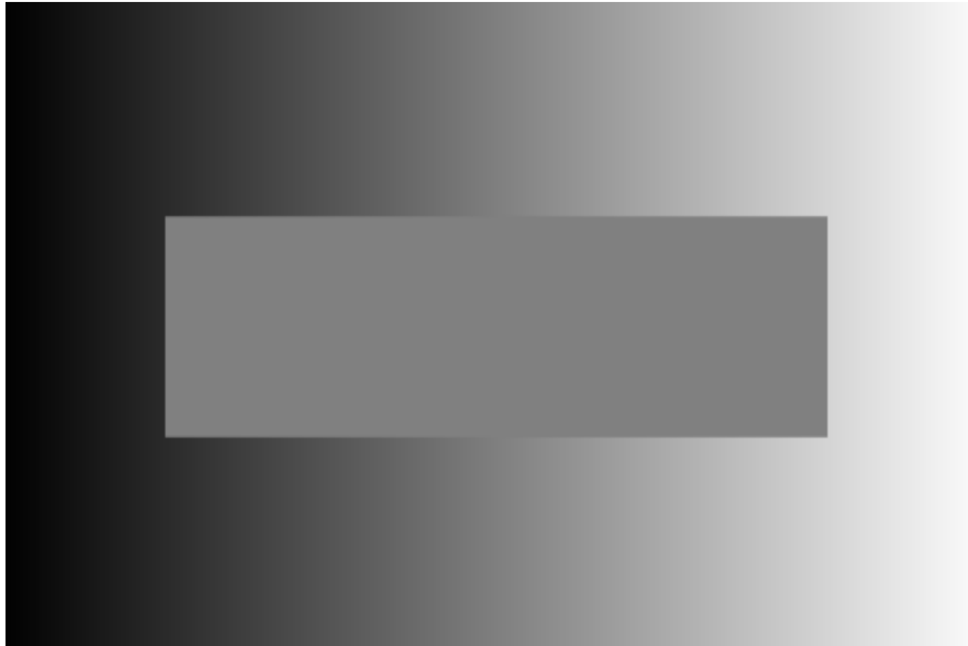
Optical Illusions

The aim of today's project is to recreate a couple of curious optical illusions, and hopefully give you some tools with which you can recreate your own. Playing around with the parameters is a good way to see just how robust some of these illusions really are, and to get a better sense of how they work.

Our first illusion is, on the surface, very simple, but will give us a good excuse to learn how to edit the colour of individual pixels in p5:

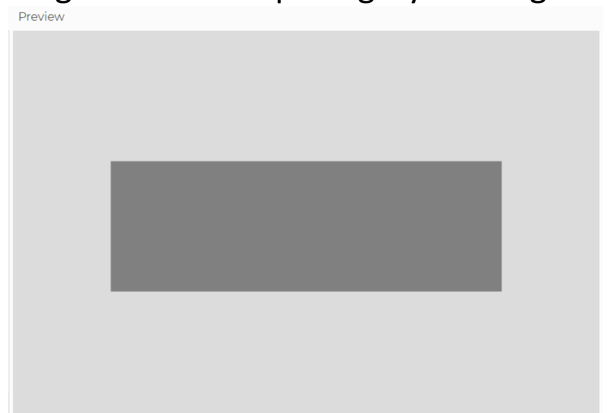


Take a good look at the grey rectangle in the middle. What do you notice?
Try covering up the border (or taking a snip of the central rectangle so you can view it without the distraction of the surrounding background). What do you notice now?

Making the rectangle

This is the easy part. The fact is, the rectangle is nothing more than a plain grey rectangle:

```
1  function setup() {
2    createCanvas(600, 400);
3  }
4
5  function draw() {
6    background(220);
7    noStroke();
8    fill(128);
9    rect(width/6, height/3, 4*width/6, height/3);
10 }
```



We create a canvas in *setup*, then in the *draw* loop we set *noStroke()* so our rectangle will have no boundary line, then *fill(128)* which gives it a uniform grey fill (128 is halfway between black at 0 and white at 255: think of the number as the amount of colour, where white is all the colour and black is the absence of it). *rect* just needs $x, y, width, height$ as its arguments, and I've set these so that it sits in the centre.

The pixel array

The clever part of this is the graduated colour for the background. To get this effect we need to have direct control over the colour information for each individual pixel. In p5, we can do this by accessing and editing the pixel array (literally an array with numbers that represent the colour of the pixels).

The first thing you might be wondering is, shouldn't this be a 2-dimensional data structure? After all, pixels are arranged in rows and columns. However, presumably for reasons of memory efficiency, the pixel data is stored in a single long list.

If we had a tiny 5 by 4 canvas, for instance, the pixel data would be stored in this order, reading left to right and top to bottom:

	0	1	2	3	4
0	0	1	2	3	4
1	5	6	7	8	9
2	10	11	12	13	14
3	15	16	17	18	19

However, there's one further complication. Each pixel requires not just one number to be stored, but *four*: the red, green, blue and alpha (opacity) values.
Here's one way of visualising the array:

	0	1	2	3	4
0	0	1	2	3	4
1	5	6	7	8	9
2	10	11	12	13	14
3	15	16	17	18	19

[255, 0, 255, 255,   255, 0, 0, 255,   0, 120, 120, 255, …]

The first four entries (indices 0, 1, 2 and 3) turn the first pixel a fetching shade of purple (red and blue), the next four (indices 4, 5, 6 and 7) make the next pixel bright red, etc.

We want to be able to access and edit the four colour values of any given pixel based on its row and column, so it pays to think through the best way to convert a row and column index into a pixel array index. Consider this modified diagram which shows not only the order in which each pixel appears, but the four positions within the pixel array that its information occupies:

	0	1	2	3	4
0	0, 1, 2, 3	4, 5, 6, 7	8, 9, 10, 11	12, 13, 14, 15	16, 17, 18, 19
1	20, 21, 22, 23	24, 25, 26, 27	28, 29, 30, 31	32, 33, 34, 35	36, 37, 38, 39
2	40, 41, 42, 43	44, 45, 46, 47	48, 49, 50, 51	52, 53, 54, 55	56, 57, 58, 59
3	60, 61, 62, 63	64, 65, 66, 67	68, 69, 70, 71	72, 73, 74, 75	76, 77, 78, 79

Can you work out how you could go from a row index (say i) and a column index (say j) in our 5 by 4 canvas to the first of the four pixel index values?

Eg, if I want to change the pixel with row index 2 and column index 3, I would need to edit the four pixel array index values starting from 52.

Can you generalise the formula for any canvas size?

Pixel Array housekeeping

Before you can modify the pixel array in p5, you have to tell it you want to by running the command *loadPixels()*, and after you're done, use the command *updatePixels()* to implement any changes you've made. Depending on the type of display your user has, there may be additional complications: high density displays effectively hold four pixels for every one pixel in a normal display. To remove this complexity, we can force the pixel density to be 1 for our sketches using *pixelDensity(1)*.

```
1  function setup() {
2    createCanvas(600, 400);
3    pixelDensity(1);
4  }
5
6  function draw() {
7    background(220);
8    noStroke();
9    fill(128);
10   rect(width/6, height/3, 4*width/6, height/3);
11   loadPixels();
12   pixels[0] = 255;
13   updatePixels();
14 }
```

We set the pixel density inside *setup*, then sandwich any code that modifies pixels between *loadPixels()* and *updatePixels()*.

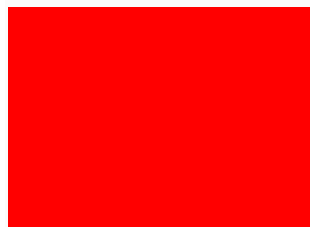
Note that this code will change how much red the top-left pixel has (but it'll be too small to detect!)

Looping through the pixel array

To access every single pixel by its row and column index, we can use a nested loop:

```
function draw() {
  background(220);
  noStroke();
  fill(128);
  rect(width/6, height/3, 4*width/6, height/3);
  loadPixels();
  for (let j=0; j<height; j++){
    for (let i=0; i<width; i++){
      let index = (j * width + i) * 4
      pixels[index+0] = 255;
      pixels[index+1] = 0;
      pixels[index+2] = 0;
      pixels[index+3] = 255;
    }
  }
  updatePixels();
}
```

Preview



The outer loop iterates through the rows, from row *0* to row *height*.
The inner loop iterates along each row, from *0* to *width*.
index represents the pixel array index that corresponds to the *i* and *j* values.
Finally, I use *index* to set the four values that relate to that particular position. In this case, everything is set to (R,G,B,A) = (255, 0, 0, 255), which is a fully opaque red.

Note that even if you set a more transparent colour, you'll still not be able to detect the rectangle we drew in the background. That's because when you edit a pixel directly, it completely overwrites anything that previous informed its colour. If instead you drew a couple of shapes overlapping, with different levels of alpha (opacity), the true pixel value would be set to some combination of the two colours. When we edit the pixel array directly, we overwrite all of this.

Making the blended colour gradient

If we want to make a smoothly blended colour change across our canvas, all we need to do is set the individual R, G and B values so that they depend up on the horizontal position, *i*.

```
  rect(width/6, height/3, 4*width/6, height/3);
  loadPixels();
  for (let j=0; j<height; j++){
    for (let i=0; i<width; i++){
      let index = (j * width + i) * 4
      pixels[index+0] = map(i, 0, width, 0, 255);
      pixels[index+1] = map(i, 0, width, 0, 255);
      pixels[index+2] = map(i, 0, width, 0, 255);
      pixels[index+3] = 255;
    }
  }
  updatePixels();
}
```

All that remains is to move the code that draws the rectangle so that it is rendered after we change the pixels:

```
loadPixels();
for (let j=0; j<height; j++){
  for (let i=0; i<width; i++){
    let index = (j * width + i) * 4
    pixels[index+0] = map(i, 0, width, 0, 255);
    pixels[index+1] = map(i, 0, width, 0, 255);
    pixels[index+2] = map(i, 0, width, 0, 255);
    pixels[index+3] = 255;
  }
}
updatePixels();
rect(width/6, height/3, 4*width/6, height/3);
```

Extension: animation

A nice adaptation to this is to incorporate a changing variable which removes the background, so the visual effect comes and goes:

```
1   let t = 0;
2   function setup() {
3     createCanvas(600, 400);
4     pixelDensity(1);
5   }
6   function draw() {
7     background(220);
8     noStroke();
9     fill(128);
10    loadPixels();
11    for (let j=0; j<height; j++){
12      for (let i=0; i<width; i++){
13        let index = (j * width + i) * 4
14        let lower = map(sin(t), -1, 1, 0, 255);
15        pixels[index+0] = map(i, 0, width, lower, 255);
16        pixels[index+1] = map(i, 0, width, lower, 255);
17        pixels[index+2] = map(i, 0, width, lower, 255);
18        pixels[index+3] = 255;
19      }
20    }
21    updatePixels();
22    rect(width/6, height/3, 4*width/6, height/3);
23    t += 0.2;
24  }
```

This sets a value *t* which increments each time the draw loop runs. I take advantage of the smooth oscillation of a sine wave, and I've used the *map* function to map values to between *0* and *255* instead of between *-1* and *1*.

Note: if your animation runs a bit slowly, it's because we're editing nearly a million numbers every frame (4 values per pixel for 240,000 pixels). Try reducing the canvas size to 300 by 200 for a smoother effect. Now we're animating, consider optimising:

```
      let bright = map(i, 0, width, lower, 255);
      pixels[index+0] = bright;
      pixels[index+1] = bright;
      pixels[index+2] = bright;
//      pixels[index+3] = 255;
```

By making a separate variable, we only have to run the *map* code once per pixel. And there's no need to update the alpha value every time either.