

Course Outline

I began this lecture with an overview of the course and in particular a discussion of the Course Outline. A link to the official Course Outline can be found on the course web page. I will not discuss the Outline in these lecture notes.

RGB

This course will look at methods for defining images on a rectangular pixel¹ grid. The coordinates of the grid are integers. For example, they might be $(x, y) \in \{1, 2, \dots, 640\} \times \{1, 2, \dots, 480\}$. The color of a pixel is defined by three numbers called RGB. These are the intensities of the red, green, and blue components on a pixel. Typical images have 8 bits for each RGB value and hence 24 bits of intensity information per pixel i.e. values from 0 to 255.

We will discuss color much later in the course, but for now you should at least know that 0 means darkest and 255 means brightest. Here are some examples of colors:

- (0,0,0) black
- (255,255,255) white
- (255,0,0) red
- (0, 255, 0) green
- (0, 0, 255) blue
- (255,255,0) yellow
- (255,0,255) magenta (purple-ish)
- (0,255,255) cyan

Notice that RGB space defines a cube: $\{0, \dots, 255\} \times \{0, \dots, 255\} \times \{0, \dots, 255\}$. The above list of colors define the corners of the cube. Much later in the course, we will return to this color cube.

Drawing a line

One basic problem in computer graphics is how to represent a continuous two dimensional geometric object (such as a sloped line segment, or curve) on a rectangular pixel grid. ² There are many ways to represent a continuous geometric object on a rectangular grid. Today we will look at one straightforward method. We consider the case of a line only.

Consider a line segment joining two pixels (x_0, y_0) and (x_1, y_1) where x is a column number and y is a row number. Assume integer valued coordinates.

¹“Pixel” stands for “picture element”.

²This problem is known as *scan conversion*. The term “scan” comes from the mechanism by which CRT televisions and old computer screens display an image, namely they scan/display the image from left to right, one row at a time. Regrettably, this traditional terminology doesn’t have much to do with the algorithm described here.

The slope of the line is $m = \frac{y_1 - y_0}{x_1 - x_0}$. The line may be represented as $y = y_0 + m(x - x_0)$. Note that the slope m will typically not be an integer.

If the absolute value of the slope is less than 1, then we draw the line by sampling one pixel per column (one pixel per x value). Although the equation of the line (and in particular, the slope) doesn't change if we swap the two points, let's just assume we have labelled the two points such that $x_0 \leq x_1$. We draw the line segment as follows.

```
m = (y1 - y0)/(x1 - x0);
y = y0;
for x = x0 to x1 {
    writepixel(x, Round(y)). // Color to be determined elsewhere...
    y = m * (x - x0) + y0;
}
```

This is a relatively expensive way to draw a line. Notice that each y assignment within the loop requires several operations (two additions and a multiplication). We can speed it up by noticing that the y value changes by m each time.

```
m = (y1 - y0)/(x1 - x0);
y = y0;
for x = x0 to x1 {
    writepixel(x, Round(y)). // Color to be determined elsewhere...
    y = y + m;
}
```

This is much faster. However, you will notice that this method still requires a floating point operation for each y assignment since m is a float. In fact, there is a more clever way to draw the line which do not require floating point operations, which takes advantage of the fact that the x and y coordinates are integers. This algorithm was introduced by Bresenham in the mid 1960s. If you are interested³, I have provided details at <http://www.cim.mcgill.ca/~langer/557/Bresenham.pdf>

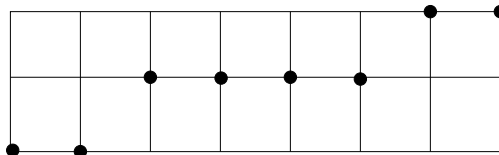
The above method assumes that we write once for each x value. This makes sense if $|m| < 1$. However, if $|m| > 1$ then this can lead to gaps where no pixel is written in some rows. For example, if $m = 4$ then we would only write a pixel every fourth row! We can avoid these gaps by changing the algorithm so that we loop over the y variable. We assume (or relabel the points so) that $y_0 \leq y_1$. The line can be written as $x = (y - y_0)/m + x_0$. It is easy to see that the line may be drawn as follows:

```
x = x0;
y = y0;
for y = y0 to y1 {
    writepixel(Round(x), y). // Color to be determined elsewhere...
    x := x + 1/m;
}
```

³You are not responsible for knowing these details. However, if you wish to work in the area of computer graphics, then you would be wise to educate yourself about this trick.

Anti-aliasing

Because the pixel coordinates are integers, we had to approximate the line by rounding off points on the line to the nearest pixel. The effect is that lines are piecewise horizontal in the case that $|m| < 1$, or piecewise vertical in the case that $|m| > 1$. The jumps that occur from row to row (or column to column) lead to a jagged appearance, sometimes called the “jaggies”. The jaggies are a specific type of “aliasing”, which we will describe later in the course.



One idea to get rid of the jaggies (called “anti-aliasing”) is to take advantage of the fact that image intensities are not binary typically, but rather have levels (i.e. 0 to 255). For example, suppose that a line is supposed to be black (0) on a white (255) background. One trick to get rid of the jaggies is to make a pixel’s intensity a shade of grey, if that pixel doesn’t lie exactly on the line. The grey value of the pixel could depend on the closeness of the pixel to the continuous line. There is no unique way to do this. Let’s look at a few ways, so you get the basic idea.

First, we need to consider how far is a pixel from a line. We could choose the intensity of a pixel based on the distance in the x or y direction only, e.g. $|y - \text{round}(y)|$ if $|m| < 1$ or $|x - \text{round}(x)|$ if $|m| > 1$.

A slightly more complicated way is to consider the distance from a pixel coordinate (x, y) to the line, measuring *in the direction perpendicular to the line* – that is, the shortest distance to the line. This distance is the dot product of the vector $(x - x_0, y - y_0)$ with the unit normal \mathbf{N} that is perpendicular to the line.

What is this unit normal? Note that for the unit normal to be perpendicular to the line, it must satisfy

$$\mathbf{N} \cdot (x_1 - x_0, y_1 - y_0) = 0.$$

By inspection, the unit normal must be parallel to $(-(y_1 - y_0), x_1 - x_0)$. The *unit* normal is thus:

$$\mathbf{N} = \frac{-(y_1 - y_0, x_1 - x_0)}{\sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}}$$

Note the denominator of \mathbf{N} requires taking a square root. This is expensive, but it’s only once per line segment.

By inspection, the perpendicular distance from any pixel (x, y) to the line can be computed by projecting any vector that joins a point on the line to (x, y) onto the unit normal, for example,

$$(x - x_0, y - y_0) \cdot \mathbf{N}.$$

There are many ways to choose the grey values so that they depend on the distance from a pixel to the line. The general idea is that points that fall exactly on the line are black, those that are very close to the the line are dark grey, and those that are far from the line (close to a pixel width) might be white (or more generally, the color of the background). We will meet similar concepts later in the course.