# CS4815 Week11 Lab Exercise

**Lab Objective:** We will consider the method of interpolation today, the idea that is used throughout lighting and colouring models. There is a very useful and motivational YouTube video that describes much of the task here.

Here's a **quick summary** of the tasks:

❶ Copy the main source file for this week's lab

❷ Compile the `.c` program and run it

❸ Now write a program called `fill.c` to simulate what OpenGL does internally

❹ Submit your completed program using the handin command

```
~cs4815/progs/handin -m cs4815 -p w11
```

**In Detail**

❶ Copy the program `cube.c` from this week's lab directory.

❷ Compile the program `cube.c`. Look back at previous labs to determine how to compile this `C` file. Look at the source code in order to figure out how to use the user interface; it's very primitive and easily figured out. Run the program to see how the colours are interpolated.

Look carefully at the code to see how the function `polygon()` is used to define faces of the cube. Note how the POLYGON primitive definition can accommodate specifying the *colour* and *normal* vector of each of the (four) vertices of the cube faces. We have seen in the lighting lectures how the normal at a vertex is taken to be the average of the normals on all of the faces incident to that vertex.[1] By virtue of the way the cube is set up (centre located at $(0, 0, 0)$) the normal vector at each vertex has the same value as the vertex itself. Draw the analogous picture in 2-D (a square) to convince yourself of why this is.

Internally, the colours assigned to each vertex are then *interpolated* to get the colour of every pixel.

❸ Now, simulate this process in 2-D by writing a program that does the same thing. You

[1] To more accurately account for some faces being larger than others we might instead use some weighted version of the average.

should proceed by creating an OpenGL program `fill.c` and defining a 4-vertex polygon with vertices $(0, M), (-M, 0), (0, -2M)$ and $(2M, 0)$, for some appropriate $M$.[2] Assign them the colours red, green, blue and black, respectively.

Now use linear interpolation on the four lines to determine the colours of the pixels of the lines. From this you can fill the polygon by doing the same thing across each horizontal, *scanline*. If you use OpenGL in Orthogonal mode one OpenGL unit is one pixel, according to the documentation, so you should, in effect, be able to determine scan lines. You should play with drawing some closely spaced horizontal lines in order to test this and to make sure that the idea works. Figure 1 illustrates what the output should look like; colors don't have to be those shown, though.

**Important note:** In OpenGL it is possible to specify a polygon, give the colors of each of the corners and have *it* do the interpolation. That is a huge shortcut and is **not** what I want.

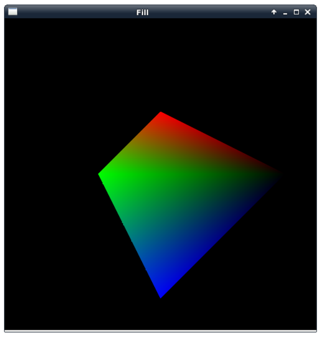

Figure 1: Typical output of `fill`.

Note that for each line you will need to store the colours of each $y$-value which are the input requirements for interpolating along scan-lines.

**An alternative method** Scan lines are very important in graphics for determining depths of faces (which face is closer to the camera) when performing hidden surface removal. However, it is not the only way to do the filling that I am asking you to do. In the spirit of the way we derived the Bézier curve algorithm you could, instead, take three consecutive points on the perimeter of the shape and, using the centre point as the starting point walk in steps of, say, $\alpha = 0.01$ towards the two other vertices. For each value of $\alpha$ we ask what is the corresponding point on each of the two "walks." Using linear interpolation we can determine the colors at each of those points. And with that information we can linearly interpolate across the line that joins those two points.

If you don't get that just look at the Bézier Curves wikipedia page that we browsed in the lectures, paying particular attention to the *quadratic* (three points) case.

**Something to consider** The polygon I have asked you to work with is very simple. Imagine if the polygon was non-convex – there is a horizontal line segment such that the

[2] You should set the variable `M` to some value at the top of your program.

two end points are *in* the polygon yet part of the line is *outside* the polygon. We won't go there but if you can make your algorithm work for the polygon where the black point above, $(2M, 0)$, is moved to $(2M, N), N > 0$ then there's an extra 20% of the assignment's marks in it for you.

❹ Using the handin command given at the top of the lab sheet please submit your lab exercise by the usual deadline. As usual a sliding scale of penalties will apply for a further three days). The submission process will be opened later this week.