Worksheet 1

Ray tracing is the easiest and the most general technique for visualising 3D models. It is easy to create many sophisticated lighting effects using ray tracing, but ray tracing is still slower than rasterization. Hence, ray tracing is used primarily for rendering of photorealistic images in applications where image quality rather than a high frame rate is the primary concern. The purpose of the exercises for the first weeks is to help you learn how a ray tracer works.

Learning Objectives

- Implement ray casting (tracing rays from eye to first surface intersection).
- Use a pinhole camera model for generating rays in a digital scene.
- Compute intersection of rays with three-dimensional primitive objects (planes, triangles, spheres).
- Compute shading of diffuse surfaces using Kepler's inverse square law and Lambert's cosine law.

Getting Started

We will build a minimalistic framework for ray tracing using WebGPU. This means that we will write HTML and JavaScript files and test them by opening the HTML file in an internet browser. The advantages are that we get all the loading, display, and interface functionalities of the browser and it runs on any platform. All you need is an internet browser that supports WebGPU (check current support status: https://caniuse-.com/webgpu) and your favorite text editor for writing code.

Useful resources for you to familiarize yourself with WebGPU:

- Your first WebGPU app: https://codelabs.developers.google.com/your-first-webgpu-app
- B. Kenwright. Introduction to Computer Graphics and Ray-Tracing Using the WebGPU API. ACM SIGGRAPH Asia 2022 Courses, Article 1. 2022. https://doi.org/10.1145/3550495.3558218
- B. Kenwright. Web Programming Using the WebGPU API. ACM SIGGRAPH 2023 Courses, Article 21. 2023. https://doi.org/10.1145/3587423.3595543
- WebGPU specification: https://www.w3.org/TR/webgpu/
- WGSL - WebGPU Shading Language - specification: https://www.w3.org/TR/WGSL/

We will do most of the ray tracing work using WGSL. The WebGPU code needed on the JavaScript side will be kept at a minimum.

Ray Casting

In this first exercise, your job is to implement the very basics of ray tracing: ray generation, ray-object intersection, and shading of diffuse surfaces. This simple, non-recursive, visible-surface ray tracing is typically referred to as ray casting.

Save a .html and a .js file for each part that you solve. In the end, your lab journal will be a collection of these files documenting your progress and providing us with easy inspection of code and results.

1. Create an html file with a canvas element of resolution 512×512, and let it load a JavaScript file that initializes WebGPU and clears the canvas to black.
2. Create a script in the html file with WGSL code that defines a rectangle from $(-0.9, -0.9)$ to $(0.9, 0.9)$ and set its colour to $(r, g, b, a) = (0.1, 0.3, 0.6, 1.0)$. In the JavaScript file, get the text in the script and use it to create a shader module and a render pipeline. Set up a render pass that draws the four vertices of the rectangle as a triangle strip.

3. Change the vertices of the rectangle to be from $(-1.0, -1.0)$ to $(1.0, 1.0)$ so that the rectangle fills out the canvas and pass the image plane location from the vertex to the fragment shader. Create a struct (`Ray`) for holding ray information and generate rays using a pinhole camera model.[1] As a test, output the ray direction as the pixel color. Multiply by one half and add one half to ensure that you get positive color values. The result should be a dark blue image with red increasing from left to right and green increasing from bottom to top.

4. Create a struct (`HitInfo`) for recording hit information and implement ray-plane intersection, ray-sphere intersection, and ray-triangle intersection. Call these intersection functions in the fragment shader to render the default scene (specified below). Assign the designated object colour to the pixel when a ray intersects it. Use the light blue rectangle colour from before as background colour if no intersection was found. Adjust the maximum trace distance of your ray ($t_{\max}$) if you have an intersection to ensure that you end up with the closest intersection.

5. Create two uniform variables, one for the aspect ratio of the canvas and one for the camera constant (zoom). Make a zoom interface in HTML/JavaScript (use a slider or the scroll wheel of the mouse) that changes the camera constant. Re-render the scene when the zoom changes and demonstrate that you can accurately use the aspect ratio of the canvas to render with a non-square resolution.

6. Create a struct (`Light`) for returning the radiance (L_i) incident from and the direction ($\vec{\omega}$) toward a light source. Implement a point light `sample` function and a `shade` function for diffuse surfaces sampling the point light in the scene. Use Kepler's inverse square law of radiation and Lambert's cosine law.

Default scene description

Camera:	eye point	look-at point	up-vector	camera constant	
	$(2.0, 1.5, 2.0)$	$(0.0, 0.5, 0.0)$	$(0.0, 1.0, 0.0)$	1.0	
Plane:	position	normal	rgb colour		
	$(0.0, 0.0, 0.0)$	$(0.0, 1.0, 0.0)$	$(0.1, 0.7, 0.0)$		
Triangle:	$\boldsymbol{v}_0$	$\boldsymbol{v}_2$	$\boldsymbol{v}_2$	rgb colour	
	$(-0.2, 0.1, 0.9)$	$(0.2, 0.1, 0.9)$	$(-0.2, 0.1, -0.1)$	$(0.4, 0.3, 0.2)$	
Sphere:	center	radius	refractive index	shininess	rgb colour
	$(0.0, 0.5, 0.0)$	0.3	1.5	42	$(0.0, 0.0, 0.0)$
Point light:	position	intensity			
	$(0.0, 1.0, 0.0)$	π			

Reading Material

The curriculum for Worksheet 1 is (105 pages)

- **B** Chapters 1–2. *Introduction* and *Miscellaneous Math*.
- **B** Chapters 3–5. *Raster Images*, *Ray Tracing*, and *Surface Shading*.

Supplementary reading material:

- Frisvad, J. R. *Ray Generation Using a Pinhole Camera Model*. Lecture Note, Technical University of Denmark, August 2012.
- Frisvad, J. R. *Ray-Triangle Intersection*. Lecture Note, Technical University of Denmark, July 2011.

[1] See the short lecture note "Ray Generation Using a Pinhole Camera Model" for more details. It is available on DTU Learn.