# Code Modernisation

# University College Cork (UCC)

# Advanced Programming with Java

# Refactoring and Redesigning a Ray-Tracing Framework

## Introduction

The original ray-tracing framework was developed as an instructional tool, relying heavily on JavaFX and had a monolithic design. While functional, it lacked encapsulation, modularity, and extensibility. Its rendering pipeline relied heavily on JavaFX, which limited its flexibility and made integration with non-graphical workflows difficult. The goal of this refactoring project was to modernize the framework, improve its usability, and establish a robust foundation for future development.

## Key Design Decisions

### Modularization and Separation of Concerns

The framework was restructured to emphasize encapsulation and modularity. Responsibilities were divided among new distinct classes, including Scene, Controller, Camera and Renderer. Each class now handles specific tasks: the Scene class manages objects and lights, Renderer focuses on pixel calculations, and RayTraceAPI provides a user-friendly interface for scene setup and rendering. The Main and Driver classes, which were tightly integrated into the framework, were replaced with the new RayTraceAPI. This change separated the core functionality from its dependency on JavaFX, enabling the framework to function as an independent library. Additionally, RayTraceAPI abstracts away technical complexities, offering high-level methods for adding objects and lights.

### Scene Management Improvement

The introduction of a Scene class further streamlined scene management. This class organizes objects and lights through intuitive public methods, making the codebase clearer and easier to work with. The methods for adding objects (e.g., addSphere, addPlane) and lights (e.g., addAmbientLight, addPointLight) were refactored to ensure consistency, making the process easier for users. A "current surface" system was implemented, allowing users to set a surface once and apply it to multiple objects without redundant configuration.

### Enhancements to Rendering Pipeline

The rendering pipeline was transitioned from JavaFX to BufferedImage, which allowed the framework to function independently of any specific user interface library. This change made the framework more portable and easier to integrate with other workflows. Multithreading was incorporated into the rendering process, enabling parallel computation of pixels, which significantly improved rendering performance for large images. Complex methods like Shade were decomposed into smaller, more focused functions (e.g. addReflections, addAmbientLightColour). This not only improved code readability but also adhered to the single responsibility principle, making the rendering logic easier to maintain and extend. The Renderer class was updated to handle pixel shading calculations. Methods like exportImage now include file validation and error handling, ensuring reliability during image generation and export. Additionally, recursive ray tracing was implemented to support reflection, with the maximum number of bounces made configurable through the API.

### Project Management and Testing

Git was used extensively to manage the refactoring process, allowing easy comparison of changes, rollback of errors, and traceability of contributions. Extensive documentation was added throughout the codebase, including both inline comments and Javadocs. This ensures that the framework is accessible to future developers. Testing focused on verifying that refactored components produced results consistent with the original implementation, as well as ensuring stability under various configurations. Tests were implemented after each change to catch any unintentional errors introduced by the modification.

## Conclusion

The modernization of the ray-tracing framework introduced a modular structure with key improvements, such as the creation of the RayTraceAPI for simplified scene creation, a shift from JavaFX to BufferedImage for rendering, and the implementation of multithreading for parallel pixel computation. By replacing the monolithic design with clear responsibilities across classes and adopting modern Java features, the framework is now more efficient, easier to use, and better suited for future development.