Graphics and Imaging Lab

Assignment #2

Monte Carlo Direct Illumination

Modeling and Simulation of Appearance

Master on Robotics, Graphics and Computer Vision

Overview

In this assignment you will implement your first Monte Carlo renderer working with direct illumination. It will generalize the point-light renderer you implemented on Assignment #1, so that it efficiently handles multiple area light sources of arbitrary shape. After completing this assignment, you will be able to render area light sources and soft shadows with your rendering code. Note that this assignment builds on code from Assignment #1.

This assignment consists of two parts. In the first, you will get to implement Monte Carlo sampling techniques that warp an initially uniform distribution into a number of different target distributions. Note that all work you do in this assignment will serve as building blocks in later assignments when we apply Monte Carlo integration to render images.

In the second part, you will implement a very basic stochastic rendering algorithm, which computes direct light coming from area light sources and environment maps.

Important: Note that this assignment assumes you implemented `src/pointlight.cpp` and `src/direct_whitted.cpp` in Assignment #1.

Important: You do not need to implement every part of this assignment to earn the maximum grade. There are two parts marked as **Extra** (one in Section 1, other in Section 2). The parts that do not have an **Extra** label are mandatory, and their code will be re-used in future assignments.

0 Preliminaries

First, **download** the new scenes from the patch in Moodle (extract patch/* to Nori2/*, it also contains a small Windows fix). Then, before coding, compile and run

```
$ ./nori ../scenes/assignment-2/aliasing/aliasing.xml
```

Running this will render a scene similar to Figure 1 (left), with significant artifacts due to poorly sampling a high-frequency signal (the checkerboard). This is because a single (Monte Carlo) sample per pixel has been used here. Now open the scene at `../scenes/assignment-2/aliasing.xml`, and at the begining of the scene description you will see the following lines

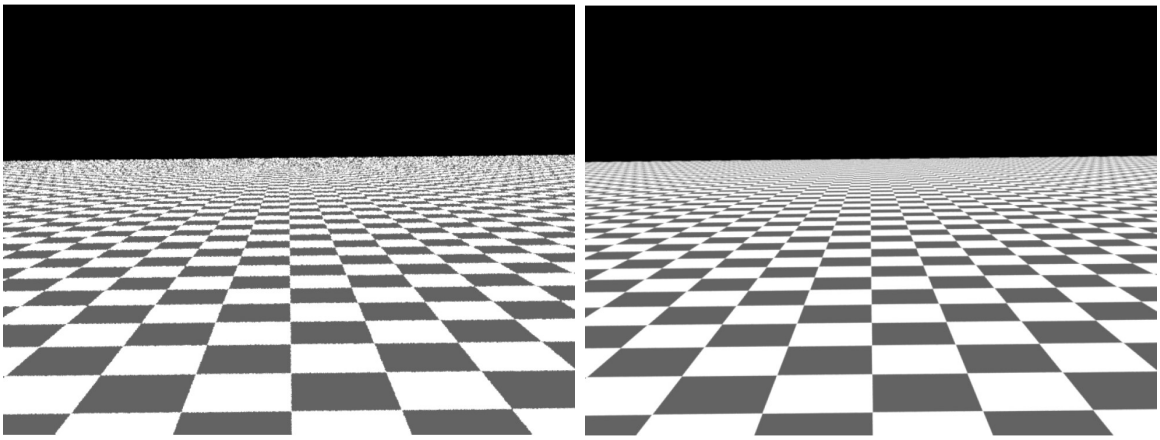

Figure 1: ***Left:*** *Example of aliasing due to undersampling a high-frequency signal (the checkerboard), rendered with a single sample per pixel.* ***Right:*** *A noise-free render of the same scene with* $\times 64$ *more samples per pixel; note how the reconstructed image is now smooth.*

```
...
<!-- Independent sample generator -->
<sampler type="independent">
  <integer name="sampleCount" value="1"/>
</sampler>
...
```

; there you can set the number of samples (per pixel) used to render your image. Setting `sampleCount` to 64, and rendering again, you will obtain an artifact-free version of the previous image (Figure 1, right), at the price of a significantly larger computation time ($\times 64$). In the following, set your sample count by changing this parameter in all the `xml` files describing the scene.

1 Monte Carlo Sampling (40%)

In this exercise you will generate sample points on various domains: the plane, spheres, and hemispheres. In particular, you are asked to implement a distribution function (a *pdf*) and a matching sample warping scheme. It is crucial that both are consistent with respect to each other (i.e. that warped samples have exactly the distribution described by the density function). Significant errors can arise if inconsistent warpings are used for Monte Carlo integration.

To validate the consistency of the warping procedures and associated pdfs, the base code includes an interactive visualization and testing tool to make working with point sets as intuitive as possible. After compiling all the code, you should see an executable named `warptest`. Run this executable to launch the interactive warping tool, which allows you to visualize the behavior of different warping functions given a range of input point sets

(`independent`, `grid` and `stratified`). Up to now, we only discussed uniform random variables which correspond to the "independent" type, and you need not concern yourself with the others for now. The provided `warptest` tool implements a χ^2 test to ensure that this consistency requirement is indeed satisfied. Note that passing the test does not generally imply that your implementation is correct–for instance, the test may not have enough "evidence" to generate a failure, or potentially the warping function and the density function are both incorrect in the same manner. Use your judgment and do not rely on this test alone.

What you have to do Implement the following missing functions in class `Warp` found in `src/warp.cpp`. This class consists of various warp methods that take as input a 2D point $(s,t) \in [0,1) \times [0,1)$ and return the warped 2D (or 3D) point in the new domain. Each method is accompanied by another method that returns the probability density with which a sample was picked. Our default implementations all throw an exception, which shows up as an error message in the graphical user interface of `warptest`. The slides on the course website provide a number of useful recipes for warping the samples and computing the densities, and the PBR textbook also contains considerable information on this topic that you should feel free to use.

Hint: When developing your sampling routines, remember that:

- A uniform distribution on a surface has probability $p(\mathbf{x}) = 1/A$ with A the area of the surface.
- In the case of constant $p(\mathbf{x})$, take into account that when $f(\mathbf{x}) = 0$ its pdf $p(\mathbf{x}) = 0$ (e.g. when uniformly sampling a 2D shape, $p(\mathbf{x}) = 0$ when $\mathbf{x}$ outside of the shape).

- **`Warp::squareToUniformTriangle` and `Warp:squareToUniformTrianglePdf` (10%):** Implement a method that transforms uniformly distributed 2D points on the unit square into uniformly distirbuted points on a triangle. Fill in code for uniformly sampling points from a unit triangle. Next, implement a probability density function that matches your warping scheme.
- **`Warp::squareToUniformSphere` and `Warp::squareToUniformSpherePdf` (10%):** Implement a method that transforms uniformly distributed 2D points on the unit square into uniformly distributed points on the unit sphere centered at the origin. Implement a matching probability density function.
- **`Warp::squareToUniformHemisphere` and `Warp::squareToUniformHemispherePdf` (5%):** Implement a method that transforms uniformly distributed 2D points on the unit square into uniformly distributed points on the unit hemisphere centered at the origin and oriented in direction (0,0,1). Add a matching probability density function.
- **`Warp::squareToCosineHemisphere` and `Warp::squareToCosineHemispherePdf` (5%):** Transform your 2D point to a point distributed on the unit hemisphere with a

cosine density function $p(\omega) = \frac{\cos\theta_\omega}{\pi}$, where θ_ω is the angle between a direction ω on the hemisphere and the north pole (i.e. direction (0,0,1)).

Extra `Warp::squareToBeckmann` and `Warp::squareToBeckmannPdf` (10%): Transform your 2D point to a point distributed on the unit hemisphere with a Beckmann half-vector distribution:

$$D(\omega_h) = \frac{e^{\frac{-\tan^2\theta_h}{\alpha^2}}}{\pi\,\alpha^2\cos^4\theta_h}, \tag{1}$$

where α is a user-specified roughness parameter, and θ_h is the half angle defining the angle between the half vector ω_h and the north pole. Remember that the pdf $p(\omega_h)$ of a normal distribution function $D(\omega_h)$ is $p(\omega_h) \propto D(\omega_h)\cos\theta_h$, and therefore it holds

$$p(\omega_h) = \mathcal{C}\int_0^{2\pi}\int_0^{\frac{\pi}{2}} D(\omega)\ \cos\theta\ \sin\theta\ \mathrm{d}\theta\ \mathrm{d}\phi = \mathcal{C}\int_0^{2\pi}\int_0^{\frac{\pi}{2}} \frac{e^{\frac{-\tan^2\theta}{\alpha^2}}}{\pi\,\alpha^2\cos^3\theta}\ \sin\theta\ \mathrm{d}\theta\ \mathrm{d}\phi = 1 \tag{2}$$

for the normalization constant $\mathcal{C}$. Use this identity to normalize your pdf.

`Warp::squareToUniformDisk` and `Warp::squareToUniformDiskPdf` (Extra 10%): Implement a method that transforms uniformly distributed 2D points on the unit square into uniformly distributed points on a planar disk with radius 1 centered at the origin. Next, implement a probability density function that matches your warping scheme.

Remember that you should validate whether your sampling routines are correct, passing all χ^2 tests of the above warpings using `warptest.exe`. Take screenshots of the result of the test for each implemented warping method and store them as `png` images with names `UniformTriangle.png`, `UniformSphere.png`, `UniformHemisphere.png`, `CosineHemisphere.png`, `Beckmann.png`, and optionally `UniformDisk.png`, respectively; submit these images following the instructions at the end of this document.

2 Direct Light – Emitter Sampling (60%)

In this second part, you will extend the simple Whitted-style renderer you implemented in Assignment #1 (`direct_whitted.cpp`) to take advantage of Monte Carlo integration when handling arbitrary light sources beyond simple point lights.

Your new integrator will compute the local illumination integral discussed in class:

$$L_o(\mathbf{x},\omega_o) = L_e(\mathbf{x},\omega_o) + L_r(\mathbf{x},\omega_o) \tag{3}$$

$$= L_e(\mathbf{x},\omega_o) + \int_\Omega L_i(\mathbf{x},\omega_i)\, f_r(\mathbf{x},\omega_i,\omega_o)\ \cos\theta_i\,\mathrm{d}\omega_i, \tag{4}$$

where L_o is the outgoing radiance at a surface position $\mathbf{x}$ in direction ω_o as the sum of L_e and L_r the emitted and reflected radiance at $\mathbf{x}$ respectively. The reflected radiance is defined

as the integral on the hemisphere Ω of the incoming radiance L_i at $\mathbf{x}$ from direction $\omega_i \in \Omega$, times the BRDF f_r. The angle θ_i is the angle between ω_i and the surface normal at $\mathbf{x}$. Generally, L_i and L_o are related to each other using the ray tracing operator $\mathrm{r}(\mathbf{x}, \omega)$, which returns the nearest surface position visible along the ray with origin $\mathbf{x}$ and direction ω, i.e.

$$L_i(\mathbf{x}, \omega) = L_o(\mathrm{r}(\mathbf{x}, \omega), -\omega), \tag{5}$$

and the above integral is thus defined recursively. In this assignment, we focus on direct illumination only and therefore truncate the recursion after the first light bounce. This means that the integral is now given by

$$L_o(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + \int_\Omega L_e(\mathrm{r}(\mathbf{x}, \omega_i), -\omega_i)\, f_r(\mathbf{x}, \omega_i, \omega_o)\, \cos\theta_i \, \mathrm{d}\omega_i. \tag{6}$$

During rendering, NORI will approximate this integral with the Monte Carlo estimate

$$L_o(\mathbf{x}, \omega_o) \approx \frac{1}{N} \sum_{k=1}^{N} \left(L_e(\mathbf{x}, \omega_o) + L_e(\mathrm{r}(\mathbf{x}, \omega_i^{(k)}), -\omega_i^{(k)})\, f_r(\mathbf{x}, \omega_o, \omega_i^{(k)})\, \cos\theta_i^{(k)} \right), \tag{7}$$

where $\omega_i^{(k)}$ is the **sampled** incoming direction. Note that the summation and averaging is taken care of by NORI itself, and you do not need to write code for this. The integrator in this assignment (and future ones) only need to compute the values of a single sample at a time (i.e. the inner term of the sum in Equation (7)).

We will be be working with emitters that have a finite extent, which means that they can be directly observed by the camera. This is why the first summand (L_e) in Equation (7) is needed: It returns emitted radiance towards the camera when light sources are visible on screen. Also note that Equation (7) defines outgoing radiance at surface points. To visualize incident radiance at the camera, you will need one more invocation of the ray tracing operator. Like in the first assignment, we provide an API that you should use to implement the sampling and evaluation operations for emitters, as well as storage and query of emitters in the scene.

To complete this task, you will need to fill in code for the following steps:

1. Generate a sample from a randomly chosen light source, and evaluate its direct contribution (`direct_ems.cpp`). Remember that NORI will call the integrator several times per pixel, and average each individual contribution to evaluate the sum in Equation (7).
2. Uniformly sample a point on a triangle mesh (`mesh.h` and `mesh.cpp`).
3. Generate samples from a triangle mesh, and computer the sample radiance/PDF (`area.cpp`).

To help you with completing the assignment, we've broken up the steps into smaller subtasks:

2.1 Integrator (20%)

Create a `direct_ems.cpp` file, and create a new integrator called `DirectEmitterSampling`. Use the NORI_REGISTER_CLASS to register the new integrator as "`direct_ems`" as:

```
NORI_REGISTER_CLASS(DirectEmitterSampling, "direct_ems");
```

> Use `DirectWhittedIntegrator` that you implemented in `direct_whitted.cpp` as a base for this new integrator. Note that `DirectWhittedIntegrator` does iterate over all light sources (see l.31 of `direct_whitted.cpp`) and does not account for visible light sources.
>
> Do not forget to add `direct_ems.cpp` to `CMakeLists.txt` and run CMake before compiling.

The `DirectEmitterSampling` integrator should approximate the Equation (6) as follows

$$L_o(\mathbf{x}, \omega_o) \approx L_e(\mathbf{x}, \omega_o) + \frac{1}{N} \sum_{k=1}^{N} \frac{L_e(\mathrm{r}(\mathbf{x}, \omega_i^{(k)}), -\omega_i^{(k)})\, f_r(\mathbf{x}, \omega_o, \omega_i^{(k)})\, \cos\theta_i^{(k)}}{p_\Omega(\omega_i^{(k)})} \tag{8}$$

$$= L_e(\mathbf{x}, \omega_o) + \frac{1}{N} \sum_{k=1}^{N} \frac{L_e(\mathbf{x}_l^{(k)}, -\omega_i^{(k)})\, V(\mathbf{x}_l^{(k)} \to \mathbf{x})\, f_r(\mathbf{x}, \omega_o, \omega_i^{(k)})\, \cos\theta_i^{(k)}}{p_\Omega(\mathbf{x}, \mathbf{x}_l^{(k)})} \tag{9}$$

where $\omega_i^{(k)} = |\mathbf{x}_l^{(k)} - l|$ was generated using emitter sampling (i.e. sampling the position $\mathbf{x}_l^{(k)}$ at the light source). For that, first randomly sample a light source in the scene using `Scene::sampleEmitter`; then, sample the light source to obtain the sampled `EmitterQueryRecord` using `Emitter::sample`. Do not forget to initialize your sample point at `EmitterQueryRecord::ref` to the point you are illuminating **before** sampling. Remember that $p_\Omega(\mathbf{x}, \mathbf{x}_l^{(k)})$ is the product of the pdf of choosing the light source l and the pdf of $\mathbf{x}_l^{(k)}$ at the light source.

Finally, make sure to include the directly visible radiance (the first L_e term). You can check if the intersected material is an emitter using `its.mesh->isEmitter`, and may call its `Emitter::eval` method to compute the visible radiance.

Demonstrate your integrator by rendering the two scenes `serapis_whitted.xml` and `serapis_ems.xml` in folder `scenes/assignment-2/serapis/`: The first one will render the scene by using the `DirectWhittedIntegrator` from Assignment #1, while the second will use `DirectEmitterSampling`, in both cases with one sample per pixel. The result should look like something like Figure 2, where `DirectWhittedIntegrator` shows a significant aliasing in the borders but the shading is smooth, while `DirectEmitterSampling` exhibits lots of variance since only one light source is randomly sampled at each sample. Now increase the sample budget of both scenes to 64 samples per pixel. In both cases the result should converge to the exact same solution.

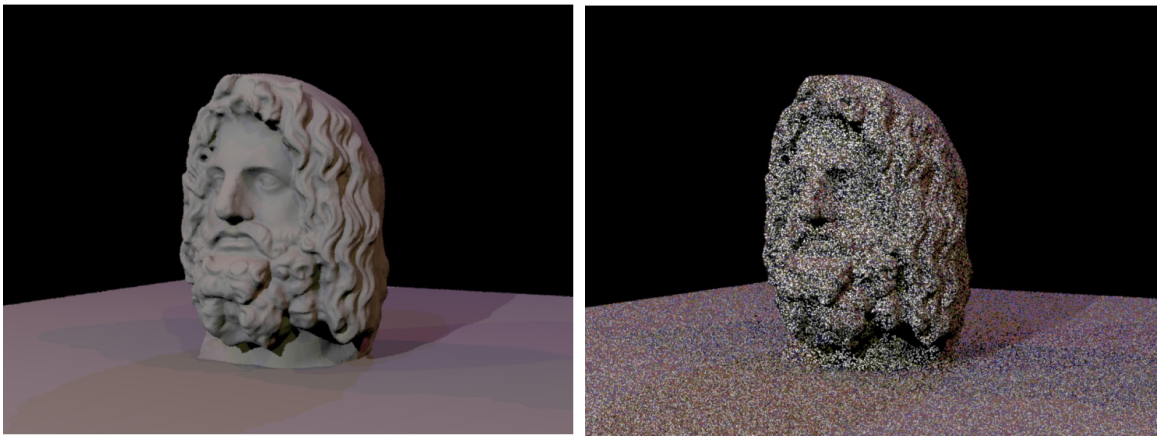

Figure 2: *The* SERAPIS *bust illuminated by seven colored point light sources, rendered using the* `DirectWhittedIntegrator` *(left) and the* `DirectEmitterSampling` *integrator (right) with one sample per pixel. The rendering cost of each image is 453 and 137 ms respectively. Note that in both cases there are significant aliasing in the borders, though in the image on the right it is masked by the Monte Carlo noise of light sampling.*

Extra 10%: To improve convergence, you can implement importance sampling of the light sources, so that brighter sources would potentially receive more samples. In order to support importance sampling, update `Scene::sampleEmitter` so that the probability of sampling a light source is proportional to their emitted radiance. You can use `DiscretePDF` for sampling the light sources. You can construct your discrete sampler before you start rendering, in `Scene::activate`. Show the improvement of using the naïve sampling against the radiance-based one.

2.2 Mesh Area Light (30%)

Here you will implement a new type of emitter: an *area light*. As opposed to the point lights implemented in Assignment #1, area lights have finite area. In this case, you will implement area lights associated to a triangle `Mesh`. For convenience, we split this task into multiple subtasks.

Triangle Sampling Familiarize yourself with the `Mesh` class (`Mesh.cpp` and `Mesh.h`) to see how vertices, faces and normals are stored. Fill the function `Mesh::samplePosition` that samples a point in a triangle of the mesh. Use `Warp::squareToUniformTriangle` from Part 1 to uniformly sample the barycentric coordinates of the triangle, and use these barycentric coordinates to interpolate between the positions of the triangle's vertices. You should compute the sampled point, the surface normal, and the UV coordinates (if any) at that point. For now, assume the mesh only has a single triangle.

Then extend your code for `Mesh::samplePosition` to handle multiple triangles. To do

this, sample a triangle of the mesh at random, with probability proportional to the triangle area: You can use `DiscretePDF Mesh::m_pdf` (see `include/dpdf.h` for details on `DiscretePDF`) for sampling all triangles. Fill the function `Mesh::pdf` that returns the PDF of sampling a point on the mesh (**Hint:** For uniform distributions, the PDF is always $1/\text{area}$; you can compute the area of each triangle by using the function `Mesh::surfaceArea`, and the area of the full mesh is the normalization factor of the discrete PDF, which you can obtain through `DiscretePDF::getNormalization`). Note that `DiscretePDF` needs to be initialized before use; do that in `Mesh::activate`.

> **Hint:** The function `Mesh::samplePosition` receives a 2D random sample point. You can reuse one of the dimensions for sampling the both the triangle and one barycentric coordinate of the triangle. For that you can use the function `DiscretePDF::sampleReuse` for both sampling the triangle, and updating the random sample.

Area Emitter Take a look at `area.cpp`. The `AreaEmitter` is a helper class that can be attached to a `Mesh` to turn it into an emitter. Fill in `AreaEmitter::eval`: This method should return the radiance of the light, as seen from a specified direction. This function takes in an `EmitterQueryRecord`, which is defined in `emitter.h`. Be careful to check that the outgoing direction is above the emitting hemisphere of the triangle. Also note that `AreaEmitter` supports textured area lights; you can access the radiance of these area lights using `AreaEmitter::m_radiance->eval()`, passing the texture coordinates in the introduced `EmitterQueryRecord`.

Using the methods implemented in `mesh.cpp`, fill in `AreaEmitter::sample` and its corresponding pdf in `AreaEmitter::pdf`. Remember to convert the positional pdf p_S returned by `Mesh::pdf` to solid angle p_Ω measure. These are related by the factor

$$p_\Omega(x, x_l) = p_S(x_l)\frac{||x - x_l||^2}{|n_l \cdot \omega_i|}. \tag{10}$$

> **Hint:** To help you narrow down bugs, you can test your integrator for a scene with a single emitter and a single triangle. This scene setups excludes any potential bugs introduced from incorrect use of `DiscretePDF`.
> Also, be careful when filling the sampled normal in `Mesh::samplePosition`, since some meshes do not have explicit normals in their vertices (i.e. the normals vector `Mesh:m_N` might be empty): In this case, you must compute explicitly the normal of the triangle. An example on how to compute the point, normal, and UV coordinates of a point from the barycentric coordinates of a triangle in NORI can be found in l.452-493 of `accel.cpp`.

We have prepared a couple of scenes for demonstrating your code: `bunny`, `cbox`, and `table`. Of course, you are more than welcome to prepare additional scenes to demonstrate

your code. For helping with comparisons, we have also included a converged version of the `cbox` scene (1024 spp), which you can use as ground truth (see `scenes/assignment-2/references`).

2.3 Environment Light (10%)

Here you will implement a new type of emitter: an *environment light*. This type of light source is an spherical light source that is placed at the infinite. Think of it as the light coming from the sky. While the basic stuff is already implemented in `environment.cpp`, you need to fill the sampling routines needed to support illumination from the environment map. In particular, implement the functions `EnvironmentEmitter::sample()` and `EnvironmentEmitter::pdf()` by uniformly sampling a direction on the sphere. You can use the provided `serapis/serapis_env.xml` to demonstrate your code. We also include a converged render (2048 spp) that you can use as ground truth for validation (see `scenes/assignment-2/references`).

3 References

You may find the following general references useful:

- *"Physically Based Rendering, Third Edition: From Theory To Implementation"* by Matt Pharr, Wenzel Jakob, and Greg Humphreys. Morgan Kaufmann, 3rd edition, Nov 2016. `http://www.pbr-book.org/`
- *"Global Illumination Compendium - The Concise Guide to Global Illumination Algorithms"*, Philip Dutre, 2003. `https://people.cs.kuleuven.be/~philip.dutre/GI/`

Feel free to consult additional references when completing projects, but remember to properly acknowledge them in your submission. When asked to implement feature X, we request that you don't go and read the source code of the implementation of X in some other renderer, because you will likely not learn much in the process. The PBR book is excluded from this rule. If in doubt, get in touch with the course staff.

4 What to submit?

You should submit all your rendered figures and the source code you generated in this assignment. The submission should be a a `zip` file with name `p2_NIP1_NIP2.zip`, and with the following structure:

- A file `README.txt` including the name of the authors (as sanity check) and all references consulted during the development of the assignment.
- A folder `./figures` including all generated figures (both `exr` and `png`); the figures should be named with the name of the `xml` scene, appended with the number of samples used to generate these figures (e.g. if you generated a render of the scene `table_env.xml` with 16 samples per pixel, then you should submit the figures `table_env_16.exr` and `table_env_16.png`. Do not forget to include the screen captures from Part 1 of the assignment.
- A folder `./src` including `warp.cpp`, `area.cpp`, `mesh.cpp`, `direct_ems.cpp`, `enviroment.cpp`, and any other source file you modified or added in your implementation.

In case your file gets too big for the submission system, you can just include a link to a shared folder (Google Drive, OneDrive, etc.) with additional figures in `README.txt`.

The **deadline** for this task will be just before the first session of Assignment #3 corresponding to the lab group you are enrolled on:

- **Group 1 (Tuesdays B): November 4, 2024.**
- **Group 2 (Thursdays A): October 30, 2024.**