Lab 02: Smoothing, Coordinates, Modes & Colour

In this lab you will learn:

- How screen resolutions are specified.
- What the Processing `smooth()` function does.
- How to specify points on the Processing screen.
- How to change Processing's drawing modes.
- How to specify RGB colours.
- How to use greyscale.
- How to set the background, fill, and stroke colours in Processing.
- How to use variables to store frequently-used colours.
- How to use the `map()` function.

Introduction

In animation and graphics programming, it's important to know how to specify the location of objects on the screen. Whilst this was introduced in the first lab, we will go over it again today in more detail.

Screen Resolution

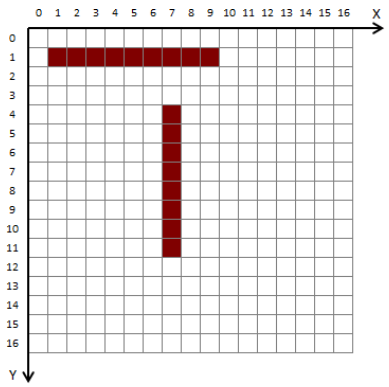
Conceptually, most computer screens are a rectangular grid of pixels. A pixel is the smallest region of colour on the screen that we can change. When we talk about a point on the screen, we usually mean a pixel.

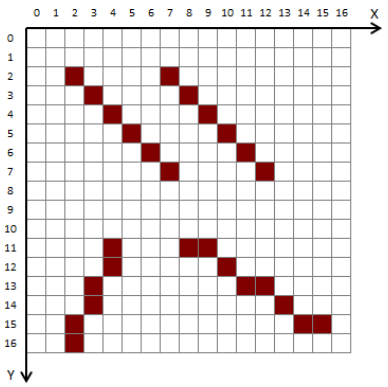
The width and height of the screen in pixels is known as its resolution. For example, a desktop computer with a big monitor might have a resolution of 1920×1200 pixels, i.e. the screen has 1920 columns of pixels, and 1200 rows of pixels (for a total of 1920×1200 = 2,304,000). In contrast, an iPhone 4 has a resolution of 640×960 pixels, while earlier iPhones have a lower resolution of 320×480.

Getting a program to work on screens with different resolutions is surprisingly tricky, and many professional graphic designers simply ignore the issue and work with a fixed resolution. In this course we will do the same, and assume that our programs will always be displayed on a monitor with at least 500×500 pixels.

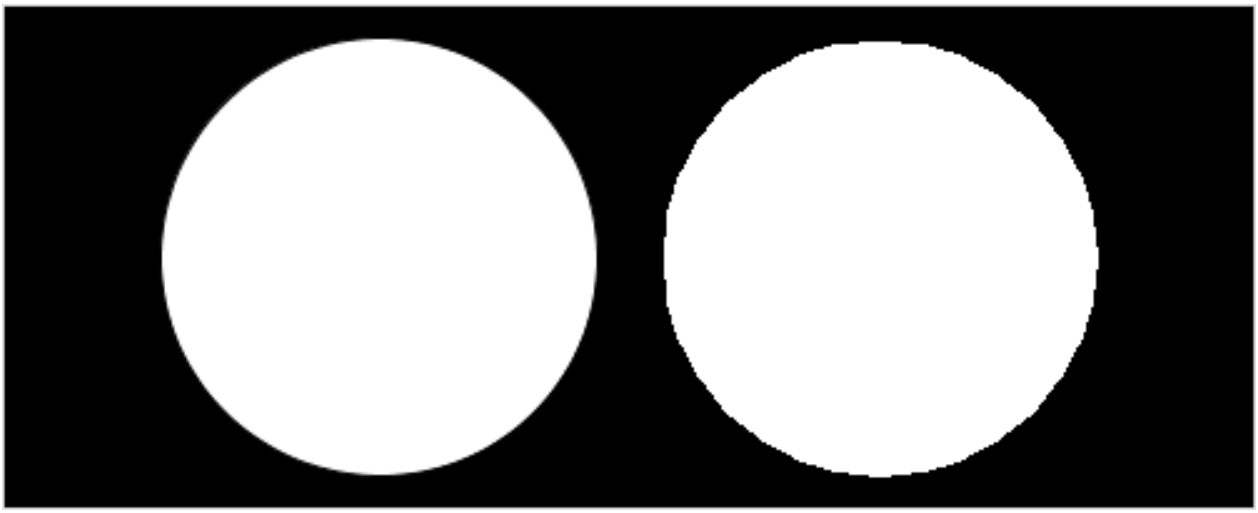
The `smooth()` Function

As you've no doubt seen, computer images can have jagged-looking lines for edges that ought to be smooth. For example, drawing a perfectly horizontal or vertical line is easy, as shown in the left figure. But slanted lines necessarily have some jagged edges, as shown in the right figure:





Processing has a function called `smooth()` that can help hide jagged edges by subtly changing the colours of the pixels around the jags (a technique known as anti-aliasing). For instance, the circle on the left is smoothed, while the circle on the right is not:

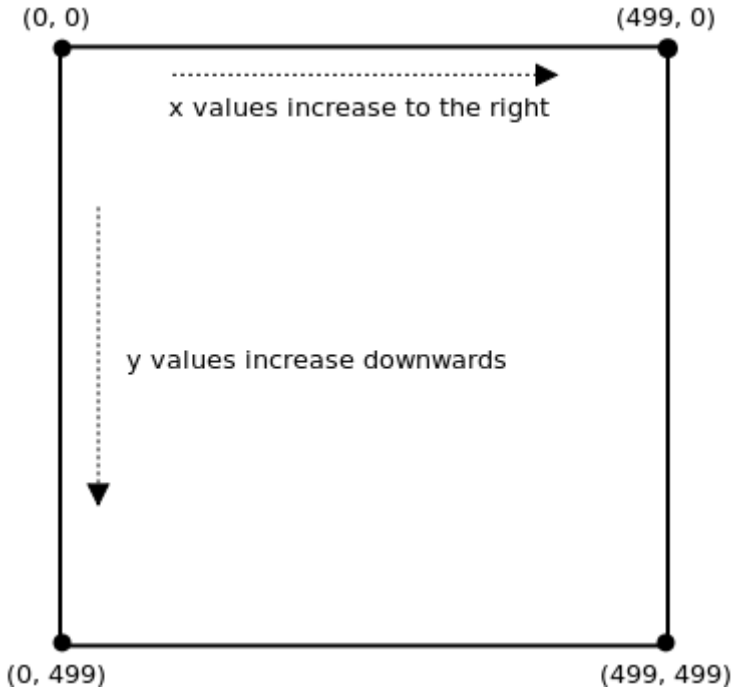


The main downside of using `smooth()` is that it slows your program down: smoothing-out the pixels takes a little bit of extra time. For many small programs, though, this speed decrease is not noticeable. Thus we'll often call `smooth()` in `setup()` as a simple way to make our output look a little nicer.

(x, y) Points on the Processing Screen

As in mathematics, Processing uses (x, y) coordinates to label every pixel on the screen: x is the column and y is the row.

Suppose the screen is 500 pixels wide and 500 pixels high. Then the pixel at the upper-left corner is (0, 0), and the pixel at the lower-right corner is (499, 499).



You might wonder why the bottom-right corner has coordinates (499, 499) instead of (500, 500). The reason is that we start counting pixels at 0. For example, if the screen were 5 x 5 pixels, then the pixels would have these coordinates:

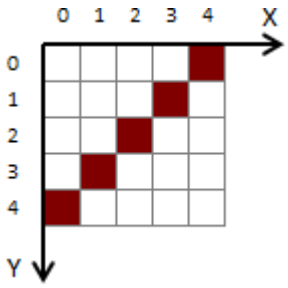
(0, 0)	(1, 0)	(2, 0)	(3, 0)	(4, 0)
(0, 1)	(1, 1)	(2, 1)	(3, 1)	(4, 1)
(0, 2)	(1, 2)	(2, 2)	(3, 2)	(4, 2)
(0, 3)	(1, 3)	(2, 3)	(3, 3)	(4, 3)
(0, 4)	(1, 4)	(2, 4)	(3, 4)	(4, 4)

You can see that the lower-right corner is (4, 4) and not (5, 5). Again, that's because we start counting from 0.

As the diagram above shows, the x-values increase from left to right, and the y-values increase from top to bottom. Unfortunately, that's not quite how coordinates work in mathematics. In math class, the y-values increase from bottom to top. This can be confusing if you are using a mathematical equation to, say, draw a shape in processing.

Example: Drawing a Diagonal Line

Suppose you want to draw a diagonal line from the lower-left corner of the screen to the upper-right corner, e.g. (For simplicity, assume the screen is 5 x 5 pixels):



Recall from math class that the equation for this line is y = x. But that doesn't quite work here for two reasons:

- Y-values in Processing coordinates increase from top to bottom, not bottom to top;
- The equation y = x makes the line go through the origin, i.e. the point (0, 0). But here our line does not go anywhere near (0, 0) (which is the top-left corner of the screen).

Let's look at the coordinates of the plotted points to see what is going on:

(4, 0)

(3, 1)

(2, 2)

(1, 3)

(0, 4)

How are the x and y values related in these points? Notice that they sum to 4, i.e. x + y = 4 for each point. Re-arranging that a bit gives us the equation y = -x + 4.

And that's the line we want: y = -x + 4. The -x takes care of the flipped y-axis, and the 4 moves the line away from the origin.

We always need to make such adjustments when we convert mathematical equations to Processing.

Example: Mirror Drawing

Recall this program, which draws an ellipse wherever the mouse pointer goes:

```
void setup() {
  size(500, 500);
}

void draw() {
  ellipse(mouseX, mouseY, 10, 10);
}
```

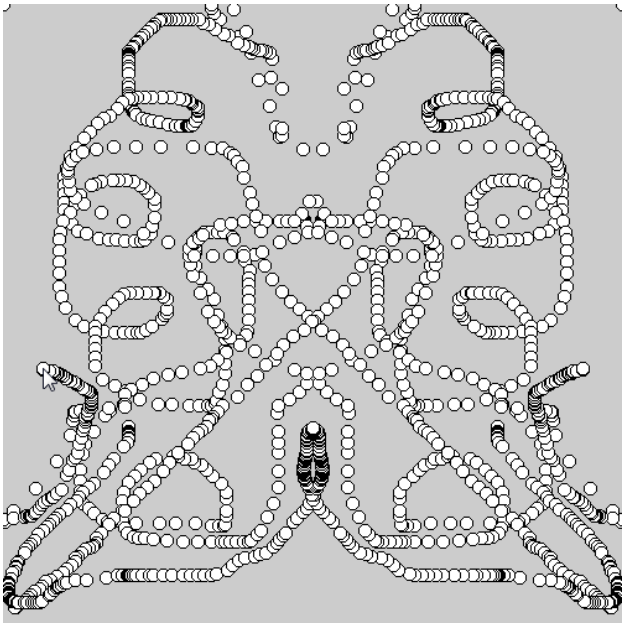
With a bit of arithmetic, we can create a mirror effect:

```
void setup() {
  size(500, 500);
}

void draw() {
  ellipse(mouseX, mouseY, 10, 10);

  // reflected point
  ellipse(500 - mouseX, mouseY, 10, 10);
}
```

In this program, every time `draw()` is called, two circles are drawn: one at the mouse position, and one at the mouse position reflected through the vertical centre line of the screen. Note the coordinates of the second circle: it's centred at (`500 - mouseX`, `mouseY`).



Plotting Points

We've seen that Processing has functions for plotting shapes (such as rectangles and ellipses), but sometimes we may want to plot individual points. There are a couple of ways of do this.

One way to plot a point in Processing is to use the `point(x, y)` function, e.g.:

```
void setup() {
  size(5, 5);
}

void draw() {
  point(4, 0);
  point(3, 1);
  point(2, 2);
  point(1, 3);
  point(0, 4);
}
```

Of course, a 5 x 5 window is pretty small, so you might have to look hard to see the line.

Another common way to plot points is to draw small shapes, such as circles or rectangles. This way you have control over the size, colour, and shape of the point, which is useful when, say, plotting data on a graph. Let's plot a point in the middle of a 500 x 500 window:

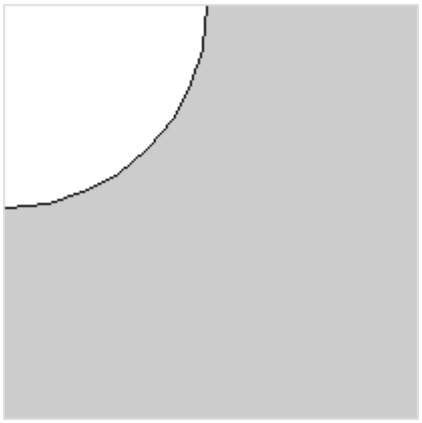
```
void setup() {
  size(500, 500);
}

void draw() {
  ellipse(500 / 2, 500 / 2, 10, 10);
}
```

The call to ellipse in `draw()` plots a circle of size 10 in the middle of the screen, i.e. at coordinates (250, 250). The / is division, and we use it here to show that Processing can do arithmetic for you.

Drawing Modes

When you call `ellipse(0, 0, 200, 200)`, it draws a circle centred at location (0, 0):



Since (0, 0) is the upper-left corner of the screen, only a quarter of the circle is visible: three-quarters of it are off-screen.

If you want to, you can change how Processing positions a circle using the `ellipseMode` function. For example, this code sets the upper-left corner of the ellipse at (0, 0):

```
ellipseMode(CORNER);
ellipse(0, 0, 200, 200);
```

Try it and see the difference it makes!

The "corner" of the ellipse means the corner of the smallest rectangle that can be drawn around the ellipse (and whose edges are parallel to the sides of the screen). This surrounding rectangle is known as the **bounding box** for the ellipse, and it's a common way to specify the size and position of a graphical object.

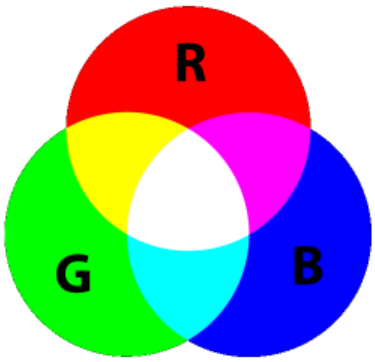
Rectangles drawn with `rect(x, y, width, height)` also have a mode that you can set with `rectMode`. By default, the (x, y) in a call to `rect` is the location of the rectangle's upper-left corner. You can change it to be the centre like this:

```
rectMode(CENTER);
rect(0, 0, 50, 50);
```

What drawing mode to use is up to you. You never need to change it, but sometimes it might be convenient. We will occasionally change the drawing mode when it is useful.

RGB Colour

RGB is short for “red, green, blue”: all RGB colours are combinations of different amounts of red, green, and blue.



For example, the colour orange is written as (255, 165, 0) in RGB. We call (255, 165, 0) an RGB triplet because it consists of three numbers. RGB triplets always have the form (red, green, blue), where the numbers red, green, and blue represent the intensity of that colour. So orange, which is (255, 165, 0), is a combination of red at intensity 255, green at intensity 165, and blue at intensity 0 (i.e. there is no blue in orange).

For each red/green/blue value, the minimum intensity is 0, and the maximum intensity is 255. Thus each of red, green, and blue have 256 intensity levels, which together can name exactly 16,777,216 different colours, i.e. over 16 million unique colours. On a good colour monitor this is enough to display photo-realistic images.

Why 16,777,216 different colours? In the RGB triple (r, g, b), each of r, g, and b can take on one 256 different values. Thus, there are 256 \times 256 \times 256 = 256^3 = 16,777,216 different RGB triples.

It’s useful to know a few basic RGB colours. Black is (0, 0, 0) and can be thought of as the absence of any colour. In contrast, white is (255, 255, 255).

Pure red is (255, 0, 0), and (128, 0, 0) is a darker shade of red; it’s darker because (128, 0, 0) is closer to black. Similarly, pure green is (0, 255, 0) and pure blue is (0, 0, 255).

Greyscale

A number of Processing functions support greyscale colour. Greyscale is important when you need to display an image somewhere that doesn’t support full RGB. For instance, many newspapers are greyscale, and so must convert colour photographs to greyscale. In RGB, any triplet of the form (c, c, c) specifies a shade of grey, and is considered to be a greyscale colour. Since there are 256 different possible values for c, these greyscale colours have 256 different shades:

Greyscale need not use the colour grey. Sometimes different shades of a colour can be used in greyscale. For instance, here are three different greyscales, using 256 shades of red, green, and blue respectively:

Setting the Background Colour

To set the background colour of the drawing window, uses the `background` function, e.g.:

```
background(255, 165, 0);  // orange
```

To set the background to be a shade of grey you could do this:

```
background(128, 128, 128);
```

Or, equivalently, this:

```
background(128);
```

Setting the Fill Colour

Shapes, such as ellipses and rectangles, can have their interiors filled with any colour using the `fill` function. For example:

```
fill(255, 165, 0);  // orange fill
ellipse(100, 100, 60, 40);

fill(100, 100, 100); // grey fill
rect(100, 100, 60, 40);
```

The fill colour is always the same as the last call to `fill`.

For greyscale colours you can write `fill(c)`, or `fill(c, c, c)` if you like typing.

You can also specify that a shape has no fill colour at all, i.e. a completely transparent interior that will let whatever is underneath show through. To do this, use the `noFill()` function, e.g.:

```
noFill();
ellipse(100, 100, 60, 40);
```

Setting the Stroke Colour (and size)

Shapes also have an edge with a colour that can be different from its fill colour. The `stroke` function set the colour of the edge, e.g.:

```
fill(255, 165, 0);  // orange fill
stroke(0, 50, 0);   // dark green stroke
ellipse(100, 100, 60, 40);
```

If you don't want an edge on a shape, then use `noStroke()`. As with the other colour-setting functions, calling `stroke(c)` will set the stroke to be the greyscale colour (c, c, c).

You can set the thickness of the stroke using the `strokeWeight` function, e.g.:

```
strokeWeight(10);
```

Colour Variables

We haven't formally introduced variables yet, but they are basically containers that have a name. The contents of the container can be changed, but the name stays the same. Remembering and writing RGB triplets is rather tedious, and so it is often convenient to store colours in variables. For example:

```
color orange = color(255, 165, 0);
```

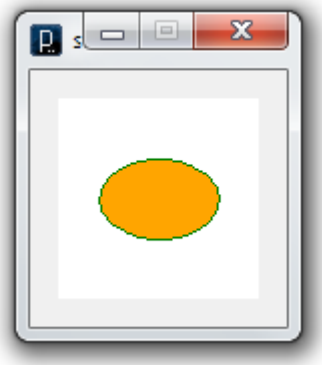
This statement creates a new variable named `orange` that labels a `color` object. Note that the keyword "`color`" has to be written in American English, without the "u". Otherwise, Processing will report an error.

In a Processing program we would use colour variables like this:

```
color orange     = color(255, 165,   0);
color dark_green = color(  0, 128,   0);
color white      = color(255, 255, 255);

void setup() {
  size(100, 100);
}

void draw() {
  background(white);
  fill(orange);
  stroke(dark_green);
  ellipse(50, 50, 60, 40);
}
```



Notice that we define the three colour variables outside of any function. That way they can be used by any function that needs them: they are global variables.

Using colour variables has a couple of advantages. First, they make the program more readable. When you see the statement `fill(orange)` you have a pretty good idea about what it does. In contrast, the statement `fill(255, 165, 0)` is not as readable. Second, it is easier to change the colours later. For instance, suppose you decide that you want your green to be darker, and so all you need to do is change its definition in one place:

```
color dark_green = color(0, 30, 0); // 30 used to be 50
```

Everywhere in your program where the variable `dark_green` is used will now have this new colour.

Transparency

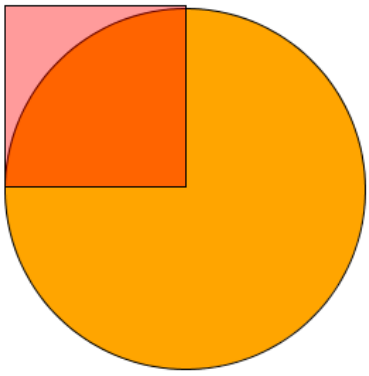
Processing also lets you specify how transparent a colour is. For example, in this program the moving rectangle has a transparency value of 100:

```
color orange          = color(255, 165,   0);
color transparent_red = color(255,   0,   0, 100); // transparent
color white           = color(255, 255, 255);

void setup() {
  size(255, 255);
}

void draw() {
  background(white);
  fill(orange);
  ellipse(255/2, 255/2, 255, 255);
  fill(transparent_red);
  rect(mouseX, mouseY, 128, 128);
}
```

It is not uncommon to see the same function name but with different numbers of parameters. This is a concept called function overloading which we will cover in more detail later in the course.

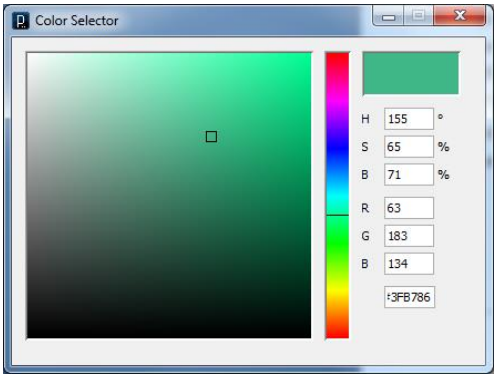


The rectangle acts a bit like red cellophane, i.e. you can see through to whatever is underneath. By changing 100 to different values you can get different levels of transparency.

Transparent colours have the format (r, g, b, alpha), where the so-called alpha-value is, like the others, a number from 0 (totally transparent) to 255 (totally opaque).

The Processing IDE Colour Selector

If you are using the programming editor that comes with Processing, then you may want to use the colour selector tool that comes with it. To use it, open the Tools menu and choose “Color Selector”. A window with a colour wheel and various boxes should appear. Click on any colour in the wheel and the RGB values for that colour will be instantly displayed.



Notice that in addition to RGB colour, the selector also supports HSB (hue, saturation, brightness) colour codes, which can sometimes be more convenient than RGB.

More Colour

We've just scratched the surface of colour in Processing, but it will suffice for most of the programs we'll write in this course. The Processing documentation lists many other colour-related functions, and they are a good start for learning more about colours.

Example: Colours and Arithmetic

It's often useful or interesting to set colours automatically. For instance, lets modify the program from the previous section to draw circles whose fill colour is a shade of grey that depends upon `mouseX` and `mouseY`:

```
void setup() {
  size(500, 500);
}

void draw() {
  noStroke();
  fill(255 * mouseX * mouseY / (500 * 500));
  ellipse(mouseX, mouseY, 50, 50);
}
```

Take a look at this line:

```
fill(255 * mouseX * mouseY / (500 * 500));
```

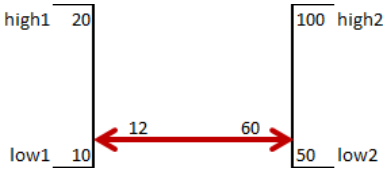
What is it doing? It sets the fill colour of the circle to be a greyscale value between 0 and 255. The chosen colour depends upon the values of `mouseX` and `mouseY`, i.e. the colour of the circle is determined by its location.

Since the screen size is 500 by 500, `mouseX` and `mouseY` both have a maximum value of 500. Thus, the product `mouseX * mouseY` has a maximum value of 500 * 500. We can't use that value directly to select a colour because it will often have a value much bigger than 255.

So we need to shrink it. The way to shrink a value in arithmetic is to use division. If we divide `mouseX * mouseY` by 500 * 500, then we get `mouseX * mouseY / (500 * 500)`, which has a maximum value of 1. Now that's too small, and so we need to stretch the number by multiplying it by 255. This gives us the expression `255 * mouseX * mouseY / (500 * 500)`, which is guaranteed to be a number from 0 to 255.

Using the `map()` Function

Shrinking and stretching numbers like this is so common in graphics programming that Processing provides a helper function to do it for you. The `map(x, low1, high1, low2, high2)` function converts x from the source range [low1, high1] to a new value in the target range [low2, high2]. The key fact is that the new value is at proportionally the same position in the target range as x is in the source range.



For example, suppose you have a point 20% of the way between 10 and 20 (i.e. the point 12), then `map(12, 10, 20, 50, 100)` returns the point 60, which is exactly 20% of the way between 50 and 100.

As another example, in the program from the previous section the first range is [0, 500 * 500], i.e. the range of possible values for `mouseX * mouseY`. The second range is [0, 255], the range of all legal greyscale colours. We can associate each number in the range [0, 500 * 500] with a number in [0, 255] using this function call:

```
map(mouseX * mouseY, 0, 500 * 500, 0, 255)
```

This always returns a number in the target range, i.e. a number from 0 to 255. And using this function we can choose colours based on the mouse position like this:

```
void setup() {
  size(500, 500);
}

void draw() {
  noStroke();
  fill(map(mouseX * mouseY, 0, 500 * 500, 0, 255));
  ellipse(mouseX, mouseY, 50, 50);
}
```

Since `map` is so useful, let's look at one final example. This time, let's draw an ellipse whose diameters are always between 25 and 100, and change in proportion to the mouse location:

```
color orange = color(255, 165, 0);
color white  = color(255, 255, 255);

void setup() {
  size(500, 500);
  noStroke();
  smooth();
}

void draw() {
  background(white);
  fill(orange);
  ellipse(mouseX, mouseY, map(mouseX, 0, 500, 25, 100),
                          map(mouseY, 0, 500, 25, 100));
}
```

Since the screen is 500 by 500, we know that `mouseX` and `mouseY` must be in the range [0, 500]. The legal diameter values are in the range [25, 100], and so `map(mouseX, 0, 500, 25, 100)` returns a value in [25, 100] that corresponds to the same location as `mouseX` in the range [0, 500].

Theory Quiz

Before attempting the below programming tasks make sure that you have a go at the theory test on AUTonline. A theory test is given each week and is just a means for you to gauge how much you have remembered from the tutorial material. The number of quizzes you complete will be tracked to determine how much effort you are putting in to learning programming.

Programming Questions

1. Write a program that draws 5 circles on a 500 x 500 screen at these screen locations:
    a. the upper-left corner
    b. the upper-right corner
    c. the lower-left corner
    d. the lower-right corner
    e. the middle

    Make sure the circles are entirely visible, i.e. don't let any part of them go off the screen.
2. Write a program that draws a circle and its bounding box, both centred in the middle of the screen. The circle should just touch the middle of each of the four sides of the box.
    Make the size of the circle and box change with respect to the mouse pointer (i.e. use `mouseX` and `mouseY` to set their width and height) so that the shapes will get bigger or smaller when the mouse moves. They should always stay centred in the middle of the screen.
3. Modify the mirror-drawing program in the notes so that the circles drawn by the user are reflected through the horizontal centre line.
4. Modify the mirror-drawing program so that three reflected circles are drawn (so there will be four circles drawn simultaneously). The three circles should reflect through:
    a. the horizontal centre line
    b. the vertical centre line
    c. both the horizontal and vertical centre line
5. Write a program that draws a purple square with a yellow edge at the mouse pointer. Make the width and height of the square equal to `(mouseX + mouseY) / 5`, and use `strokeWeight` to give the edge a thickness of at least 10.
6. Modify the previous program so that, in addition to what it already does, the colour of the square's interior is based on `mouseX + mouseY`. Use the map function to convert `mouseX + mouseY` into a legal RGB colour.
7. Write a program that draws four differently coloured triangles on the screen. The base of each triangle should be one of the four edges of the screen, and one point of each triangle should follow the mouse pointer. As the pointer moves, the four triangles change shape, but their bases always stay attached to the screen edges.
    Hint: If there's a `rectangle` function, maybe there's something for `triangles`, too?