1 Methods and Materials

1.1 Initial Approach Using Ray Tracing in C++

Ray tracing is a rendering technique that simulates the way light interacts with objects to generate images with high visual realism. Unlike rasterization, which is used in most real-time graphics, ray tracing calculates the color of pixels by tracing the path that light would take as it travels through a scene. This path is traced backwards from the viewer's eye to the light source, a method known as *backward ray tracing*.

Initially, the simulation of optical flat measurements was attempted using a ray tracing technique in C++. Ray tracing is a powerful computational method for simulating the path of light through media. It models the propagation of light rays and their interactions with surfaces, which is particularly useful in optical studies where the understanding of light behavior in precise environments is necessary. However, this approach did not yield successful results due to complexities in accurately modeling the intricate interference patterns that are critical in optical flat evaluations.

1.2 Successful Simulation Using Python

After the initial setbacks, a more successful simulation was developed using Python. This method utilized the concept of intersecting planes with the surface under test (sft) (see figure 1.1). Each plane was separated by half the wavelength of the light transmitted through the optical flat, allowing for the simulation of interference patterns by modeling how these planes interact with the sft irregularities as can be seen in figure 1.2.

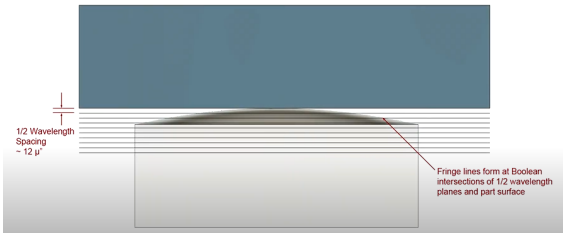

Figure 1.1: The blue area at the top represents an optical flat placed over a test surface shown in grey. The white lines, spaced half a wavelength apart, are interference fringes. These fringes form at points where the height difference between the optical flat and the surface matches multiples of half the light's wavelength. The pattern of these fringes reveals the topography of the surface beneath the flat. Figure taken from [**?**].

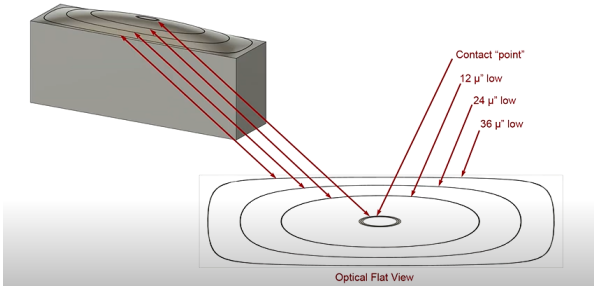

Figure 1.2: A 3D model with an optical flat above a test surface and a 2D projection. Red lines represent light rays reflecting at different gaps between the optical flat and the surface. These variations are visible in the 2D "Optical Flat View" as concentric circular fringes, each indicating a constant air gap. This method precisely measures the deviations in surface flatness, with each ring in the diagram quantifying specific irregularities. Figure taken from [**?**].

1.2.1 Backend Development: Python Scripts

To implement this method, several Python scripts were created:

- **Intersection** - This function, as can be seen in figure 1.3, handles the mathematical computation of plane and surface intersections, critical for modeling how light interacts with different surfaces.
- **Disk, Cylinder, Flat_surface, Shape3D, STLFigure** - These classes contribute to generating and handling various geometrical shapes and rendering the final 3D images which represent the simulation results.

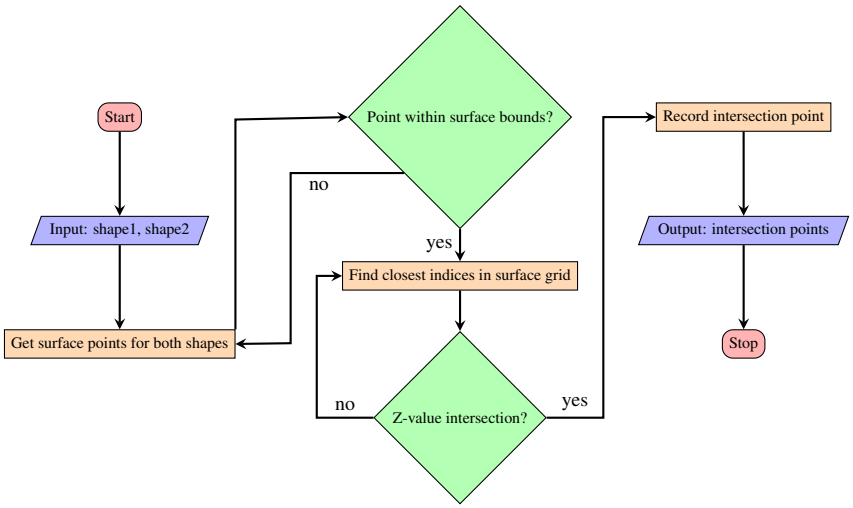

Figure 1.3: Flowchart of the intersection function: initially, it extracts the x, y, and z coordinates from both (`shape1`) and (`shape2`) using their `get_surface()` methods. Lists are then initialized to store the coordinates of the intersection points. The function determines the bounds of (`shape1`) and iterates over each (`shape2`) point, checking if it lies within these bounds. For each in-bound point, it identifies the closest surface points in the x and y dimensions. If a (`shape2's`) z-value matches closely with the z-value at these points, within a tolerance of ± 0.01, the point is recorded as an intersection. The function ultimately returns these intersection points as NumPy arrays.

1.2.2 Frontend Development: Graphical User Interface (GUI)

A comprehensive GUI developed through `Gui.py` facilitates user interaction with the simulation, allowing for real-time adjustments and visualization of results. The GUI is structured into several functional tabs, each enhancing usability and educational value. Screenshots of the GUI can be found in the appendix.

1.2.2.1 GUI Components Detailed Descriptions

File Tab This tab manages file operations such as importing and exporting configurations. Functions include:

- **Export Configuration** - Opens a file dialog to save configurations in JSON format. After selection, it executes the export shapes command and confirms the operation.
- **Import Configuration** - Similar to export, this function loads configurations from a JSON file, redraws shapes based on the new data, and confirms the import.

Insert Tab This tab facilitates the insertion of various 3D shapes into the simulation. Each subtab provides specialized controls for inserting and configuring objects such as cylinders, disks, flat surfaces, optical flats and STL figures.

View Tab This tab allows users to customize the visual aspects of the application: users can modify the text color, background color, and font.

Help Tab This tab includes a PDF viewer for accessing documentation directly within the GUI so users can effectively utilize the simulation tool without Internet connection.

Log Tab This tab records and displays logs of user actions and system responses.

3D View Tab This tab provides a dynamic 3D visualization space where users can interact with the models.

1.2.2.2 3D View Tab Detailed Description

The `ThreeDViewTab` class is the most important tab of the GUI, designed to provide interactive 3D visualizations of the simulated environment. This tab is important to assess the accuracy and spatial relationships of the models within the simulation. It integrates a Matplotlib 3D plot within the interface and renders graphical representation of the shapes.

Interaction and Visualization

- **Shape Manipulation Methods**: Includes adding new shapes, drawing updates on the Matplotlib canvas, and managing shape deletions — all reflected visually in real-time.
- **Interactive Features**: Allows users to interact with the 3D models via mouse and keyboard inputs to rotate, zoom, and pan across different views, providing a comprehensive understanding of the model's geometry and spatial properties.

Utility Functions

- **Update States**: Dynamically adjusts the state of GUI controls based on the context (e.g., whether shapes are present or a particular shape is selected).
- **Calculate Intersections**: Uses the intersection function as described above to determine and visualize intersections between different shapes.

1.3 Reconstruction

The `Help Tab` and the `Log Tab` have the same functionalities as above, so I won't discuss them here.

Image Processing Tab The `Image Processing Tab` class extends the capabilities of the graphical user interface by incorporating image processing techniques.

1. **Fourier Transforms:** This tab utilizes the Fourier transform, a mathematical technique that transforms spatial data into frequency data. This is crucial for identifying periodic structures in images, as well as for noise reduction and image enhancement.
2. **Frequency Filtering:** After transforming the image to the frequency domain, specific frequencies like the carrier frequency can be isolated or suppressed to enhance certain image features or remove unwanted artifacts. This process involves applying a mask to the Fourier transform of the image that selectively retains or removes certain frequencies.
3. **Phase Extraction:** This feature extracts the phase from the complex representation of the Fourier-transformed image. The phase of the image carries important structural information that is not visible in the amplitude alone. Extracting and analyzing the phase is essential to reconstruct three-dimensional shapes from two-dimensional images.
4. **Phase Unwrapping:** Phase images, by nature, are wrapped around a 2π interval which creates discontinuities that can complicate further analysis. Phase unwrapping corrects these discontinuities to produce a continuous phase map that accurately represents changes across the image.
5. **Height Map Calculation:** Utilizing the unwrapped phase, this function calculates height maps from phase images, where the optical path difference at each pixel translates to physical height variations. This is particularly useful in surface profilometry and interferometric microscopy.
6. **Height Map Smoothing:** The calculated height maps can often contain noise and other high-frequency components that obscure the true surface features. Smoothing, typically using Gaussian filters, helps to mitigate these effects, enhancing the visual quality and interpretability of the height maps.
7. **Visualization and Interaction:** Integration with `matplotlib` for visualizing results allows users to see the processed images in various forms, including 3D plots of height maps. This interactive visualization aids in the qualitative analysis of the data and provides a dynamic way to explore the impact of different processing parameters.