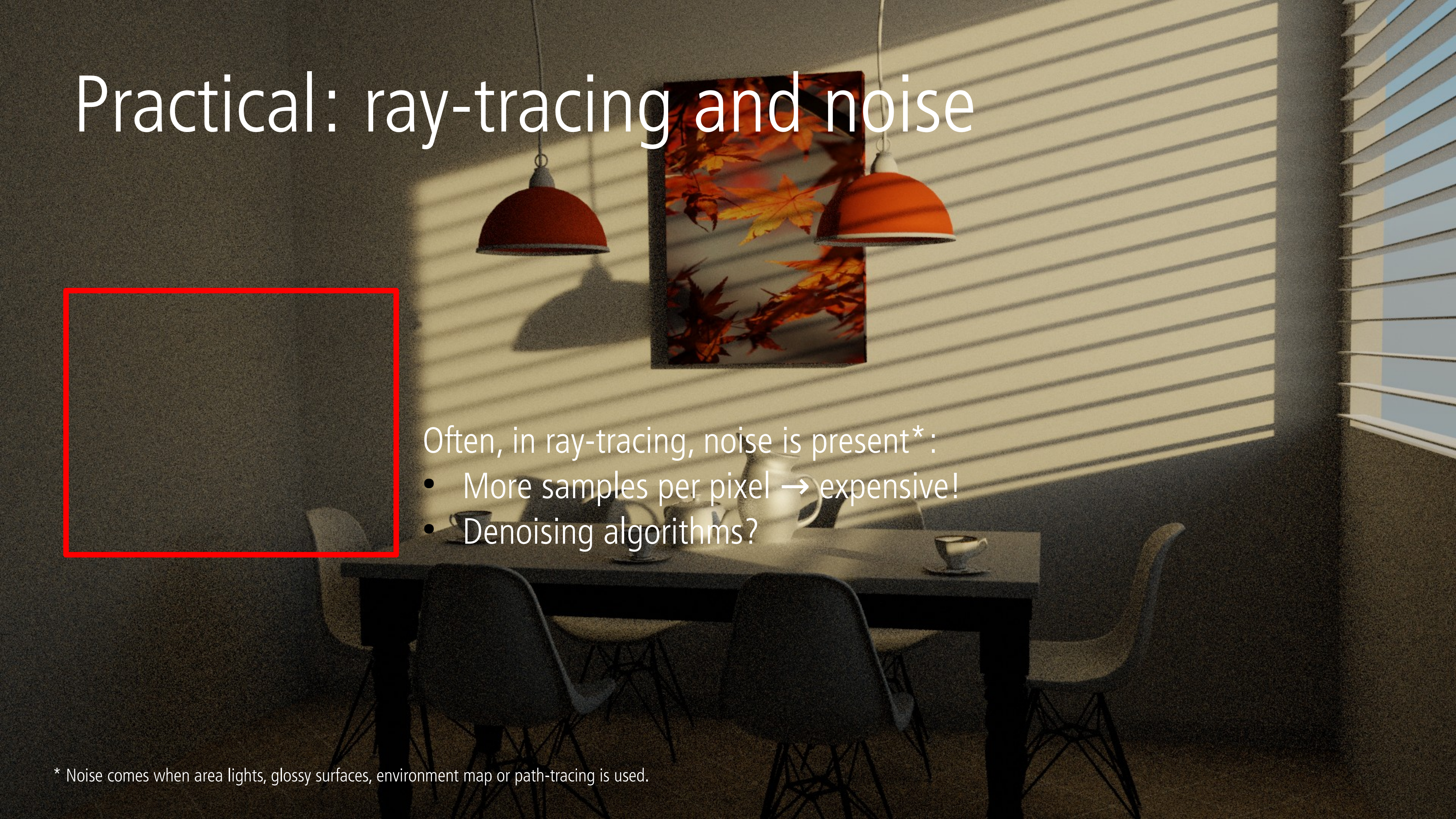
Backface culling

- https://learnopengl.com/Advanced-OpenGL/Face-culling

Occlusion culling

- https://www.gamedeveloper.com/programming/occlusion-culling-algorithms

Practical: ray-tracing and noise

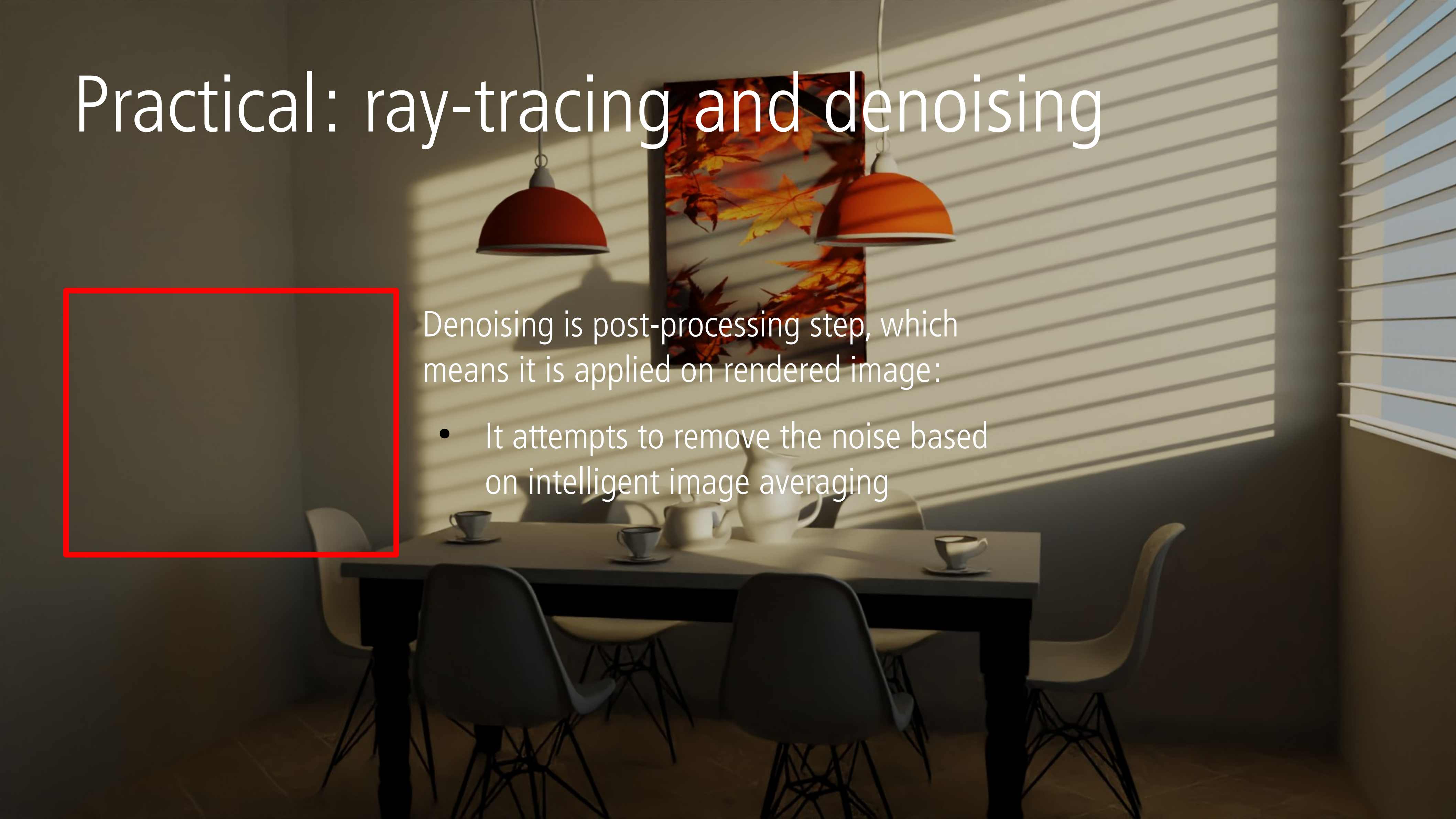


Often, in ray-tracing, noise is present*:

- More samples per pixel → expensive!
- Denoising algorithms?

* Noise comes when area lights, glossy surfaces, environment map or path-tracing is used.

Practical: ray-tracing and denoising



Denoising is post-processing step, which means it is applied on rendered image:

- It attempts to remove the noise based on intelligent image averaging

Towards real-time ray-tracing

- Additional techniques and hardware
- Denoising is promising solution.
- For short-term, clever combinations are expected.
  - https://docs.unrealengine.com/4.26/en-US/RenderingAndGraphics/RayTracing/
  - https://unity.com/ray-tracing
  - Rasterization seems to be not going anywhere any time soon!

re-read

Note: compexity of scenes and rendering

- Although HW and methods a
  rendering the scenes stayed
  complexity of the 3D scenes
  algorithms increased.
  - **Blinn law**

re-read

Raytracing and rasterization

re-read

- Raytracing is historically early method for imag n
- Raytracing has always been recognized as best nages. However, up until 1990-2000s it hasn’t been u
- Due to hardware advancements, rasterization-b
- Raytracing is intuitive and straightforward meth f how image is created.
  - Based upon raytracing, more advanced rendering methods are build.
  - Furthermore, understanding ray-tracing gives good foundations for understanding rasterization-based rendering systems.

Practical ray-tracing and rasterization

- Raytracing method can be separated in:
  - **Determining visibility from camera**: determine which point in 3D s **ve projection** is inherent due to camera which is used for generating
  - **Shading**: calculating the color of visible point – this operation bes o collect all light which influences the color and intensity of that point.
- Both steps require extremely time-consuming intersections b
  - Efficient ray-object intersection method is needed
  - 3D scene must be represented for fast and efficient spatial search of objects which should be tested for intesection
- On the other hand, rasterization can solve the visibility problem (camera-object) very fast. That is why real-time graphics is predominantly using GPU rasterization approach. But when it comes to shading, due to the available information on 3D scene, it is not as good as raytracing.
- Raytracing is slow for solving visibility problem, but it enables high-quality shading. For high-quality, photo-realistic production rendering, ray-tracing is almost always used*.

re-read

* However, real-time ray-tracing is now possible to certain degree with certain hardware and it is hot research topic!

Rasterization and Raytracing in practice

- Examples of rasterization-based renderers in producti
  - Godot, Unity, Unreal
- Examples of raytracing-based renderers in production
  - Appleseed, Arnold, etc.

<IMAGE: differences and comparison between rasterize

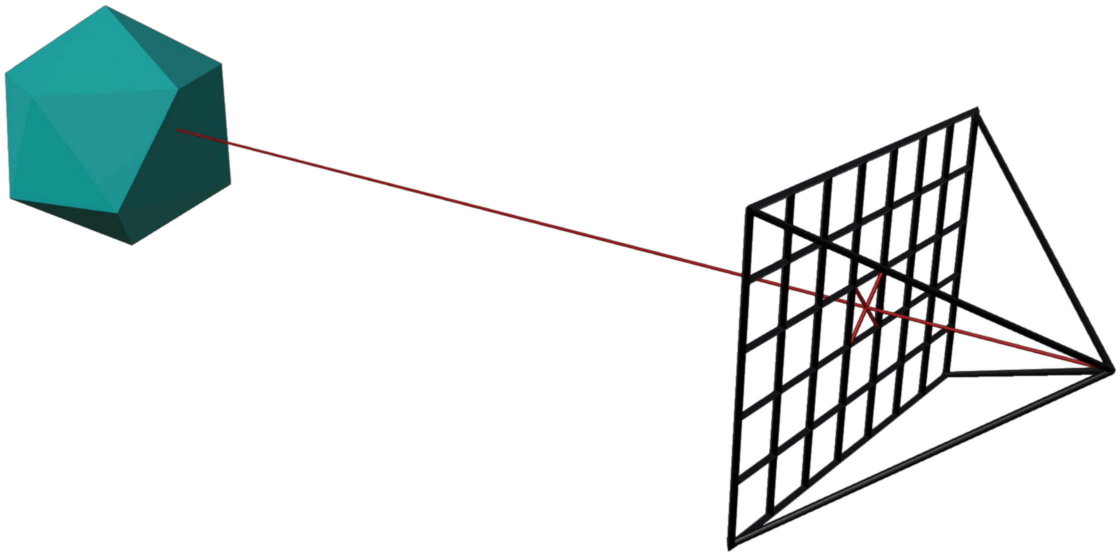
re-read

- Important note that we can learn here, that occurs in almost any field of computer graphics is **trade-off between speed and quality.**
  - Depending on application, best option for speed of rendering and quality of rendered images must be found. For games, it is important that frames are rendered in real-time (>30fps) and certain quality must be sacrificed. For animated films, rendering time can take up to several hours for only one frame therefore, focus can be put on quality.

Rendering: intuition via ray-tracing

Intuition: camera in ray-tracing

- Camera defines from where we look at 3D scene and a 2D surface on which 3D scene will be projected as an image.
  - Rendering goal: compute color of pixels in the image.
  - Viewing rays are generated from aperture position through each pixel of film plane
  - Each ray will return information about color of the object it intersected in 3d scene using **shading**



Intuition: lights and objects in ray-tracing

- To compute color of intersected point, shading: light-object interaction is computed
- For shading calculation we need information about objects in 3D scene as well as light sources.
  - Without light we wouldn’t see objects, without objects we wouldn’t see light.

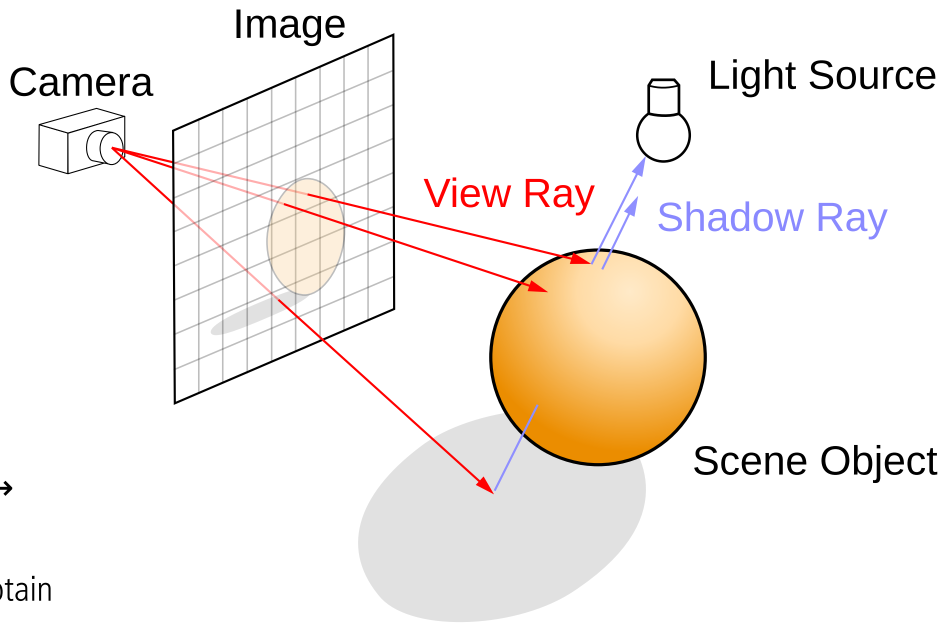
Simulating light

- Simulating physical (real-world) light-object interaction means simulating all light rays paths from light and their interactions with objects. Furthermore, only small amount of that light actually falls on camera. This kind of simulation is called **forward ray-tracing or light tracing.** Simulating this is not tractable! Therefore, we simplify:
  - Only light-object interaction that are visible from camera should be simulated
  - Therefore, we reverse the process: we start with rays that fall into camera and build from there.
- Based on previous discussion, we trace rays from sensor to the objects. This simulation is called **backward ray-tracing or eye-tracing***.
- In rest of the lecture, by ray-tracing we mean backward ray-tracing**.**

* Introduced by Turner Whitted in paper "An Improved Illumination Model for Shaded Display". Method in computer graphics using the concept of shooting and following rays from light or eye is called path-tracing.

Ray-tracing-based rendering idea

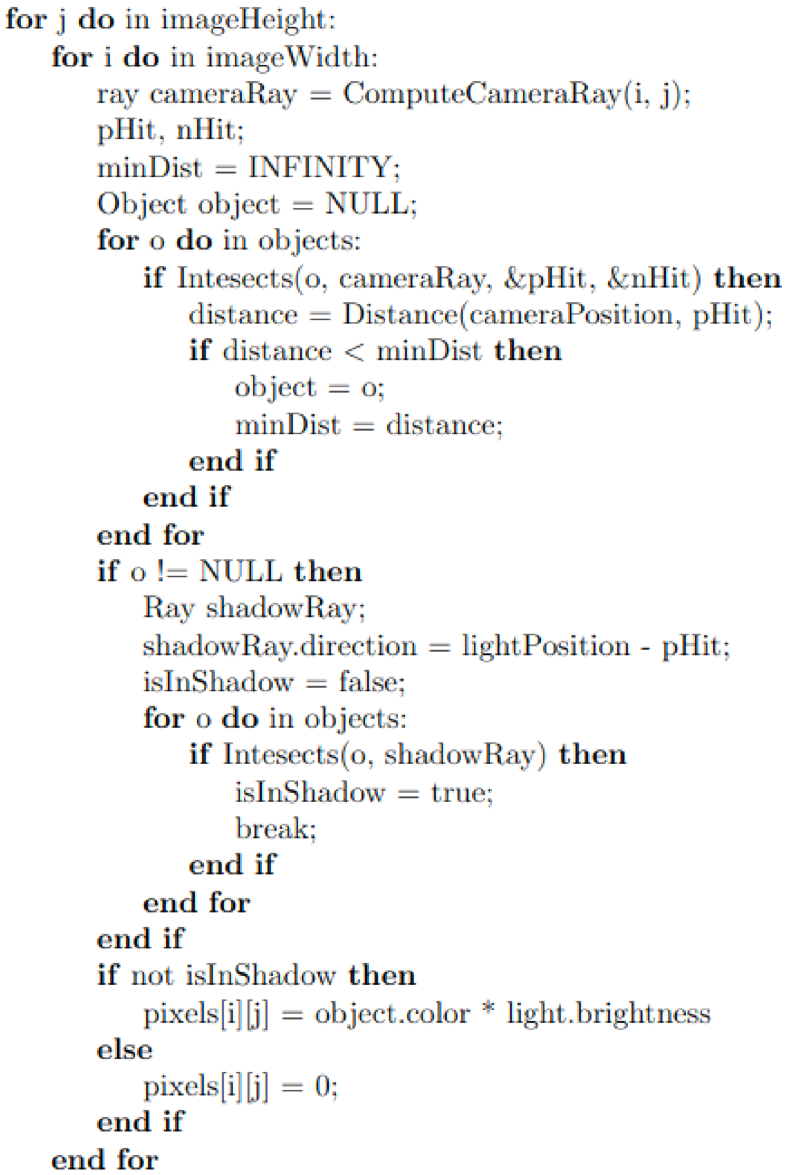
- Generate viewing rays
  - Generate ray using camera (eye and film) → primary/camera/visibility ray
- Tracing ray into the scene. Possible outcomes:
  - Intersects objects, find closest
  - No intersection →hit background
- Object intersection:
  - Calculate amount of light falling on intersection point
    - Shadow/light ray: directly traced from intersection to light source → direct illumination
    - Advanced: sample various directions from this point in 3D scene to obtain reflections of other objects → global illumination
  - Calculate amount of light reflected in primary ray direction using incoming light and material specification → shading



Practical raytracing

Algorithm*:

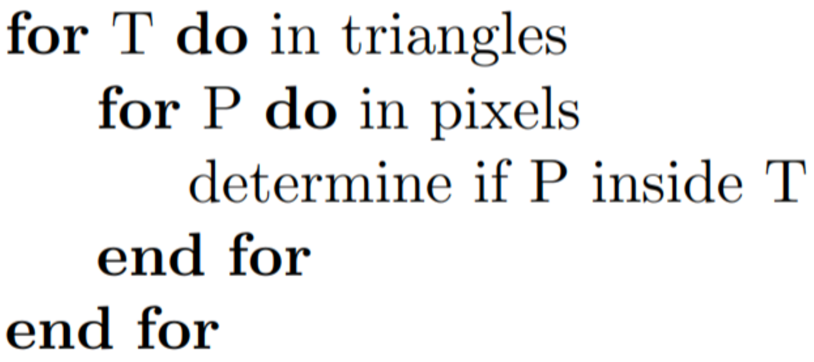
- For each pixel in raster image:
  - Generate primary ray by connecting eye and film using camera information
  - Shoot primary ray into the scene
  - For each object in the scene
    - Check if intersected with primary ray. If intersect multiple objects, take one closest to the eye
  - For intersection, generate shadow ray from intersection point to all lights in the scene
    - If shadow ray is not intersecting anything, lit the pixel



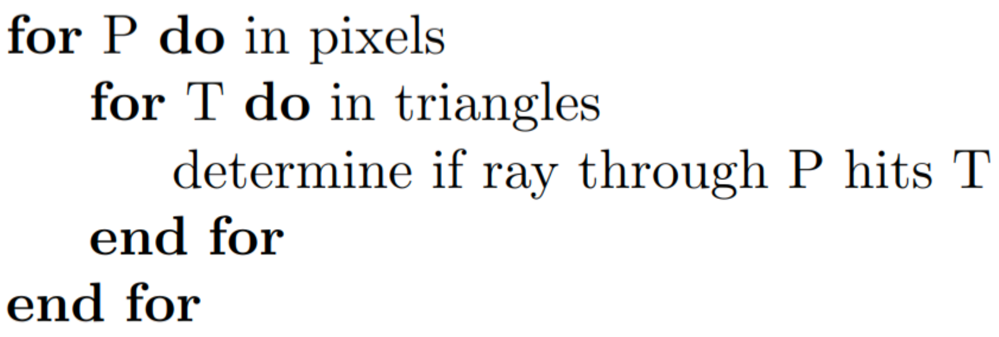
```
for j do in imageHeight:
    for i do in imageWidth:
        ray cameraRay = ComputeCameraRay(i, j);
        pHit, nHit;
        minDist = INFINITY;
        Object object = NULL;
        for o do in objects:
            if Intesects(o, cameraRay, &pHit, &nHit) then
                distance = Distance(cameraPosition, pHit);
                if distance < minDist then
                    object = o;
                    minDist = distance;
                end if
            end if
        end for
        if o != NULL then
            Ray shadowRay;
            shadowRay.direction = lightPosition - pHit;
            isInShadow = false;
            for o do in objects:
                if Intesects(o, shadowRay) then
                    isInShadow = true;
                    break;
                end if
            end for
        end if
        if not isInShadow then
            pixels[i][j] = object.color * light.brightness
        else
            pixels[i][j] = 0;
        end if
    end for
```

* Arthur Appel in 1969 - "Some Techniques for Shading Machine Renderings of Solids"

Rasterization: algorithm reminder



```
for T do in triangles
    for P do in pixels
        determine if P inside T
    end for
end for
```



```
for P do in pixels
    for T do in triangles
        determine if ray through P hits T
    end for
end for
```

Rasterization and raytracing

- Both **raytracing and rasterization are used to solve the visibility problem** (which also contains **perspective/orhographic projection** solving).
- Solving the visibility problem is only a part of rendering. Other part is determining the color and intensity of visible surface. This step is called **shading** and it uses **light transport** to gather light on visible surface and calculates light-matter interaction

Shading and light transport

- Note that the in algorithm that was used to introduce raytracing and idea of rendering, we only determine objects which are visible from camera and shade them by calculating visibility between visible surface and light source.
- With this approach, we disregard that other objects in the scene can also reflect light on objects that are visible from camera. Furthermore, objects that are visible from camera can be very reflective (imagine mirror like surface) and they would certainly reflect objects visible from them.
- This effects are called indirect illumination and global illumination effects.
- They can be simulated by generating additional rays in intersections: propagating rays and building paths that light might travel to the surface which is being shaded.
- Classic examples that can be started with are mirror-like and transparent surfaces
- https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing/adding-reflection-and-refraction
- Depending on surfaces, light will scatter differently. Material defines, among others, scattering model* which is used to determine where light scatters (direction) and how much it scatters in particular direction (intensity).
- Categorization of light transport algorithms based on lighting effects they simulate (Diffuse, Specular and other types of reflections and refractions).

* Scattering models might be also called illumination models. This name might make more sense in GPU-rasterization-based rendering (graphics rendering pipeline) since after determining surfaces visible from camera, illumination model is employed to directly calculate color and intensity taking in account positions of light and material of the objects. In ray-tracing-based rendering, shading part relies on more advanced light-transport which uses scattering models to determine how light bounces (scatters of the objects) in the scene.

Importance of visibility calculation for rendering

- Based on previous discussions, we can highlight the importance of visibility calculation and how it is related to all concepts in rendering.
- Visibility problem is concerned with determining visibility between points
- First, when determining visible objects from camera, we solve visibility problem and utilize perspective/orthographic projection
- Then, when calculating shading we require light transport information which is again based on visibility between surface and lights in 3D scene
  - This is required to calculate shadows, soft shadows, global illumination effects such as reflection, refraction, indirect reflection and.
- Rasterization is good for solving the visibility from camera to object. But it is not good for finding visibility between surfaces in 3D scene which is important for light transport and shading
- Raytracing is good for solving the visibility from camera to object. And it also can be used for finding visibility between surfaces in 3D scene which is important for light transport and shading

<IMAGE: visibility in rasterization and raytracing>

Comparison with rasterization

- Recursive ray-casting used in ray-tracing is something that rasterization-based rendering must approximate with different techniques to achieve just a sub-set of effects that can be obtained with ray-tracing
- <image: example of approximation and ray-traced effect>

Summary questions

- https://github.com/lorentzo/IntroductionToComputerGraphics/tree/main/lectures/11_rendering_overview

Reading Materials

- https://github.com/lorentzo/IntroductionToComputerGraphics