Computer Graphics (COMS30115)

Practical 3: Getting Started with Rasterising

This is quite a complex practical topic, therefore this lab sheet will run for a couple of weeks.
(Some of the tasks near the end might prove tricky until after later lectures)

In this practical we will start to work in 3 dimensions.
To help you complete the tasks, we have provided you with the following data structure classes:

- ModelTriangle: A class to represent a triangular plane of a model within 3D space.
  This structure contains the positions of the three vertices (as vec3s) and the colour of the surface.
- CanvasTriangle: A class to represent a "flattened" 2D triangle for drawing onto the image plane (Canvas)
  This structure contains the positions of the three vertices (as CanvasPoints) and the colour of the surface.
- CanvasPoint: A data structure to represent a single 2D point on the image plane (Canvas).
  This structure contains the x and y coordinates of this point on the canvas.
  There is also a variable for recording the Z depth of the model element to which this point relates.
  Just ignore the *brightness* element for the moment - we will need that for a later exercise.
- Colour: A structure to represent the colour of a triangle or pixel.
  This structure contains an optional name for the colour and values for red, green and blue channels.

You will find these structures in the libs folder of the template project.
Take a look inside the files to find out more about them.
Each has a number of data elements, at least one constructor and a "print" function (to help with debugging !)
For this exercise you will also need to make use of the *vec3* and *mat3* from the glm library !

Task 1: Read in OBJ files

The first task in this practical is to implement a low-level file reader that can parse in OBJ files.
First write a function to read in the OBJ material file and use it to populate a "palette" of Colours.
You could just use a vector to store this palette of colours or...
if you want something a bit fancier, use a hashmap for more efficient colour lookup.

You may like to make use of the an *ifstream* and the *getline* function to read in the file data.
A *split* function has also been provided (in *Utils.h*) to make tokenising of lines easier.
But is really is up to you how you parse in the files.

Once you have read in the colours from the material file, you next job is to read in the triangle geometry.
Write a function that reads in a specified OBJ file and populates a vector of *ModelTriangles*
(see this tutorial on using vectors in c)

TIP: Faces in OBJ files are indexed from 1 (whereas vectors are indexed from 0)

TIP: You might also like to pass in a scaling factor (float) to adjust the size of the model
when you read it in (different models are to different scales !)

TIP: You should try to make your file loader as versatile and flexible as possible
(not all models will adherent stringently to the OBJ files specification)

Once a file has been loaded in, loop through all of the triangles and print them out,
(you should be able to send *ModelTriangle* to cout !) just to make sure they have all been successfully loaded.

.

Task 2: Wireframes

Now that you are able to read in OBJ files, the next step is to draw the triangles onto the image plane (canvas).
To start with, just project the wireframes onto image plane (unfilled triangles with white "stroke" lines).
You will need to decide on a position for the camera, but as general guidance, it should be...
Centred (in X & Y), but "stepped back" a bit in the Z direction.

NOTE: Your "stepped back" distance might to be negative, depending on which direction your Z axis points !

You will also need to experiment with the focal length (distance from camera to image plane).
Placing the image plane halfway between the camera and the centre of the world seems like a good place to start.

TIP: Make use of your line drawing code from the last practical session to help you in this task.

HINT: For each *ModelTriangle* you will need to generate a *CanvasTriangle* by
mapping the 3D vertices from the model onto the 2D vertices of the canvas (image plane).

Task 3: Filled Triangles

After you have successfully rendered a wireframe of the model, the next step is to fill those triangles.
Implement a triangle rasteriser that draws filled triangles instead of just wireframes on the image plane.
Make sure you use the correct colours that you read in from the materials file when rasterising the faces/facets.

TIP: Make use of your triangle drawing code from the previous practical session to help you in this task.

Task 4: Depth Buffer

Depending on the order that triangles appear in the OBJ file, rendered triangles may overlap in appropriately.
Use a 2D array of floats (the same resolution as the display window) to record the ***inverse*** of the Z depth
of the model element represented by each pixel drawn in the window (i.e. 1/Z).
Use this "depth buffer" in your drawing functions to decide what colour to set a particular window pixel
(if/when there are overlapping triangles in a model).

TIP: When you initialise the depth array use `std::numeric_limits<float>::infinity()`
This signifies that (at the start) that there is no model element in that particular position.

TIP: Don't forget to recalculate the values in this depth buffer if anything changes in the scene
(e.g. the camera moves or model artefacts get shifted)

NOTE: Check out this chapter from scratch-a-pixel for more details on why we need to use the inverse of Z depth.

TAUNT: "Less than" or "Greater than" which is it to be ? There is a lot going on - it's up to you to decide !

Task 5: Camera Movement

Extend your rendering engine to allow the position and orientation of the camera to be altered
Add some event handlers that will allow the user to manipulate the camera in the following ways:

- Translate the camera in 3 dimensions (up/down, left/right, forwards/backwards)
- Rotate the camera orientation in the Y axis (panning) and X axis (tilting)

NOTE: It is important that you only manipulate the camera position and orientation
(i.e. Do not alter the vertices of the model artefacts themselves)

HINT: To store the state of the camera use a 3x3 orientation matrix and a 3 element position variable.
OR, if you are feeling more ambitious, use a composite 4x4 homogeneous matrix instead.
(this however will require all variables in your code to be 4x4 matrices and 4 element vectors !)

.

Task 6: Orbit & LookAt

Add some additional code that will rotate (orbit) the position on the camera about the scene origin.
Implement a *LookAt* function so that the camera is always oriented to face the centre of the scene.

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 !