

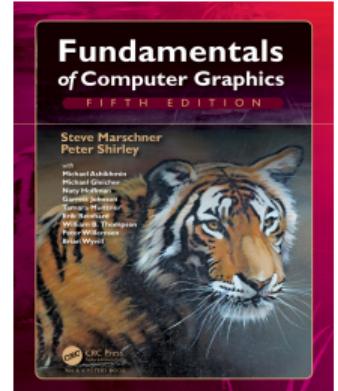
Welcome to 02562: Rendering - Introduction

Jeppe Revall Frisvad

September 2025

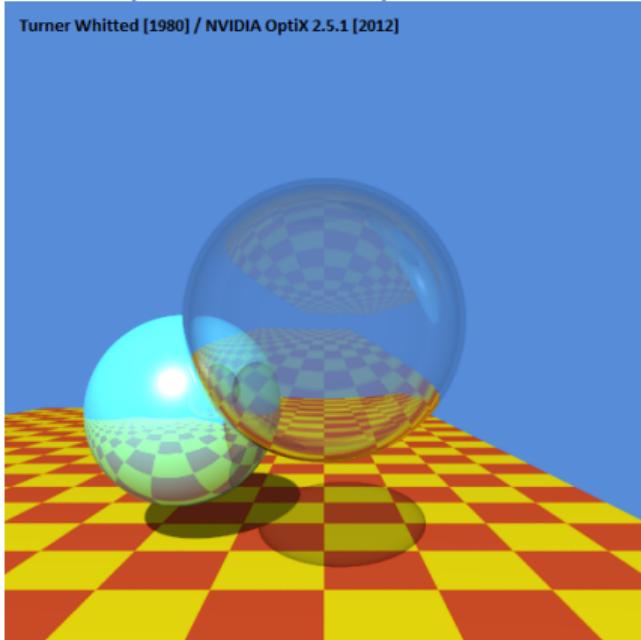
<https://www.disneyanimation.com/technology/hyperion/>

0:00-5:06, 8:56-9:31



Ray tracing (weeks 1–3)

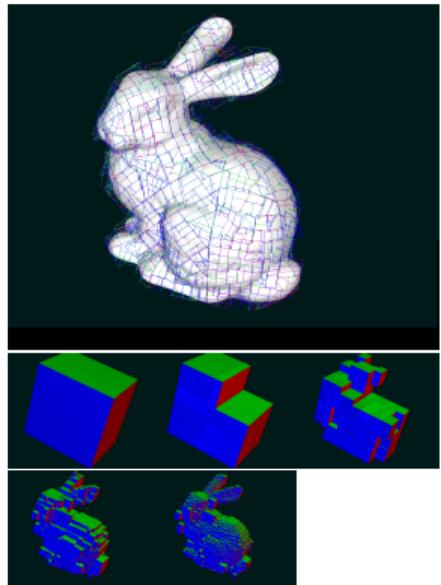
Turner Whitted [1980] / NVIDIA OptiX 2.5.1 [2012]



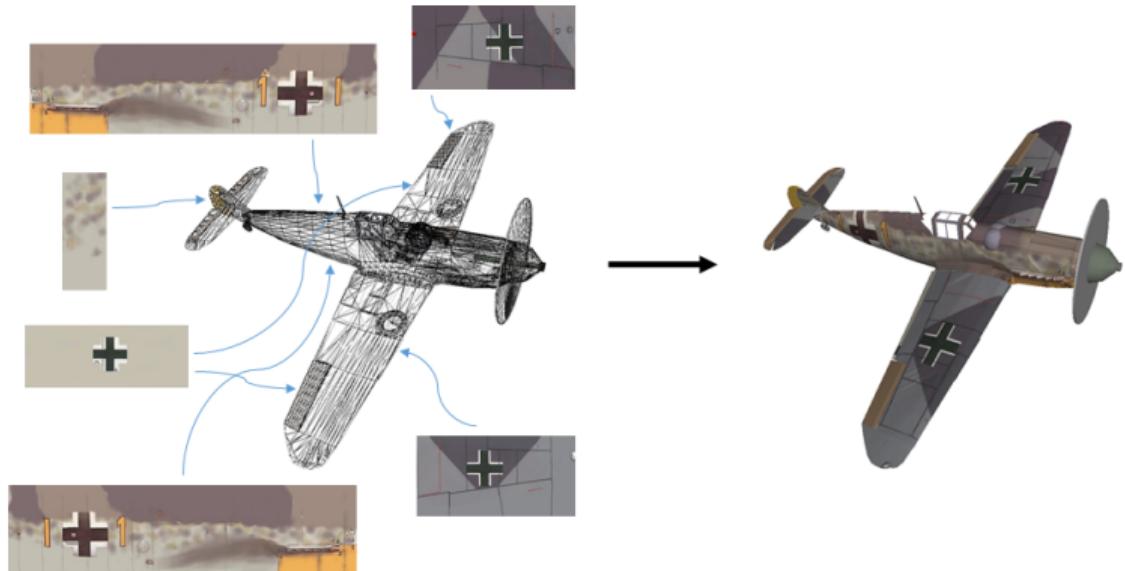
- ▶ 74 minutes in the original ray tracing paper by Turner Whitted [1980].
- ▶ More than 300 frames per second using GPU ray tracing in 2012 [Parker et al. 2010].
- ▶ That's a 6 orders of magnitude speed-up! (Beats Moore's law, but 950 fps on RTX in 2022.)
- ▶ Ray tracing is now a widely spread production-ready rendering technique.

Content (weeks 4, 6, and 7)

- Triangle meshes, spatial subdivision, texturing.



[Kammaje and Mora 2007]



[Blinn and Newell 1976]

Radiometry and light transport (weeks 5 and 8)

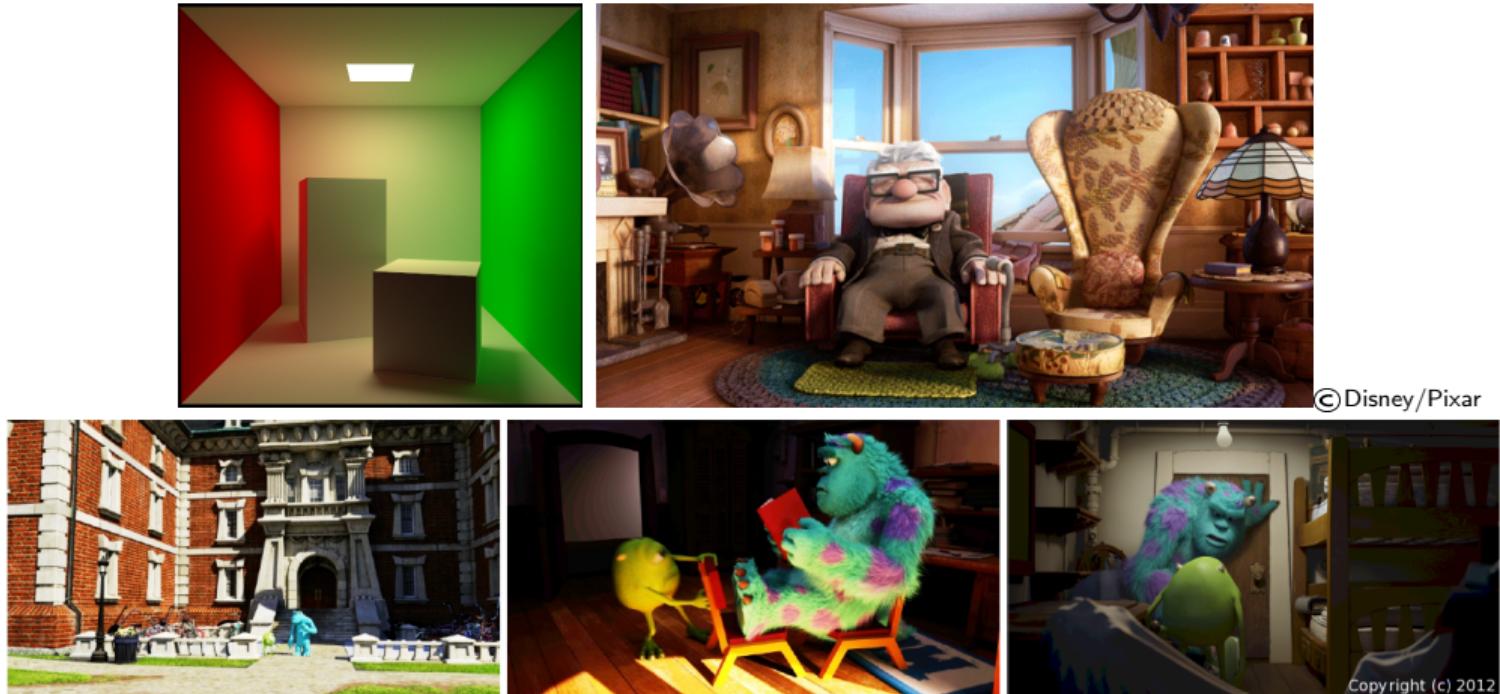
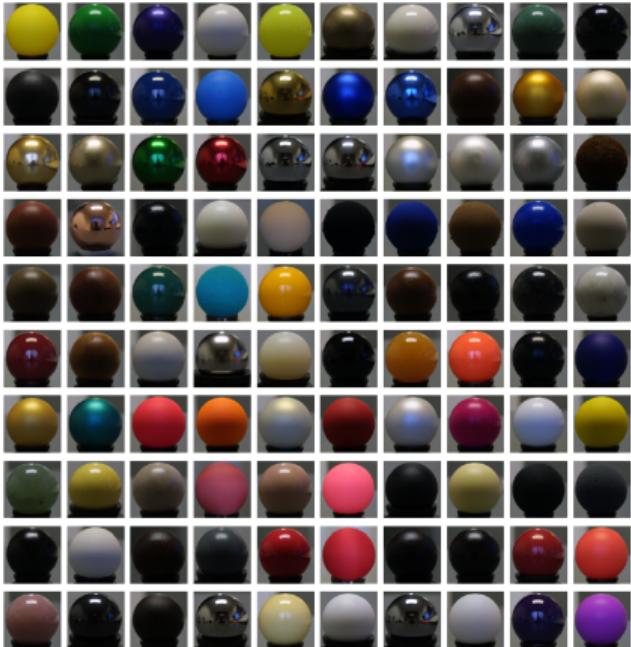
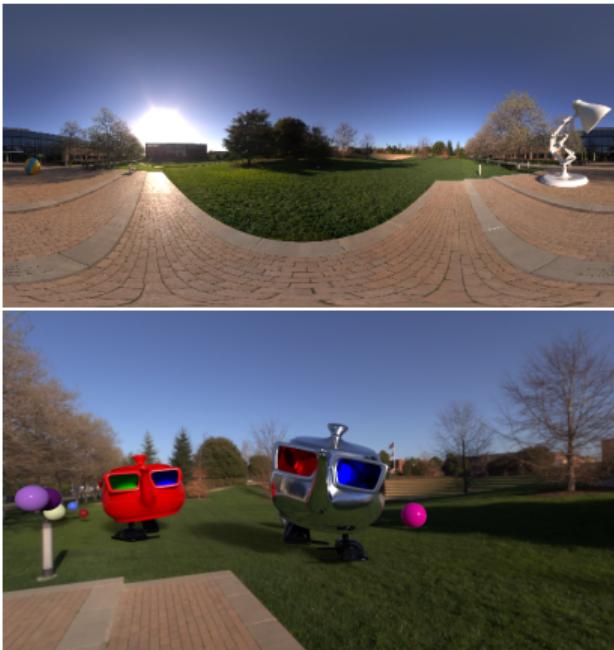


Figure 1: Global illumination images from 'Monsters U.' These images render more than $30\times$ faster with radiosity caching. © Disney/Pixar.

- ▶ Diffuse reflection of indirect illumination [Cohen and Greenberg 1985; Christensen 2010; Christensen et al. 2012].
- ▶ Used in production since the mid 2000s [Christensen et al. 2018].

Connection to the real world (weeks 9-10)

Pixar RenderMan



[Matusik et al. 2003]

- ▶ Use of 360° photographed environments for natural lighting of a scene.
- ▶ Use of measured (BRDF) functions to reproduce the appearance of real materials.
- ▶ Realism is needed to render synthetic datasets for learning.

Other parts of this course (offered as projects)

- ▶ Production path tracing (denoising or material appearance modelling).
- ▶ Tone mapping of images rendered with high dynamic range (HDR).
- ▶ Importance sampling techniques for material shaders and/or environment lighting.
- ▶ Rendering for computer vision (with a matrix obtained from camera calibration).
- ▶ Rendering with a textured light source (like a monitor or a projector).
- ▶ Appearance representation using a neural radiance field ([NeRF](#)).
- ▶ [3D Gaussian ray tracing](#) for fast rendering of captured point clouds.
- ▶ Differentiable rendering using [Slang](#).
- ▶ [NVIDIA RTX rendering](#) to exploit the most recent GPU hardware features.
- ▶ Better faster spatial data structures: implement a bounding volume hierarchy (BVH) perhaps with treelet restructuring (or use [Vulkan/OptiX](#)).
- ▶ Test ray tracing in [PyOptiX](#) or [SlangPy](#).

Instructor and teaching style

- ▶ Course responsible: Jeppe Revall Frisvad
- ▶ TA: Phi Dinh Tran
- ▶ Lecture (13-15)
- ▶ Exercises (15-17)
- ▶ Lab journal: Save a set of solution files for each part of each worksheet.
- ▶ Project: Select, study and implement, describe in a report.
- ▶ Hand-in: HTML, JavaScript, and PDF (include all code in a ZIP file)
- ▶ Deadline: **19 December 2025 at 23:59**

References (chronologically)

- Blinn, J. F., and Newell, M. E. Texture and reflection in computer generated images. *Communications of the ACM* 19(10), pp. 542–547. October 1976.
- Whitted, T. An improved illumination model for shaded display. *Communications of the ACM* 23(6), pp. 343–349. June 1980.
- Cohen, M. F., and Greenberg, D. P. The hemi-cube: a radiosity solution for complex environments. *Computer Graphics (SIGGRAPH '85)* 19(3), pp. 31–40. July 1985.
- Matusik, W., Pfister, H., Brand, M., and McMillan, L. Efficient isotropic BRDF measurement. In *Proceedings of EGSR 2003*, pp. 241-248. June 2003.
- Kammaje, R. P., and Mora, B. A study of restricted BSP trees for ray tracing. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing (RT '07)*, pp. 55–62. October 2007.
- Parker, S. G., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A., and Stich, M. OptiX: a general purpose ray tracing engine. *ACM Transactions on Graphics* 29(4), Article 66. July 2010.
- Christensen, P. H. Point-based global illumination for movie production. In *Global Illumination Across Industries*. ACM SIGGRAPH Courses, Article 8. July 2010.
- Christensen, P. H., Harker, G., Shade, J., Schubert, B., and Batali, D. Multiresolution Radiosity Caching for Efficient Preview and Final Quality Global Illumination in Movies. Pixar Technical Memo #12-06. Pixar, July 2012.
- Eisenacher, C., Nichols, G., Selle, A. and Burley, B. Sorted deferred shading for production path tracing. *Computer Graphics Forum* 32(4). pp. 125-132. July 2013.
- Christensen, P., Fong, J., Shade, J., Wooten, W., Schubert, B., Kensler, A., Friedman, S., Kilpatrick, C., Ramshaw, C., Bannister, M., Rayner, B., Brouillant, J., and Liani, M. RenderMan: an advanced path-tracing architecture for movie rendering. *ACM Transactions on Graphics* 37(3), Article 30. June 2018.
- Burley, B., Adler, D., Chiang, M. J.-Y., Driskill, H., Habel, R., Kelly, P., Kutz, P., Li, Y. K., and Teece, D. The design and evolution of Disney's Hyperion renderer. *ACM Transactions on Graphics* 37(3), Article 33. June 2018.

02562 Rendering - Introduction

Ray Casting

Jeppe Revall Frisvad

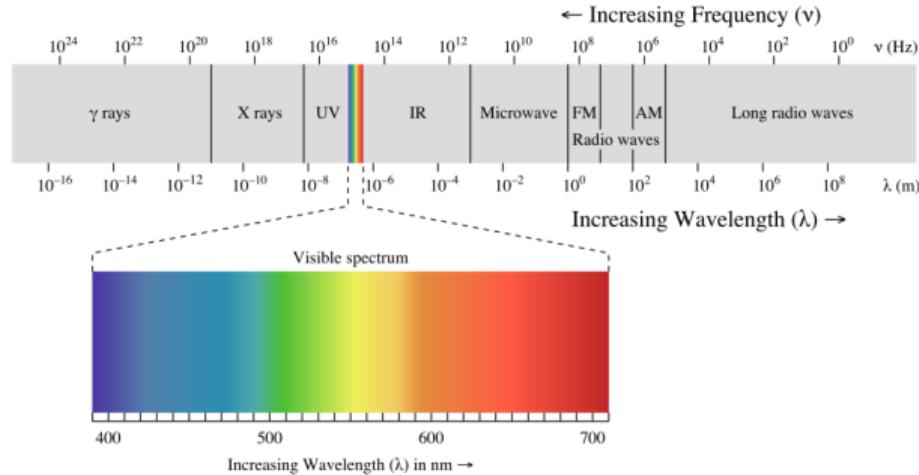
August 2024

Building blocks of realistic rendering

- ▶ Think of the experiment: “taking a picture”.
- ▶ What do we need to model it?
 - ▶ Camera (light sensor).
 - ▶ Objects (scene geometry).
 - ▶ Light (light transport).
 - ▶ Luminaires (light sources).
 - ▶ Materials (light scattering).
- ▶ Mathematical models for these elements are required as a minimum in order to render an image.
- ▶ We can use very simple models:
 - ▶ Pinhole camera (perspective, lines from a point).
 - ▶ Mathematical primitives (planes, spheres, triangles).
 - ▶ Ray optics (light propagates along straight lines).
 - ▶ Practical lights (point, directional, spot).
 - ▶ Lambertian materials (constant reflectance).
- ▶ If we desire a high level of realism, more complicated models are required.

Light propagation

- ▶ Visible light is electromagnetic waves of wavelengths (λ) from 380 nm to 780 nm.



- ▶ Electromagnetic waves in transparent media propagate as *rays of light* for $\lambda \rightarrow 0$.
- ▶ Rays of light follow the path of least time (Fermat).
- ▶ How does light propagate in air? In straight lines (almost).
- ▶ The parametrisation of a straight line in 3D is therefore a good, simple model for light propagation: $\mathbf{r}(t) = \mathbf{o} + t\vec{\omega}$, $t \in [t_{\min}, t_{\max}]$, $t_{\max} > t_{\min} > 0$.

Light sources

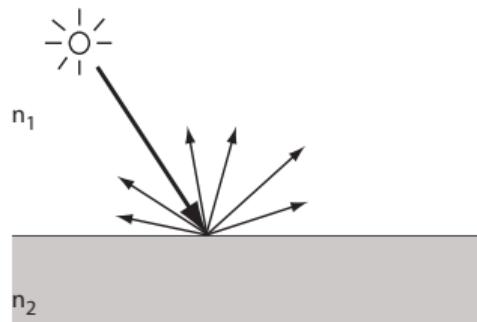
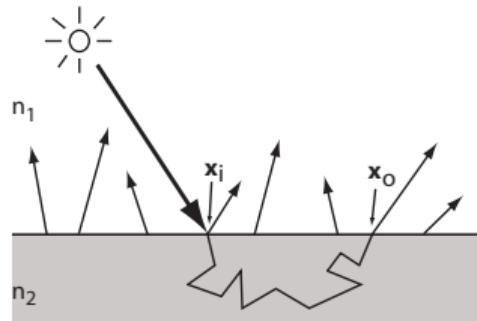
- ▶ A light source is described by a spectrum of light $L_{e,\lambda}(x, \vec{\omega}_o)$ which is emitted from each point on the emissive object.
- ▶ A simple model is a light source that from each point emits the same amount of light in all directions and at all wavelengths, $L_{e,\lambda} = \text{const.}$
- ▶ The spectrum of heat-based light sources can be estimated using Planck's law of radiation. Examples:



- ▶ The surface geometry of light sources is modelled in the same way as other geometry in the scene.

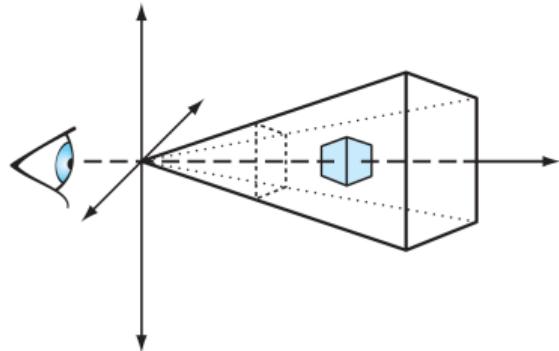
Materials (scattering and absorption of light)

- ▶ Optical properties (index of refraction, $n(\lambda) = \eta(\lambda) + i\kappa(\lambda)$).
- ▶ Reflectance distribution functions, $f(\mathbf{x}_i, \vec{\omega}_i; \mathbf{x}_o, \vec{\omega}_o)$.



Camera

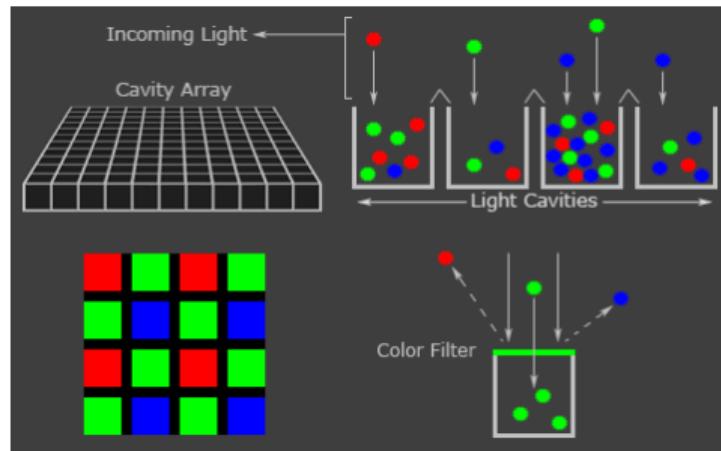
- ▶ A camera consists of a light sensitive area, a processing unit, and a storage for saving the captured images.
- ▶ The simplest model of a camera is a rectangle, which models the light sensitive area (the chip/film), placed in front of an eye point where light is gathered.



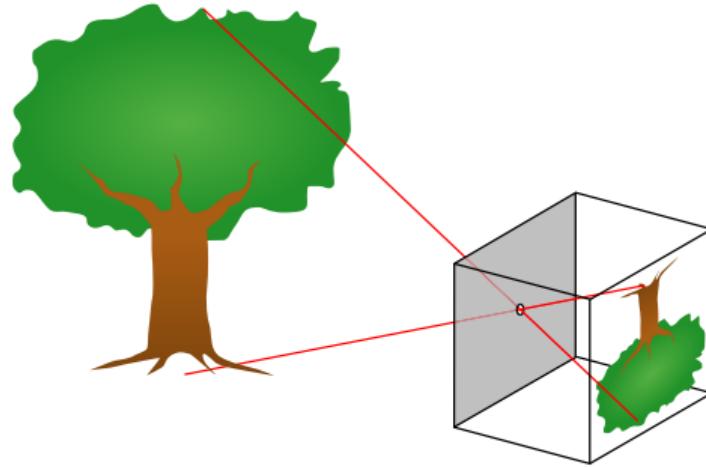
- ▶ We can use this model in two different ways:
 - ▶ Follow rays from the eye point through the rectangle and onwards (ray casting).
 - ▶ Project the geometry on the image plane and find the geometry that ends up in the rectangle (rasterization).

The light sensitive Charge-Coupled Device (CCD) chip

- ▶ A CCD chip is an array of light sensitive cavities.
- ▶ A digital camera therefore has a resolution $W \times H$ measured in number of pixels.
- ▶ A pixel corresponds to a small area on the chip.
- ▶ Several light sensitive cavities are used for one pixel because the light measurement is divided into red, green, and blue.
- ▶ Conversion from this colour pattern to an RGB image is called demosaicing.



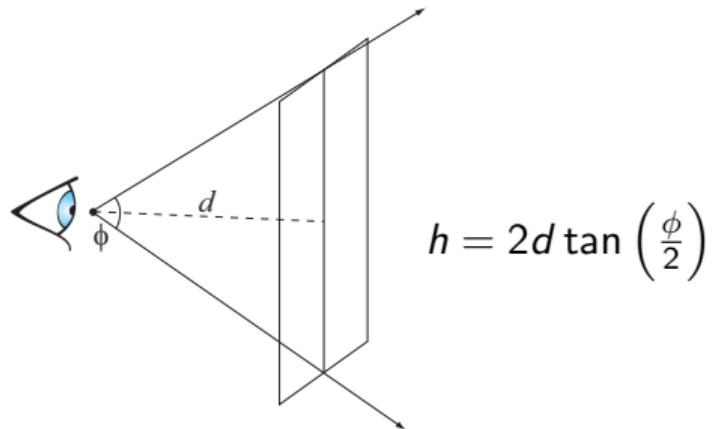
The pinhole camera model



- ▶ The lens is modelled as a point **e** (an infinitely small hole).
- ▶ At the distance d behind the point **e**, we place the CCD chip.
- ▶ But is it really necessary to turn the image upside down?
Not if we place the image plane in front of **e** (the eye point).

The lens as an angle and a distance

- ▶ The lens system determines how large the field of view is.
- ▶ The field of view is an angle ϕ .



- ▶ In the model, it is not important whether d and the size of the light sensitive area correspond to reality.
- ▶ We choose $h = 1$ and use only d or only ϕ to describe the lens system (and thereby the zoom level of the camera).
- ▶ We often use d , which is called the camera constant.

What is ray tracing?

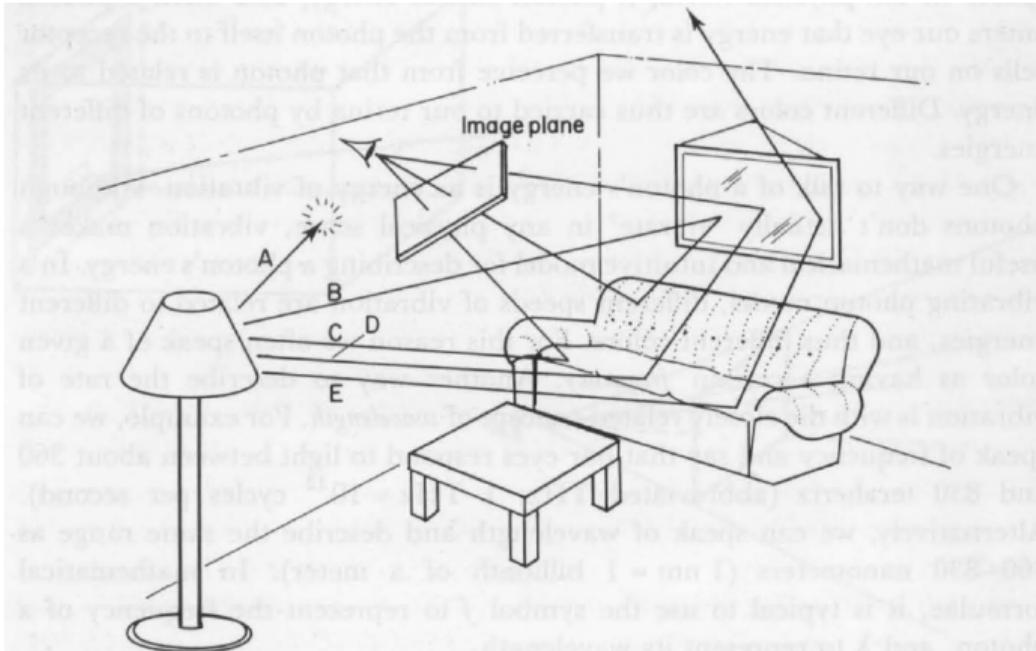


Fig. 5. Some light rays (like A and E) never reach the image plane at all. Others follow simple or complicated routes.

References

- Glassner, A. An Overview of Ray Tracing. In *An Introduction to Ray Tracing*, Chapter 1, pp. 1–17, Morgan Kaufmann, 1989.

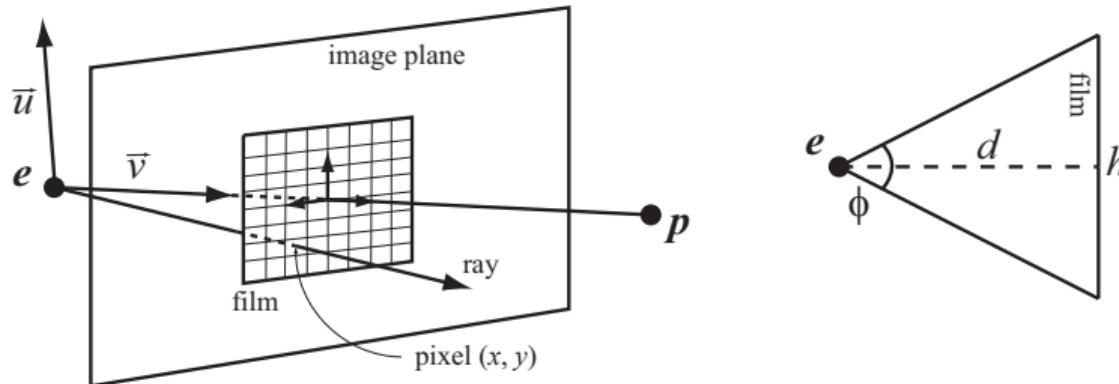
Ray generation

Camera space coordinates: $\mathbf{q}_c = (x_{ip}, y_{ip}, d)$, $(x_{ip}, y_{ip}) = \left(x + \frac{1}{2}, y + \frac{1}{2}\right) \frac{1}{H} - \left(\frac{1}{2} \frac{W}{H}, \frac{1}{2}\right)$
Change of basis matrix: $\begin{bmatrix} \vec{b}_1 & \vec{b}_2 & \vec{v} \end{bmatrix} = \begin{bmatrix} \frac{\vec{v} \times \vec{u}}{|\vec{v} \times \vec{u}|} & \vec{b}_1 \times \vec{v} & \frac{\mathbf{p} - \mathbf{e}}{|\mathbf{p} - \mathbf{e}|} \end{bmatrix}$

► Camera description: Ray direction: $\vec{\omega} = \frac{\mathbf{q}}{|\mathbf{q}|}$, $\mathbf{q} = [\vec{b}_1 \quad \vec{b}_2 \quad \vec{v}] \mathbf{q}_c = \vec{b}_1 x_{ip} + \vec{b}_2 y_{ip} + \vec{v}d$

	Extrinsic parameters	Intrinsic parameters
\mathbf{e}	Eye point	ϕ Vertical field of view
\mathbf{p}	View point	d Camera constant
\vec{u}	Up direction	W, H Camera resolution

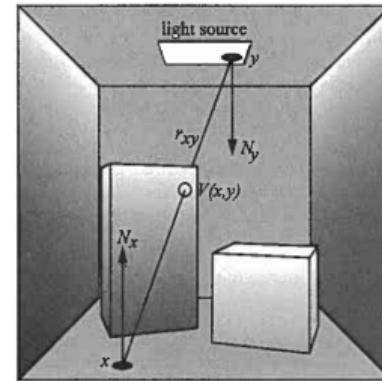
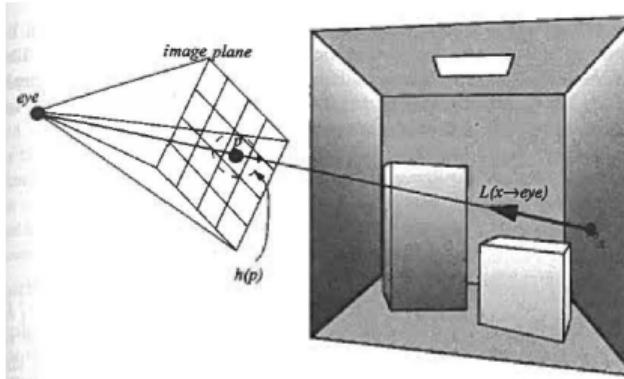
- Sketch of ray generation:



- Given pixel index (x, y) , find the direction $\vec{\omega}$ of a ray through that pixel.

Rays in theory and in practice

- ▶ Parametrisation of a straight line: $\mathbf{r}(t) = \mathbf{o} + t\vec{\omega}$, $t \in [t_{\min}, t_{\max}]$.
- ▶ Camera provides origin (\mathbf{o}) and direction ($\vec{\omega}$) of “eye rays”.



- ▶ Rays in code (WGLS):

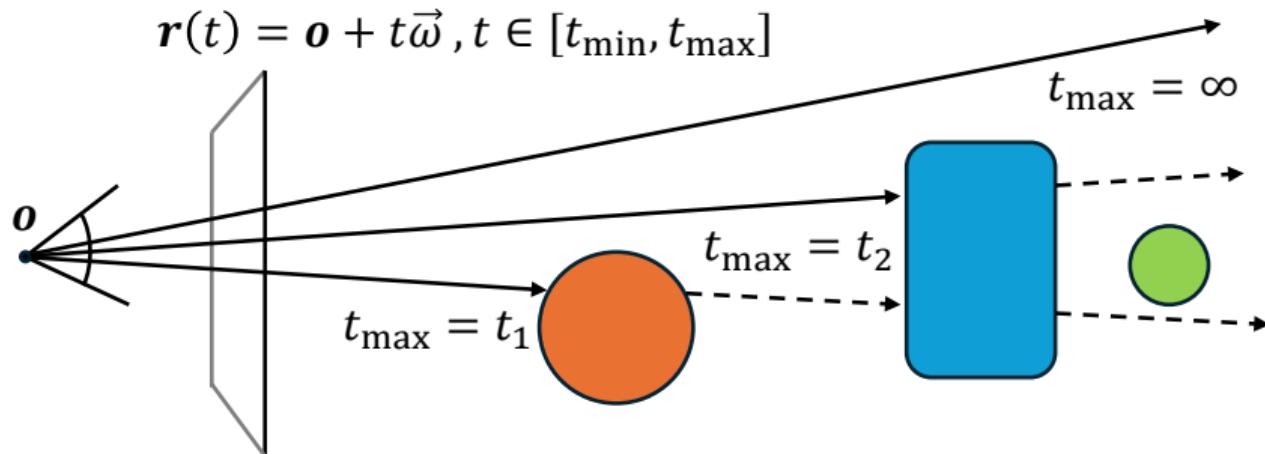
```
struct Ray {  
    origin: vec3f,  
    direction: vec3f,  
    tmin: f32,  
    tmax: f32  
};
```

Hidden surface removal

- When computing ray-object intersection, we must always check that the distance to the intersection t is within the limits:

$$t_{\min} \leq t \leq t_{\max}$$

- When searching for the closest hit, modifying t_{\max} ensures that intersections further away will no longer be considered.
- If searching for any hit, terminate the search as soon as an intersection is found.



What do you need in a ray tracer?

- ▶ Render engine (for looping over all pixels).
- ▶ Camera (for ray generation).
- ▶ Objects (for ray-object intersection).
- ▶ Accelerators (for finding the closest hit point).
- ▶ Light sources (for shading and shadows).
- ▶ Materials (for assigning material properties to objects).
- ▶ Shaders (for computing the shade at the closest hit point).
- ▶ Tracer (for calling functions to shade a pixel).
- ▶ Scene (container).

Using WebGPU as the render engine

- ▶ The application runs in a [web browser](https://webgpureport.org/) (<https://webgpureport.org/>).
- ▶ The browser environment provides many render engine features for free:
 - ▶ Window management.
 - ▶ GPU rendering context.
 - ▶ User interface functionality.
 - ▶ Image file input and output.
 - ▶ Text rendering.
 - ▶ Platform independence.
- ▶ The native programming language of a browser is JavaScript.
- ▶ The browser provides developer tools
(console for error output and debugger for stepping through code).
- ▶ An application that runs in a browser is easily published on the web.
- ▶ Handing in your lab journal is significantly simpler (no need to compose a large pdf with all the code snippets and rendering results).



Exercises

Week 1

- ▶ Implement a render engine based on WebGPU.
- ▶ Generate rays using a (modified) pinhole camera model.
- ▶ Pass data from CPU to GPU to enable interaction and load balancing.

Week 2

- ▶ Compute ray-plane, ray-triangle, and ray-sphere intersection.
- ▶ Implement conditional acceptance of intersections to find the closest hit (hidden surface removal).
- ▶ Compute shading of diffuse surfaces by point lights.
Use Kepler's inverse square law and Lambert's cosine law.

Getting started with HTML and JavaScript

HTML for a webpage with a canvas
(get MV.js from DTU Learn):

```
<!DOCTYPE html>
<html>
<head>
    <title>My Page</title>
    <script type="text/javascript" src="MV.js"></script>
    <script type="text/javascript" src="myscript.js"></script>
</head>
<body>
    <canvas id="my-canvas" width="512" height="512">
        Please use a browser that supports HTML5 canvas.
    </canvas>
</body>
</html>
```

Call an **asynchronous** main function
(in myscript.js) when the page loads:

```
"use strict";
window.onload = function() { main(); }
async function main()
{
    const gpu = navigator.gpu;
    const adapter = await gpu.requestAdapter();
    const device = await adapter.requestDevice();
    const canvas = document.getElementById('my-canvas');
    const context = canvas.getContext('webgpu');
    const canvasFormat = navigator.gpu.getPreferredCanvasFormat();
    context.configure({
        device: device,
        format: canvasFormat,
    });

    // Create a render pass in a command buffer and submit it
    ...
}
```

Working with WebGPU (W01P1a)

- The render pass command buffer:

```
// Create a render pass in a command buffer and submit it
const encoder = device.createCommandEncoder();
const pass = encoder.beginRenderPass({
    colorAttachments: [
        {
            view: context.getCurrentTexture().createView(),
            loadOp: "clear",
            storeOp: "store",
        }
    ]
});
// Insert render pass commands here
pass.end();
device.queue.submit([encoder.finish()]);
```



- Writing WGSL shaders in the HTML file:

```
<script id="wgsl" type="x-shader">
@vertex
fn main_vs(@builtin(vertex_index) VertexIndex : u32) -> @builtin(position) vec4f
{
    const pos = array<vec2f, 4>(vec2f(-0.9, 0.9), vec2f(-0.9, -0.9), vec2f(0.9, 0.9), vec2f(0.9, -0.9));
    return vec4f(pos[VertexIndex], 0.0, 1.0);
}

@fragment
fn main_fs() -> @location(0) vec4f
{
    return vec4f(0.1, 0.3, 0.6, 1.0);
}
</script>
```

- Using a WGSL file: <script id="wgsl" type="x-shader" src="myshaders.wgsl"></script>

Setting up a render pipeline (W01P1b)

- ▶ Load the WGSL shaders from the HTML file and use them for the render pipeline:

From HTML file

```
const wgpu = device.createShaderModule({
    code: document.getElementById("wgsl").text
});

const pipeline = device.createRenderPipeline({
    layout: 'auto',
    vertex: { module: wgpu, entryPoint: 'main_vs' },
    fragment: {
        module: wgpu,
        entryPoint: 'main_fs',
        targets: [{ format: canvasFormat }],
    },
    primitive: { topology: 'triangle-strip' },
});
```

From WGSL file

```
const wgslfile = document.getElementById('wgsl').src;
const wgslcode
= await fetch(wgslfile, {cache: "reload"}).then(r => r.text());
const wgpu = device.createShaderModule({
    code: wgslcode
});
```



- ▶ Using a WGSL file requires that you set up a local server (recommended).
- ▶ Add the pipeline and a draw call to the render pass:

```
pass.setPipeline(pipeline);
pass.draw(4);
```

Getting image plane coordinates (W01P1c)

- We need to pass information from the vertex to the fragment shader:

```
struct VSOut {
    @builtin(position) position: vec4f,
    @location(0)         coords : vec2f,
};

@vertex
fn main_vs(@builtin(vertex_index) VertexIndex : u32) -> VSOut
{
    const pos = array<vec2f, 4>(vec2f(-1.0, 1.0), vec2f(-1.0, -1.0), vec2f(1.0, 1.0), vec2f(1.0, -1.0));
    var vsOut: VSOut;
    vsOut.position = vec4f(pos[VertexIndex], 0.0, 1.0);
    vsOut.coords = pos[VertexIndex];
    return vsOut;
}

// Define Ray struct
:

fn get_camera_ray(ipcoords: vec2f) -> Ray
{
    // Implement ray generation (WGSL has vector operations like normalize and cross)
    :
}

@fragment
fn main_fs(@location(0) coords: vec2f) -> @location(0) vec4f
{
    let ipcoords = coords*0.5;
    var r = get_camera_ray(ipcoords);
    return vec4f(r.direction*0.5 + 0.5, 1.0);
}
```



Getting user input from the CPU side

- ▶ Data that are the same for all pixels are called Uniforms.
- ▶ We can define them at the top of our WGSL code:

```
struct Uniforms {  
    aspect: f32,  
    cam_const: f32,  
};  
@group(0) @binding(0) var<uniform> uniforms : Uniforms;
```

and use them like other structs: `let ipcoords = vec2f(coords.x*uniforms.aspect*0.5, coords.y*0.5);`

- ▶ But they require some infrastructure on the JavaScript side:

```
let bytelength = 5*sizeof['vec4']; // Buffers are allocated in vec4 chunks  
let uniforms = new ArrayBuffer(bytelength);  
const uniformBuffer = device.createBuffer({  
    size: uniforms.byteLength,  
    usage: GPUBufferUsage.UNIFORM | GPUBufferUsage.COPY_DST,  
});  
const bindGroup = device.createBindGroup({  
    layout: pipeline.getBindGroupLayout(0),  
    entries: [{  
        binding: 0,  
        resource: { buffer: uniformBuffer }  
    }],  
});
```

- ▶ and a command needs to be added to the render pass before the draw call:

```
pass.setBindGroup(0, bindGroup);
```

Making the application interactive and writing uniforms (W01P2(&5))

- ▶ Put data in an array (note the vec4 padding) before writing them to the GPU:

```
const eye = vec3(2.0, 1.5, 2.0);
const aspect = canvas.width/canvas.height;
const cam_const = 1.0;
new Float32Array(uniforms, 0, 8).set([aspect, cam_const, 0.0, 0.0, ...eye, 0.0]);
device.queue.writeBuffer(uniformBuffer, 0, uniforms);
```

- ▶ Put the render pass command buffer code in a render function.
- ▶ Suppose we have a mouse with a scroll wheel (if not, choose something else).
- ▶ The browser provides an interface for it.
- ▶ We can let it modify the camera constant, and pass this to the GPU as a uniform.

```
addEventListener("wheel", (event) => {
    cam_const *= 1.0 + 2.5e-4*event.deltaY;
    new Float32Array(uniforms, 4, 1).set([cam_const]);
    requestAnimationFrame/animate);
});

function animate()
{
    device.queue.writeBuffer(uniformBuffer, 0, uniforms);
    render(device, context, pipeline, bindGroup);
}
animate();
```

When to do the work on the CPU

- ▶ If some computation is the same for all pixels, it is much more efficient to do it on the CPU side (once instead of millions of times).
- ▶ Computing the basis vectors of the image plane is the same for all pixels.
- ▶ Shift the computation of basis vectors to the JavaScript side and upload the basis vectors as uniforms (W01P2).
- ▶ When computing the basis vectors on the CPU side, the camera extrinsics are on the CPU side. Consider adding an interface for interactive view control.
- ▶ Note (again) that if you use `vec3f` as a uniform, it will be considered a `vec4f` in the buffer. Pad with a zero in the JavaScript array.

- ▶ The file `MV.js` included initially is a matrix-vector library for JavaScript (with `vec3`, `vec4`, `dot`, `cross`, `normalize`, etc.). This file is available on DTU Learn.

Next steps

- ▶ Define a HitInfo struct for recording information when an intersection between a ray and an object is found.
- ▶ Write ray-object intersection functions (for plane, triangle, and sphere) that take a generated ray as input and outputs HitInfo.
- ▶ Write an overall ray-scene intersection function that defines the objects in the scene and finds the closest intersection.
- ▶ Define a Light struct for recording information when a light source is sampled from a point of interest in the scene.
- ▶ Implement a function for shading of diffuse objects.

- ▶ More on these topics next week, but feel free to give it a try.