WORKSHEET C

Version 1.0
Computing
COMP270

Kate Bergel

Introduction

In this worksheet you will use 3D vectors and matrices to complete the implementation of a basic ray caster.

Begin by **forking** the following git repository:

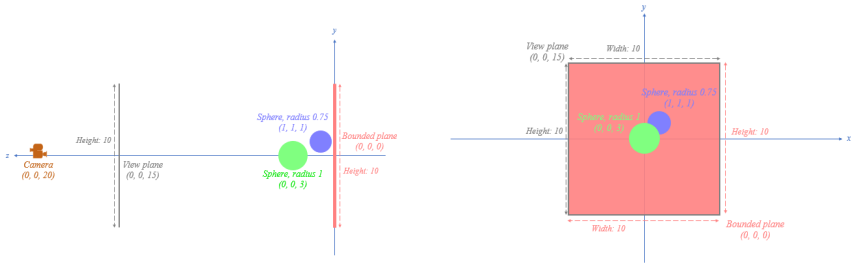
`https://github.com/Falmouth-Games-Academy/comp270-worksheet-C`

Complete the tasks described below, remembering to **commit** your work regularly. To submit your work, open a **pull request** from your forked repository to the original repository.

Note: The tasks below can be completed by filling in the relevant functions marked in `Camera.cpp`, `Object.cpp` and `Matrix3D.cpp`, however you are welcome (and encouraged) to add helper methods and member data to these classes (and the `Plane` class, which derives from `Object`), expecially for Task 4. It should not be necessary to deviate from the main program structure; if you do decide to do so, your reasons should be stated in code comments and/or git commit messages.

Aside from the first task, which must be completed in order to actually see anything, the implementations do not depend on one another and may be attempted in any order.

Background



(a) Side view (b) Top view

Figure 1: A diagram of the scene setup.

The provided skeleton project implements a basic raycasting camera setup, as shown in Figure 1. The objects - two spheres and a plane - are already in the scene, though they cannot be seen to start with. The camera is initially at the point (0, 0, 20) in world space, facing along the negative z-axis, and can be moved in a limited fashion, in directions along the world coordinate axes, with the following key presses:

- `a`: translates the camera in the negative x direction ("left")

- `d`: translates the camera in the positive x direction ("right")
- `s`: translates the camera in the negative y direction ("up")
- `w`: translates the camera in the positive y direction ("down")
- `q`: translates the camera in the negative z direction (towards the objects)
- `e`: translates the camera in the positive z direction (away from the objects)

In addition, the camera can be "zoomed" in/out by pressing the up/down arrow keys, which moves the view plane away from/towards the camera.

Task 1: let there be light

The camera creates a picture by choosing a colour for each 'pixel' of the view plane based on which object in the scene is visible "through" the pixel's location in the view plane from the camera's position. The visibility test is carried out in `Camera::updateScreenBuffer` by finding which objects are encountered by "rays" sent from the camera through the centre of each pixel. The number and directions of the rays will be determined by the view plane properties stored in the `Camera::m_viewPlane` struct (**note**: the view plane distance is measured along the *camera's viewing direction*).

Implement `Camera::generateRays` to compute a unit-length `Vector3D` direction for each ray and store it at the indices corresponding to the pixel it passes through in `Camera::m_pixelRays` - you will need to make sure the data structure is sized appropriately! **Note:** Pixel indices are assumed to correspond to the coordinate frame of the scene, where the y axis points upwards, *not* the SDL screen coordinates.

Once you've generated the rays, you should be able to see the spheres as in Figure 2.

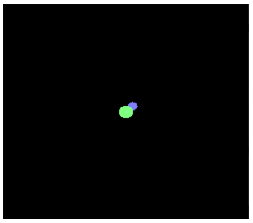


Figure 2: Scene view with ray generation.

Task 2: in plane sight

There is another object in the scene that can't be seen yet, because the test to determine whether a ray intersects it is missing. **Implement** `Plane::getIntersection` (in `Object.cpp`) to return `true` if the ray intersects it within the bounds of its width and height, and set the input parameter `distToFirstIntersection` to be the distance along the ray from its source that the intersection occurs.

Following this, your scene should look like Figure 3.

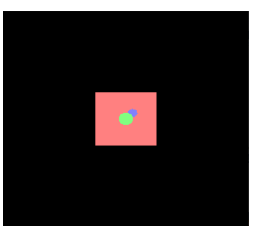


Figure 3: Scene view with plane intersection.

Task 3: spinning around

Now you can see things, but only in a limited way, as the camera direction is fixed: although holding `shift` down when pressing the camera control keys modifies the values in `Camera::m_rotation`, these values are not used when determining the scene layout relative to the camera. **Modify** `Camera::updateWorldTransform` to compute the transform matrix that will transform the objects into camera space before the ray intersections are performed, and `Matrix3D::inverseTransform` so that they can be transformed back again!

Note: there are various ways you in which you can perform the rotation, which may be relative to the camera (as a first person player view) or the world origin/other point of interest (as in a modelling tool such as Maya). Any solution is acceptable, provided it allows reasonably intuitive movement (you may edit the key mappings in `Application::processEvent` if it helps) and preserves the proportions of the scene.

With rotations in place, you should be able to view the scene from more interesting angles, such as in Figure 4.

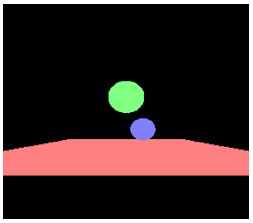


Figure 4: Scene view from a different angle.

Task 4: round things that look round

So far, everything is looking rather flat because we're setting all the pixels to the same colour, regardless of "how much" they intersect the object. One way to add depth to a scene is to use Phong shading; shadows can also be created by sending rays that intersect an object back to a light source to see if any other objects block their path, and modifying the resulting pixel colour accordingly.

Choose (or **invent**) and **implement** a method for giving the scene a "more 3D" look - points will be awarded for creativity as well as adherence to existing techniques! You will almost certainly need to extend the functionality of more than one class; make sure to explain your chosen strategy and/or provide references to source material in the comments.

Marking Rubric

Criterion	Weight	Refer for Resubmission	Adequate	Competent	Very Good	Excellent	Outstanding
Basic competency threshold	30%	A reasonable attempt at the worksheet was not submitted by the formative deadline.	A reasonable attempt at the worksheet was submitted by the formative deadline. There is no evidence of academic misconduct.				
Functional coherence	40%	None of the tasks have been attempted.	Task 1 has been attempted and partially completed.	Task 1 has been successfully completed.	Two tasks (including task 1) have been successfully completed.	Three tasks (including task 1) have been successfully completed.	All four tasks have been successfully completed.
Maintainability	30%	The code is only sporadically commented, if at all, or comments are unclear. Few identifier names are clear or inappropriate. Code formatting hinders readability.	The code is well commented. Some identifier names are descriptive and appropriate. An attempt has been made to adhere to a consistent formatting style. There is little obvious duplication of code or of literal values.	The code is reasonably well commented. Most identifier names are descriptive and appropriate. Most code adheres to a sensible formatting style. There is almost no obvious duplication of code or of literal values.	The code is reasonably well commented, with appropriate high-level documentation. Almost all identifier names are descriptive and appropriate. Almost all code adheres to a sensible formatting style. There is no obvious duplication of code or of literal values. Some literal values can be easily "tinkered".	The code is very well commented, with comprehensive appropriate high-level documentation. All identifier names are descriptive and appropriate. All code adheres to a sensible formatting style. There is no obvious duplication of code or of literal values. Most literal values are, where appropriate, easily "tinkered".	The code is commented extremely well, with comprehensive appropriate high-level documentation. All identifier names are descriptive and appropriate. All code adheres to a sensible formatting style. There is no duplication of code or of literal values. Nearly all literal values are, where appropriate, easily "tinkered".