Lecture 11: Rendering overview

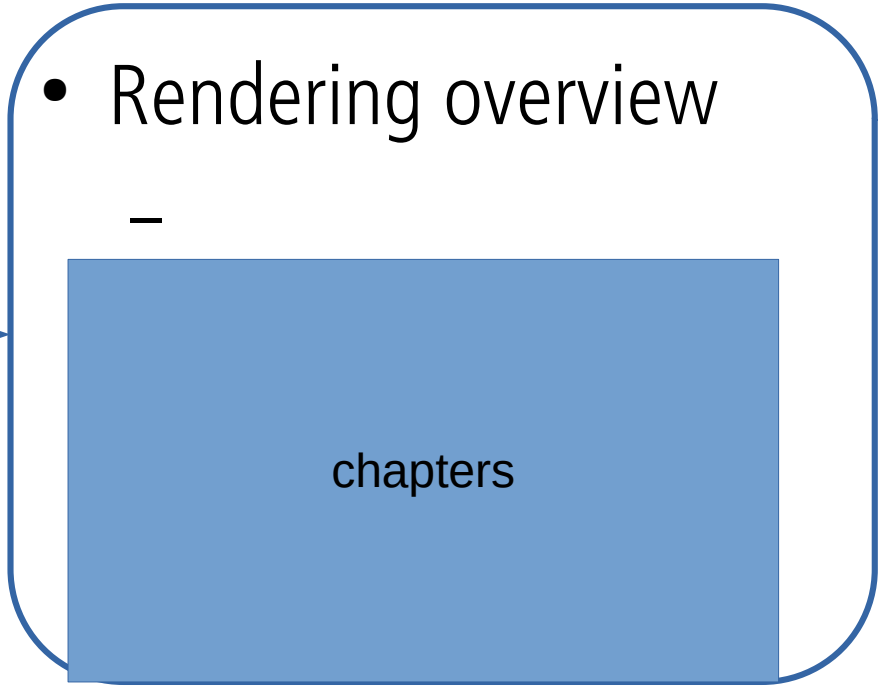
DHBW, Computer Graphics

Lovro Bosnar

8.3.2023.

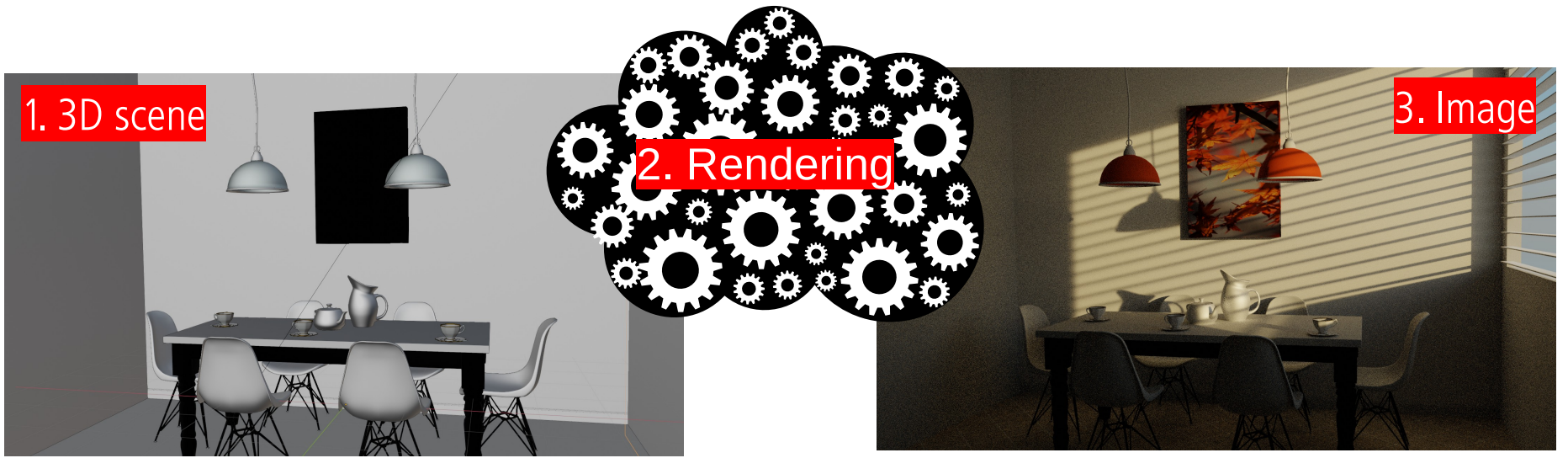
Syllabus

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



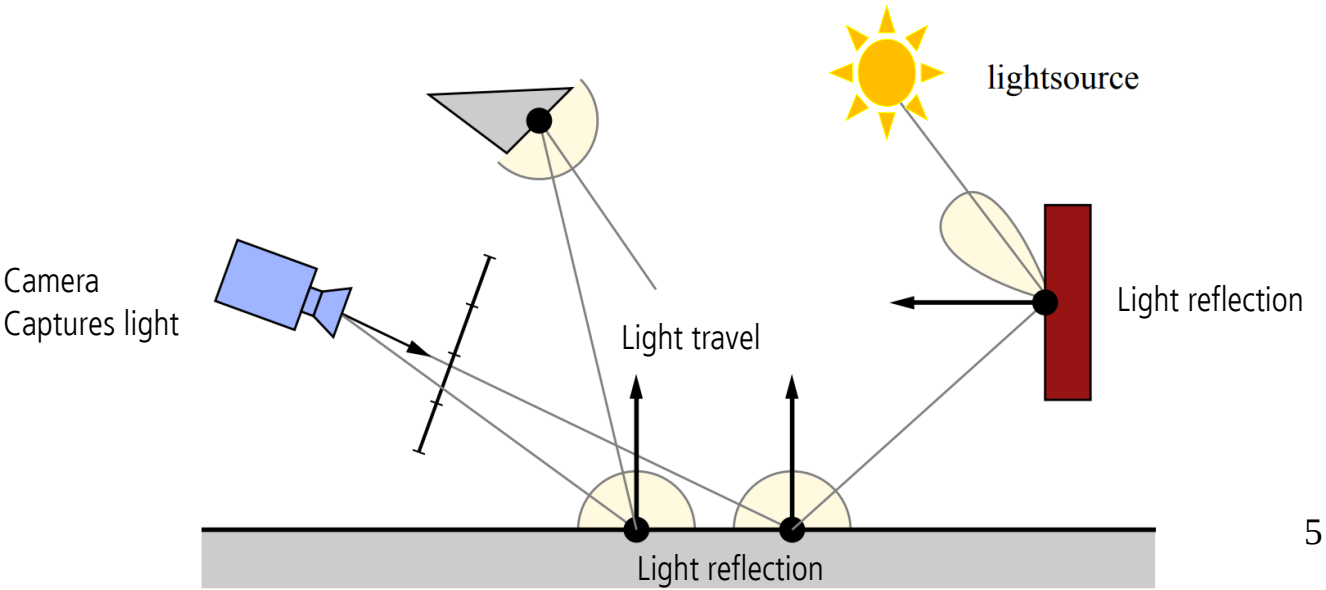
Rendering

- 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** of virtual camera in 3D scene
  - Find which objects are visible virtual camera
  - Calculate color of visible objects



Rendering: light simulation

- Idea of computing pixel colors of virtual image plane is similar to how digital camera sensor forms an image
- Digital camera sensor is made of array of photo-sensitive cells (pixels) which convert incoming light into colors
- Light falling on camera is reflected from objects in the scene
- Therefore, we are only interested into light falling on camera sensor



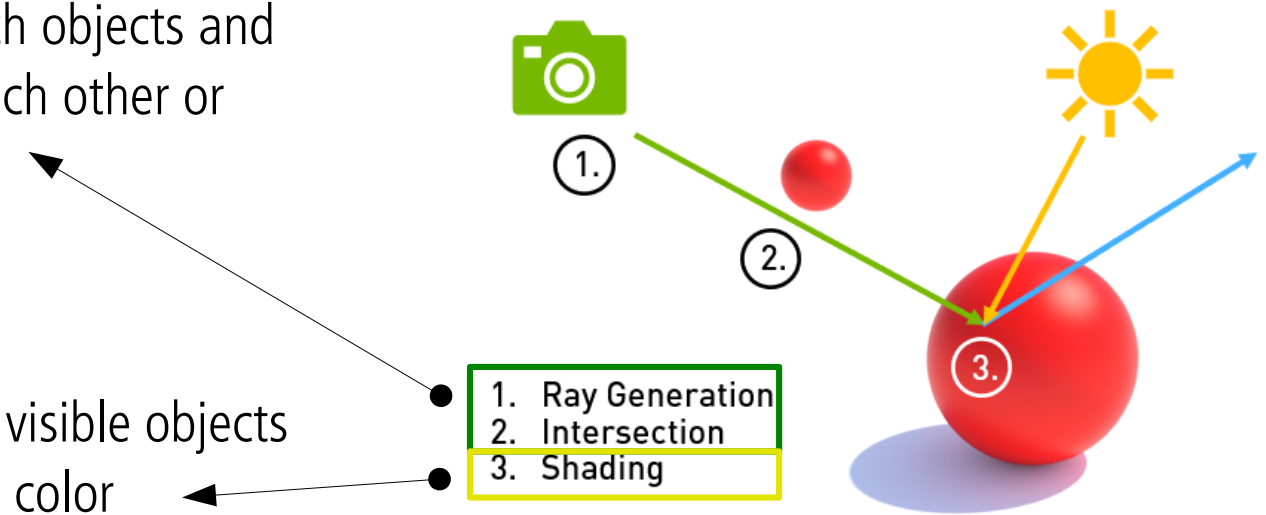
Rendering: physical and perceptual considerations

- Light emission, travel and interaction with objects and camera sensor is well described in physics (wave and geometrical optics)
- We also 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

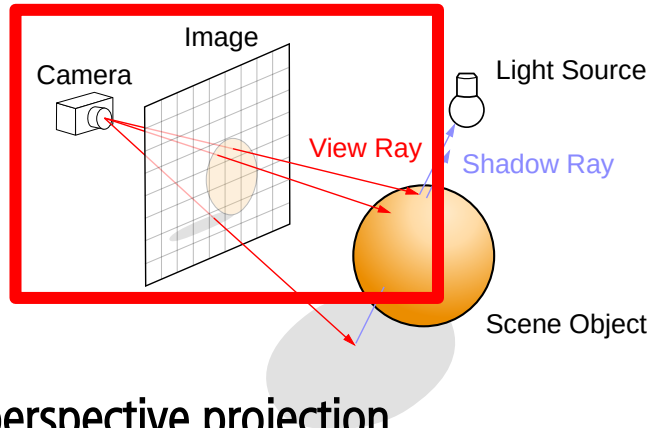
Rendering is solving:

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

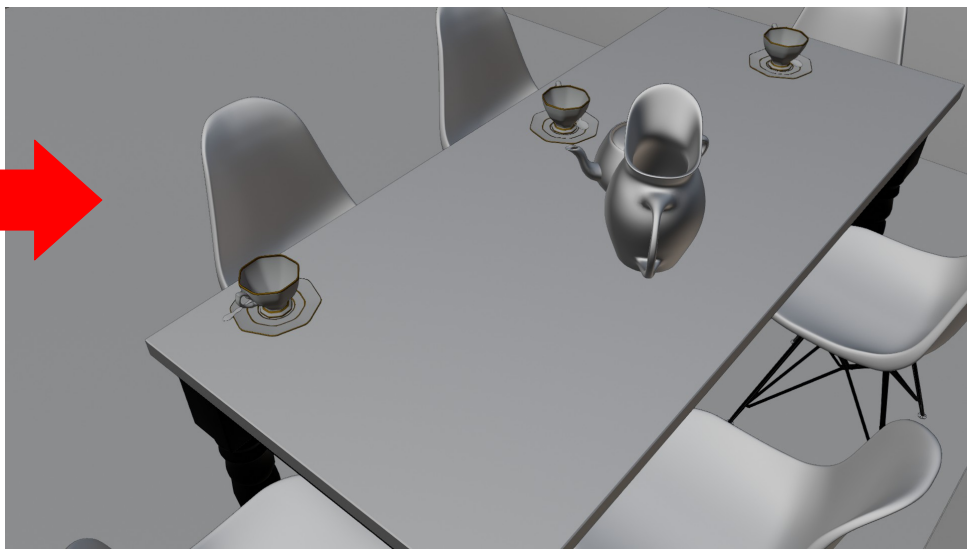


https://developer.nvidia.com/blog/rtx-best-practices/

Visibility

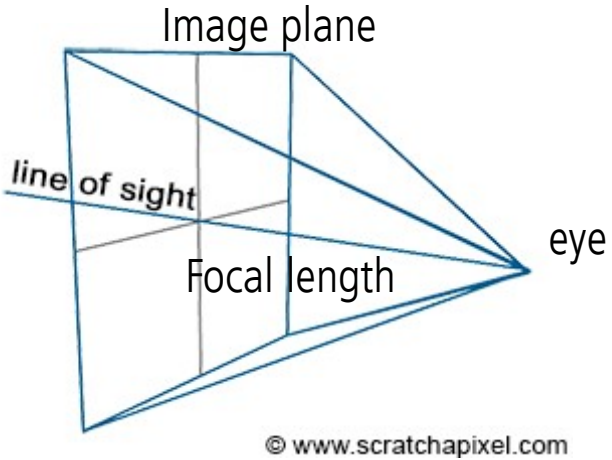


- **Visibility determines if two points are visible one to another**
- Often, used to determine **which objects are visible to camera**
  - Virtual camera is used to simulate human visual system → **perspective projection**

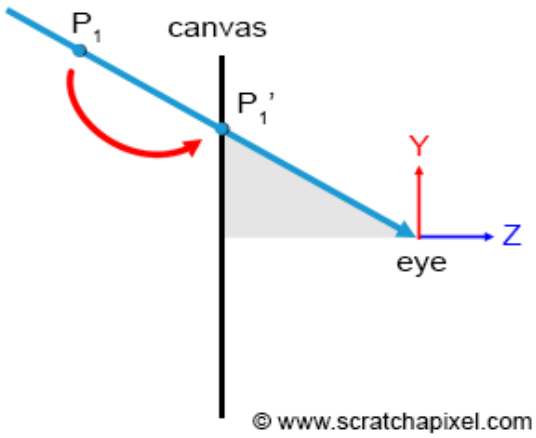


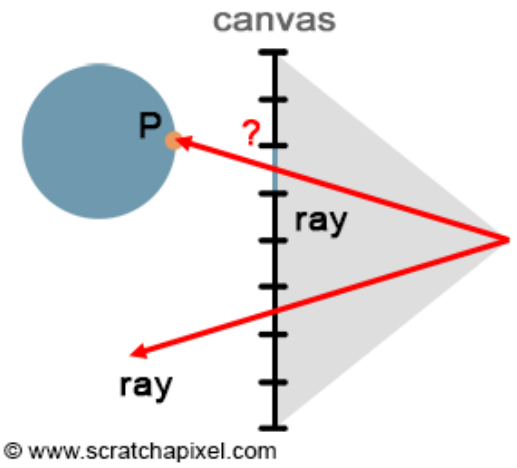
Perspective projection

- **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**



Visibility and projection



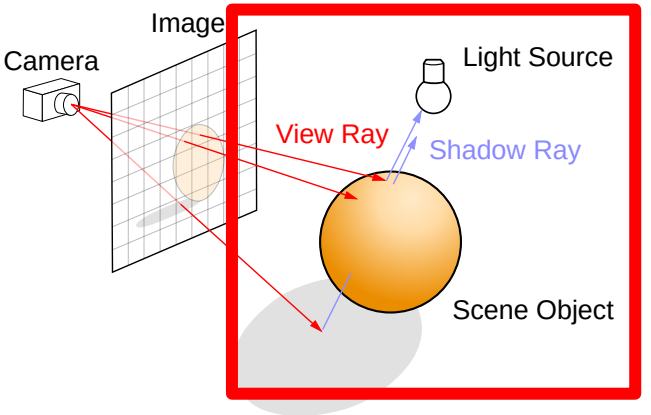


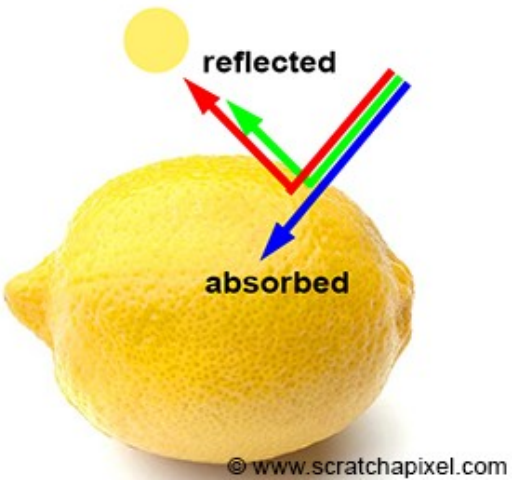
- **Rasterization-based** rendering projects object vertices on screen to solve visibility
  - Objects are transformed in camera space (world-to-camera matrix)
  - Objects are projected using perspective projection matrix

- **Ray-tracing based** rendering generates rays from camera to solve visibility
  - Rays are generated from camera aperture for each pixel
  - Perspective projection will be inherently present

Shading

- Shading calculates appearance (color) of objects visible to camera
- Color (hue) and brightness of objects is determined by:
  - Incoming light
    - **Light amount** (intensity) determines brightness of the object
    - **Light color**
  - Material
    - How light **reflects**
    - How much light is **absorbed** determines object color (hue)
    - Object can not reflect more light that it receives
    - Color of object: **ratio** of reflected light over amount of incoming (white) light → RGB in `[0,1]` (note that reflected light may be larger than 1)

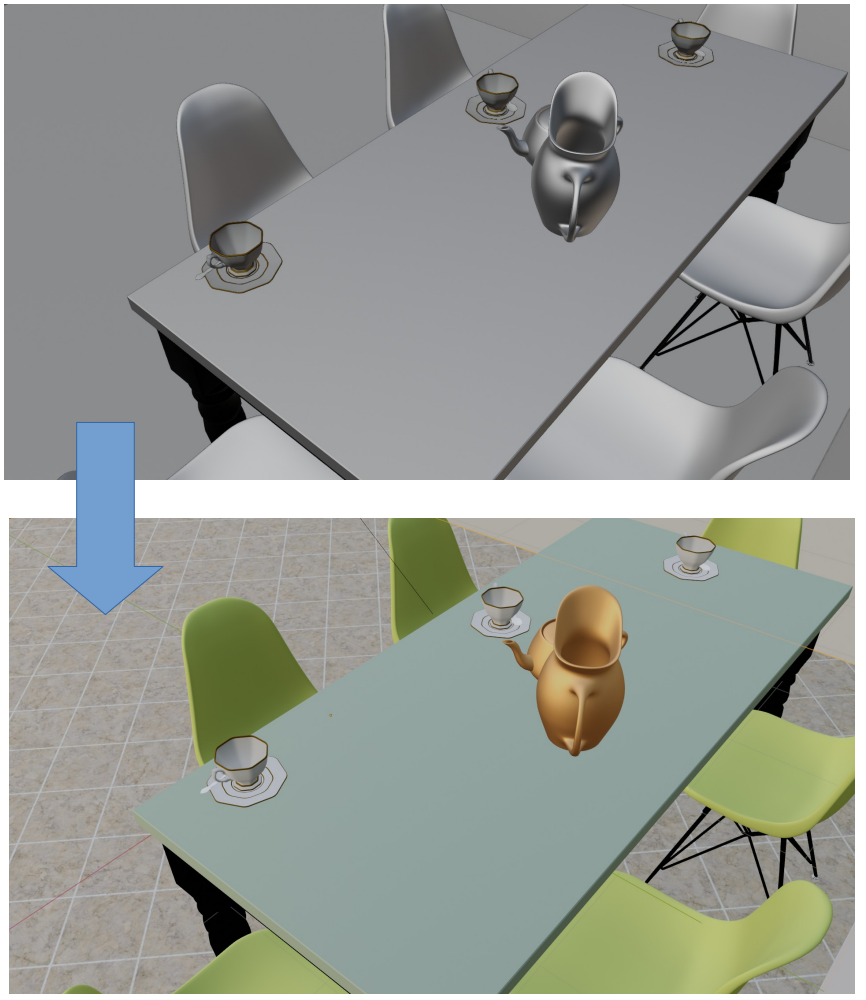




https://www.scratchapixel.com/lessons/3d-basic-rendering/rendering-3d-scene-overview/introduction-shading.html

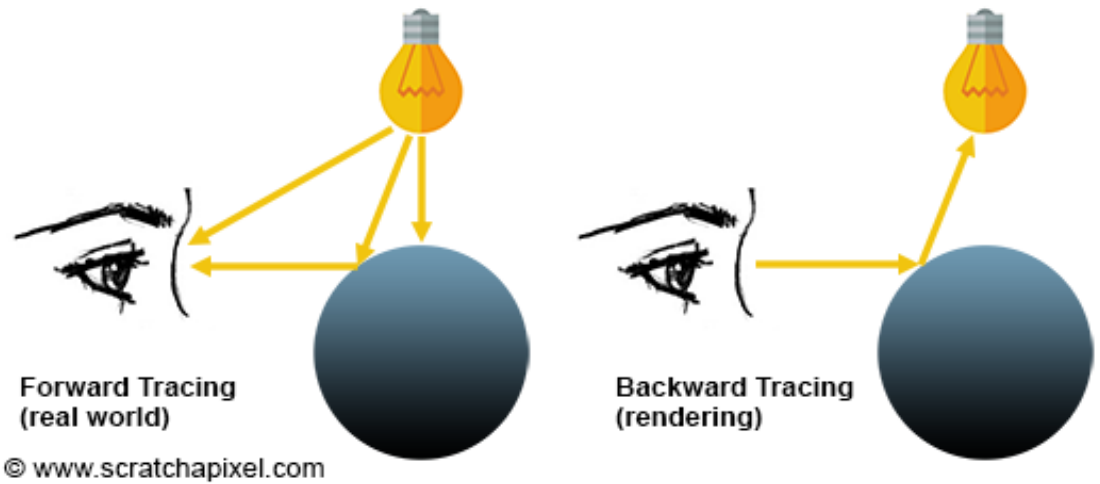
Shading

- **Mathematical model** describing **light-object interaction** is called **shader** and it depends on:
  - Object shape (e.g., normal)
  - Object material (e.g, diffuse, specular, etc.)
  - Viewing direction
  - Incoming light and its direction
- **Shading depends on amount of light falling on surface**
  - Visibility problem: which light sources and surfaces reflecting light are visible from shaded surface
  - This computation is called **light transport**



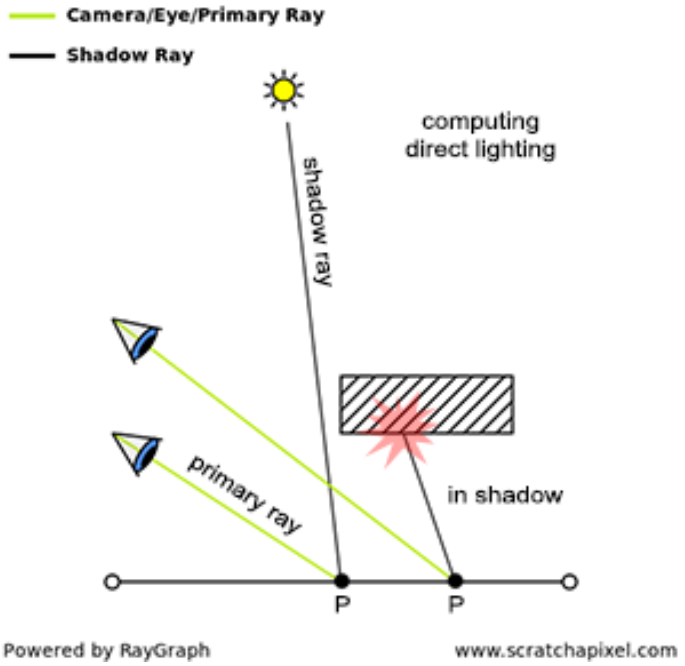
Light transport

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

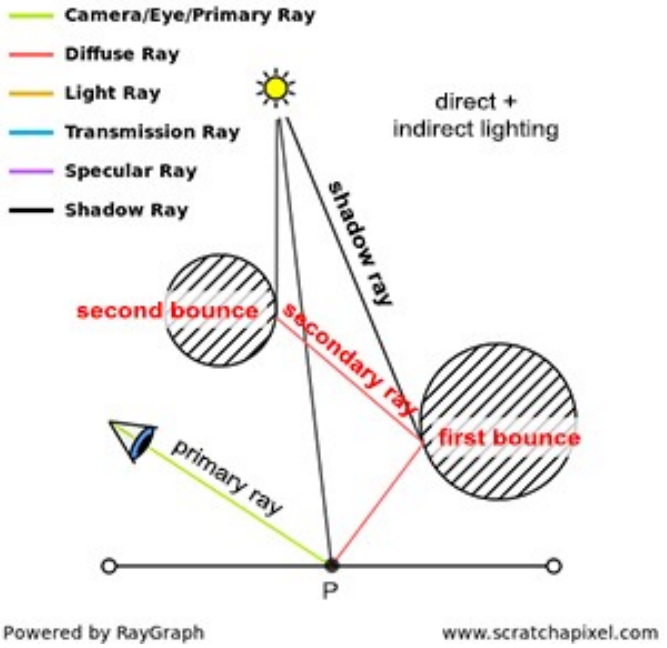


Light transport

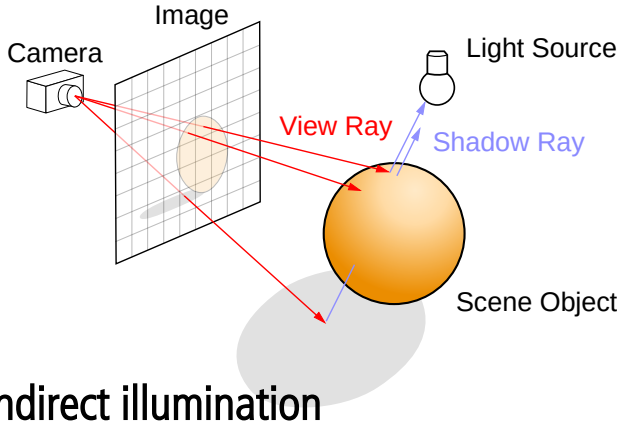
- Backward tracing: direct illumination
  - Only takes in account direct contribution from light source (shadow ray)



- Backward tracing: global illumination
  - Takes in account both direct and indirect light contribution - reflections from other surfaces (secondary rays)



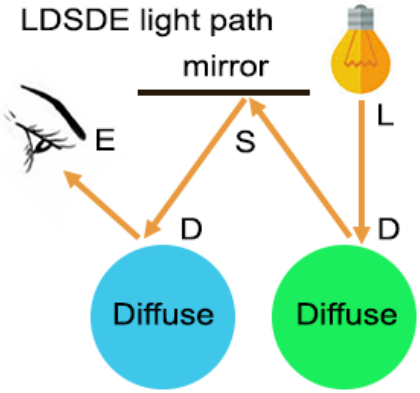
Light transport

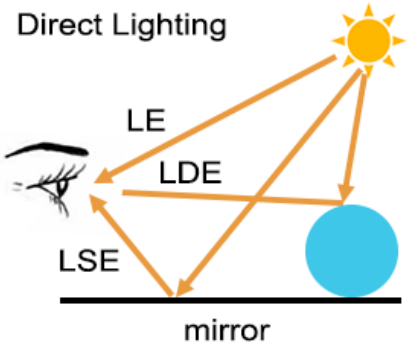


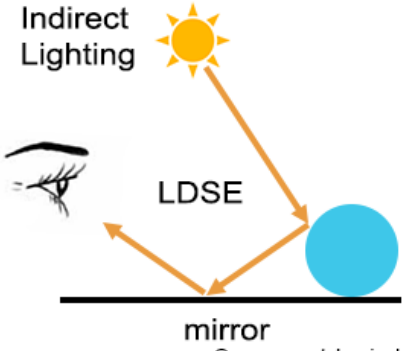
- **Relies on concept of visibility**
  - Which light sources are visible to shaded surface → **direct illumination**
  - Which surfaces are visible to shaded surface that may reflect light → **indirect illumination**
- **Light transport significantly determines the resulting image realism**

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**)
- Different light transport algorithms can be characterized using **path labeling**: L, E, D, S
  - Paul Heckbert: "Adaptive Radiosity Textures for Bidirectional Ray Tracing"
- **Rendering equation can be described as L(D|S)*E**
  - "*" - represents zero or more occurrences, "|" - represents either/or

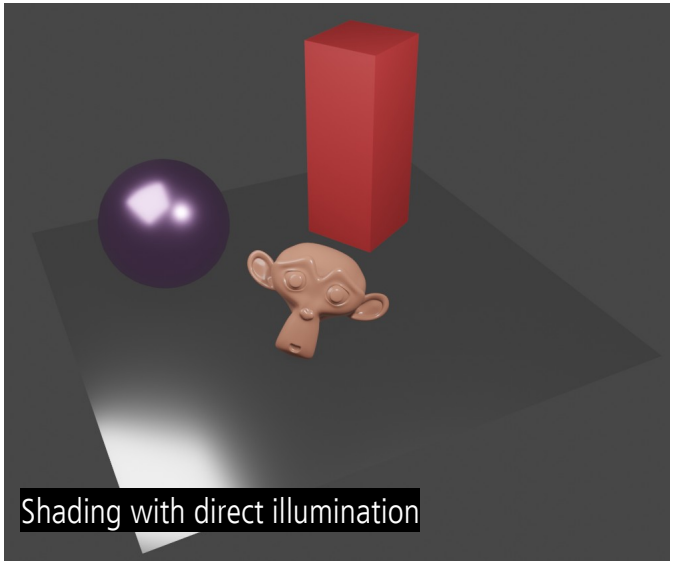


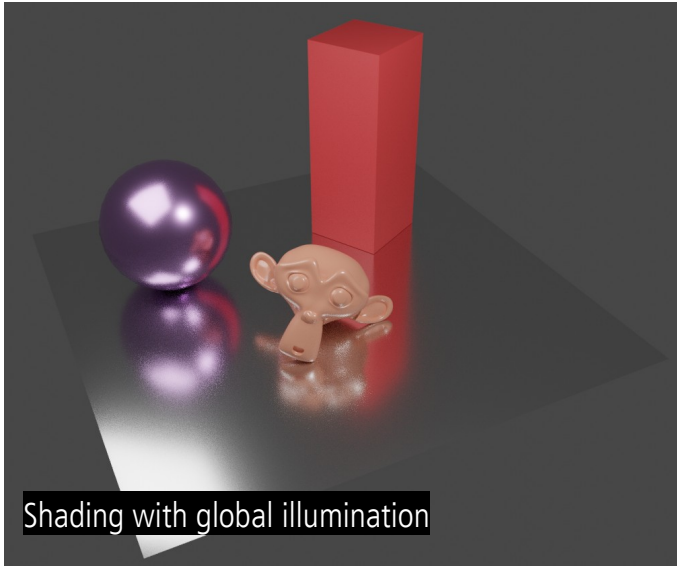




Shading and light transport

- Object appearance (color) depends on:
  - How much light falls on object and from which direction
  - How much light is absorbed and reflected into view direction
- **Shading**: light-object interaction computation
  - 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**





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**
  - foundation of physically-based rendering

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

Emission — Scattering → reflection

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

Emission

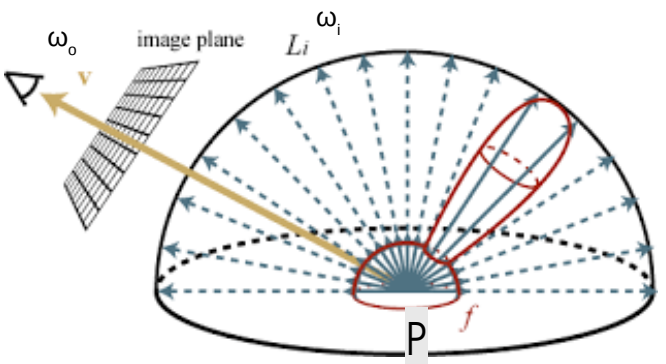
BRDF — Incoming light — Attenuation due to surface orientation

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

Rendering equation: incoming light

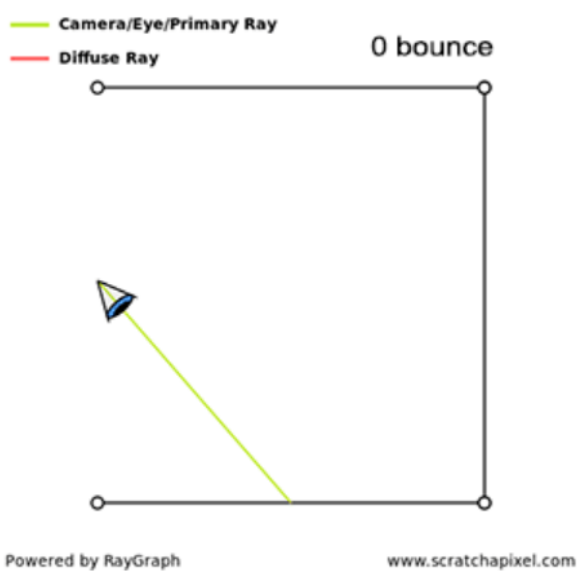
- Incoming light is possible from any direction (thus integration is needed) on surface point P
- Not possible to be solved analytically, solutions:
  - Simplify possible incoming directions: **direct illumination** – only look for light sources
  - **Approximate indirect illumination** by sampling smaller number of incoming light directions
    - Those represent directions from which other surfaces may contribute

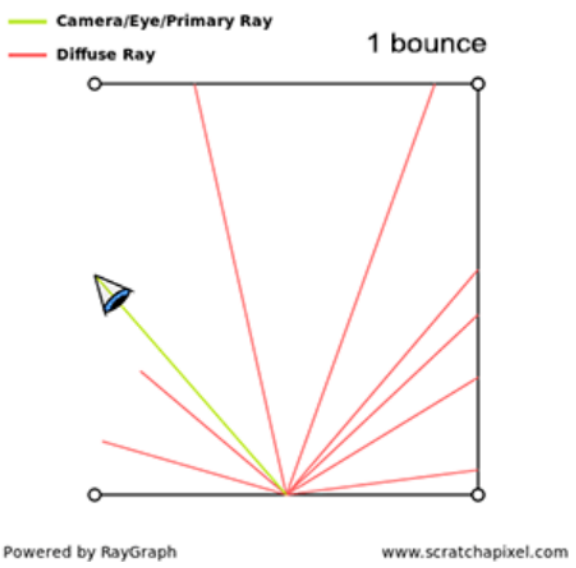
$$L_o(p,\omega_o) = L_e(p,\omega_o) + \int_\Omega f(p,\omega_o,\omega_i) L_i(p,\omega_i)(\omega_i \cdot n) d\omega_i$$

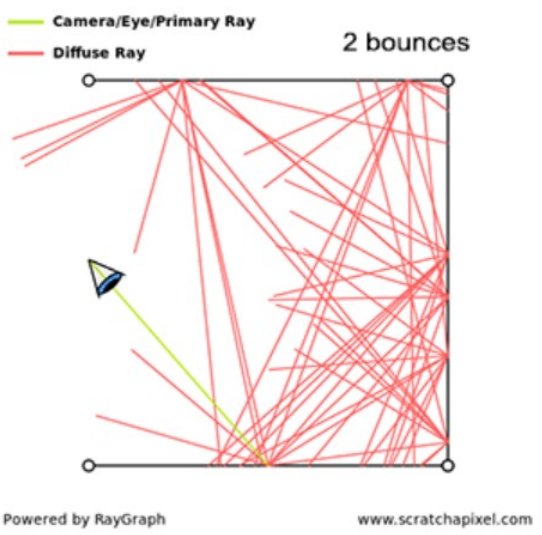


Incoming light: recursive nature

- For each light direction incoming from another surface, the rendering equation must be evaluated again to calculate amount of light falling on that surface → recursive equation





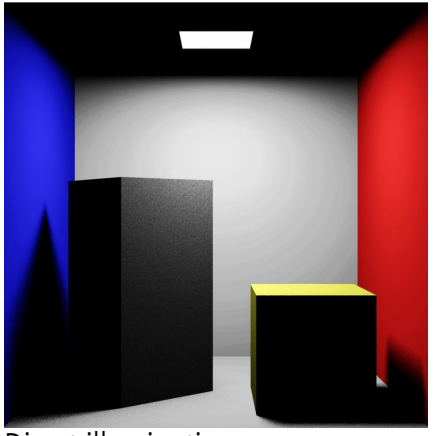


Light transport solvers

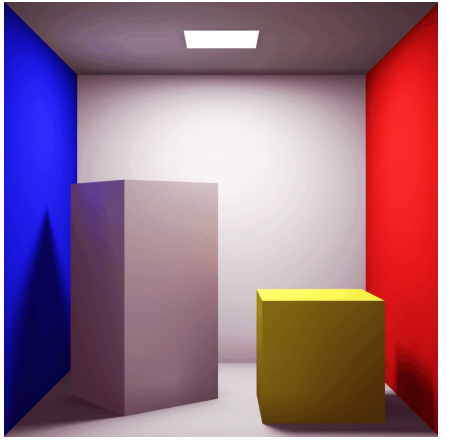
- Rendering equation is very general: describes wide range of light transport phenomena
  - Not possible to solve analytically
- Different methods compute incoming light with simplifications or approximations:
  - **Direct illumination:** usually done rasterization-based rendering
  - **Whitted ray-tracing:** consider only light sources, and light from specular reflection and transmission:
  - **Direct and indirect illumination:**
    - **Monte-Carlo methods based on ray-tracing → path tracing, bidirectional path tracing, metropolis light transport, etc.**
    - **Finite element methods → radiosity**

Radiosity

- First computer graphics technique to **simulate light transport on diffuse surfaces**
  - Assumption: **all indirect light is from diffuse surfaces**
  - Using Heckbert notation it can be described as **LD*E**
- Idea: turn on the light, light reaches equilibrium in the scene, consider each surface as light source
- It can compute **inter-reflections and soft shadows from area lights**



Direct illumination

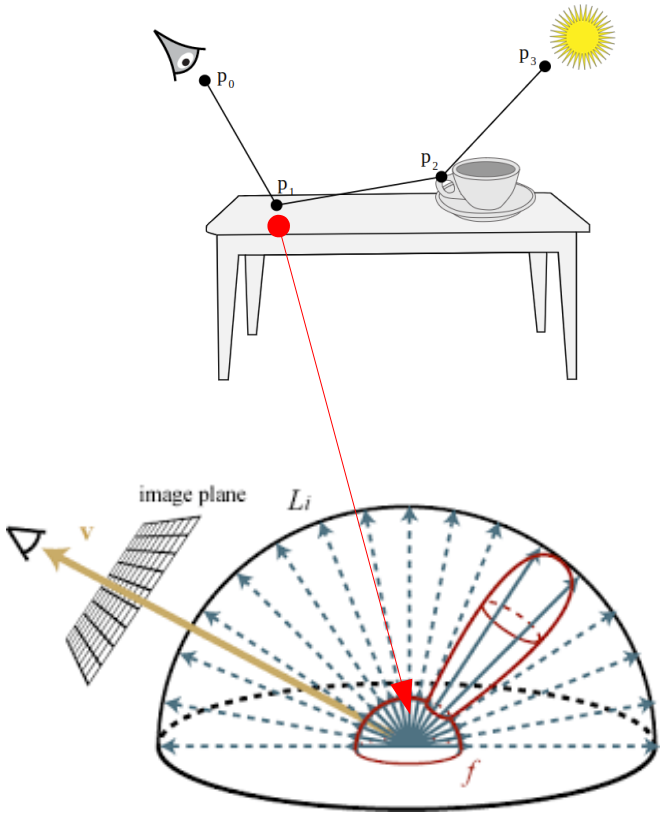


Radiosity

https://dl.acm.org/doi/10.1145/964965.808601
https://people.cs.kuleuven.be/~philip.dutre/GI/TotalCompendium.pdf
https://en.wikipedia.org/wiki/Radiosity_%28computer_graphics%29

Path-tracing

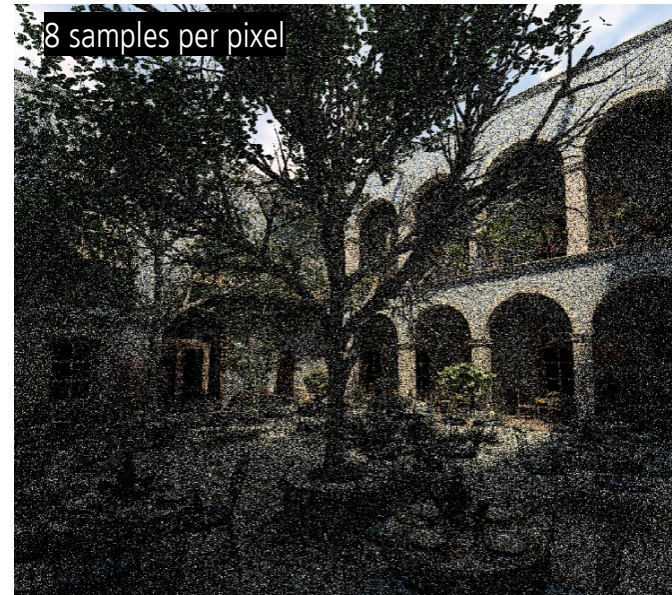
- Principles of ray-tracing: shooting rays and evaluating how much light they carry is used to solve full rendering equation
  - Random rays are generated across hemisphere and traced: Monte-Carlo sampling
  - For each intersection, rendering equation is evaluated again (recursive nature)
  - As ray bounces across scene, a path is built
    - Light carried along each path represents one evaluation of rendering equation
- Used to solve specular (glossy) or diffuse surfaces: L(D|S)*E
- Ensure rendering of soft shadows, transparent objects and caustics
- Can be extended for handling participating media and sub-surface scattering



Offline path tracing: https://pbrt.org/

Path-tracing

- Problem of path-tracing is high variance → noise in the rendered image
- High visual fielity (more correct estimate of rendering equation) is achieved with:
  - Large number of rays per pixel
  - Large number of traced paths
- Solutions:
  - Importance sampling – rays are traced in direction where most of the light comes
  - Denoising rendered image



Monte-Carlo light transport: http://graphics.stanford.edu/papers/veach_thesis/
Rendering equation: https://www.cs.cmu.edu/afs/cs/academic/class/15462-s13/www/lec_slides/86kajiyaRenderingEquation.pdf

Noise

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

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

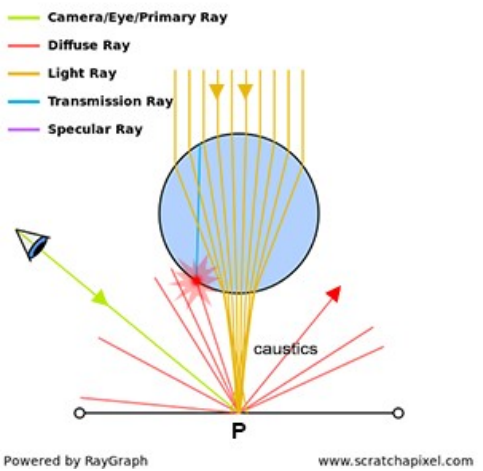
Practical: ray-tracing and denoising

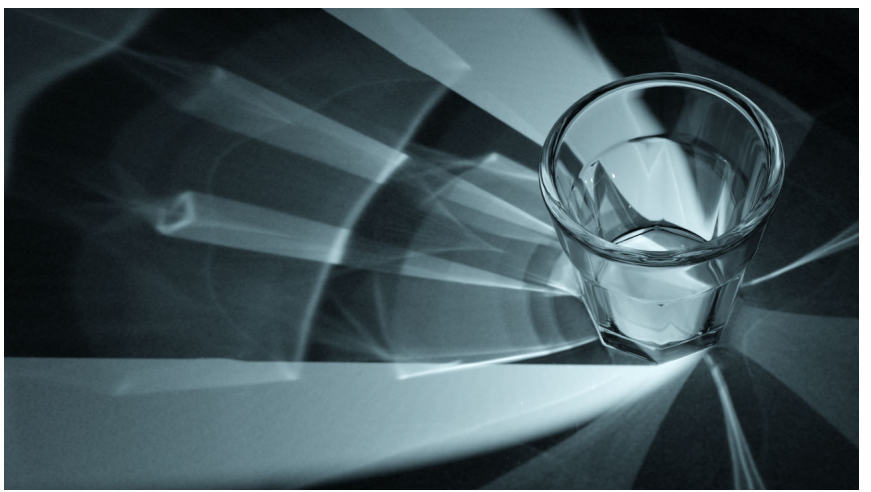
Denoising is post-processing step, which means it is applied on rendered image:

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

Light transport complexity example: caustics

- Light transport depends on different types of surfaces which reflect light differently
- Example: hard problem are specular surfaces (S) illuminating diffuse surfaces (D): **LSDE**
  - Foward tracing: light is emitted, refracted through glass object and due to refraction, light is concentrated towards few points → **caustics**
  - Solving via backward tracing: start from eye, intersect diffuse surface and try to find incoming direction → since diffuse surfaces reflect in all direction and specular surface resulted in concentrated set of directions it is hard to find incoming light





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

Practical rendering

Ray-tracing and Rasterization

- **Two main rendering approaches are based on: Ray-tracing and 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.

Visibility solvers

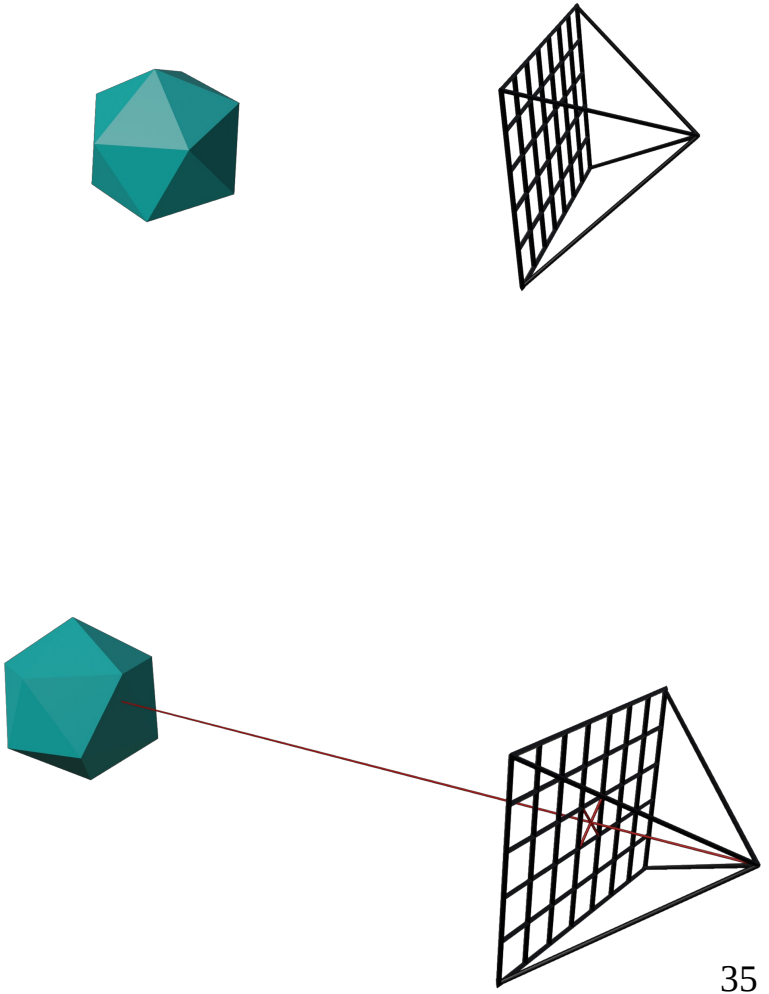
- **Rasterization and ray-tracing are methods for solving visibility**
- Both methods solve **visibility from camera**
- **Ray-tracing can be further used to solve visibility between any points in 3D scene**
  - particularly useful for shading and light transport
- **Global illumination effects** are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

Visibility: ray-tracing

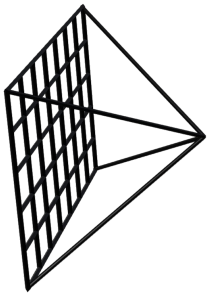
Image centric approach:

- 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

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



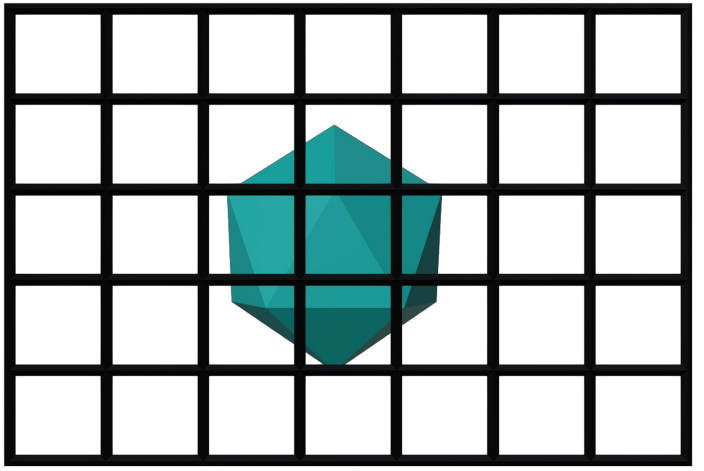
Visibility: rasterization



Object centric approach:

- Objects are transformed into camera space (world-to-camera matrix)
- Camera information is used to construct (perspective) projection matrix
- Objects are projected on virtual image plane using projection matrix
- For each pixel of virtual image plane find which objects are covered by pixel and take closest

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



Importance of visibility computation

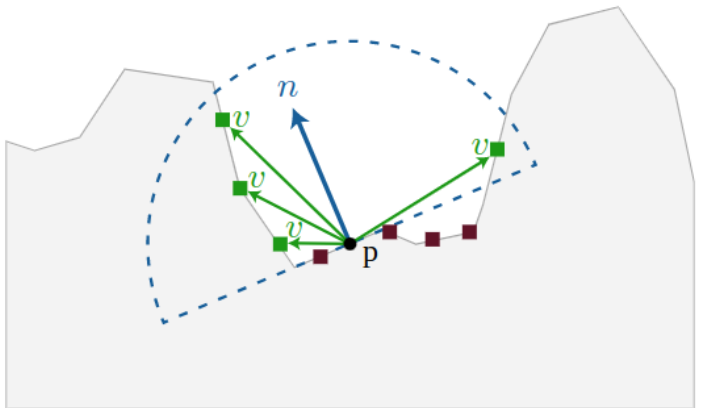
- **Visibility**: determine if two points in 3D scene are visible to each other
  - **Objects visible from camera**: solve visibility problem using perspective/orthographic projection
  - **Shading, that is, light transport** is based on visibility between surface and lights in 3D scene
    - This is required to calculate shadows, soft shadows, global illumination effects such as reflection, refraction, indirect reflection and.
- Rasterization is solving the visibility from camera to object very fast. But it is not good for finding visibility between surfaces in 3D scene which is important for light transport and shading
  - Real-time rendering is predominantly done using rasterization approach on GPU
- Ray-tracing 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.
  - Both steps require extremely time-consuming intersections between rays and scene objects (geometry)

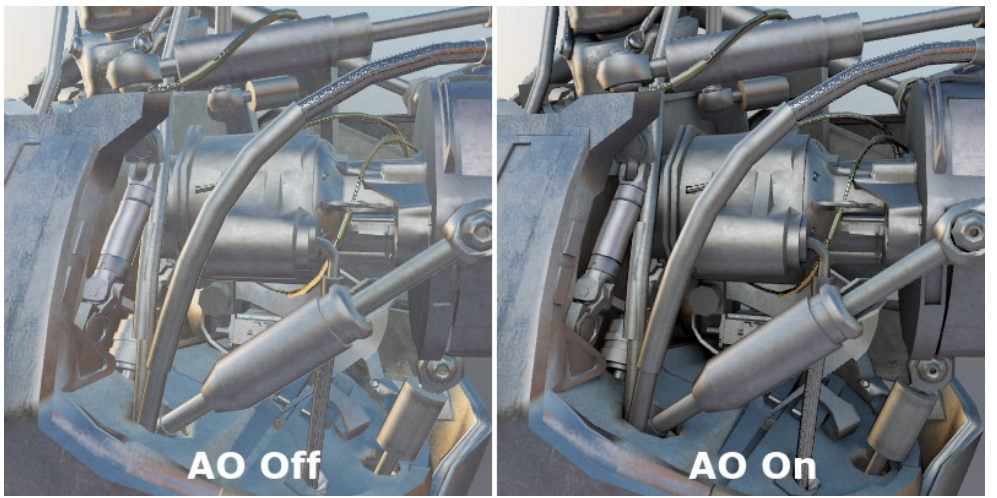
Practical tip: Ray-tracing vs Rasterization

- Important difference is that **ray-tracing** can shoot rays in any direction, not only from eye or light source
  - This makes it possible to **render reflections, refractions and approximate rendering equation**.
- Content creation is simpler for ray-tracing than rasterization since for realistic effects are just present
  - Additional artist work is required when rasterization-based rendering is used

Ambient occlusion: global illumination approximation

- Simplest approximation of global illumination possible for rasterization-based rendering
- Idea: some incoming light directions may be blocked by other parts of the object
  - Blocked light will have lower or zero color and intensity





https://learnopengl.com/Advanced-Lighting/SSAO
http://frederikaalund.com/a-comparative-study-of-screen-space-ambient-occlusion-methods/
https://developer.playcanvas.com/en/user-manual/graphics/lighting/ambient-occlusion/

Other global illumination approximations

- Ambient occlusion gives poor approximations for visibility when dealing with large area light sources or small point light sources → **directional occlusion** methods are used and can be categorized into:
  - Precomputed, e.g., using spherical sign distance functions*
  - Dynamic, e.g, using cone tracing**
- Light can be pre-computed using path-tracing or radiosity
  - Works fine for static objects
- Dynamic computation of indirect light with simplifications:
  - Only diffuse or specular surfaces**
  - Example: voxel cone tracing**

Interactive indirect illumination using voxel cone tracing: https://research.nvidia.com/sites/default/files/publications/GIVoxels-pg2011-authors.pdf

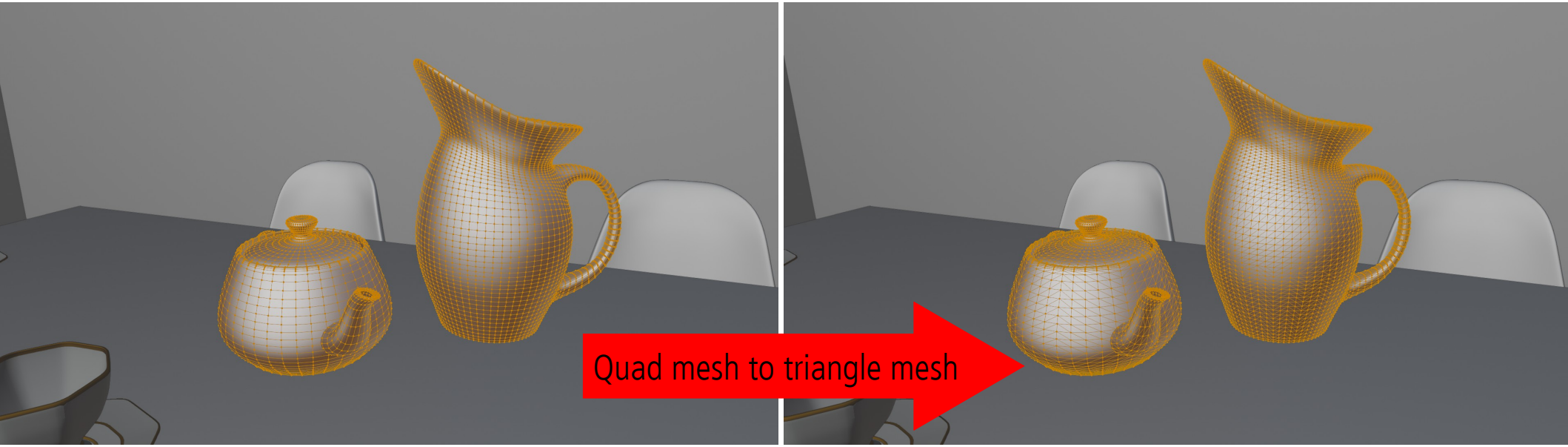
* https://www.microsoft.com/en-us/research/wp-content/uploads/2009/12/sg.pdf
** https://godotengine.org/article/vulkan-progress-report-5/
**https://bc3.moe/vctgi/

Back to visibility: shape representation

- **Raytracing** iterates through all pixels of image plane, generates rays and then iterates over all objects (triangles) in the scene.
- **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.

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.



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

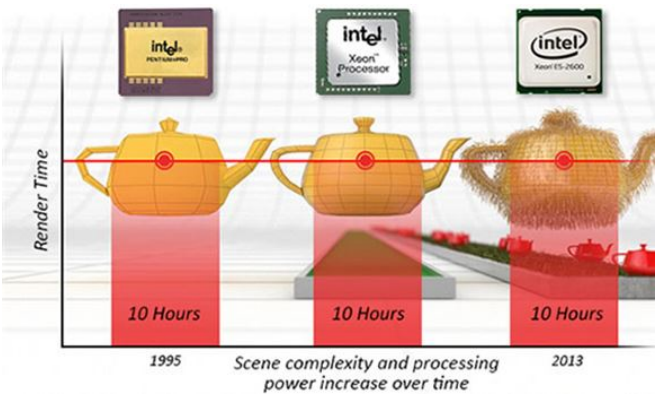
https://www.pbr-book.org/3ed-2018/Primitives_and_Intersection_Acceleration

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
- **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 time: **Blinn's law**

- Rendering time tends to remain constant although GPU hardware gets faster
- This is because complexity of the 3D scenes as well as rendering algorithms increased



https://en.wikipedia.org/wiki/Jim_Blin

Speed: Ray-tracing vs Rasterization

- Both algorithms are conceptually simple
- To make them fast and usable in practice:
  - **Ray-tracing** is utilizing **acceleration spatial data structures** for achieving running time of `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
    - GPUs are adapted for rasterization-based rendering
    - Deferred shading*: shading only points which will end on screen

`n` – number of primitives, e.g., triangles

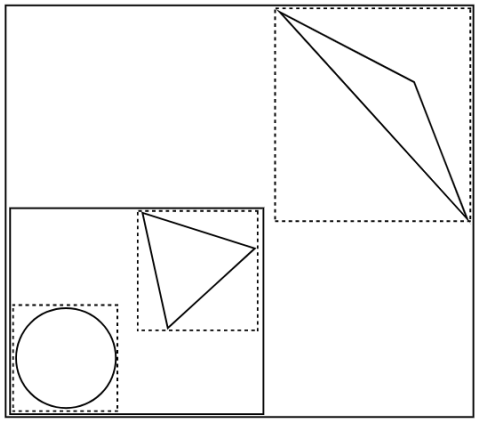
* https://learnopengl.com/Advanced-Lighting/Deferred-Shading

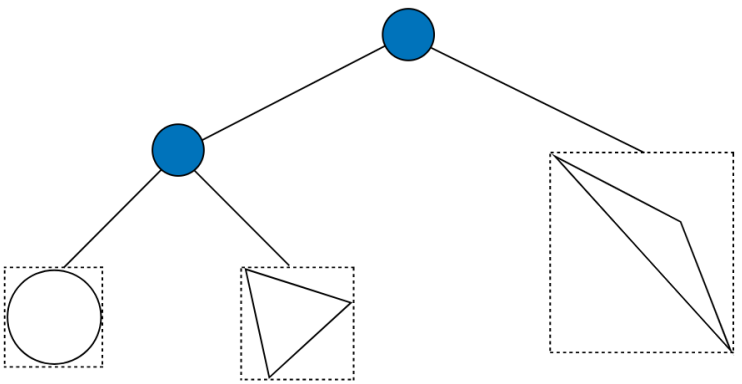
Ray-tracing acceleration

- 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

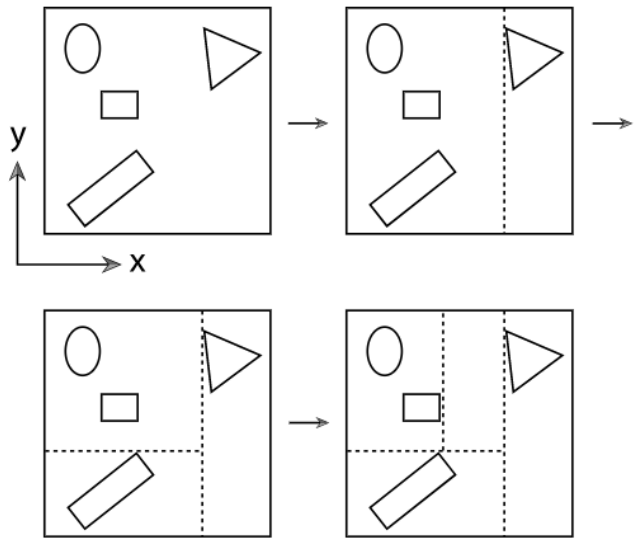
https://www.pbr-book.org/3ed-2018/Primitives_and_Intersection_Acceleration/Bounding_Volume_Hierarchies

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

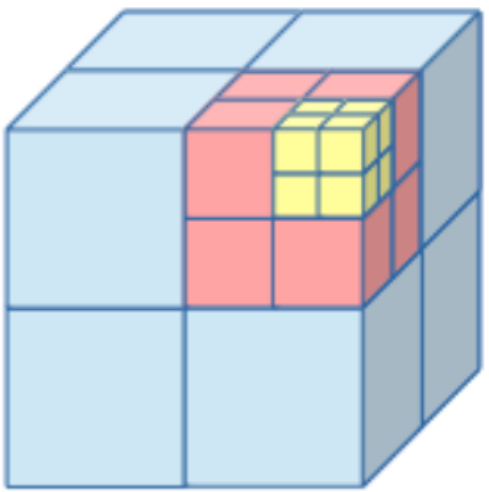
Spatial subdivisions: kd-trees

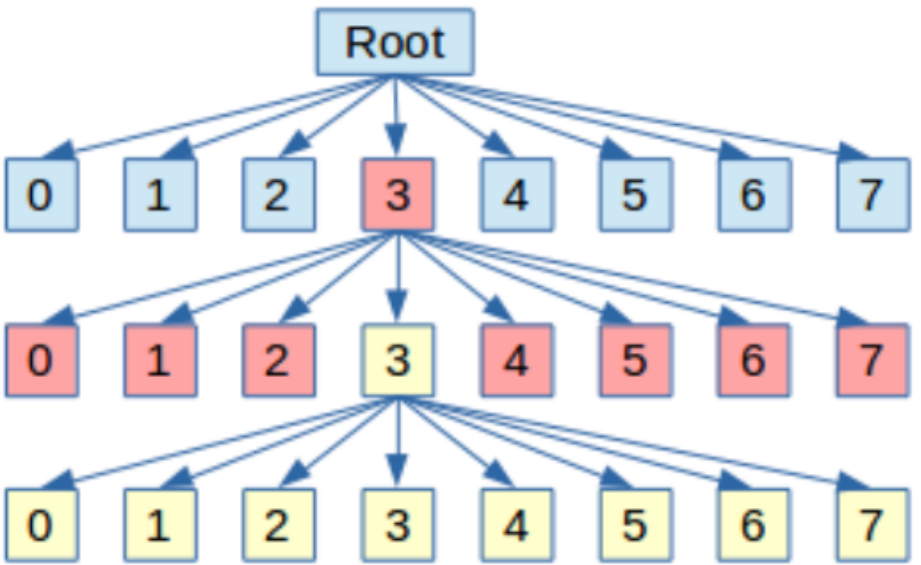
- Subdivision rule: planes must be perpendicular to on of coordinate axes



Spatial subdivisions: octrees

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

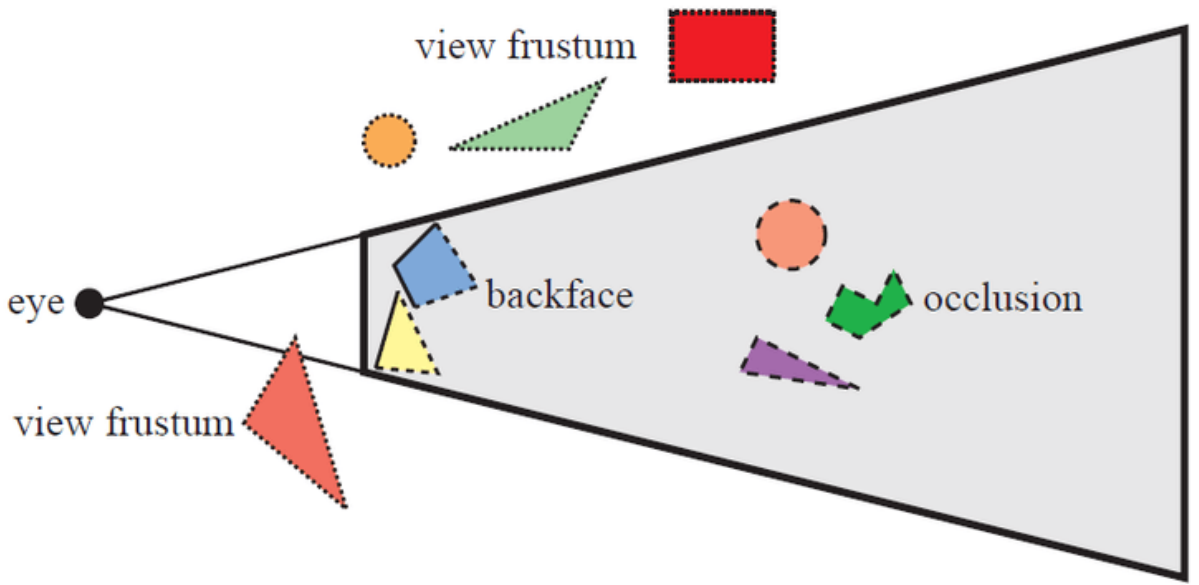




https://geidav.wordpress.com/2014/07/18/advanced-octrees-1-preliminaries-insertion-strategies-and-max-tree-depth/

Rasterization acceleration

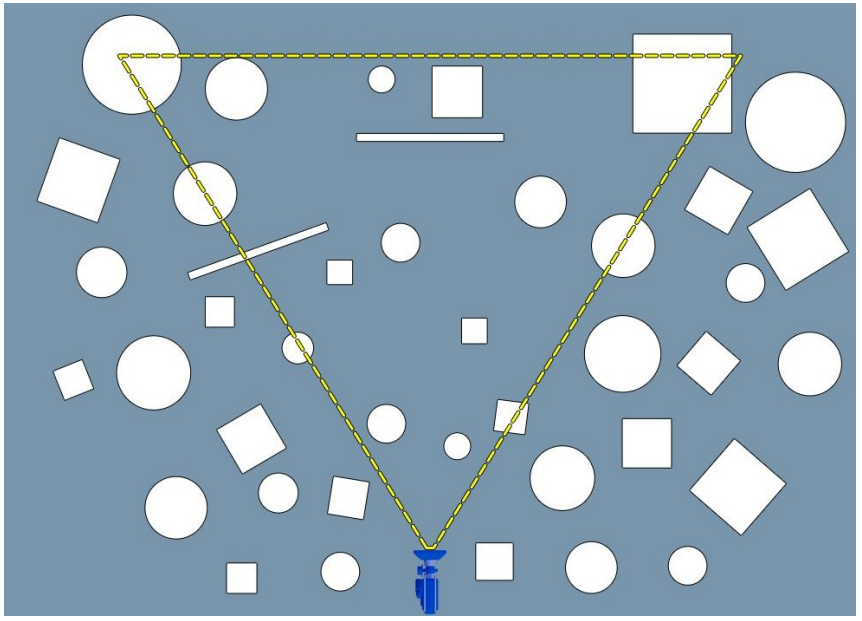
- Whole scene processing is avoided using:
  - Frustum culling
  - Occlusion culling
  - Back face culling

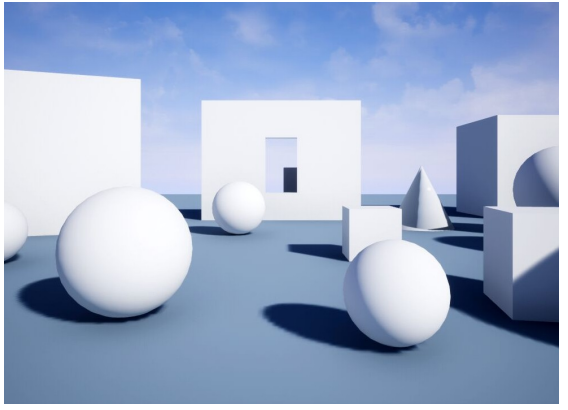


https://bruop.github.io/frustum_culling/

Frustum culling

- Instead of processing all triangles, determine only those visible to camera frustum

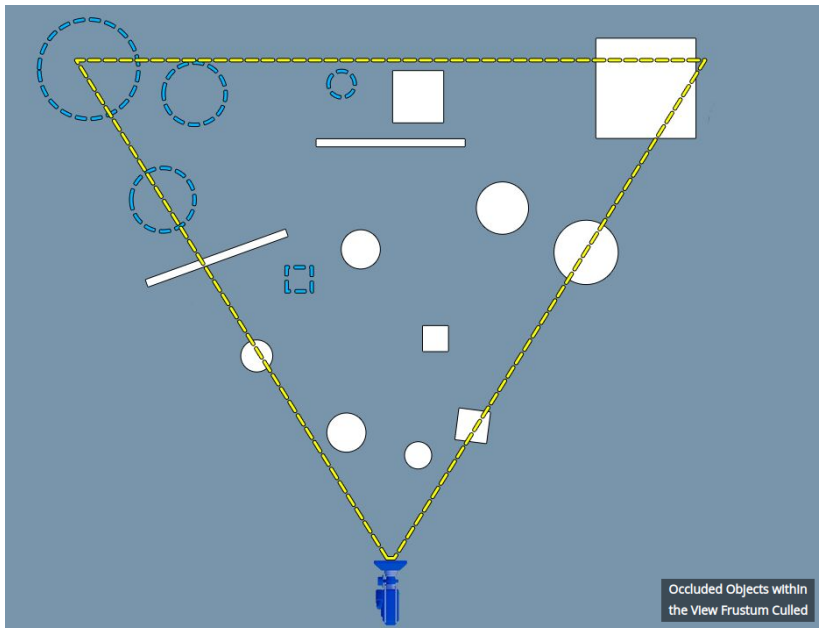


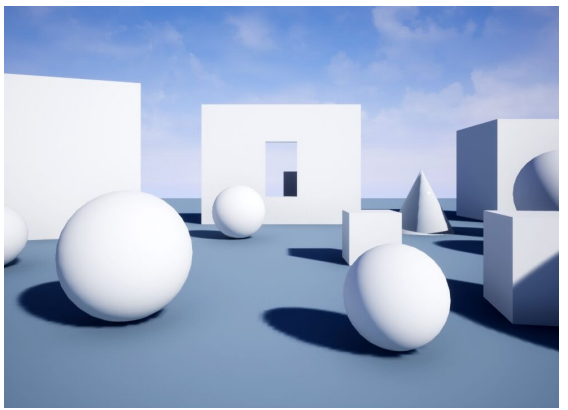


https://learnopengl.com/Guest-Articles/2021/Scene/Frustum-Culling
https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/VisibilityCulling/

Occlusion culling

- Objects which are occluded by other objects from current camera view are not rendered.

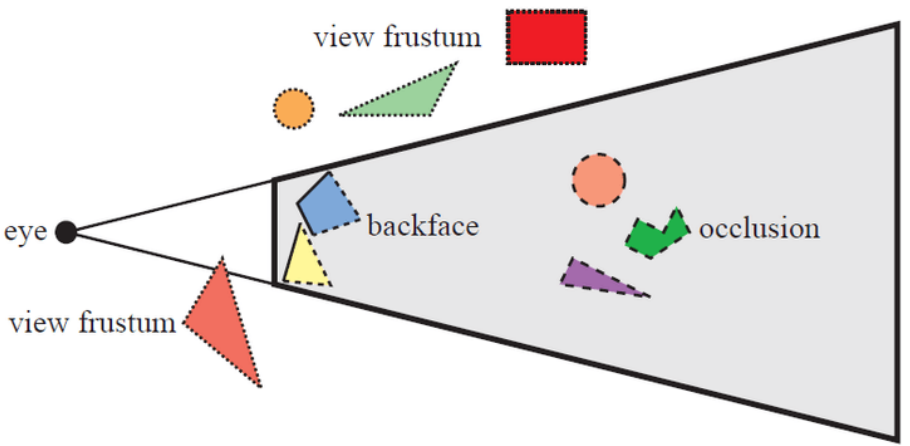




https://www.gamedeveloper.com/programming/occlusion-culling-algorithms
https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/VisibilityCulling/

Back face culling

- Determine if triangle is visible from camera's point of view
  - Front facing triangles are rendered
  - Back facing triangles are discarded
- Front or back facing triangles are determined using triangle normal or vertex winding order



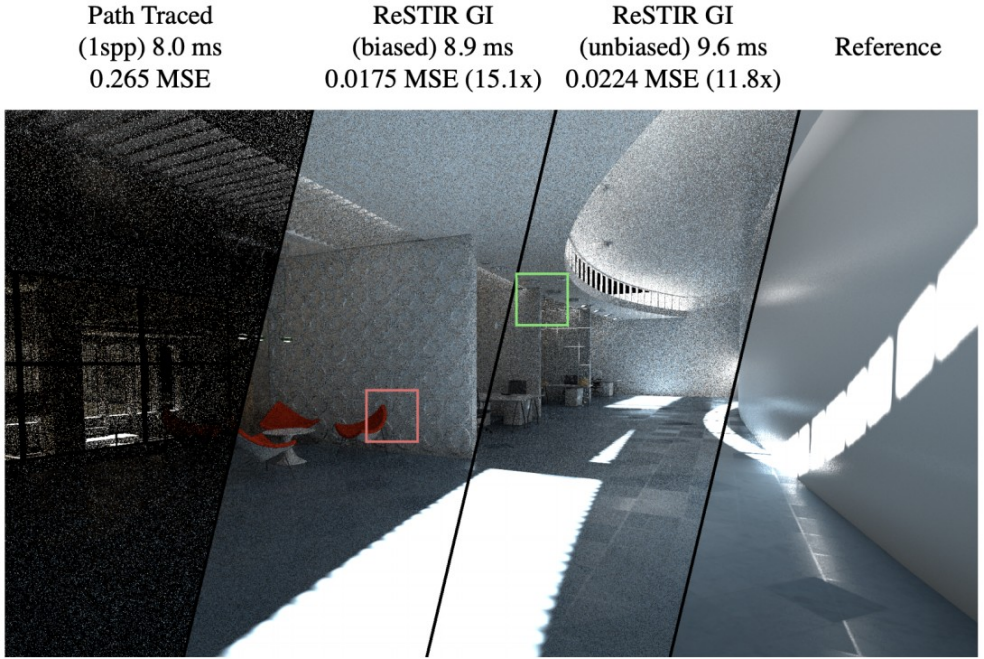
https://learnopengl.com/Advanced-OpenGL/Face-culling

Rasterization and Raytracing in practice

- **Rasterization-based renderers: interaction and games**
  - Godot: https://docs.godotengine.org/en/stable/tutorials/rendering/index.html
  - Unity: https://unity.com/srp/High-Definition-Render-Pipeline
  - Unreal: https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/Overview/
  - Vulkan-based renderers: https://vulkan.org/tools#engines
- **Examples of raytracing-based renderers: animation and VFX**
  - Renderman: https://renderman.pixar.com/
  - Arnold: https://arnoldrenderer.com/
  - Appleseed: https://github.com/appleseedhq/appleseed

Real-time ray-tracing*

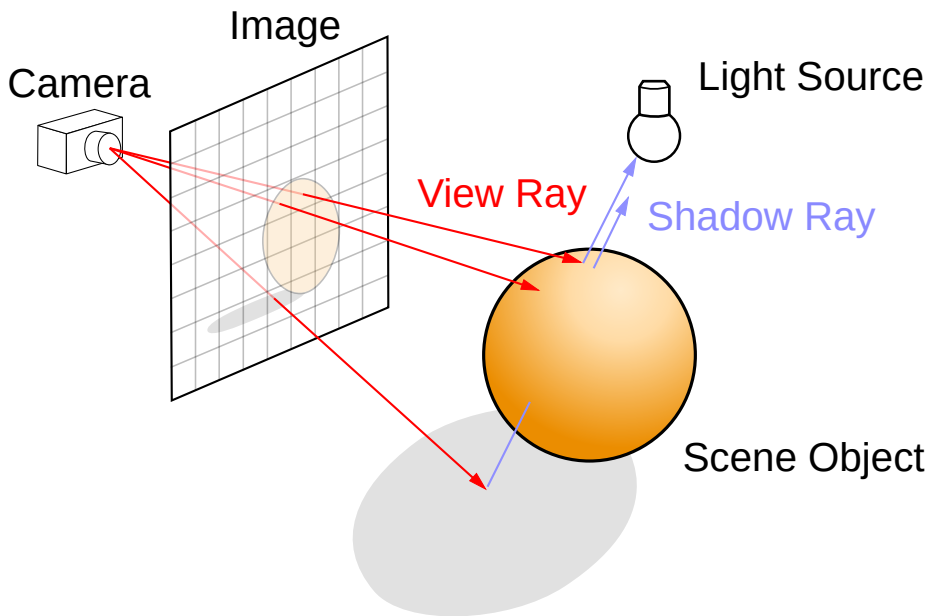
- GPU hardware is optimized for rasterization-based rendering enabling real-time graphics
- Graphics hardware is being developed for ray-tracing support
  - https://developer.nvidia.com/rtx/ray-tracing
  - https://www.khronos.org/blog/ray-tracing-in-vulkan
  - https://learn.microsoft.com/en-us/samples/microsoft/directx-graphics-samples/d3d12-raytracing-samples-win32/
- Generating large number of rays per pixels with long paths is still expensive → noise
  - Denoising is promising solution enabling real-time ray-tracing
  - https://www.youtube.com/watch?v=NRmkr50mkEE&ab_channel=TwoMinutePapers
  - https://research.nvidia.com/publication/2021-06_restir-gi-path-resampling-real-time-path-tracing
- For short-term, clever combinations of rasterizations and ray-tracing are used
  - Example: ray-tracing for shadows
  - https://docs.unrealengine.com/4.26/en-US/RenderingAndGraphics/RayTracing/
  - https://unity.com/ray-tracing



Rendering: intuition via ray-tracing

Ray-tracing based rendering

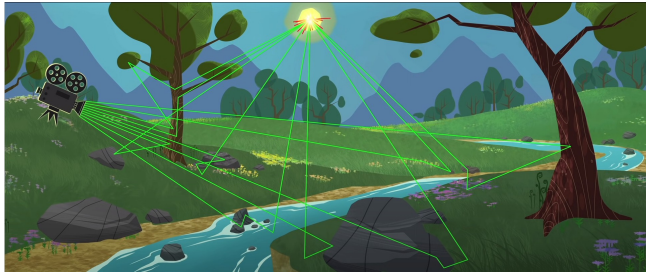
- Rendering: process of creating 2D image from 3D scene
- Task: calculate pixel colors of virtual image plane of virtual camera in 3D scene
  - This implies computing light falling into camera



Simulating light

- Simulating real-world light-object interaction means simulating all light rays paths from light and their interactions with objects
  - Only small amount of that light actually falls on camera forming an image
  - This kind of simulation is called **forward ray-tracing or light tracing**
  - Too expensive!

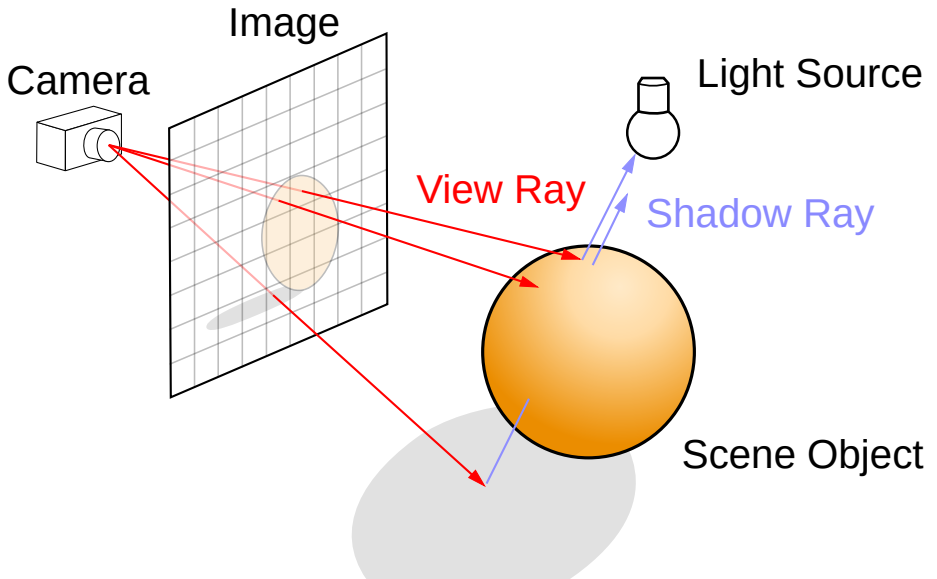
- It is enough to compute only light which enters the camera
  - Therefore, rays are traced from sensor to the objects which may reflect light into camera
  - This simulation is called **backward ray-tracing or eye-tracing***



* Introduced by Turner Whitted in paper "An Improved Illumination Model for Shaded Display".

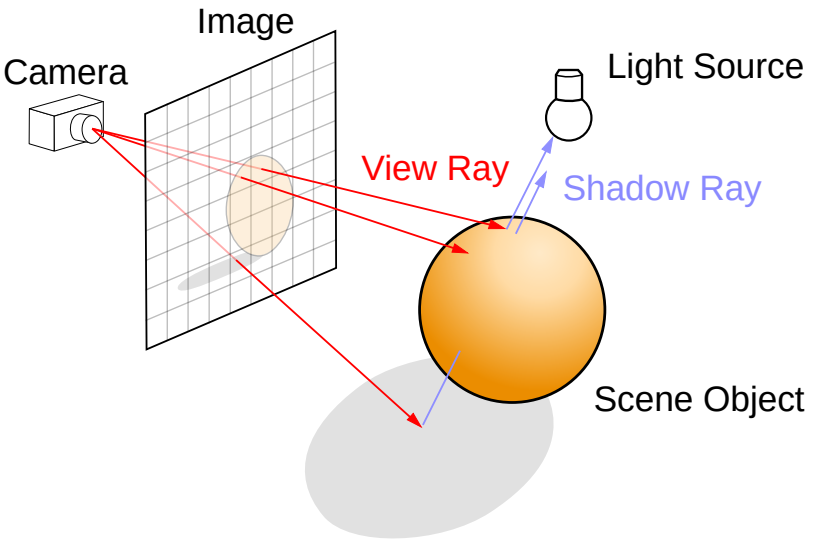
Pixel colors → camera rays

- Camera defines from where we look at 3D scene and a 2D surface on which 3D scene will be projected as an image
- Compute only light rays which are contributing to virtual image plane: **camera or view rays**
  - Start from virtual camera and generate camera rays for each pixel of virtual image plane into the scene
  - Camera rays are generated from aperture position through each pixel of film plane
- The color (light) which will be computed for each ray will correspond to pixel color from which this ray was generated



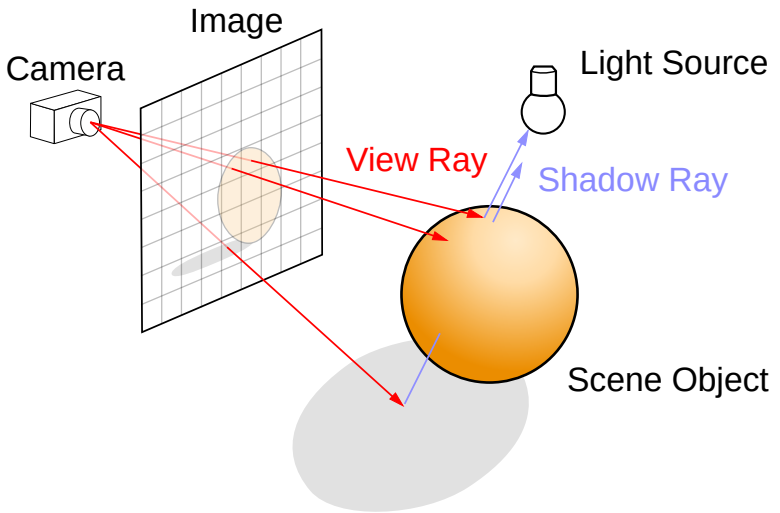
Camera ray intersections

- **To compute color (light) for each camera ray, find closest intersection of this ray and objects in 3D scene**
- This way object closest to camera, that is, **objects visible to camera** which will reflect light into camera are found
- Ray-object intersection is geometrical problem and it **depends on object shape representation**
- Although magnitude of different shapes exist, we can assume that in rendering time we will have **triangulated mesh**



Intersection point color → shading

- Once intersection with object is found we need to find amount of light reflected from this point into camera
  - This is process is generally described with **rendering equation**
- Therefore, **task is to calculate object color taking in the account:**
  - Current view ray direction on shading point
  - Incoming light ray direction and color on shading point → light sources and/or other surfaces
  - Object material and shape (normal) in shading point
- Intersection point can be now also called **shading point**



Shading

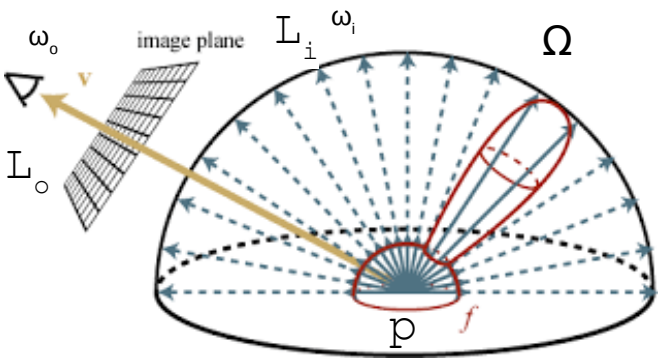
- Amount of light reflected in view direction, that is color of visible surface is defined with **rendering equation**
- Computing incoming light on shading point is called **light transport**

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

BRDF
- Defines surface material
- Uses texture information

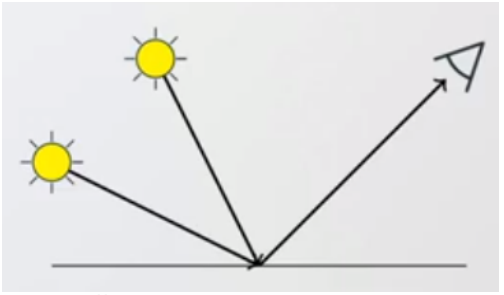
Incoming light over hemisphere of directions

Attenuation due to orientation of surface towards light.
- Depends on surface shape (normal)

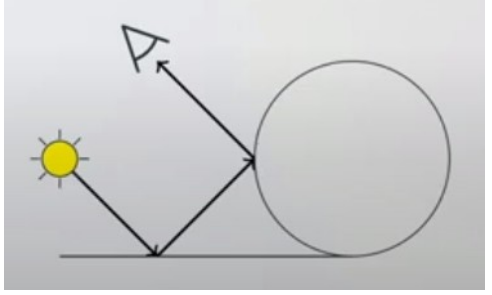


Light transport: incoming light

- **Direct illumination**: calculating visibility between shading point and light source
- **Indirect illumination**: calculating visibility between shading point and all possible surfaces above shading point which may contribute with light
  - Additional rays are generated from shading point into hemisphere of directions above surface
  - For any found intersection, rendering equation must be re-evaluated
  - Material determines determines direction and amount of light scattering
  - Different light transport algorithms simulate different light paths: diffuse, specular, etc.
- **Global illumination**: direct and indirect illumination
  - Path-tracing represents method for solving global illumination which extends ray-tracing



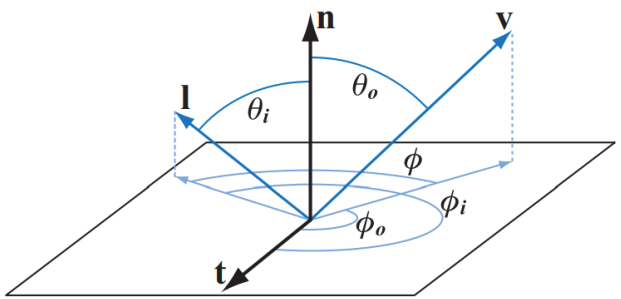
Direct illumination



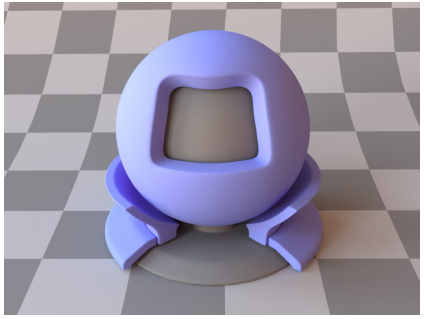
Indirect illumination

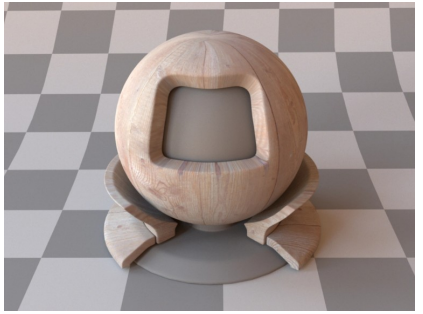
Shading: light-matter interaction

- Once incoming light on shading point is computed, material information is used to determine amount of light being absorbed and reflected in view direction
- For this purpose, BRDF is used
  - Given light `(l)` and view `(v)` directions, BRDF determines ratio of reflected light in view direction
  - BRDF further uses texture parameters which varies with shading point position



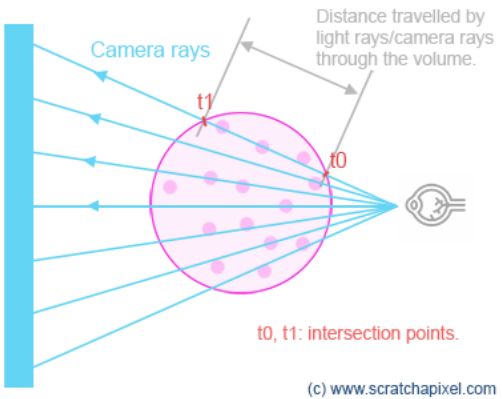
BRDF notation

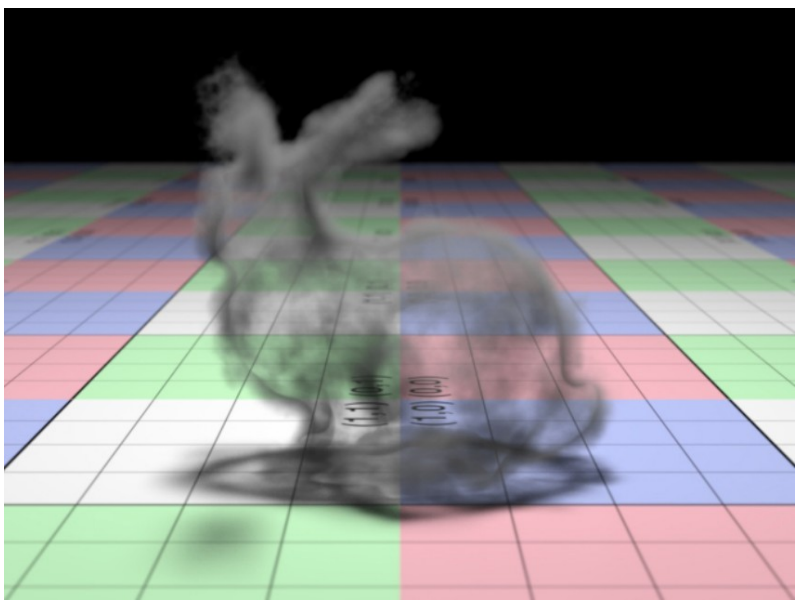

Diffuse BRDF


Diffuse BRDF with texture

Participating media*

- We assume that media between object surfaces is vacuum
  - In this case, light which is reflected from object or incoming on object will not be attenuated
- If other medium is present then light will be attenuated by scattering and absorption
  - This is called **participating media**
  - To solve this, **volumetric rendering equation** is used

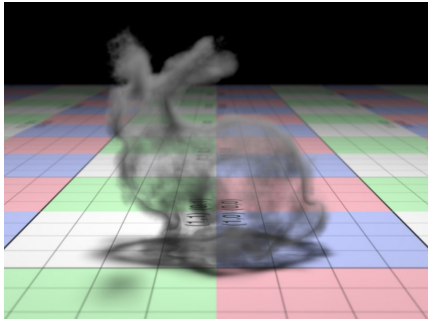




https://graphics.pixar.com/library/ProductionVolumeRendering/paper.pdf
https://www.scratchapixel.com/lessons/3d-basic-rendering/volume-rendering-for-developers/intro-volume-rendering.html

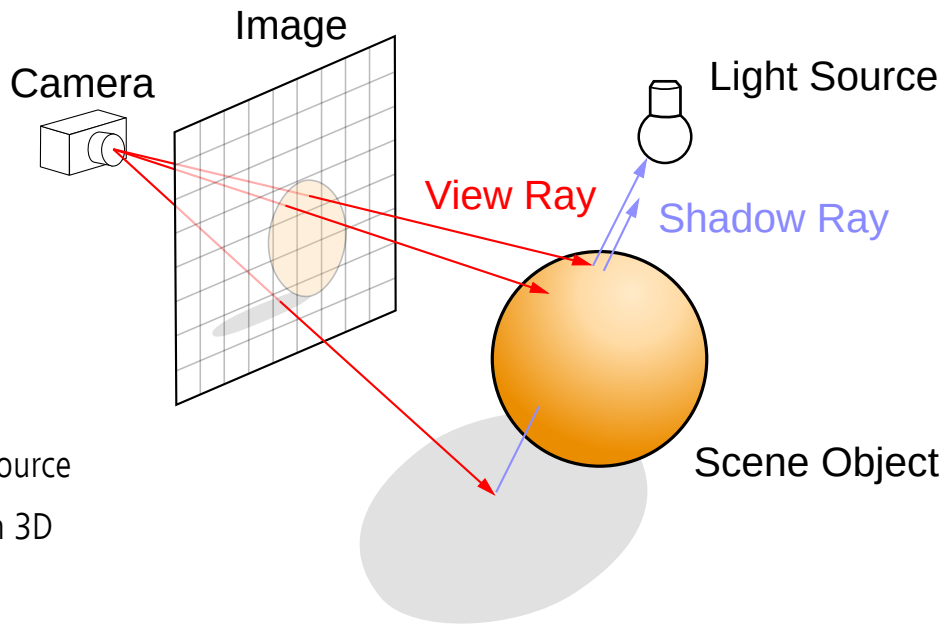
Surface vs volume rendering

- **Surfaces**:
  - **Shape representation**: meshes, parametric surfaces, implicit surfaces, etc.
  - **Material**: transmissive, reflective, glossy, diffuse, specular, etc.
- **Volumes** (participating media):
  - **Shape representation**: voxels, meshes (consider its volume), etc.
  - **Material**: sub-surface scattering, participating media density scattering



Surface rendering summary

- Generate ray using camera information → **primary/camera/visibility ray**
- **Trace ray into the scene**. Possible outcomes:
  - Intersects objects, **find closest surface**
  - No intersection →hit background
- **Surface intersection:**
  - **Calculate amount of light falling on intersection point**
    - **Direct illumination:** directly traced from intersection to light source
    - **Indirect illumination**: sample various directions from this point in 3D scene to obtain reflections of other contributing surfaces
  - Calculate amount of light reflected in primary ray direction using incoming light and material specification → **shading**



Surface rendering summary

For each pixel in raster image:

- Generate primary ray by connecting eye pixel position
- Trace primary ray into the scene
- For each object in the scene:
  - Check if it is intersected with primary ray. Find closes intersection.
- For intersection, generate shadow ray from intersection point to all lights in the scene
  - If shadow ray is not intersecting anything, calculate color
- Calculated color is color of 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, 1969, "Some Techniques for Shading Machine Renderings of Solids"

More into topic

- Surface rendering
  - https://www.pbr-book.org/3ed-2018/Light_Transport_I_Surface_Reflection
  - https://www.scratchapixel.com/lessons/3d-basic-rendering/rendering-3d-scene-overview/computer-discrete-raster.html
- Volumetric rendering
  - https://www.scratchapixel.com/lessons/3d-basic-rendering/volume-rendering-for-developers/volume-rendering-summary-equations.html
  - https://www.pbr-book.org/3ed-2018/Light_Transport_II_Volume_Rendering
  - https://graphics.pixar.com/library/ProductionVolumeRendering/paper.pdf
  - Sub-surface scattering: https://www.pbr-book.org/3ed-2018/Volume_Scattering/The_BSSRDF
- Wave optics rendering
  - https://ssteinberg.xyz/2022/04/03/practical_plt/

Summary questions

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

Reading Materials

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