Project 1: Ray Tracer

COMP30019 - Graphics and Interaction
Semester 2, 2022

This project is **individual** work (30 marks).

Due: 4th September 2022, 23:59 AEST

Assignment Brief

You are tasked with building a *ray tracer*. Your ray tracer will output a single static PNG image, based on an input 'scene' file and command line arguments. We have provided you with a template C# implementation that you will need to complete. **We are not using the Unity engine in this project**, however, you may find that some of the theory in this assignment will be transferable to Unity development (particularly the maths). The assignment is broken down into numerous stages and steps. Our expectations for each stage and the respectively allocated marks are outlined in detail below. You should aim to complete each ste

There are variou ene. Almost always, th tional complexity and realism. A ray tracing based approach can p , however this comes with a significant computational cost t able for real-time rendering. Even if there are no real-time rende *all* rays in a scene is computationally intractable.

Template code

You will be given a GitHub repository to work on your project that is already pre-initialised with the template code. This is a **private** repository, so you may commit/push to it without worry of other students having access to your work. You are expected to use GitHub from the start through to the end of the project, and should commit and push frequently. **We won't accept** submissions not hosted in your private repository.

A link to accept the assignment and automatically create your template repository is provided on the Canvas project page (where you found this specification document). Note that you may submit the assignment as many times as you wish – only the **latest** will be marked.

Stage 1 - Basic ray tracer (9 marks)

You will first implement the **basic** functionality of a ray tracer. At its core, ray tracing is an application of geometry and basic linear algebra (vector maths will become your bread and butter!). For example, a ray of light can be modelled by two three-dimensional vectors: a starting **position** and **direction**. Surfaces, light sources, and other entities in the environment can also be defined using vectors. Using geometry, it is possible to calculate how a ray *reflects* off a surface, or perhaps even *refracts* through it. Ultimately we are interested in simulating rays of light propagating throughout the environment, interacting with various surfaces, before finally reaching the viewer as pixels on their screen. If we are clever in utilising 'real-life' physical models for these interactions, we can generate incredibly realistic scenes.

In this first stage you will implement some basic vector functionality, and figure out how to shoot a ray for each pixel in a rendered image. We won't yet be worrying about materials, lighting, shading, etc. Such fancy stuff will come later in the assignment.

Stage 1.1 - Familiarise yourself with the template

Before writing any code, try to understand how the template provided to you works. We have already taken care of quite a few details for you, such as input and output handling. A sample input scene is provided to you in a text file (`tests/sample_scene_1.txt`), and a parser for this file has been written so you can access objects and resources directly within the `Scene` class (`src/scene/Scene.cs`). The core ray tracing logic (which you will write) should be imp `e.cs`. This method takes an `Image` as derive propertie will automaticall

Try running the project so that you can see this in action. Open u Visual Studio Code (or your preferred environment), and ru

```
dotnet run -- -f tests/sample_scene_1.txt -o output.png
```

Although this looks like a bit of a mouthful at first, all it is doing is running the project with two command line arguments: an input text file (`-f`) and an output image file (`-o`). The input file will be read and parsed, and the output image written accordingly. Open the generated output file, and you will notice the entire image is black, since no ray tracing has been implemented yet. Before continuing, test your understanding by modifying the project code to output the image entirely in white instead.

Hint: Try using some loops inside the `Render()` method. The `Image` class has `Width` and `Height` properties which should be handy for determining the loop bounds. These properties are already determined by the command line arguments `-w` and `-h`, if specified.

[1] Note that you may need to install the .NET SDK if you haven't already, otherwise `dotnet run` won't be available. Make sure you install **version 6.0**. You can find it at `https://aka.ms/dotnet-download`.

Now take a look at the main `Program.cs` file. In the `OptionsConf` class, you can see all of the potential command line arguments and their default values (these are the values used if that argument is not specified at runtime – e.g., not entered on the command line). Don't change these default values, instead, pass values using the appropriate flags on the command line, if you want to change parameters. At this point it's worth stressing that **you should not modify the `Program.cs` file at all**. Doing so risks our automated test suites breaking when running your project during marking (see the 'Submission' section for details).

Stage 1.2 - Implement vector mathematics

We have provided you with a C# struct template for representing a three-dimensional vector (`src/math/Vector3.cs`). Write code to complete the missing operations which are currently empty methods. Note that for convenience we have overloaded operators[2] such as `+, *, /`. This is a handy language feature that allows us to perform vector arithmetic concisely:

```
Vector3 a = new Vector3(0, 1, 0);
Vector3 b = new Vector3(1, 1, 0);
Vector3 c = a + b; // We overloaded '+' so c = (1, 2, 0)
```


As well as basic arithmetic, you will also need to complete functions to compute the dot product and cross product (at least). The dot product will tell you how much two vectors point in the sa
the vector which is p
give you the z-axis).

It is **strongly** recommended that you te e thoroughly. Vectors are utilised everywhere in this can lead to a major headache down the line.

Stage 1.3 - Fire a ray for each pixel

We have already provided you with a 'ray' structure (`src/math/Ray.cs`). Notice that it is simply a position (origin) and a direction, both represented as vectors. While it is possible to trace rays forwards from light sources in the scene, it is far more efficient to trace rays backwards from the camera. This is because most rays in the scene will never be seen by the viewer, and computing these would be a waste of resources.

Inside the `Render()` method, write code to iterate through each pixel and construct a corresponding ray that fires into the world. The biggest challenge here will be converting from a two-dimensional pixel coordinate to a three-dimensional ray. You might find it useful to consult external resources about the maths here.

[2] `https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/operator-overloading`

In this project we want you to use a left-handed coordinate system. The camera should be situated at the origin of the scene – (0, 0, 0) – looking forward along the positive z-axis ('into the screen'), with the positive x-axis pointing 'right', and the positive y-axis pointing 'up'. You should ensure that there is a horizontal field-of-view (FOV) of 60°. As a sanity check, a ray at the very center of the rendered image should point in the direction (0, 0, 1). Rays at the corners of the image should have directions $(\pm i, \pm j, k)$, where $i = j$ if the image is square. You should ensure that your solution works when different output image widths/heights are specified. For non-square images the vertical FOV should vary to maintain the correct aspect ratio.

Stage 1.4 - Calculate ray–entity intersections

In this project a scene can contain three types of primitive entities – planes, triangles and spheres. If you haven't already, open the template classes provided in the `src/primitives` folder:

- `Plane.cs` – Represented by a point (`center`), and a vector representing the direction it faces (`normal` - i.e., perpendicular to the actual surface of the plane). Note this defines an 'infinite' plane.
- `Triangle.cs` – Represented by three points (`v0, v1, v2`). A clockwise winding order defines the front face of the triangle.
- `Sphere.c`

All of these classes should contain a method called `RayHit` as its output. The returned `RayHit` structure contai ed during ray tracing: the *incident* ray direction, the *pos* *normal* of the surface at that position. It is also possible that there is no hit at all, in which `null` should be returned instead of a `RayHit` i t the `Intersect()` method for all three primitive entities. Again, you may find it helpful to research common mathematical approaches to these problems if you are stuck.

Stage 1.5 - Output primitives as solid colours

You are finally ready to generate some graphical output! Earlier you computed a ray for each pixel in the image. Extend this code to check for intersections with primitives in the scene. You will need to make further additions to the `Render()` method. For each ray, iterate through every entity in the scene and check whether there is an intersection between the ray and the entity. If so, you should set the corresponding pixel colour to the colour of the object. Ensure you correctly handle cases where there is more than one entity that coincides with a ray.

In case your object-oriented programming is rusty, here is a template for looping through all primitives/entities in the scene and checking if `ray` intersects with them:

```
foreach (SceneEntity entity in this.entities)
{
    RayHit hit = entity.Intersect(ray);
    if (hit != null)
    {
        // We got a hit with this entity!
        // The colour of the entity is entity.Material.Color
    }
}
```

Note that `entity` is an interface, so we don't know exactly which type of primitive it is (plane, triangle or sphere), but that does not matter since we are only interested in the intersection itself.

We have provided you with sample outputs in the `images` folder, so you have an indication of how your output should look for stages 1 and 2 respectively.

Stage 2 - Lighting and materials (9 marks)

In this stage you will extend the ray tracer to handle lighting, and model different types of *materials*. Some materials are more trivial to compute than others, and this complexity ultimately boils down to how light rays interact with them.

Note that every entity is assigned a *material*. The material contains properties that allow us to calculat her it is opaque, reflecti cs to see our representation of a materi olor property in the previous stage.

Stage 2.1 - Diffuse materials

We will first consider the case where a ray coincides with a *diffuse* surface which is directly illuminated by a light source. When light hits an 'ideal' diffuse material, it scatters uniformly in all directions. This means it is viewer-independent, and the intensity only varies depending on the angle of incidence between the light source and the surface. Diffuse lighting is so trivial to compute that it is regularly used in real-time rendering techniques (not just ray tracing).

In this stage you need to extend the ray tracer to handle materials with the `Diffuse` type. Objects should be smoothly lit when this is implemented correctly. As a starting point, take note of where you set the colour of a pixel currently. Instead of outputting the material colour directly, you should compute it based on the following function:

$$C = (\hat{N} \cdot \hat{L}) C_m C_l$$

...where $\hat{N}$ is the normal of the surface at the hit point, $\hat{L}$ is the direction to the light source from the hit point, C_m is the material colour, C_l is the light colour and C is the

resultant output colour. Note that all light sources are available in the `Scene` class, so you will likely have to iterate through these. You should sum the outputs of multiple light sources into the final pixel colour (if there is more than one).

Stage 2.2 - Shadow rays

Consider the fact that light rays may be blocked by objects in the scene. This should lead to visible shadows. Extend your implementation to check whether a hit point is in fact in a shadow. You can do this by firing another ray towards the light source from that point, and checking if it hits a (closer) surface along the way. If there is a hit, then that light source should not contribute to illumination at that point.

Be careful when firing a ray away from the surface of an object. Numerical error could lead to a 'premature' hit with that same object! One solution is to offset the origin of the ray slightly away from the surface.

Stage 2.3 - Reflective materials

This is where ray tracing really starts to shine – no pun intended! Extend the ray tracer to handle materials with the `Reflective` type. When a ray hits a reflective material, another ray should be *recursively* traced to determine the colour at that point. To do this you need to calculate a reflection vector as a function of the hit point's surface normal and the incident ray direction. This should be *pure* reflection – the colour of the material plays n
surfaces in a scene, co
place a hard limit on t
willing to 'fire').

Stage 2.4 - Refractive materials

Some materials are transparent, and allow light to . Glass is an example of such a material. Unfortunately, simulating this effect in a realistic manner is not as simple as allowing an incident ray to pass directly through the object. Indeed, you may have observed that light can 'bend' through transparent mediums (take a look at any curved glass object). This phenomenon is known as *refraction*. Extend the ray tracer once again to handle materials with the `Refractive` type. In a similar way to how you handled reflection, upon a ray hitting a surface, you should recursively trace a ray through the object according to physical laws of refraction. The colour of the material should not play any role in the calculations at this stage. Note that materials have an additional `RefractiveIndex` property, which will come in handy here.

Stage 2.5 - The Fresnel effect

In the real world, refraction does not really occur in total isolation from reflection. When light hits a refractive surface, some proportion of it is reflected, while another proportion

is refracted (these proportions sum to 1 since energy is conserved). This proportion is not uniform for all rays which hit the surface. As a ray's angle of incidence decreases, there is greater reflection versus refraction. If you look at a sheet of glass from front on, and you will see that most of the light is refracted (transmits through). However, if you look at it almost side-on, it looks a lot more reflective!

This phenomenon is known as the *Fresnel effect*. Your next task is to improve refractive materials so that reflection is *mixed* into the corresponding lighting calculations according to the Fresnel equations. Note that this means that *two* rays need to be traced for every one ray that coincides with a refractive material. If this process repeats itself multiple times, the computational burden increases exponentially, so keep this in mind when coding your solution.

Stage 2.6 - Anti-aliasing

You may have noticed that the images being produced so far contain somewhat jagged edges. This is because details in the scene can differ at the sub-pixel level when they are projected onto the final image. This is a common problem in computer graphics generally, and is called *aliasing*. Aliasing is usually quite visible where there are curved edges, or edges that are not aligned horizontally or vertically with the screen (think about why). We can use various techniques to mitigate this problem, and this process is called *anti-aliasing*.

Modify your ray tracer to incorporate optional anti-aliasing during rendering. You should do this by firing more rays per pixel and then averaging the outputs for the final colour. There is anot hich specifies the anti-al an example:


```
dotnet run -- -f tests/sample_scene.txt -o outpu
```


The argument `-x` specifies this multiplier, which in t should fire twice as many rays both horizontally and vertically (4x rays per pixel). If the multiplier is 3, then you should fire three times as many rays in both directions (9x rays per pixel). And so on. Note that we have already parsed this command-line argument for you! It is accessible within the `Scene` class as `options.AAMultiplier`, so you don't need to worry about how to read it into your program.

Stage 3 - Advanced add-ons (9 marks)

In this stage you are given the opportunity to implement some advanced add-on effects of your choosing. Some are more trivial to implement than others, and the allocated marks reflect their approximate difficulty and/or time commitment. In completing these questions to a high standard, we expect that you research various approaches, and make informed decisions to maximise the outcomes of the intended effects. You should write some detailed comments in your `README.md` file which describe the approach you have taken. It is not possible to receive more than the allocated marks for this section (**9**

marks maximum), so if you complete more add-ons than required, clearly state which ones you want to contribute to your mark!

N.B.: Regardless of whether you complete this stage or not, you are still encouraged to show off your work by submitting a custom scene (see below).

Render quality

For a few of these add-ons, it may be useful to have a render *quality* parameter, which allows the user to 'trade' between computation time and output image quality. We have provided an additional command line argument `-q` that you are free to utilise for this. For add-ons **where it's relevant**, our test suite may increment the quality level and re-render a test output (starting at the default quality level of 0). This will be repeated until some reasonable time limit is reached, or the image does not change. The exact details of this process will depend on available resources during marking, however all submissions will be tested in the same environment, and with the same time limits, in order to ensure fairness.

You do not *have to* utilise the `-q` parameter. It's there purely so you have the opportunity to show what your ray tracer is capable of without having to be ultra-conservative with respect to time. It only applies to *select* stage 3 add-ons, and won't be used elsewhere during testing. It is up to you to determine if it is relevant to the problem you are solving.

Stage 3, Option A - E

Up until this point w
imation, and not ho
bulb which is a long cylindrical shape. It would be inappropriate t
a singular point. Even a standard light globe which comes close t
a 'point', is still a physical object with a *surfa* ll such surfaces **emissive**.

For this add-on you need to extend the ray tracer to handle emissive materials. As a consequence there should also be 'soft shadows' present in the scene (have a think about why). Note that the `MaterialType` enum has an `Emissive` type, which will be used to specify whether the material should be emissive. The colour of the material can be interpreted to be the colour of the emitted rays. You may assume only spheres and triangles will be given emissive materials (not infinite planes).

Stage 3, Option B - Ambient lighting/occlusion (+6)

The current ray tracer only simulates a very small subset of rays in a scene. Consider how the illumination of a point on a diffuse object is currently calculated. We test if there is a *direct* ray originating from a light source. If there isn't, or there is an object in-between, then the point isn't illuminated at all. However, in reality, there may still be some illumination that comes from indirect rays of light (e.g. bouncing off

other surfaces in the scene). This is called *ambient lighting*. When ambient lighting is computed, we consequently compute *ambient occlusion* in the scene, since indirect light will not illuminate all surfaces equally. For example, surfaces inside crevices and cracks tend to be a lot darker since they are less 'exposed' to the other surfaces in the scene.

Extend the ray tracer so that it is possible to optionally compute ambient lighting in a rendered image. You should utilise the `-l` command line flag which is available as `options.AmbientLightingEnabled` in the `Scene` class. In other words, if and only if `-l` is passed to your program, ambient lighting should be enabled.

It is infeasible to compute *every* ray in a scene, so you may end up using some sort of randomised or 'Monte Carlo' method to assist with this.

Stage 3, Option C - OBJ models (+6)

Extend the ray tracer to read simple 3D models in the form of .obj files. Research how the .obj file format is structured. We only expect that you handle files with vertex, normal and face definitions (`v`, `vn`, `f`), and faces may be assumed to be triangles (exactly three vertices). Be sure to consider how more complex models could impact the *performance* of your ray tracer, and optimise relevant data structures accordingly. If you do implement optimisations, ensure these are discussed in your `README.md`.

We have provided you with a template class in the `src/extensions` folder called `ObjModel.cs`. In the same manner as the sphere, plane and triangle primitives defined earlier, `ObjModel` has an `Intersect()` method that needs to be implemented. You should read/parse t
intersections.

We have provid _obj.txt) that contains an OBJ, so you should examine this file first to see exactly ho
defined. In particular, observe there is a string that is the path to the .o
Also notice that we specify an `Offset` vector and en loading a model. These parameters allow us to adjust the size and positi
scene appropriately. The scale should be applied first, *then* the offset, as these are order dependent operations.

Stage 3, Option D - Glossy materials (+3)

At the moment we can model a few common materials, but there are still many material types which cannot be adequately simulated. Add support for 'glossy' materials to your ray tracer. The `MaterialType` enum has a `Glossy` type, which you should add functionality for. Unlike refractive materials, no light should transmit through the object. However, the object should not be completely reflective either. Instead, the object should be coloured based on the material's `Color` property, but also appear to reflect some light. Exactly how you achieve this effect is up to you, but you should aim to make it convincing. If there are parameters that need to be tuned to determine the amount of 'glossiness', choose suitable defaults that demonstrate the effect.

Stage 3, Option E - Custom camera orientation (+3)

Currently the camera is assumed to be at the origin (0, 0, 0) looking along the z-axis (with the y-axis oriented 'up'). Allow the user to apply a custom camera position and orientation to the camera based on the following parameters: a position vector (x, y, z), an axis of rotation vector (x, y, z) and an angle (in degrees). Note that this is an axis *of rotation*, **not** the 'direction' or local z-axis of the camera. The axis and angle are specified via the following command-line parameters where `#` denotes a floating point number:

- `--cam-pos #,#,#` – Available as `options.CameraPosition`
- `--cam-axis #,#,#` – Available as `options.CameraAxis`
- `--cam-angle #` – Available as `options.CameraAngle` (in degrees)

These parameters are automatically parsed for you by the template code, and so all you need to do is utilise the respective variables within the `Scene` class (e.g., as `options.CameraPosition`).

Stage 3, Option F - Beer's law (+3)

We currently consider transmission of light through a medium to be unaffected by that medium, other than its refractive index. *Beer's law* states that light intensity should drop as light passes th
traveled. Addition
giving rise to 'colou

Modify the ray tracer to obey Beer's law for `Refractive` materials. Currently the colour of refractive materials has no effect in lighting calculat
utilised. You should interpret this field to contain the
For example the RGB value of (4, 1, 1) means that red light will be
rate than blue and green light.

Stage 3, Option G - Depth of field (+3)

The camera is currently modelled assuming an infinitely small 'pinhole' aperture. If you think about it, a physical camera cannot function like this, since no light can pass through an infinitely small hole! We may introduce additional parameters to our ray tracer that allow us to more accurately model how a real camera works. This allows us to generate a 'depth of field' effect in our rendered images, which means some parts of the scene are in focus, and other parts are blurry depending on a given 'focal length'.

Extend your ray tracer to account for two additional parameters: a **focal length**, and an **aperture radius**. These will be passed via the following command-line parameters where `#` denotes a floating point number:

- `--aperture-radius #` – Available as `options.ApertureRadius`

- `--focal-length #` – Available as `options.FocalLength`

We have already parsed these parameters for you and you can access them in the `Scene` class (e.g. as `options.FocalLength`). The aperture radius parameter has a default value of 0, which reflects the idealised pinhole model we have used up until this point. However, if this value is larger than 0, then a circular pinhole should be simulated with the corresponding radius. Surfaces at a distance equal to the focal length (away from the camera) should be in focus.

Finally - Show off your work! (3 marks)

Congrats! At this point, your ray tracer should be able to generate some pretty impressive images. Now is your chance to show off what it can do. The `tests/final_scene.txt` file is currently empty. You should populate this scene in order to demonstrate all the core features of the ray tracer, as well as your chosen add-ons. Choose suitable settings to render your final image, and replace the `images/final_scene.png` placeholder image with it. Notice that this image is already embedded in the `README.md`.

Also record how long it takes to render the image on your PC and include that in the `README.md` (see the template provided). In case we need to verify your image, you must also include the exact command line used to generate it. We might use lower resolution/anti-aliasing/quality settings to speed up the verification process.

Verify that your image is visible on GitHub (instead of the placeholder ima

Assessment

Unless explicitly stated, grading of your submission will be e
produced by your ray tracer, **not** your code. We will s
hidden test scenes and/or specific command line arguments t
tests won't receive credit! Therefore, you must ensure that you do not change the inputs or outputs of the provided template. Furthermore, don't rely on just the provided test scenes – think about possible edge cases and *write your own*! We will run your ray-tracer with both the provided test scene, and some hidden test scenes.

Ray tracers are resource intensive by nature, so we don't expect test scenes to be rendered 'instantly'. However, you should take necessary steps to optimise the process where it is sensible. For practical reasons, time limits will apply, but they will be chosen based on our sample solution to very comfortably allow your project to generate correct output images if implemented reasonably. Occasionally the time taken to generate images may be part of your assessment (e.g., if specific optimisations are applicable to an add-on you have implemented). You are encouraged to discuss any time related dilemmas you faced in your `README.md`.

Finally, feel free to ask questions about the specification on the discussion forum (Ed). However, before doing so, please refer to our live FAQ which will be pinned for

the duration of the project. This is based on questions from previous years, and will be updated further as new questions roll in!

Submission

You must submit the project via the respective project details page on Canvas (where you found this specification). There is an embedded Gradescope interface on that page that allows you to link your GitHub account, as well as the private repository containing your work. You may re-submit your project as many times as you wish, and only the *final* submission will be marked, so you are **strongly encouraged** to attempt a test run of this process comfortably before the deadline.

Your submission must conform to the *exact* scene and command line specification that is provided with the template, since our testing system is semi-automated. If you do not modify the core template code, then this should not be an issue. Additionally, external libraries, dependencies, or packages are **not** allowed. Make it clear in your git `README.md` which add-ons you wish to be marked, and briefly describe your approach for these. Remember, you should frequently commit your work to GitHub while working on the project. Even though you are working solo, this serves as a backup and proof of your work on the project.

Late submissions will incur a penalty of 3 marks per day after the final due date. **Penalties also apply** when not using Git/GitHub correctly, not refe ng to the spec

Academic honesty

Your submission must be your own work. Sharing code with other st is **not** allowed for this assignment, although you may discuss the pr conceptual level. We will use a sophisticated code similarity checker between all student submissions, and previous years' ones. You are welcome to utilise external resources for research purposes, however, you must attribute any sources you use both in your code and your `README.md`. You should minimise *blindly* copying and paste code, even if referenced, and deductions may apply if there is little evidence of understanding due to an over-reliance on external resources. Feel free to clarify this with your tutor if you are unsure.