WORKSHEET 3: RAYCASTING

Version 1.1
Computing
COMP270

Kate Bergel

Introduction

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

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

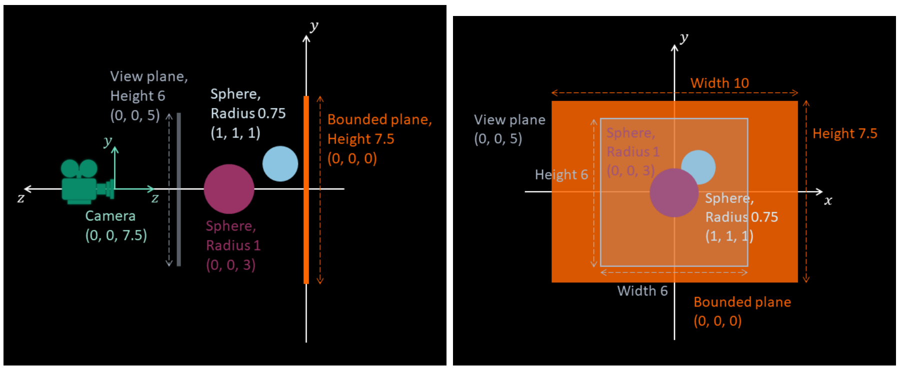
https://gamesgit.falmouth.ac.uk/projects/COMP270/repos/comp270-worksheet-3

Complete the **FOUR** 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 and Object.cpp, however you are welcome (and encouraged) to add appropriate helper methods and member data to relevant classes, especially 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) Camera view

Figure 1: A diagram of the scene setup with world-space coordinates and dimensions of the objects.

The provided skeleton project implements a basic raycasting camera setup, as shown in Figure 1. Some 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, 7.5) in world space, facing along the negative z-axis, with its x-

and y-axes aligned with the world coordinates so that they form a **left-handed** coordinate space, as opposed to the **right-handed** one of the world.

The camera 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 ("down")
- `w`: translates the camera in the positive y direction ("up")
- `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.

Additional settings: If you need to adjust the colours of the objects to see them more clearly, you can do so in `Application::setupScene()`.

Task 1: let there be light

"Happiness can be found, even in the darkest of times, if one only remembers to turn on the light."

— Albus Dumbledore

The camera creates a picture by choosing a colour for each 'pixel' of the view plane based on which object is visible through the pixel from the camera's position, which is determined by sending 'rays' through the centre of each pixel and seeing which objects they intersect (the overall process happens in `Camera::updatePixelBuffer`, but you shouldn't need to edit that - for now).

The number and directions of the rays will be determined by the view plane properties stored in the `Camera::m_viewPlane` struct, which can be accessed from within `Camera` methods as e.g. `m_viewPlane.distance`. **Note** that the view plane `distance` is measured along the **camera's viewing direction**, i.e. the positive z direction in camera space starting from the camera, so that the view plane initially has coordinates (0, 0, 2.5) in camera space and (0, 0, 5) in world space.

The view plane is divided into a grid of 'pixels' with the resolution in each direction giving the number of pixels in that direction as shown in Figure 2, which also illustrates the measurement of the distance from the camera to the view plane and the pixel indexing.

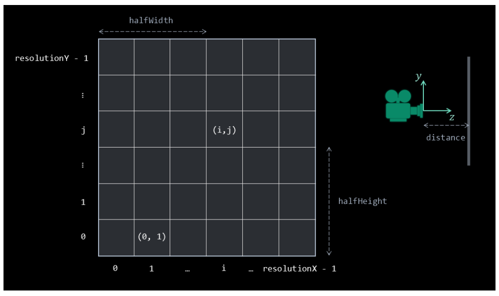


Figure 2: View plane dimensions and positioning

Complete the following two steps in Camera.cpp to determine the direction of the rays through the view plane:

(a) **Add the calculations** in `Camera::init()` for finding the width and height of each pixel (in camera-space units), storing the results in `m_pixelWidth` and `m_pixelHeight` (these values are used in various functions).

(b) **Implement** `Camera::getRayDirectionThroughPixel()` to return the **normalised direction in camera space** of the ray starting at the camera's position and passing through the **centre** of pixel in the view plane with grid index `(i, j)`. Note that the grid indices should correspond to the coordinate frame of the scene/camera, 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 3. You'll notice that the image appears fairly blocky, as a relatively low resolution has been set to keep processing times reasonable; you can reduce the blockiness by increasing the values of `Camera::m_viewPlane.resolutionX` and `Camera::m_viewPlane.resolutionY`.

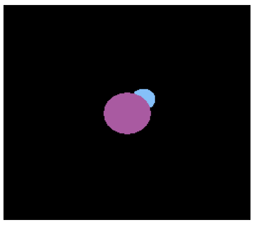
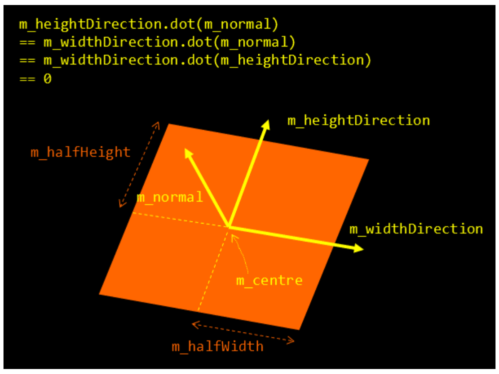


Figure 3: Scene view with ray generation.

Task 2: in plane sight

There is still one object in the scene that can't be seen yet, because the test to determine whether a ray intersects it is missing. The object is a rectangular plane, which is defined by its normal `m_normal` and (optional) dimensions `m_halfWidth` and `m_halfHeight` that give the distance of the edges from the world-space centre point (`m_centre`) along perpendicular vectors `m_widthDirection` and `m_heightDirection` as shown in Figure 4. If either of the half-width or half-height of the plane is zero, the plane is assumed to be infinite.



```
m_heightDirection.dot(m_normal)
== m_widthDirection.dot(m_normal)
== m_widthDirection.dot(m_heightDirection)
== 0
```

Figure 4: Properties of a rectangular plane.

Implement `Plane::getIntersection()` (in Object.cpp) to return `true` if the ray intersects the plane **within the bounds of its width and height**, and **set** the parameter `distToFirstIntersection` to be the distance along the ray from its source that the intersection occurs (i.e. the parameter t in the equations). Following this, your scene should look like Figure 5.

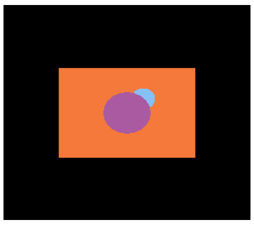


Figure 5: Scene view with plane intersection.

Test that your code works for a variety of plane orientations, not just ones that are aligned with the camera, by changing the plane normal and 'up' directions passed to the plane constructor in `Application::setupScene()` - for example, to add a plane with `m_normal` = <0.5, 0.5, 1.0> and `m_heightDirection` (or `up`) = <-0.5, 1.0,-0.25>, you would run the following code, to create a scene that looks like Figure 6:

```
m_objects.push_back(new Plane(Point3D(),
                Vector3D(0.5f, 0.5f, 1.0f),
                Vector3D(-0.5f, 1.0f, -0.25f),
                10.0f, 7.5f));
```

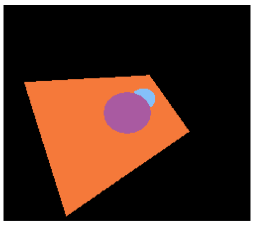


Figure 6: Scene view with tilted plane.

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.

"Unfortunately, no-one can be told what the Matrix is. You have to see it for yourself."

— Morpheus

Modify `Camera::updateWorldTransform()` to compute the **full transform matrix** of the camera in world space, which is also the matrix that will transform the objects from camera space to world space. The result should be stored in `m_cameraToWorldTransform`; you can access the element of a `Matrix3D` at row `r` and column `c` using the `()` operator for both get and set:

```
// set the value of the matrix at row r, column c
m_cameraToWorldTransform(r, c) = value;
// get the value of the matrix at row r, column c
value = m_cameraToWorldTransform(r, c);
```

Note that there are various ways in which you can perform the rotation, which may be relative to the camera (as a first person player view) or around the world origin/other point of interest (as in a modelling tool such as Maya). In addition, there are multiple ways to interpret Euler angles, resulting in several possible calculations for converting them into a matrix; any solution is acceptable, provided it allows the scene to be viewed from different angles whilst preserving the relative positions and dimensions of the objects. It is highly likely that you will want to modify the key mappings in `Application::processEvent()` to match your particular choice of rotation setup. Also, **remember** to include the translations and conversion from right- to left-handed coordinates.

With rotations in place, you should be able to view the scene from more interesting angles, such as in Figure 7.

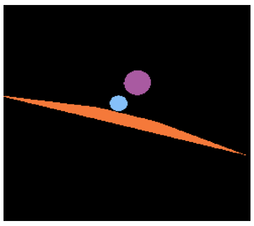


Figure 7: Scene view from a different angle.

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 far away the object is or how the intersected surface is oriented, when in reality there would be shadows, reflections and a number of other effects adding depth to the scene. There is a variety of methods to simulate these effects, known as shading, with different levels of complexity, including

- applying a 'distance fog'-like effect to modulate the colour intensity depending on the distance from the camera.
- modifying the colour based on the surface orientation (given by the surface normal).
- using ray tracing to simulate reflecting the rays from the object's surface and create shadows by testing whether those rays intersect another object before reaching a light source.
- applying a standard shading technique, such as Phong or Lambert shading, to approximate lighting effects more efficiently than with ray tracing.

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, and for interesting as well as realistic effects!

The colour for a pixel is currently computed in `Camera::getColourAtPixel()`, which is supplied with the pixel indices as input arguments, and has access to the closest object to the pixel and the distance to the point of intersection via the `ObjectInfo` stored in `m_pixelBuf` - you can also retrieve the direction of the ray that intersects the object by calling `getRayDirectionThroughPixel(i, j)`, plus all the other members of the `Camera` class are of course available. For simple techniques, this may be enough, however you could also extend the functionality by saving additional information from the intersection test (see comments in the code for how).

Marking Rubric

All submissions and assessment criteria for this assignment are individual. To **pass** this assignment (achieve 40% or more), you must submit a reasonable attempt at the worksheet by the formative deadline stated on LearningSpace.

Criterion	Weight	Near Pass	Adequate	Competent	Very Good	Excellent	Outstanding
Basic competency threshold	30%	A reasonable attempt at the worksheet was not submitted by the formative deadline. Evidence of breach of academic integrity.					
PROCESS: 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.
PROCESS: 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".