Computer Graphics (COMS30115)

Practical 4: Getting Started with Raytracing

The aim of this lab sheet is to introduce the topic of ray tracing.
You should continue to use all of the classes provided for you in the libs folder
(*ModelTriangle*, *CanvasPoint*, *Colour* etc.) as well as *vec3* and *mat3* from the glm library !

Task 1: Raytracing

Write a new draw function that draws filled triangles, using the raytracing approach outlined in the lectures.
(don't delete your old draw function - you will need that again later !)

The basic principle is to loop through each pixel on the image plane (top-to-bottom, left-to-right),
casting a ray from the camera, through the current pixel and into the scene.
Your task is to determine if this ray intersects with a triangle from the model.

TIP: a ray might intersect with more than one triangle - you are interested in the closest one.
(only the closest one will be visible to the camera - others will be hidden behind it !)

To help you with this task, we recommend you write a *getClosestIntersection* function that given:

- The position of the camera
- The direction vector of a ray being cast into the scene
- All of the triangles in the model

...It will return details of the intersected point on the *closest* triangle (if any !)

HINT: You will need to loop through all of the triangles, checking to see if the ray intersects it.

TIP: You are going to need to return various details about the intersection point found...
There is a class in the libs/sdw folder called *RayTriangleIntersection* that you might like to use for this !

GIFT: To aid you in implementing your *getClosestIntersection* function, here is a code version of the ray/triangle intersection equation that was shown in the lecture (it's not so difficult really !):

```
vec3 e0 = triangle.vertices[1] - triangle.vertices[0];
vec3 e1 = triangle.vertices[2] - triangle.vertices[0];
vec3 SPVector = cameraPosition-triangle.vertices[0];
mat3 DEMatrix(-rayDirection, e0, e1);
vec3 possibleSolution = glm::inverse(DEMatrix) * SPVector;
```

Remember that the "possible solution" produced is a three-element data structure that consists of:

- 't' - the distance along the ray from the camera to the intersection point
- 'u' - the proportion along the triangle's first edge that the intersection point appears
- 'v' - the proportion along the triangle's second edge that the intersection point appears

NOTE: Be sure to check the constraints on 'u' and 'v':

- 0.0 <= u <= 1.0
- 0.0 <= v <= 1.0
- u + v <= 1.0

This is important because the found solution might well be on the same plane as the triangle,
but then NOT within the bounds of the triangle itself !

TIP: Be sure to correctly transpose intersection coordinates into the model/world coordinate system
(and then onto the image plane !)

Task 2: Interactive Mode

Add some key event handling code to your program that allows the user to switch between
the three main modes of rendering:

- Wireframe: Stroked/Outline triangles
- Rasterised: Filled Triangles drawn using your rasterising code
- Raytraced: Filled Triangles drawn using your brand new raytracing code

This will be very useful later on as it will allow you to navigate the camera around the scene
using the FAST wireframe renderer and then switch to the (much slower) ray traced renderer
to view the fully lit scene from a particular angle (very useful for testing !)

Task 3: Proximity Lighting

Add a new vec3 variable to your code to store the location of a single-point light source
(the middle of the room, above the origin, somewhere need the ceiling seems sensible !)
Implement proximity lighting using the $1/4\pi r^2$ approach outlined in the lecture,
so that the closer a surface is to the light, the brighter a pixel will be drawn on the image plane.

HINT: You might like to get creative with the fraction (1/4 doesn't always lead to the best looking result)
TIP: You may wish to make use of the *brightness* attribute of the *CanvasPoint* object to help.
TIP: Your brightness values should be within the range 0.0 (total darkness) to 1.0 (fully illuminated)
Use this value as a multiplier to adjust the RGB channels of a pixel
(remember the "Linear Multiplication Reduction" slide from way back in the 1D lecture ?)

Task 4: Angle-of-Incidence Lighting

Now implement the Angle-of-Incidence lighting effect as outlined in the lecture.
You will need to calculate the normal of the triangle being intersected (using cross product).
Make use of the glm function called *cross* (this takes two vec3s and returns their cross product)
HINT: The "winding" used by the model will impact how your normal calculation should behave
(the best advice is to just try it ! if/when it doesn't work, just switch the vertex order !)

And then the angle-of-incidence of the light relative to the normal (using dot product)
Make use of the glm function called *dot* (this takes two vec3s and returns their dot product)
Use the result of this calculation to further adjust the brightness of each pixel.

TIP: When considering the angle that the light hits the surface,
you MUST use: *the vector direction of the LIGHT from the POINT ON THE SURFACE*
and NOT: *the vector direction of the POINT ON THE SURFACE from the LIGHT*
If you use the wrong one, you will end up lighting the back surface of the triangle !

Task 5: Ambient lighting

Finally implement a basic form of ambient lighting
The simplest way to achieve this is to set a minimum threshold (floor) for the brightness multiplier
An IF statement will do the job - when the brightness of a pixel falls below a certain value
(e.g. 0.2) just reset it to 0.2 !

.

Plagiarism

You are encouraged to discuss assignments and possible solutions with other teams.
HOWEVER it is **essential** that you only submit your own work.
This may feel like a grey area, however if you adhere to the following advice, you should be fine:

- Never exchange code with other teams (via IM/email, USB stick, GIT, printouts or photos !)
- It's OK to seek help from online sources (e.g. Stack Overflow) but don't just cut-and-paste chunks of code...
- If you don't understand what a line of code actually does, you shouldn't be submitting it !
- Don't submit anything you couldn't re-implement under exam conditions (with a good textbook !)

An automated checker will be used to flag any incidences of possible plagiarism.
If the markers feel that intentional plagiarism has actually taken place, marks may be deducted.
In serious or extensive cases, the incident may be reported to the faculty plagiarism panel.
This may result in a mark of zero for the assignment, or perhaps even the entire unit
(if it is a repeat offence).
Don't panic - if you stick to the above list of advice, you should remain safe !