Computer Graphics (COMS30115)

Practical 1: Interpolation

Overview

The aim of this practical exercise is to get to grips with the tools & libraries required for the assessed exercise.
We have tried to use an environment that is fairly agnostic to the underlying architecture
(so it should be fairly straight-forward to get running on most machines !)
We however will test your assignment on the lab machines so you should not use any platform-dependent features.
If you use an IDE (Visual Studio, Eclipse, XCode etc) make sure that your project compiles without it.

The project uses two external libraries: SDL and GLM
The aim of this unit is to learn to build things from the ground-up, so don't use anything other than SDL and GLM.

The SDL library allows us to draw pixels on the screen.
We have provided a "wrapper" object around SDL called *DisplayWindow* .
This wrapper initialises SDL and provides useful functions for drawing pixels onto the screen:

- DrawingWindow: A constructor for the drawing window that takes 3 parameters:
  - Horizontal resolution (integer width)
  - Vertical resolution (integer height)
  - Whether to show the window full-screen (boolean)
- pollForInputEvents: Checks to see if there are any events waiting to be processed (returns true if there are !)
- setPixel: sets an individual pixel (indicated by x and y coordinates) to a particular colour (a packed ARGB value)
- getPixel: gets the colour of the pixel indicated by the x and y coordinates
- clearPixels: clears all pixel colours (for whole window)
- renderFrame: draws all pixels to the screen (until you call this, all pixel changes are just in a memory buffer)

There is a template/skeleton project (*RedNoise*) that illustrates the use of the above methods.
Use this template as a starting point for each practical exercise.
Note: The template comes bundled with the GLM libraries,
however you may have to install the SDL Development Libraries yourself
(OSX users should probably try installing with homebrew/macports first - if you have it !)

Task 1: Red Noise

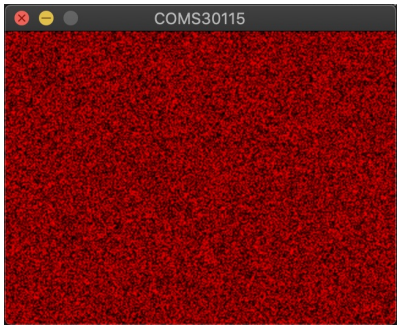
This first task is mainly to make sure you have all of the libraries installed and can compile and run SDL code.
Download the *RedNoise* template, unzip it and build it.
The project comes complete with a Makefile containing a number of build rules:

- 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 run the resultant executable.

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

When you run the *RedNoise* executable you should see a window that looks like the following...



Task 2: Single Element Numerical Interpolation

Interpolation (informally: "filling in the gaps between known numbers") is an essential operation in computer graphics.
Write a function called *interpolate* that takes 3 parameters:

- from: a floating point number to start from
- to: a floating point number to go up to
- numberOfValues: the number of steps required in the result returned by the function

This function should return an **evenly spaced** list (a vector in fact) of floats
that starts at *from*, ends at *to* and contains *numberOfValues* elements.

So, for example, calling: `interpolate( 2.2, 8.5, 7 )`
Should return a vector containing the following values: `2.2 3.25 4.3 5.35 6.4 7.45 8.5`

Task 3: Single Dimension Greyscale Interpolation

Create a greyscale gradient across the *DisplayWindow*, from left to right (as shown in the diagram below)

This is a basic form of linear interpolation - something essential for later in this unit.

You can use the *interpolate* function that you wrote previously to do the hard work for you.
You will need to construct colours by calculating RGB values and packing them into an int.
Don't worry about transparency/alpha - just set it to 255 (fully opaque).

Task 4: Three Element Numerical Interpolation

In this task, we will extend the floating point *interpolate* function to make it more powerful.
Write a new function that operates on "vec3" *from* and *to* parameters, rather than just floats.
A vec3 is a versatile 3-element vector provided by GLM. If we have a function that can interpolate
these then we can deal with colours, 3D coordinates (in fact anything with 3 elements !)

If we have the following two vec3 variables:

```
vec3 from( 1, 4, 9.2 );
vec3 to( 4, 1, 9.8 );
```

And pass them into our new interpolation function (with a size of 4), then we should get the following output:

```
( 1, 4, 9.2 )
( 2, 3, 9.4 )
( 3, 2, 9.6 )
( 4, 1, 9.8 )
```

Task 5: Two Dimension Colour Interpolation

In task 3 we were only attempting interpolation in one dimension (the x axis).
Now attempt interpolation in two dimensions (x and y)

Set the four corners of your window to 4 primary colours (red, blue, green, yellow).
Then smoothly interpolate between them in order to produce the effect shown in the image below:

HINT: Work on a row at a time - work out the colour of the left-most pixel and the right-most pixel then interpolate long the row between these two extremes.
Use the vec3 *interpolate* function you wrote previously to do the hard work for you !