Rendering introduction

Cornerstones of image synthesis

- 3D scene
- **Rendering**
- Raster image

Recap

- In previous lectures we have discussed how to model elements of 3D scene:
  - 3D model
  - Light
  - Camera
- Now, we will discuss how to use scene elements to simulate interaction of light and objects to create an image



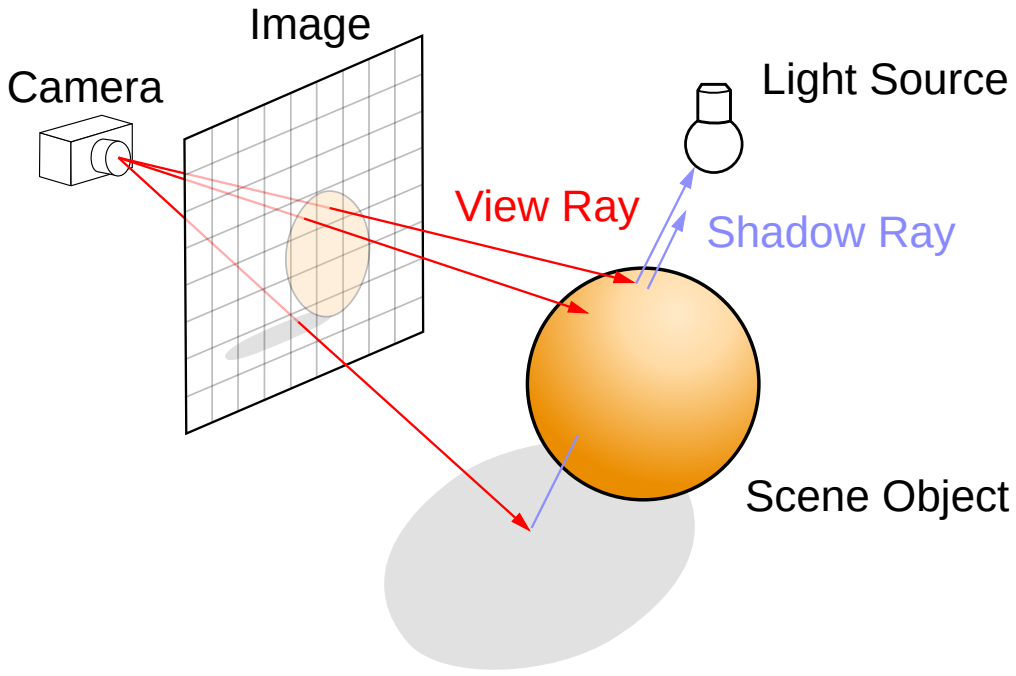
Rendering: intuition

- Photography: Taking a photo from a real world:
  - Light source is emitting light into the space
  - Light travels through space and interacts with objects
  - Small portion of that light falls into camera, where image is created
  - <image>
- Interaction of light with objects in space and camera is well described in physics (wave and geometrical optics).
- Rendering is computer graphics tool which simulates light transport and interaction of light with objects in order to create an image.
  - Rendering creates viewable 2D image from 3D scene

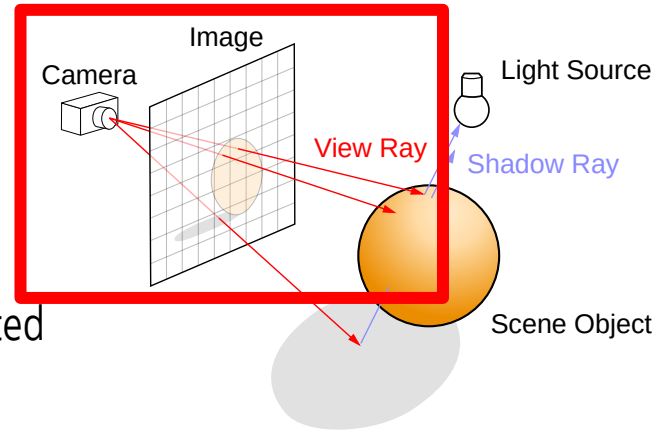
Rendering: main steps

Rendering is solving:

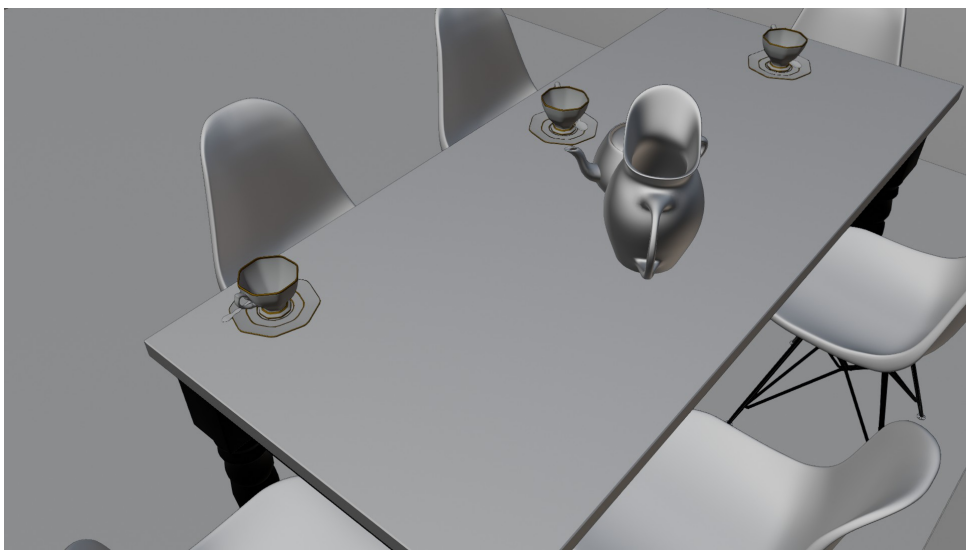
- **Visibility** problem – which objects and surfaces are visible from camera
- **Shading** problem – how does visible objects and surfaces look like



Visibility: high level

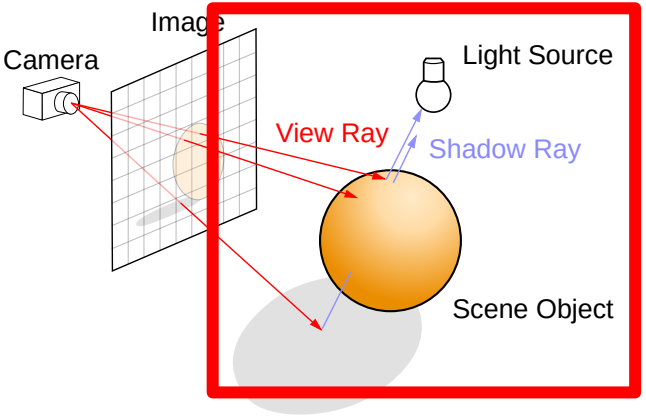


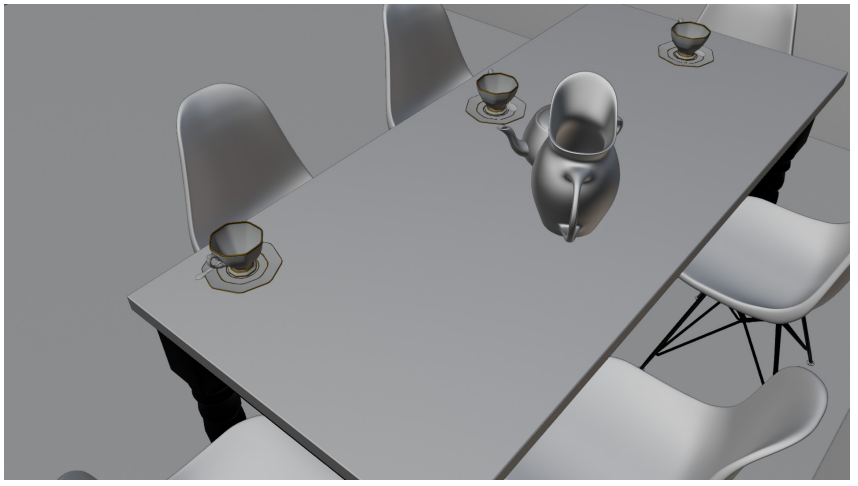
- Visibility is about finding objects in 3D scene visible from camera
- To solve visibility problem realistically, human visual system is simulated
  - **perspective projection**
- Note: Other applications might also solve visibility problem by using **orthographic projection**

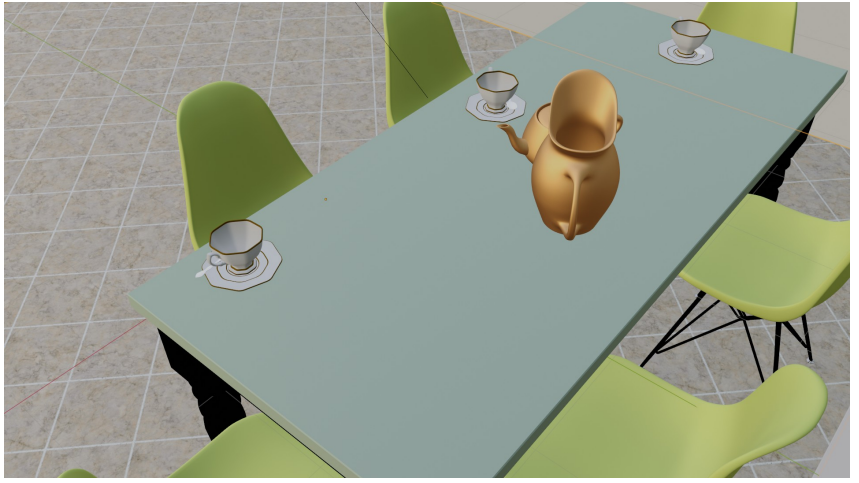


Shading: high level

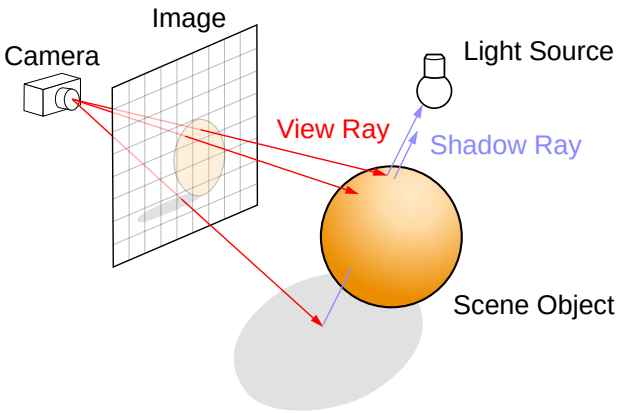
- Shading calculates color and brightness of visible surfaces and objects
- Resulting color depends on:
  - Object shape
  - Object material
  - Incoming light
- Shading results heavily depend on amount of light falling on surface
  - Form of visibility problem: which light sources and surfaces reflecting light are visible from shaded surface
  - This computation is called **light transport**







Light transport: high level



- Light transport computes the light falling on surface
- This method also relies on concept of visibility
  - In this case, visibility between surfaces and lights in 3D scene
- Complexity of light transport significantly determines the resulting image realism

Practical rendering

Practical rendering approaches

Two main practical rendering approaches are based on:

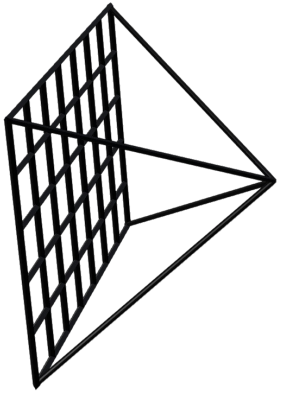
- Ray-tracing
- Rasterization

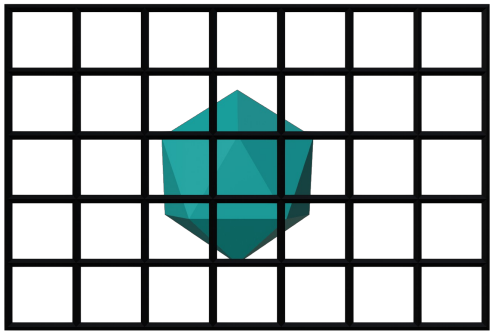
Ray-tracing vs Rasterization

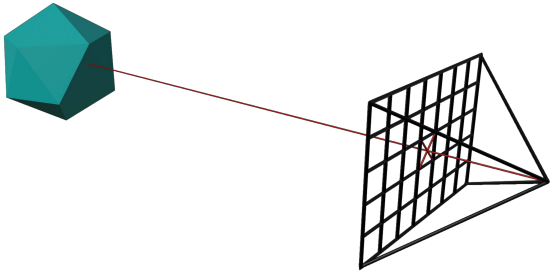
- Compared to rasterization, ray-tracing is more directly inspired by the physics of light
  - As such it can generate substantially more realistic images

Visibility solvers

- Rasterization and ray-tracing, are in their core, visibility from camera solvers!

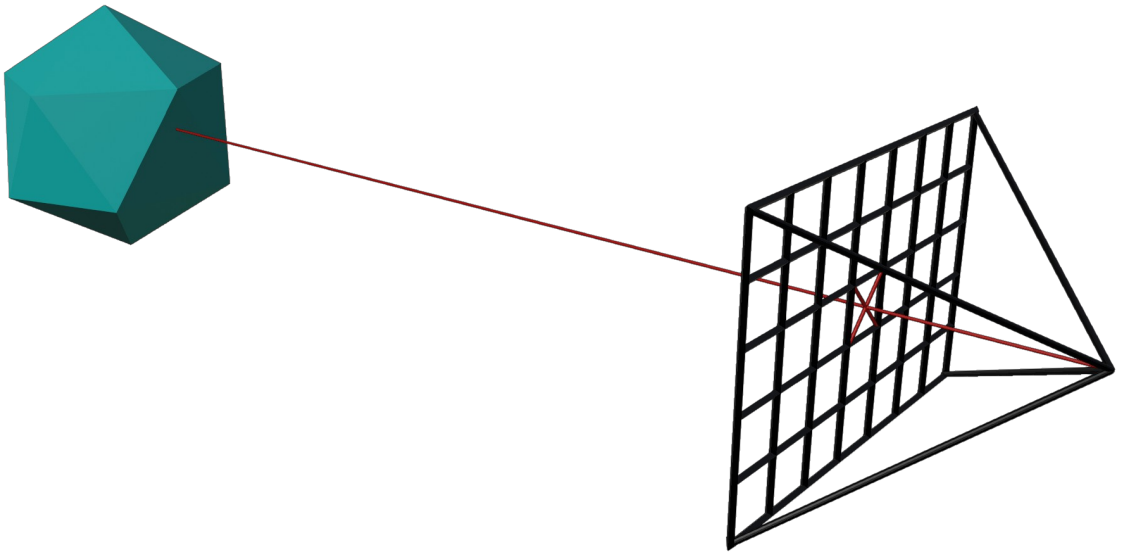






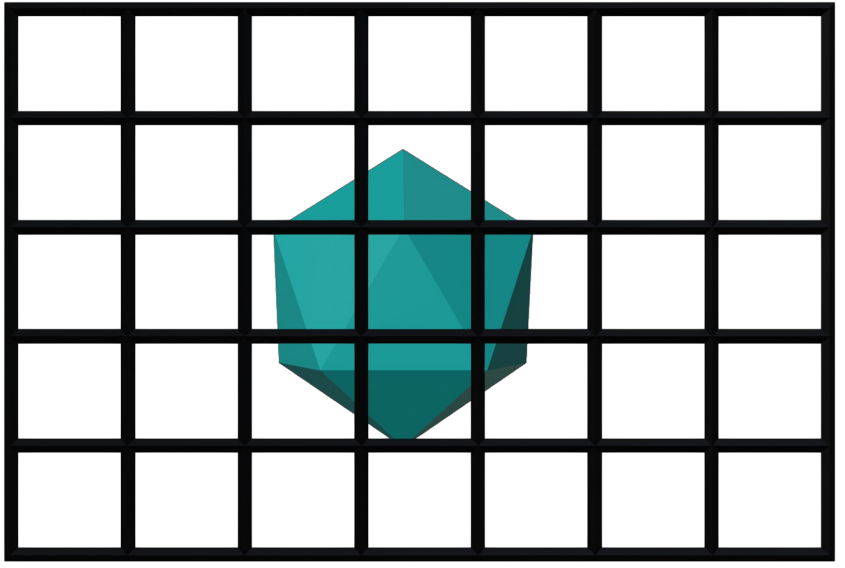
Visibility: ray-tracing

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



Visibility: rasterization

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

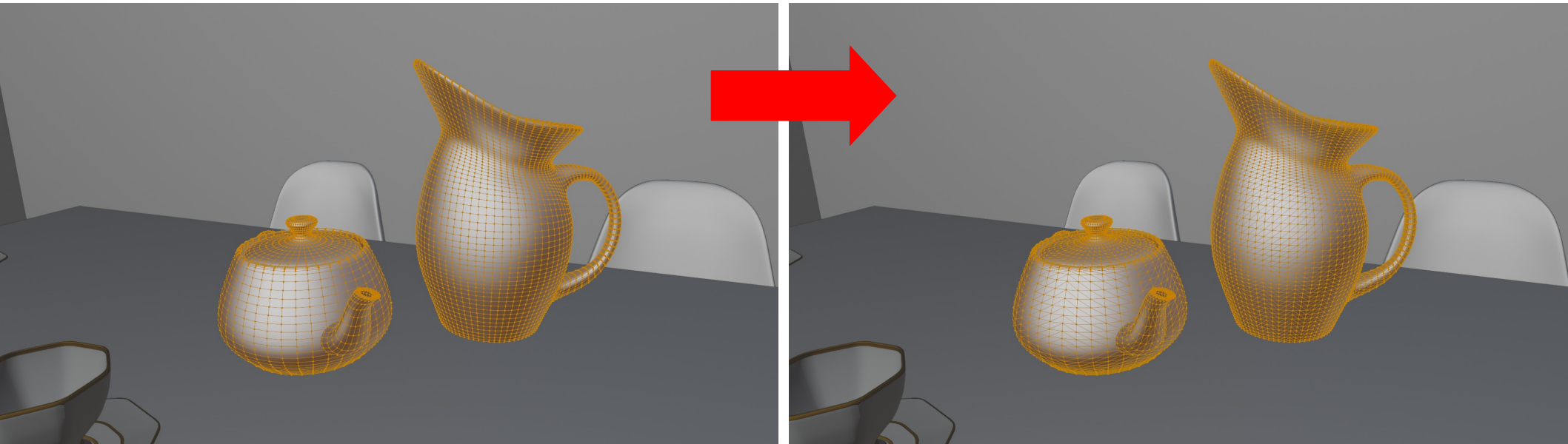


Raytracing vs rasterization

- As we can see from the algorithm, **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.**
- On the other hand, **rasterization** is reversed: it 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: triangles

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



Speed: Ray-tracing vs Rasterization

- Both algorithms are conceptually simple
- To make them fast, additional code and hardware is needed, for n objects:
  - Ray-tracing is utilizing spatial data structures* for achieving running time of O(log(n))
  - Rasterization has also better running time than O(n) by using culling** algorithms and techniques*** which avoid full scene and primitive processing. Furthermore, GPUs are adapted for rasterization-based rendering

* Bounding volume hierarchy (BVH) or Kd tree. We will discuss those later.
** Occlusion culling, frustum culling. It will be discussed later.
*** e.g., Deferred shading. It will be discussed later.

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 and full rendering equation**.
  - On other words, images just look better
- 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

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 rasterization, lot of artist hacks is needed. Sometimes ray-tracing can be used to precalculate some effects.
- <image: animation film: rasterization for preview and modeling and final shoot>
- <image: games: rasterization as final image with hacks and help from ray-tracing>

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 improvements are needed.
- Denoising is promising solution.
- For short-term, clever combinations of rasterizations and ray-tracing 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!

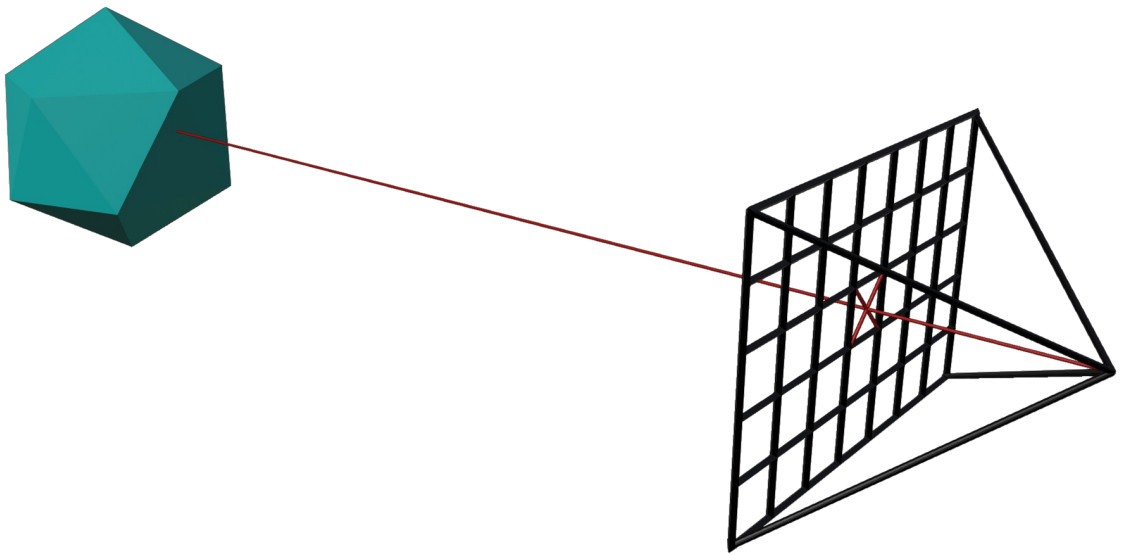
Raytracing and rasterization

- Raytracing is historically early method for image synthesis based on 3D scene description
- Raytracing has always been recognized as best method for generating photo-realistic images. However, up until 1990-2000s it hasn't been used widespread
- Due to hardware advancements, rasterization-based rendering gained a lot of attention
- Raytracing is intuitive and straightforward method of simulating physical phenomena of 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.

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.

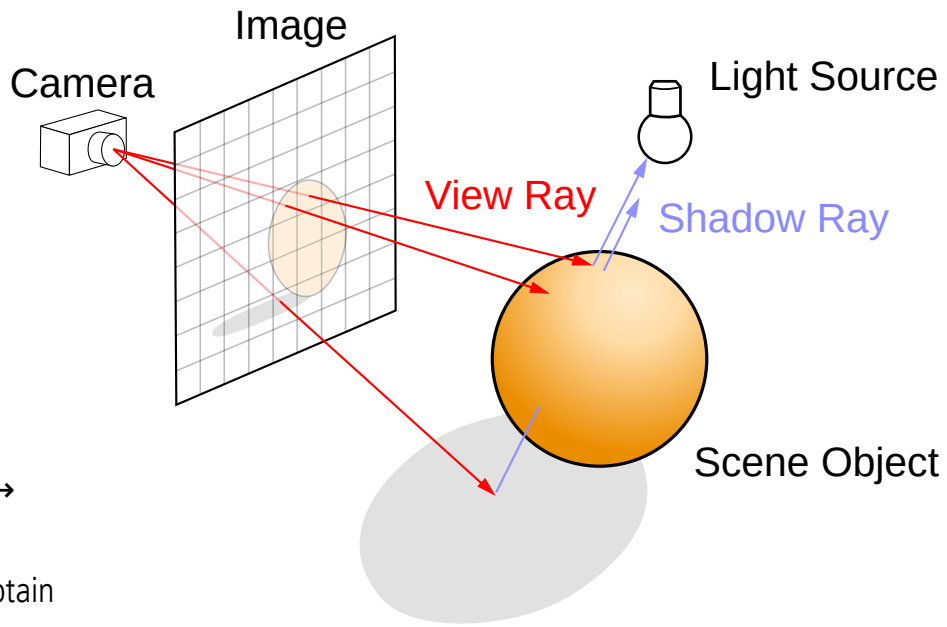
Simulating light

- 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

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

```
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
```

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>

Practical aspects

Practical ray-tracing and rasterization

- Raytracing method can be separated in:
  - **Determining visibility from camera**: determine which point in 3D scene is visible for each pixel of image. Note that **perspective projection** is inherent due to camera which is used for generating rays
  - **Shading**: calculating the color of visible point – this operation besides light-matter calculation also requires **light transport** to collect all light which influences the color and intensity of that point.
- Both steps require extremely time-consuming intersections between rays and scene objects (geometry)
  - 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*.

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

<IMAGE: differences and comparison between rasterizers and raytracers>

- 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.

Note: compexity of scenes and rendering

- Although HW and methods advanced, the time of rendering the scenes stayed the same! This is because complexity of the 3D scenes as well as rendering algorithms increased.
  - **Blinn law**

Reading Materials

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