DVA338 Raytracer Assignment

2025

Formalities

This document is a collection of tasks that need to be done to complete the assignment. You will be given a high-level descroption of the tasks, but you are **required** to supplement this with relevant parts of the course book and lectures. You will need to learn the course content, think about the problems and solve them using your own ideas and thoughts.
While the tasks have a natural progression in terms of features, it is beneficial to read through the whole section before starting on the tasks.
The assignment is separated into a practice section and two separate tasks for examination, marked as "Part 1" and "Part 2". The assignment text concludes with a series of *extra* assignments, which are optional.
To pass the assignment, "Part 1" and "Part 2" should each be submitted separately in Canvas and presented in the assigned Lab session. For information regarding the extras, please see Section 3.

Background

Ray tracing is a well-known algorithm used in different rendering tasks (e.g. movies, games), where the result is calculated based on backwards tracking of the light path to the camera. Ray tracing is well-suited to capture effects such as shadows and reflections. More advanced effects can also be produced using various ray shooting schemes, e.g., diffuse inter-reflections and caustics.
In this assignment, you will create an application that performs rendering using ray tracing.

Startup

The startup code is provided in a zip file. Notice that for this assignment, we will not draw the graphics directly on the screen, this means that you will NOT see this image on the screen. Instead, the application generates image files in the working directory of the executable file for the program: `DVA338_Raytracer_output.BMP` or `DVA338_Raytracer_output.PPM`.
Find this file in your project's working directory (on different platforms the file might end up in a different sub-folder). Opening the file using a suitable image viewer application should show you the image seen in Figure 1 below.

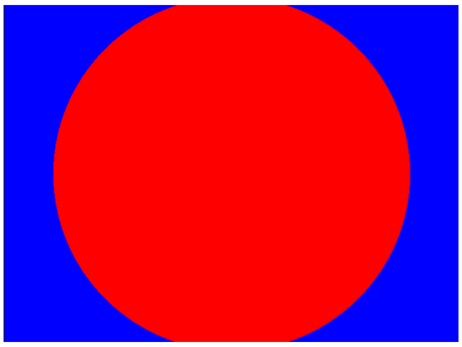


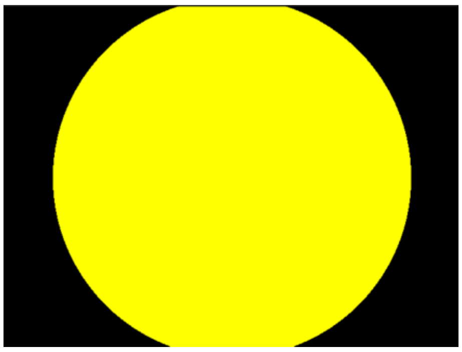
Figure 1: The image generated from the startup code. A red circle with a blue background.

Warm-up Exercises

To give you a better understanding of the given code and to prepare you for the lab tasks, the following exercises are available.
Note: Make sure that you have a way to reset the scene back to the original, before (and between) attempting each of these exercises.

Exercise - Colouring

Change the colour of the sphere. Then change the colour of the background.
For example, if you change the sphere colour to yellow and the background to black, the result should be the same as Figure 2:



bg in RayTracer.h
colour in RayTracer.cpp

Figure 2: The startup scene with different colours.

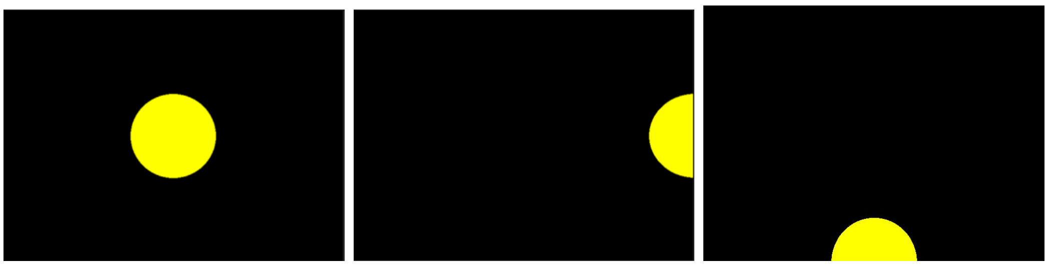
Exercise - Positioning

Recreate Figures 3a-3c below by 1) *Only* changing the position of the camera itself and 2) *Only* changing the properties (size, location) of the sphere(s).

3a - main.cpp r=0.5f
3b - main.cpp x=4.0f
3c - main.cpp y=-3.0f

Exercise – Scene Composition

Now add a few more spheres to the scene to recreate the scene in Figure 4. This should give you a good understanding of how to compose scenes programmatically.



(a) A smaller circle, centered (b) A smaller circle, right (c) A smaller circle, bottom

Figure 3: A small yellow circle at different positions of the image.

Note: *The references in Figure 4 and 5 has every sphere in a different colour to make the transformation of each variant obvious. You do not need to colour the spheres for this exercise.*

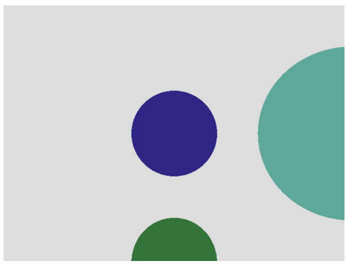


Figure 4: A scene with three circles of different colours and radius lengths. The rightmost circle is twice as big as the others.

Exercise - Ray Firing

For a better understanding of the ray firing process, reproduce the following images by, starting from the previous scene, *only* changing how the primary view rays are generated. **Note**: *you are not meant to change the location of the spheres in the code to achieve these variants. You are asked to change the way the primary rays are generated.*

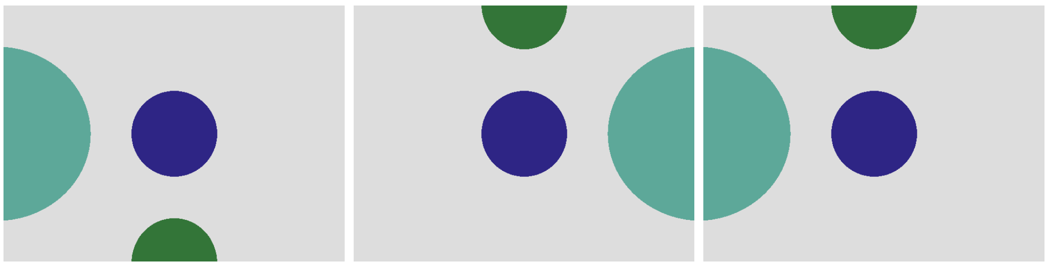


Figure 5: Different variants of the previous scene, achieved by changing properties in the ray firing process.

5a - RayTracer.cpp sizeX = -4.0f
5b - RayTracer.cpp sizeY = -3.0f
5c - both

1 Assignment Tasks – Part 1

As you have seen from the *warm-up* section, the rendering of the spheres are quite flat. Your task in this assignment is to successively build up the ray tracer to allow the rendering of nice images.

1.1 Additional Data Structures - Light and Material

In computer graphics, the look of an object is determined through a process called *shading*. **Your task** is to create data structures that can store information required for the shading process (Note: This is *not the final version of the data structure*, we will build on them as we go):

- *Material***:** This represents the colour of the material and must be represented at component level (red, green and blue), each varying between 0 and 1, where 1 is the full colour.
- *Light***:** This will be represented by a position and a colour.

We can now make spheres more visually distinct. Look at Figure 6, how many spheres are should there be in the portrayed scene? What colours do the spheres have? How are they positioned in 3D space? Recreate the following scene considering the answers that you gave to the previous questions.

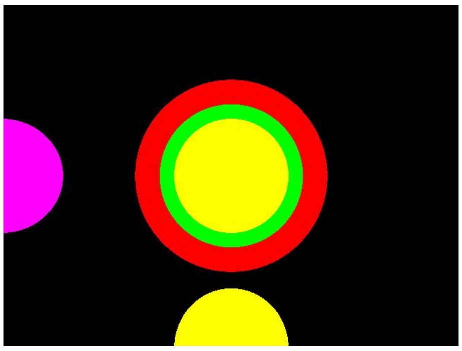


Figure 6: Different coloured circles, three of which are layered in the centre.

1.2 Improve Hit Detection to find the *Closest Intersection Point*

It is important to ensure that the intersection test for ray tracing must be designed in a way so that it returns the closest object that the ray intersects with *in the direction of the ray*. However, the provided code skeleton does not do that.
For example, if we were to change the order in which the spheres are added to the scene from section 1.1 (for instance by the middle yellow sphere being added to the scene *first*), then the expected result is what is shown in Figure 7.
Your task is to improve the hit function for the sphere so that it detects and updates the hit record structure **only when a closer hit in front of the ray is found**. Note that we are only interested in hits *in the forward direction of the ray*. **Additionally**, compute **the**

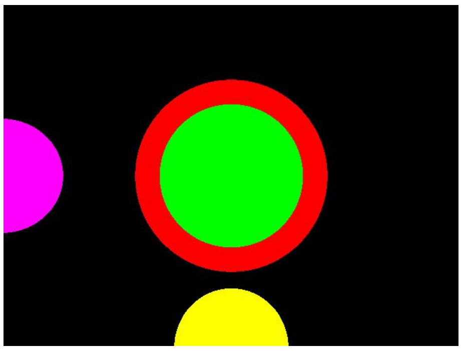


Figure 7: Different coloured circles, now only two layered in the centre.

location of the point on the sphere which is being hit as well as **the surface normal for that point**. This will be required in the next step.
Test this by creating a scene with several spheres obscuring each other, and ensure that the scene is rendered the same way irrespective of the order that the spheres are added in.

1.3 Rendering

Currently the colour of the sphere is only dependent on the material that has been set. To get more realistic renderings we will now add a local illumination model that accounts for the lighting setup in the scene. This will allow us to show depth and shadows. *If you have not already done so, please place a light into the scene.*
We shall use the Phong shading model, which consists of three *components*: *ambient, diffuse and specular.* This means that we might need to rethink and update the way material and lights are represented.
As the previous scene is not well-suited to show these effects, you shall need to reproduce the scene used in sections 1.3-1.5 below. As a hint, please note that the red sphere in the following scene is centred at (x,y,z) = (0,0,-10) with a radius of 1 length unit.

1.3.1 Render scene with only ambient

In this part, we shall render the scene using only the ambient light. The function of the ambient light is to ensure that no object is completely dark by giving every surface some base light. Thus the ambient light intensity is typically chosen to be quite low.
Your task is to render the objects only using the ambient part of the lighting model. Note: in the example we have increased the ambient intensity to clearly show the result. Additionally we have changed the background colour to a light grey to visually distinguish it better. You can see the expected result in Figure 8a below.

1.3.2 Render Scene with only Diffuse Lighting

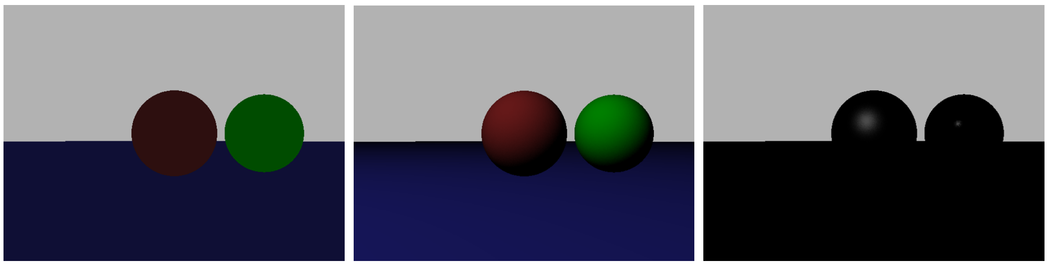
To finally get some contours and see the results as actual spheres, we need to account for the shape of the surface itself. This requires considering the surface normal, which shows up in the diffuse lighting component.
Your task is to render the objects using *only* the diffuse part of the lighting model. Note: to make it easy for you, you should be able to easily turn off the ambient part.

We can see the expected result in Figure 8b. It is now possible to see some contours. Consider your understanding of the shading calculation. Where is the light placed relative to the spheres in the scene above? Please reproduce it.

1.3.3 Render scene with only specular

The final part of the lighting model is the specular highlights. The result shall be some version of Figure 8c. Note that you should define the material in such a way that you get different sizes on the highlights.

1.3.4 Put it all together



(a) Ambient only (b) Diffuse only (c) Specular only

Figure 8: A scene rendered with only one part of the lighting equation component.

Once we have computed all components separately as shown in Figure 8, the result should be combined to get the final colour, as shown in Figure 9.

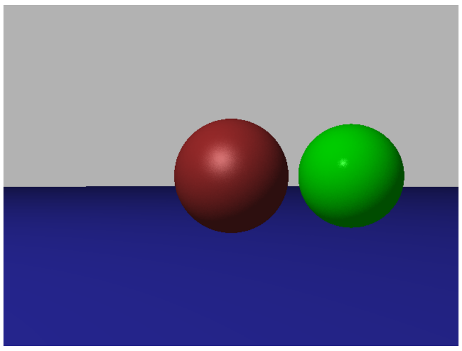


Figure 9: A scene rendered with all components of the lighting equation combined.

1.4 Shadows

In this part, we shall compute the points in shadows. Shadows occur when an object is between the intersected point and the light source.
Your task is to find such cases. We shall do this in three steps:

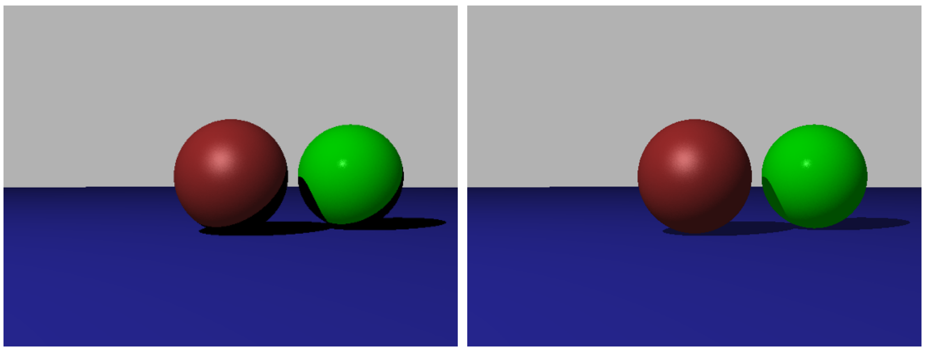
1. Make a new function that can take a ray and return whether it hits *anything*.
2. Fire a *shadow ray* from the hit point towards the light to determine whether the point is in shadow.

3. If the point is in shadow, *do not apply any lighting.*

Once we have added shadows and gotten the image in Figure 10a, we can see that the coloured spheres are in fact resting on a surface.

1.5 Shadows with Ambient Light

Now, augment the shadow lighting calculation such that points in shadow will only be rendered with ambient lighting (instead of no lighting at all). The augmented result should look like Figure 10b.



(a) Shadows are completely dark. (b) Shadows show only ambient colour.

Figure 10: The scene rendered with shadows.

In Figure 11, we are seeing a phenomenon commonly called *shadow acne.* If you encounter this result, consider the fact that floating-point numbers are represented using approximations in computers. With this in mind, what would happen if you started the shadow ray *exactly* from the intersection point?

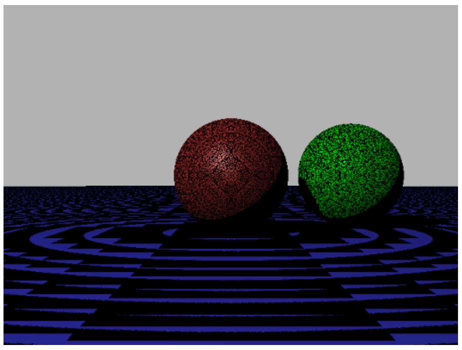


Figure 11: An example of the phenomenon known as *shadow acne.*

2 Assignment 1 - Part 2

In this part, we will wrap up the assignment by adding reflections, render another primitive type and some effect through random perturbation.

2.1 Reflections

2.1.1 Perfect Mirror Reflections

Now we are going to add mirror reflections to the scene. The effect can be achieved by firing a new ray from the intersection point and calculating the colour of the object hit by the reflection ray. This colour will be used on the original object's instead of its own colour (mirrors show other objects in them, they do not have their own colour anyway).
Your task is to implement this addition to the colour calculation. Note that you should implement some some way to avoid infinite reflections!

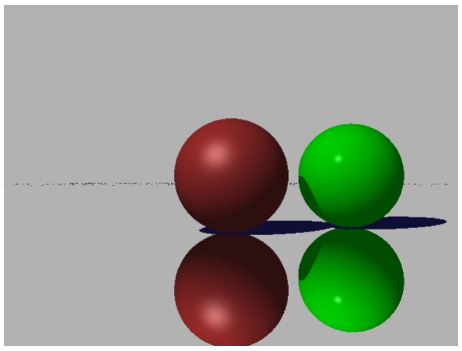


Figure 12: The scene with the floor turned into a perfect mirror.

In Figure 12, the "floor" has been made a perfect mirror. We can see that it completely reflects the spheres placed on top of it as well as the background colour.

2.1.2 Partial Reflections

While we can use this to simulate mirrors, most realistic materials only partially reflect light. It should be possible to control the amount of light that gets reflected by setting some material properties.
Your task: Add some way to control the reflectiveness of a material. Use this value to blend the reflected colour with the object's own colour.
Hint: Consider that the course book lists the following "typical values" for the Phong exponent: 10 - eggshell, 100 - mildly shiny, 1000 - glossy, 10 000 - nearly mirror-like. Clearly there is a connection between the Phong shininess component and reflectiveness.
In Figure 13, we have added handling of partial reflection. The spheres have different levels of reflectiveness. Here we can see something interesting, there is actually a *fourth sphere* that has been in the scene the whole time. See if you can spot it.

2.2 Triangle Rendering

Your task is to extend the program to handle rendering of triangles. You may consider a triangle as a set of three connected 3D points in space.

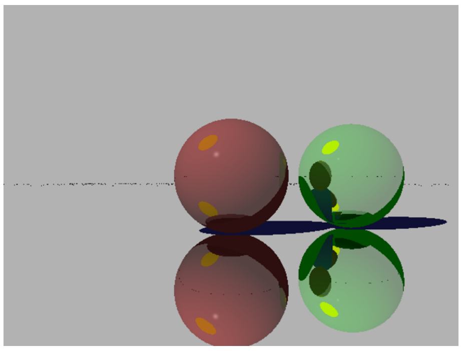


Figure 13: The scene with the floor turned into a perfect mirror and partially reflective spheres.

Note: You will need to investigate how to calculate a hit between a ray and a triangle, but once that is completed, adding triangles to the scene should be straightforward.

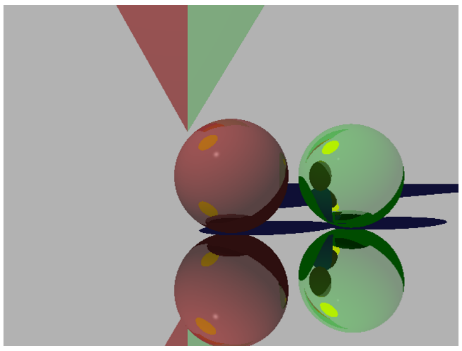
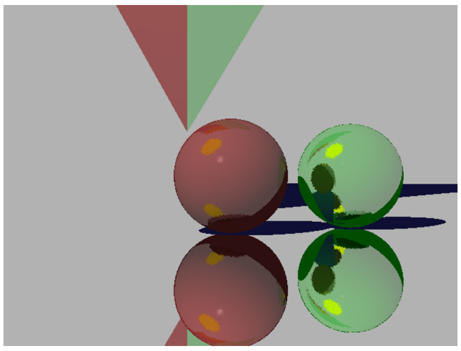


Figure 14: The scene with four triangles added.

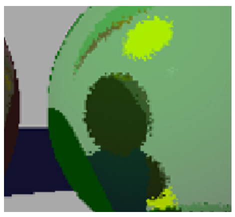
Compare Figure 13 to Figure 14. There are four triangles that have been added or replaced other geometry in the scene. **Hint:** For the non-obvious triangles, compare the figures at the horizont line.

2.3 Randomly Perturb Normals

By changing the normals we can achieve different types of effects. The simplest such effect is to randomly perturb them a small amount, which gives reflections a *fuzzy* look.
Your task is to implement such an effect. You should then be able to render an image such as the one below.
From the result in Figure 15 we can clearly see that compared to Figure 14, the reflections on the spheres are much fuzzier, especially on the edges of reflected objects, as seen in Figure 15b.



(a) The full scene.



(b) A zoomed-in view of the reflection on the green sphere.

Figure 15: Scene with fuzzy reflections.

3 Extras

We conclude the assignment by providing pointers to additional features for the interested student to continue with.
Each feature listed below gives several extra points (EP). If the "extra" feature is developed, examined and approved before the deadline for assignment 1, then you will get that number of extra points in their final written exam. Make sure to read the Study Guide carefully to learn about terms and conditions.

3.1 Soft Shadows (2 EP)

Allow the rendering of soft shadows by adding multiple light sources and extend the shadow and lighting calculation to account for several lights.

3.2 Refraction (2 EP)

Add refraction to properly render transparent materials.

3.3 Anti-Aliasing (2 EP)

As you can see in the renders, the boundaries of the objects are sometimes rendered in a jagged way. Smooth out this type of issues by sampling each pixel multiple times, then blend the resulting colours together to calculate the final pixel colour.

3.4 Multithreading (1 EP)

As you have probably experienced, ray tracing is very slow once several effects have been added. However, it is also quite parallelizable. Use multithreading to speed up the rendering. **Note:** You must provide an explanation how the work is evenly divided on each thread.