Computer Graphics (COMS30115)

Practical 2: Drawing in Two Dimensions

Task 1: Line Drawing

Starting with the *RedNoise* template, add a function to draw a simple line from one point to another.
A *CanvasPoint* class has been provided in the SDW package to help you.
You should also allow a colour to be passed in so that different coloured lines can be drawn.
A *Colour* class has been provided in the SDW package to help you.
There is no need to anti-alias the lines at this stage.

HINT: Use your knowledge of interpolation gained from the last practical sessions to help.
REMINDER Remember that the template project Makefile comes with a number of build rules:

- diagnostic: A development build rule that includes extra checking and diagnosis flags
- debug: A development build rule that will compile and link your project for use with a debugger (gdb)
- speedy: A build rule that will result in a high performance executable (to make interaction testing quicker)
- production: A build run to make an executable for release or distribution

Just typing *make* on it's own will build the *diagnostic* rule and runs the resultant executable.

NOTE: The *diagnostic* rule requires the "Address Sanitizer" library to be installed
(so probably won't work in the labs !)

Task 2: Stroked Triangles

Add a new function to your program to draw "stroked" (unfilled) triangles.
This function should draw the three outer edge lines of a triangle passed in as parameters.
You should also allow a colour to be passed in so that different coloured triangles can be drawn.
A *CanvasTriangle* class has been provided in the SDW package to help you.

HINT: Details of how to draw lines were covered in the lectures.

In order to make your application interactive, add to the event handling function so that
when the 'u' key is pressed, the triangle drawing function is called with 3 randomly generated vertices.
You might also like to randomly generate a colour for the triangle.
Test this out by running your code and pressing the `u` key (lots of times !!!)

HINT: Use the standard C *rand* function to help (e.g. `rand()%255` gives a random number between 0-255).

Task 3: Filled Triangles

Write a new function to draw *filled* triangles, again based a *CanvasTriangle* and a *Colour* .
After each filled triangle you render, draw an equivalent stroked triangle over the top of it
(this is to make sure that your rasterising is correct - both filled and stroked triangles should line up !)

HINT: Details of how to draw filled triangles were covered in the lectures.
HINT: You may like to use the standard C *swap* function if you need to do any sorting ;o)
TIP: Each data structure class has a *print* method that you might find useful for debugging !

Again, make your application interactive, so that pressing the `f` key
will cause a random filled triangle to be drawn (with a random colour !)

Task 4: Image Loading

In this task, you must load in a PPM image and show it in the SDL window.
You should parse in the PPM image file in your own code (rather than using an image loading library).
Once the pixel data has been loaded, show the image in the display window.
Your code should set each pixel of the window individually (rather than using any SDL image drawing functions).

HINT: It's up to you how you implement this, but perhaps the easiest way was to use the C++ *ifstream* class.
This provides the following functions/methods:

- *getline*: reads in whole lines as strings (for the PPM header lines)
- *get*: allows you to read in single bytes (for the RGB colour channel bytes)

HINT: For the width/height line, you can use the string *find* and *substr* methods to split the line in two.
HINT: To convert strings to integers, there is the stoi function.

Task 5: Texture Mapping

In this final task, we are going to combine all of the above task together into a single activity.
First draw a stroked triangle with the following vertices:

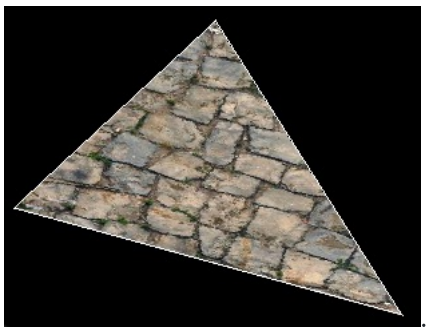
```
(160,10)
(300,230)
(10,150)
```

Now fill this triangle, not with pixels of the same colour, but with pixels from the supplied texture image.
The area of texture you should use is defined by the following three points (from the *texture image* space):

```
(195,5)
(395,380)
(65,330)
```

Each triangle vertex must be explicitly "tied" to one of these texture points.
The *CanvasPoint* class has a *TexturePoint* variable that can be used to maintain this relationship.
Any new triangle vertices you create (e.g. LEFT/RIGHT) must also be linked to a texture point
(by interpolating a suitable position within the texture image).

If all goes to plan, you should end up with a textured triangle that looks like the one shown below:



NOTE: Although the canvas triangles will be flat bottomed, the equivalent texture triangles probably won't be.
HINT: This task is going to require a LOT of interpolation ! (check lecture slides for more help)
TIP: Be careful when calculating the texture pixel array index if some of your variables are floats
(there is the danger of rounding up leading to skewed texturing !)