Project 1 - Ray Tracer

Due date: 16:00 12th September 2021 (30 marks total)

August 2021

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. We have provided you with a template C# implementation that you will need to complete. We are not using the Unity engine, 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 step in sequence since this will make the process less overwhelming.

There are various approaches to modelling how light interacts with surfaces in a scene. Almost always, the choice of approach comes down to a trade-off between computational complexity and realism. A ray tracing based approach can produce very realistic images, however this comes with a significant computational cost that generally makes it unsuitable for real-time rendering. Even if there are no real-time rendering requirements, we still have to approximate and optimise the ray tracing process, since simulating *all* rays in a scene is computationally intractable.

There are plenty of resources about ray tracing online. You are welcome to use these resources. If you do, we **strongly** encourage you to make sure you understand those resources, rather than just copying bits of their code. It is also a good idea to consult multiple sources to check whether there are significant differences in their approach. Note that the use of external libraries or code dependencies is **not** allowed – part of the fun of building a ray tracer is that it can be done from scratch!

Stage 1 - Basic ray tracer (12 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 fully 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 implemented inside the `Render()` method in `Scene.cs`. This method takes an `Image` object for which you can set the individual colour of each pixel, as well as derive properties such as its width and height. When the program is run, this image will automatically be outputted as a PNG image file.

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

```
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. You can see this by taking 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 (e.g., the default image width and height), instead pass values using the appropriate flags on the command line, if you want to change parameters.

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[1] 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 same direction, and the cross product of two vectors will give you the vector which is perpendicular to both of them (e.g., crossing the x- and y-axes would give you the z-axis).

Hint: It is ***strongly*** *recommended that you test your implementations thoroughly. Vectors are utilised everywhere in this project, so a mistake here 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.

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 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.

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:

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

- `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.cs` – Represented by a point (`center`) and a `radius`.

All of these classes implement the interface `SceneEntity`. This means they all contain a method called `Intersect()`, which takes a `Ray` as its input and returns a `RayHit` as its output. The returned `RayHit` structure contains important information used during ray tracing: the *incident* ray direction, the *position* of the hit, and the *normal* of the surface at that position. It is also possible that there is no hit at all, in which case, `null` should be returned instead of a `RayHit` instance. Your job is to implement 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. *Hint: 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 calculate how light interacts with the entity – for example, its colour, whether it is opaque, reflective, transparent, etc. Open `src/core/Material.cs` to see our representation of a material in this project. Note that you already used the `Color` 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.

Hint: 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 no role in the calculations. Note that if there are a lot of reflective surfaces in a scene, computational costs can blow out significantly, so you may wish to place a hard limit on the depth of recursion (i.e., how many new 'reflection' rays you are willing to 'fire').

Stage 2.4 - Refractive materials

Some materials are transparent, and allow light to *transmit* through them. 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 another command-line argument accessible within your program which specifies the anti-aliasing multiplier you should use when rendering the scene. Here is an example:

```
dotnet run -- -f tests/sample_scene.txt -o output.png -x 2
```

The argument `-x` specifies this multiplier, which in this example is 2. This means you 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 (6 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 relevant sections of your code which describe the approach you have taken. It is not possible to receive more than the allocated marks for this section, 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).

Stage 3, Option A - Emissive materials (+6)

Up until this point we considered lights to be infinitely small points. This is an approximation, and not how lighting typically works in the real world. Consider a fluorescent bulb which is a long cylindrical shape. It would be inappropriate to model this using a singular point. Even a standard light globe, which comes close to being modelled by a 'point', is still a physical object with a *surface area* that emits light. We call 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. *Hint: 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.

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 the .obj file in the constructor, and use this data when computing intersections.

We have provided you with a sample scene (`tests/sample_scene_obj.txt`) that contains an OBJ, so you should examine this file first to see exactly how OBJ models are defined. In particular, observe there is a string that is the path to the .obj file itself. Also notice that we specify an `offset` vector and `scale` as parameters when loading a model. These parameters allow us to adjust the size and position of the model to fit the 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 specify a custom camera position and orientation based on the following parameters: a position vector (x, y, z), an axis of rotation vector (x, y, z) and an angle (in degrees). These 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 through a medium, and that drop is proportional to the distance traveled. Additionally, some wavelengths of light may be absorbed faster than others, giving rise to 'coloured' semi-transparent objects.

Modify the ray tracer to obey Beer's law for `Refractive` materials. Currently the colour of refractive materials has no effect in lighting calculations, but now it should be utilised. You should interpret this field to contain the *absorbance* of each colour channel. For example the RGB value of (4, 1, 1) means that red light will be absorbed at a faster 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 aperture! We can 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`. Verify that it is visible on GitHub instead of the placeholder image.

Please 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 settings to speed up the verification process.

Submission

You should frequently commit your work to GitHub. Even though this is individual work, this serves as a backup and provides a record of your work on the project. The final commit on the **main** branch will be taken to be your final submission. Ensure that both your **student details** and what **stages you have attempted** is very clear in your `README.md` (use the provided template). We will batch download all submissions just after the deadline, so let us know if you are submitting the project late. Late submissions will incur a penalty of **3 marks** per day after the final due date.

We will run your ray-tracer with both the provided test scene, and some hidden test scenes. 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 have implemented (and wish to be marked), and briefly describe your approach for these.

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. Efficiency is most likely to be a consideration in stage 3 since certain add-on(s) may require a significant amount of extra computation. If you feel that a notably longer wait time is justified due to the type of effect you are producing, you should clearly state why in your `README.md` file, and a rough analysis of how much extra computation is required (versus without the add-on).

Academic honesty

Your submission needs to be your own work. Sharing code with other students is not allowed for this assignment, although you may discuss the project with classmates at a high level. You are welcome to utilise external resources for research purposes, and in fact, for some challenges in this assignment it is encouraged. However, you must attribute any sources you use both in your code and your `README.md`. We strongly recommend against just copying and pasting code, and deductions may apply if there is little evidence of understanding due to an over-reliance on external resources. Furthermore, given that you must conform to a template, direct copy-and-paste of online code may end up being more complicated than interpreting concepts and re-implementing the features yourself.