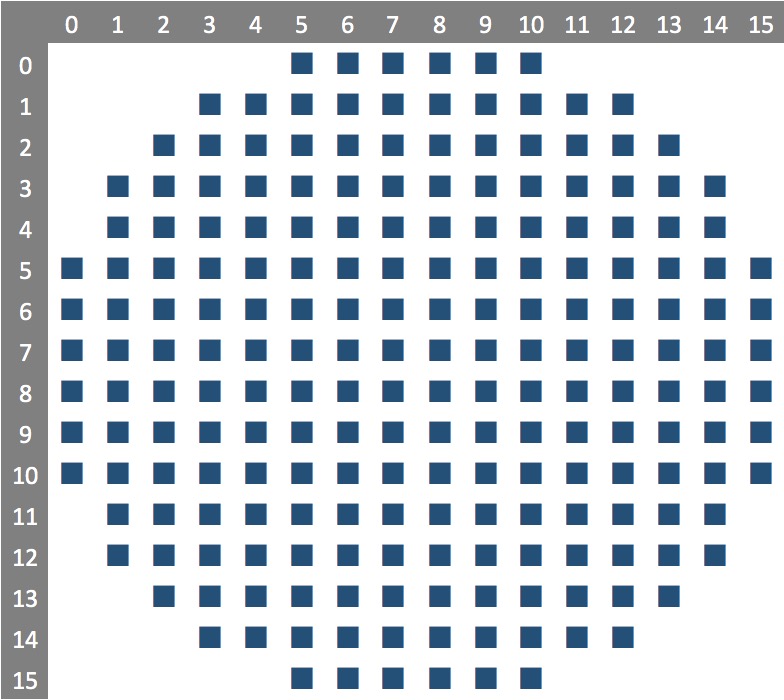
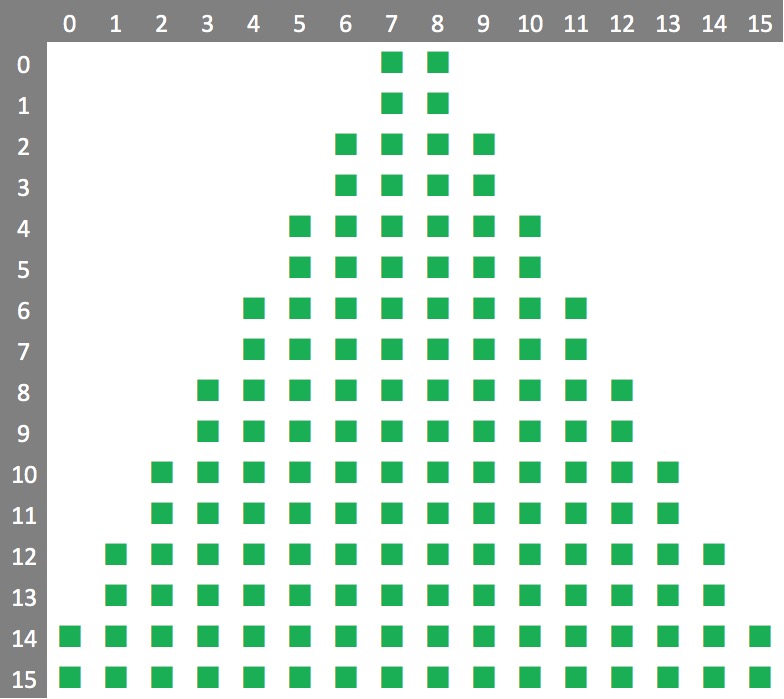
# Detecting Collisions between two Images

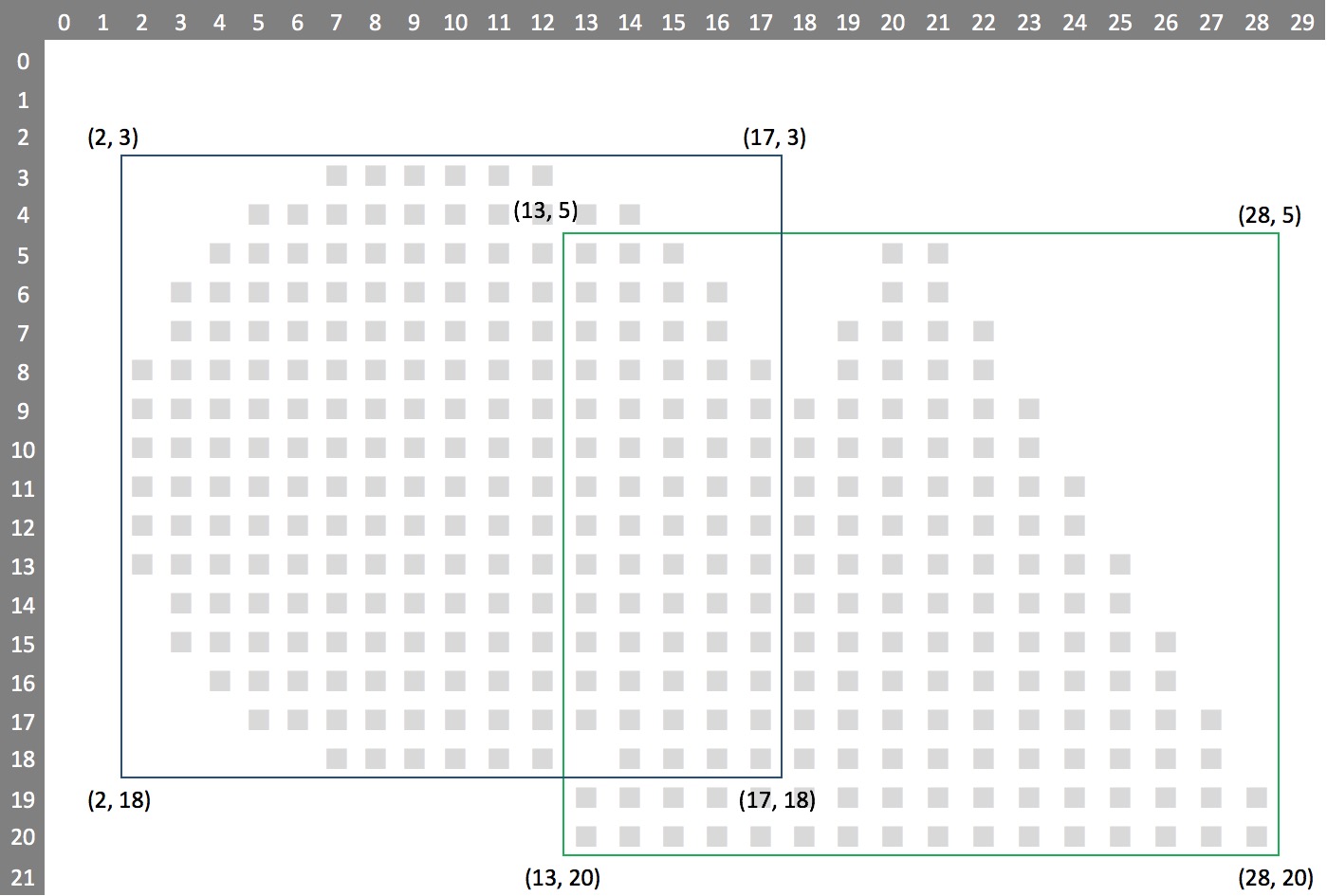
In my article last month, I covered the basics of collision detection using the Arduboy2 library. This method is very fast but suffers in that a collision might be reported when only the ’white space’ surrounding your images cross. Visually the objects may miss each other completely!

This article describes a method of comparing two images at the bit (or pixel) level to see if any of the set bits collide. I have included a sample application in my repository, <https://github.com/filmote/CollideImage>, for you to download and try.

This article will discuss the collision detection between the two images below:

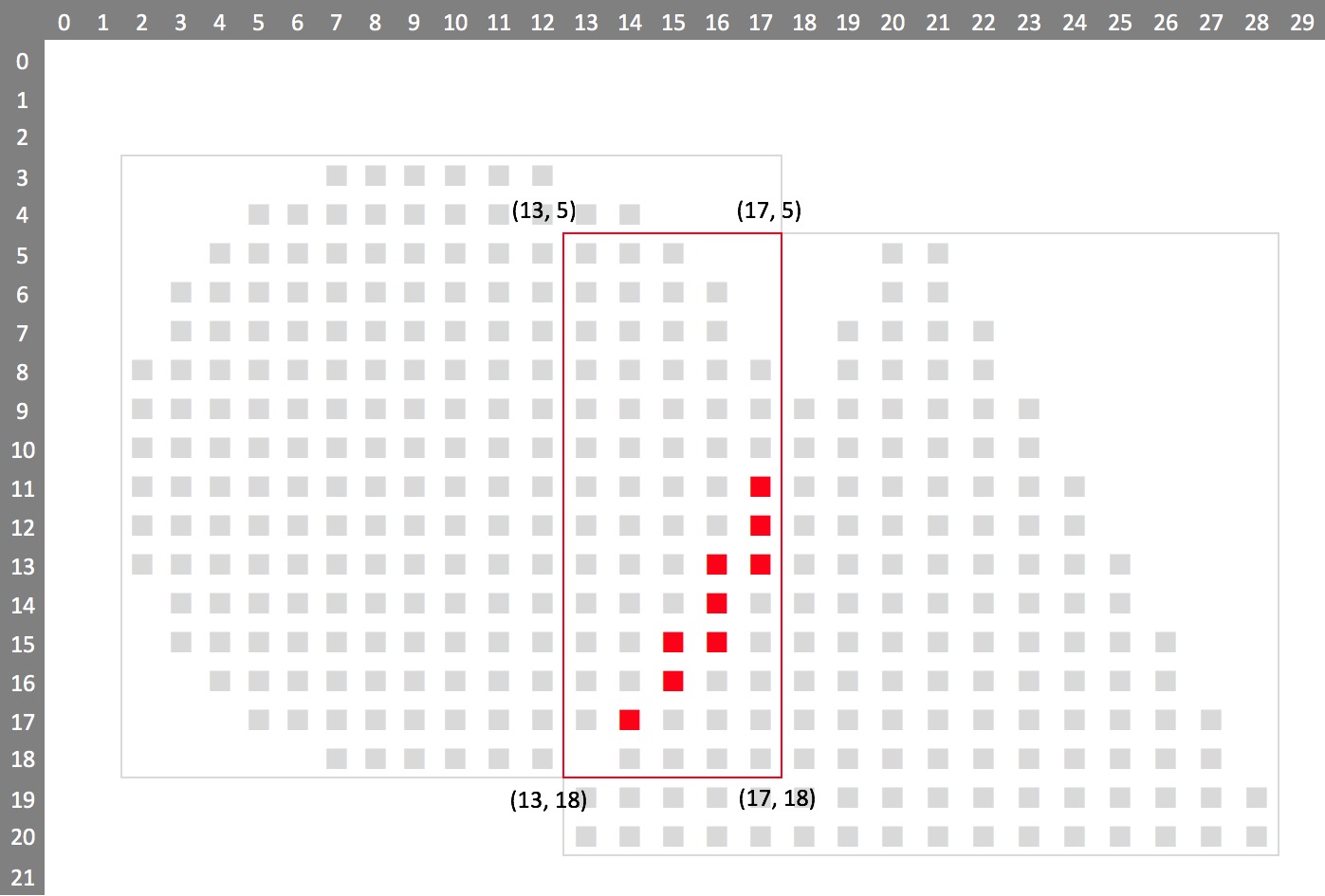
 

Assume they are rendered on the screen as shown below:



The first image (the circle) starts at position (2, 3) and extends 16 bits either way to (17, 18) inclusive.  The second image (the triangle) starts at position (13, 5) and extends 16 pixels either way to (28, 20) inclusive.

To understand if there has been a collision, we first need to calculate the overlapping section, as highlighted by the red rectangle below. For clarity, the pixels that have actually collided are coloured red.



The overlapping rectangle area can be determined by applying the following formulas:

overlap\_left = max( img1.x, img2.x );

       = max( 2, 13 );

       = 13;

overlap\_top = max(img1.y, img2.y );

        = max( 3, 5 );

        = 5;

overlap\_right = min(img1.x + img1.width, img2.x + img2.width );

          = min( 2 + 16, 13 + 16 );

          = min( 18,  29 );

           = 18;

overlap\_bottom = min(img1.y + img1.width, img2. y + img2.width );

               = min( 3 + 16, 5 + 16 );

               = min( 19, 21 );

               = 19;

Overlapping rectangle starts at (13, 5) in the top left-hand corner and extends to (17, 18) in the bottom right-hand corner, inclusive. Note that our calculation returned a right and bottom value that were exclusive (18, 19).

To retrieve the pixels that overlap each other, we need to work out which section of the image the overlapping rectangle applies to. The calculations above returned screen coordinates and take into account that the image is not located at (0, 0).

If you recall, the images are defined as an array of bytes (or uint8\_t) with the first two bytes specifying the width and height of the image and the remainder the image data itself. Each byte of the image data represents a vertical column of 8 pixels with the least significant bit at the top. The bytes are drawn as 8 pixel high rows from left to right, top to bottom. When the end of a row is reached, as specified by the width value, rendering continues on the following line.

The definition for the ‘circle’ and ‘triangle’ images are shown below. The bytes in red are those that we will ultimately be retrieving and comparing. Those in inverse red / white are the bytes that will be described in the following section:

const uint8\_t PROGMEM circle[] = {

0x10, 0x10,

0xE0, 0xF8, 0xFC, 0xFE, 0xFE, 0xFF, 0xFF, 0xFF, // Row 1

0xFF, 0xFF, 0xFF, 0xFE, 0xFE, 0xFC, 0xF8, 0xE0,

0x07, 0x1F, 0x3F, 0x7F, 0x7F, 0xFF, 0xFF, 0xFF, // Row 2

0xFF, 0xFF, 0xFF, 0x7F, 0x7F, 0x3F, 0x1F, 0x07,

};

const uint8\_t PROGMEM triangle[] = {

0x10, 0x10,

0x00, 0x00, 0x00, 0x00, 0xC0, 0xF0, 0xFC, 0xFF, // Row 1

0xFF, 0xFC, 0xF0, 0xC0, 0x00, 0x01, 0x03, 0x07,

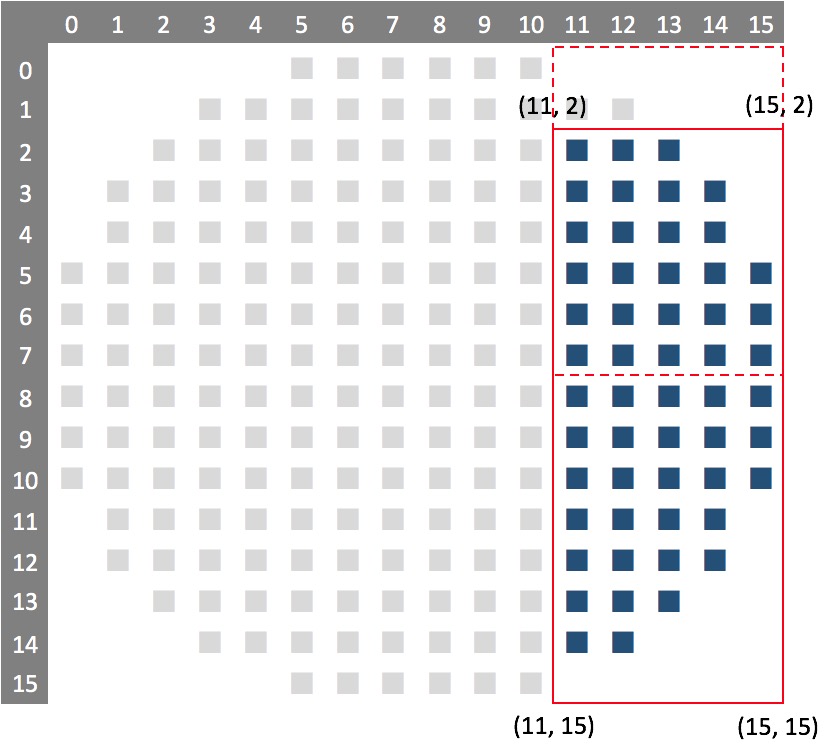
0xC0, 0xF0, 0xFC, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, // Row 2

0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0xF0, 0xC0,

};

When calculating the rectangle of the image that we are interested in, it is important to realize that although the image is defined as rows of bytes – with each byte representing a vertical column of 8 bits (or pixels) – the rectangle we will want to retrieve will often result in us needing to mask some of the bits out of the data.

In the image below, the solid red lines represent the overlapping area between the two images. The dashed lines highlight the bytes that will be retrieved. As you can see, the top two bits in the upper row of data are not needed when detected if there is a collision or not. The entirety of the bottom row is included in the overlapping area and hence no bits need to be masked off.



The following calculations determine the same coordinates relative to each image:

img1\_left = (overlap\_left – img1.x);

= 13 – 2;

= 11;

img1\_right = (overlap\_right - img1.x);

= 18 – 2;

= 16;

img1\_top\_row = (overlap\_top - img1.y) / 8;

= (5 – 3) / 8;

= 0;

img1\_top\_bit = (overlap\_top - img1.y) % 8;

= (5 – 3) % 8;

= 2;

img1\_bottom\_row = (overlap\_bottom - img1.y - 1) / 8;

= (19 – 3 – 1) / 8;

= 1;

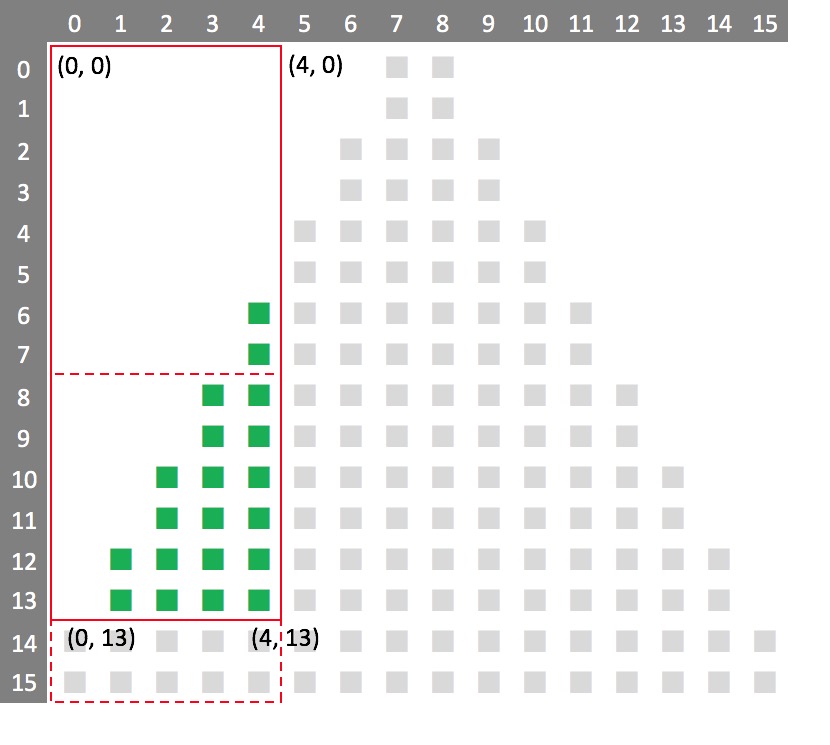
img1\_bottom\_bit = (overlap\_bottom - img1.y) % 8;

= (19 – 3) % 8;

= 0;

Rather than calculating top and bottom coordinates, we have calculated a left and right index which we can then use while iterating through the array of image data. We have also calculated a top row and the number of bits to mask – row 0 with the first two bits masked. Likewise, we have calculated the bottom row and any bits to mask – row 1 and no bits.

Repeating the process on the triangle image produces a similar result although this time we will be masking off the lower 2 bits of the bottom row.



img2\_left = (overlap\_left – img2.x);

= 13 – 13;

= 0;

img2\_right = (overlap\_right – img2.x);

= 18 – 13;

= 5;

img2\_top\_row = (overlap\_top – img2.y) / 8;

= (5 – 5) / 5;

= 0;

img2\_top\_bit = (overlap\_top – img2.y) % 8;

= (5 – 5) % 8;

= 0;

img2\_bottom\_row = (overlap\_bottom – img2.y - 1) / 8;

= (19 – 5 – 1) / 8;

= 1;

img2\_bottom\_bit = (overlap\_bottom – img2.y) % 8;

= (19 – 5) % 8;

= 6;

With these calculations done, we can retrieve the data from the image definition.

The first byte to be read is determined using the code below. It takes into account a starting byte that may be on a row other than the first and adds two additional bytes to cater for the two dimension bytes that start all image definitions. The width of the image is retrieved by reading the first byte of the image itself.

#define IMG\_DATA\_OFFSET 2

uint8\_t w1 = pgm\_read\_byte(&img1[0]);

uint8\_t w2 = pgm\_read\_byte(&img2[0]);

int16\_t i1 = (img1\_top\_row \* w1) + img1\_left + IMG\_DATA\_OFFSET;

= 13;

int16\_t i2 = (img2\_top\_row \* w2) + img2\_left + IMG\_DATA\_OFFSET;

= 2;

The first byte to be read from the circle is calculated to have an index of 13 and the triangle at index 2. A quick look at the triangle reveals that the first 4 bytes of the image are empty (0) and thus will not demonstrate the process well. Imagine we have skipped along four bytes without a collision and we are now testing:

int16\_t i1 = 17;

int16\_t i2 = 6;

Looking back at the image definitions, we can see that the value retrieved at index 17 of the circle is 0xE0 which can be represented as a binary number as 1110 0000. The value at index position 6 of the triangle is 0xC0 or 1100 0000.

const uint8\_t PROGMEM circle[] = {

0x10, 0x10,

0xE0, 0xF8, 0xFC, 0xFE, 0xFE, 0xFF, 0xFF, 0xFF, // Row 1

0xFF, 0xFF, 0xFF, 0xFE, 0xFE, 0xFC, 0xF8, 0xE0,

0x07, 0x1F, 0x3F, 0x7F, 0x7F, 0xFF, 0xFF, 0xFF, // Row 2

0xFF, 0xFF, 0xFF, 0x7F, 0x7F, 0x3F, 0x1F, 0x07,

};

const uint8\_t PROGMEM triangle[] = {

0x10, 0x10,

0x00, 0x00, 0x00, 0x00, 0xC0, 0xF0, 0xFC, 0xFF, // Row 1

0xFF, 0xFC, 0xF0, 0xC0, 0x00, 0x01, 0x03, 0x07,

0xC0, 0xF0, 0xFC, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, // Row 2

0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0xF0, 0xC0,

};

Our collision code compares 8 bits from each image together to see if there is a collision. However, as we discovered when working out the overlapping portion that the top of the area will not always align with the underlying bytes of data. In our calculations, the top two bits of the circle’s data would be discarded leaving us with only 6 bits. Reading the first byte from the triangle image would return a full 8 bits as no bits were to be discarded. To provide a valid comparison, we need to ensure we are comparing the same number of bits from each image.

To achieve, this, we need to read and append the first two bits of the byte directly below the one we are processing to complete our eight bits. Depending on the top and bottom bit masking values, we may need to mask the top and / or bottom.

The first byte value is retrieved from each image using the code below:

uint16\_t d1 = pgm\_read\_byte(&img1[i1]) & (img1\_top\_row == img1\_bottom\_row && img1\_bottom\_bit != 0 ? pgm\_read\_byte(&lookup[img1\_bottom\_bit]) : 0xFF);

uint16\_t d2 = pgm\_read\_byte(&img2[i2]) & (img2\_top\_row == img2\_bottom\_row && img2\_bottom\_bit != 0 ? pgm\_read\_byte(&lookup[img2\_bottom\_bit]) : 0xFF);

Breaking it down reveals how it works. First the top row of data is retrieved:

i1 = 17;

i2 = 6;

uint16\_t d1 = pgm\_read\_byte(&img1[i1])

= 0xE0;

uint16\_t d2 = pgm\_read\_byte(&img2[i2])

= 0xC0;

If we are only reading one row of data or we are reading the last row of data, we may need to mask the bottom of pixels off. In both images we are reading the first of two lines, so the remainder of the statement is ignored. We will cover this little bit of code later on in the article.

As mentioned earlier, it may be necessary to retrieve the byte of data directly below the one currently being assessed to build a full complement of 8 bits. This logic only applies if the calculate top bit is not zero and only if we are not yet processing the last row in the image.

if (img1\_top\_bit > 0 && img1\_top\_row < img1\_bottom\_row) {

d1 = d1 | ((pgm\_read\_byte(&img1[i1 + w1]) & (img1\_top\_row + 1 ==

img1\_bottom\_row ? pgm\_read\_byte(&lookup[img1\_bottom\_bit]) : 0xFF )) << 8);

}

Again, breaking the code down we can see the condition that tests for the top bit being non-zero and ensures we are not processing the last row of the image.

if (img1\_top\_bit > 0 && img1\_top\_row < img1\_bottom\_row) { …

If this evaluates to true, the byte of image data directly below the one retrieved previously is also retrieved. Its index can easily be determined by adding the width of the image to the index of the byte retrieved previously.

… (pgm\_read\_byte(&img1[i1 + w1]) …

If we are only reading one row of data or we are reading the last row of data, we may need to mask the bottom of pixels off. This is achieved with the second portion of the statement that calculates a mask to apply to the second value just retrieved.

Breaking it down further - if the second byte of image data was retrieved from the last row of data, we may need to apply a mask to remove the lower bits ..

… img1\_top\_row + 1 == img1\_bottom\_row …

.. then we work out the mask to apply. If no mask is needed, a default mask is returned that ensures no pixels are affected.

… ? pgm\_read\_byte(&lookup[img1\_bottom\_bit]) : 0xFF …

To speed up to the process, I have defined the mask values as constants rather than calculating them on the fly. To mask the lowest bit of an image, the mask needs to be 0111 1111. To mask the lower two bits, the mask needs to be 0011 1111 and so on. These same masks can be defined as 1111 1111 >> {number of rows to mask} or 0xFF >> {number of rows to mask}. As you can see, this is exactly how the lookup has been defined.

const uint8\_t PROGMEM lookup[] { 0xFF >> 8, 0xFF >> 7, 0xFF >> 6, 0xFF >> 5,

0xFF >> 4, 0xFF >> 3, 0xFF >> 2, 0xFF >> 1};

Finally, the value retrieved is shifted to the left 8 times (the equivalent of multiplying it by 256) and logically OR’ed to the image data we retrieved from the top line to produce a 16-bit number with the high order byte representing the second row of image data and the low order byte representing the first row. If there was a top bit to mask, this is done be right shifting the resultant 16-bit number and then masking all but the lower 8 bits to create a single byte.

d1 = d1 | ( *lower byte* ) << 8);

d1 = (d1 >> img1\_top\_bit) & 0xFF;

Let’s plug in the numbers from our two images. The data from the top line of the image was determined to be 0xE0 and 0xC0 for the circle and triangle respectively. The other important values are repeated below:

img1\_left = 11;

img1\_right = 16;

img1\_top\_row = 0;

img1\_top\_bit = 2;

img1\_bottom\_row = 1;

img1\_bottom\_bit = 0;

img2\_left = 0;

img2\_right = 5;

img2\_top\_row = 0;

img2\_top\_bit = 0;

img2\_bottom\_row = 1;

img2\_bottom\_bit = 6;

i1 = 17;

i2 = 6;

d1 = 0xE0;

d2 = 0xC0;

if (img1\_top\_bit > 0 && img1\_ top\_row < img1\_bottom\_row) {

d1 = d1 | ((pgm\_read\_byte(&img1[i1 + w1]) & (img1\_ top\_row + 1 ==

img1\_bottom\_row ? pgm\_read\_byte(&lookup[img1\_bottom\_bit]) : 0xFF )) << 8);

}

if (img2\_top\_bit > 0 && img2\_ top\_row < img2\_bottom\_row) {

d2 = d2 | ((pgm\_read\_byte(&img2[i2 + w2]) & (img2\_ top\_row + 1 ==

img2\_bottom\_row ? pgm\_read\_byte(&lookup[img2\_bottom\_bit]) : 0xFF )) << 8);

}

As the top bit on image 2 is equal to zero, the code is not executed for it. For the first image (the circle) reading the value at pgm\_read\_byte(&img1[i1 + w1])returns 0x07. We are retrieving this value from the last row of the image so we can further simplify the code to:

d1 = d1 | ((0x07 & pgm\_read\_byte(&lookup[img2\_bottom\_bit])) << 8);

Which, after the lookup, becomes:

d1 = d1 | ((0x07 & (0xFF >> 2)) << 8);

= d1 | ((0x07 & 0x3F) << 8);

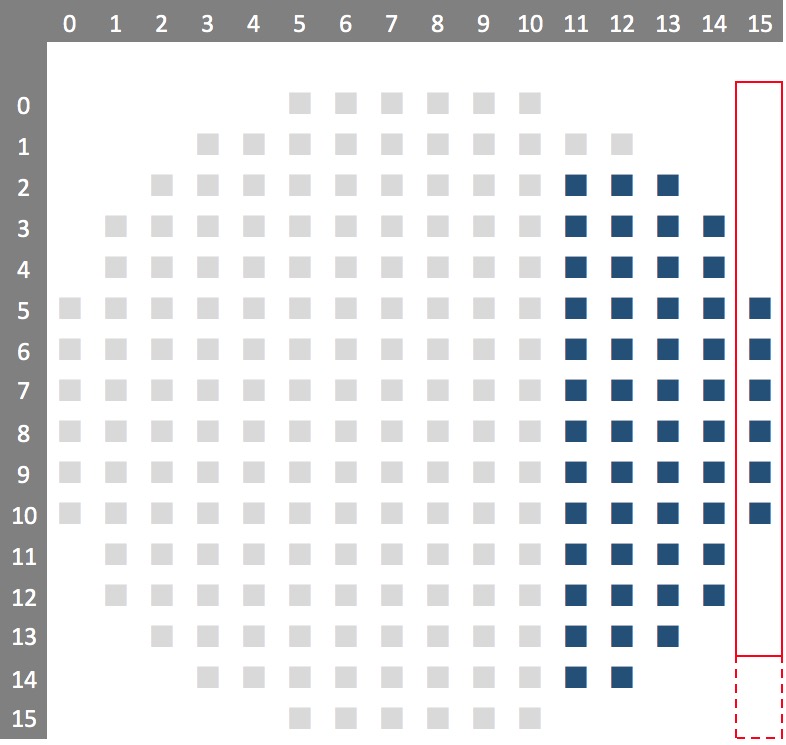
= d1 | (0x07 << 8);

= 0xE0 | 0x0700;

= 0x07E0;

As you can see, we have created a 16 bit value from the two bytes we have read. The value retrieved for the second line, 0x07 was masked using the value 0x3F (binary 0011 1111) which had the effect of masking off the two most significant bits of the value. These bits correspond to the two bottom pixels we wanted to remove.

The value 0x07E0 can be represented as 0000 0111 1110 0000 which you can see is the right most column of our image. We have already masked off the lower two bits – although they happened to be blank anyway.



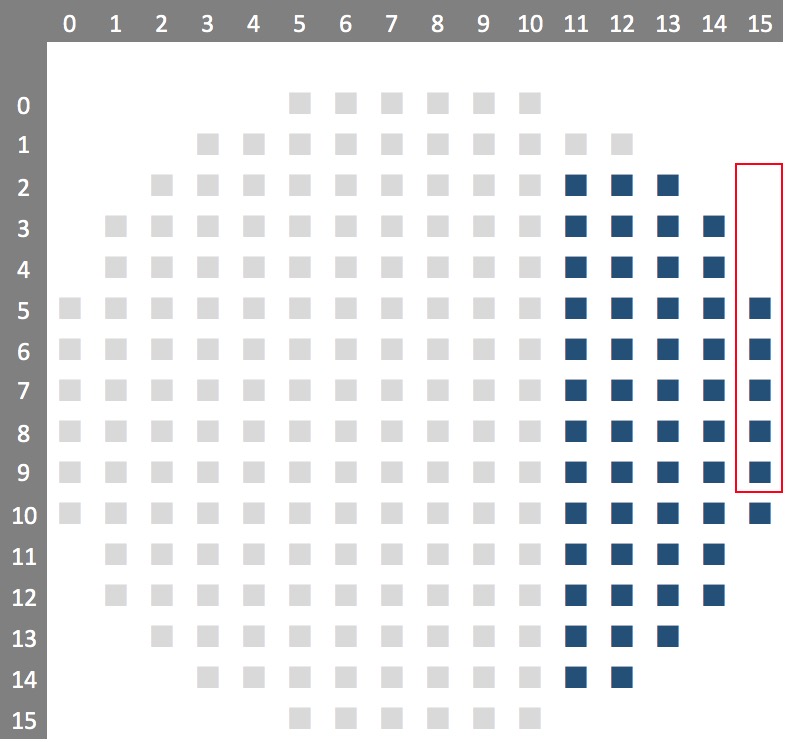
The final piece of the puzzle is to shift the result to remove any unwanted pieces at the top of the image and to truncate the data to 8 bits for comparison.

d1 = (d1 >> img1\_top\_bit) & 0xFF;

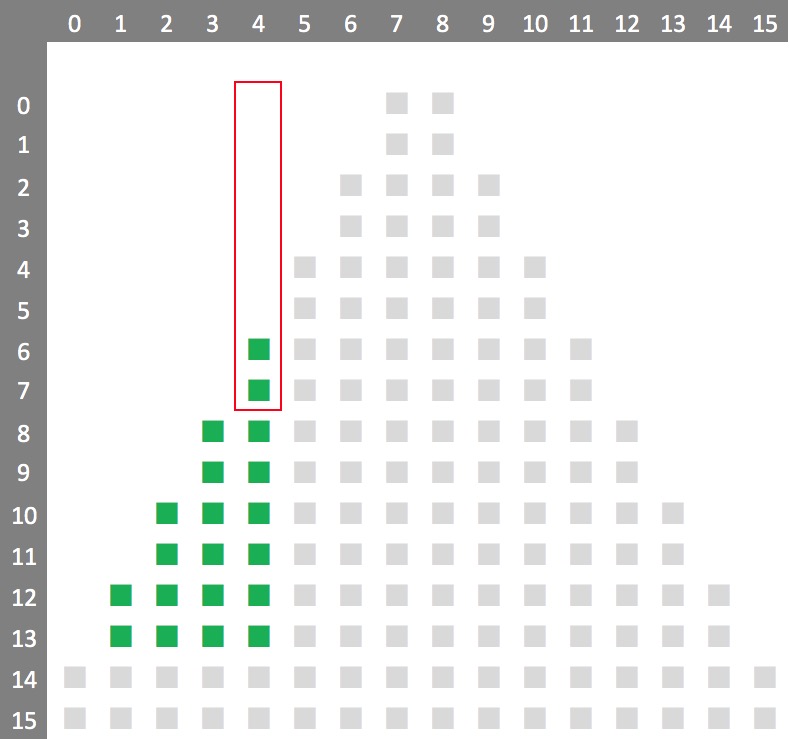
= (0x07E0 >> 2) & 0xFF;

= (0x1F8) & 0xFF;

= 0xF8 or 1111 1000;



Applying the same logic to the triangle results in:



d2 = (d2 >> img2\_top\_bit) & 0xFF;

= (0xC0 >> 0) & 0xFF;

= 0xC0 or 1100 0000;

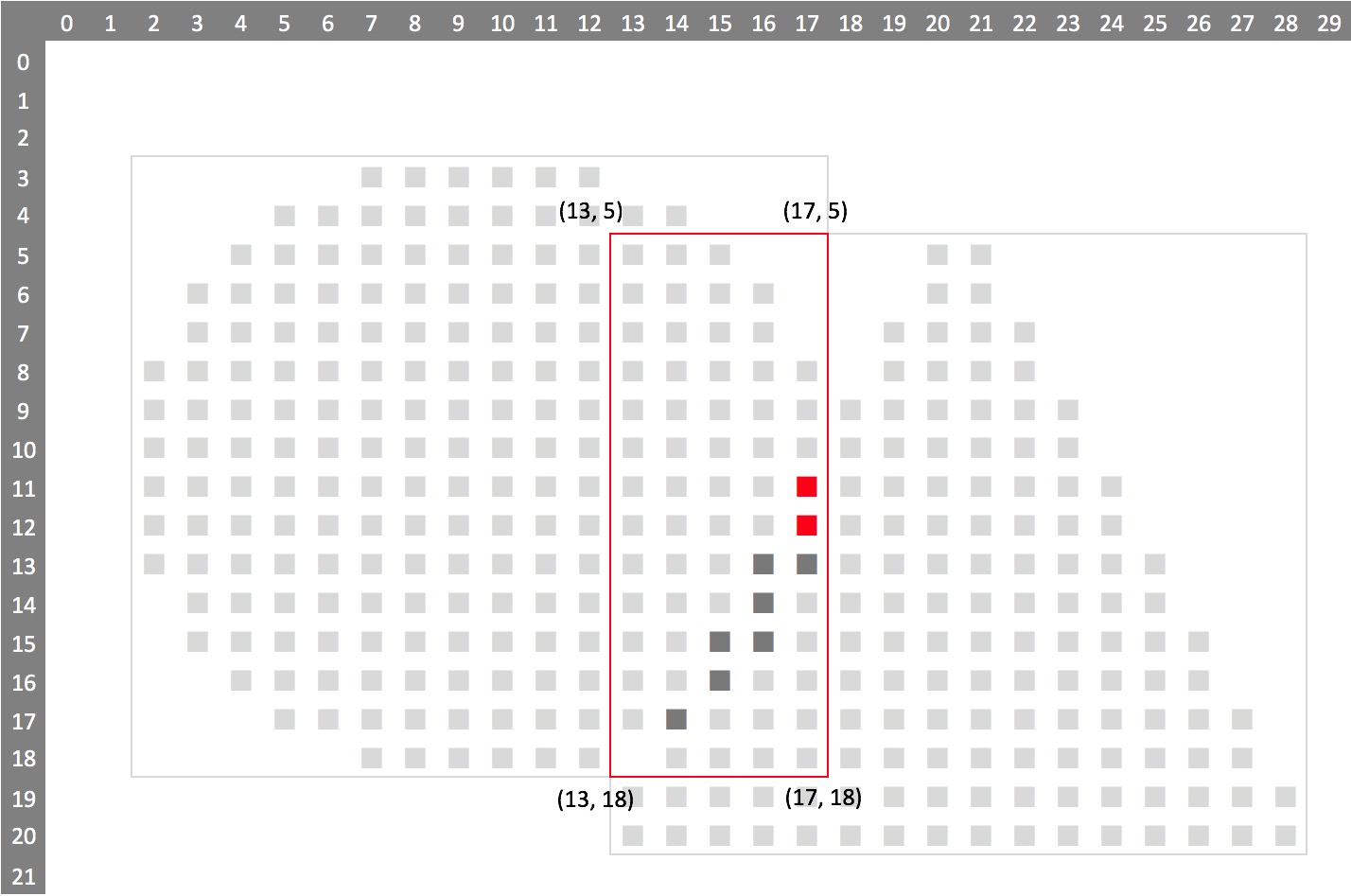
Now we have our two pieces of data we can compare them together to see if they collide at all. Converting the two values 0xF8 and 0xC0 to binary, make it easier to visualise the collision. The image data from the circle, 0xF8, converts to 1111 1000 whereas the triangle’s value 0xC0 converts to 1100 0000. The lower two bits collide!

if ((d1 & d2) > 0) {

return true;

}

The two bits we have detected are highlighted in the image below. Congratulations, your sprites have collided and you have stuck this far into the tutorial!



If no collision has been detected, we increase the index and do it all again. When the right most column is reached, the index is wrapped around to the next row and processing continues until a match is detected or all columns and rows have been compared.

if (i1 < (img1\_top\_row \* w1) + img1\_right + IMG\_DATA\_OFFSET) {

++i1;

++i2;

}

else {

if (img1\_top\_row < img1\_bottom\_row) {

++img1\_top\_row;

++img2\_top\_row;

i1 = (img1\_top\_row \* w1) + img1\_left + IMG\_DATA\_OFFSET;

i2 = (img2\_top\_row \* w2) + img2\_left + IMG\_DATA\_OFFSET;

}

else {

return false;

}

}

Earlier on when we retrieved the very first pieces of image data, we skipped over a section of code as it did not apply and I said we would cover it later. If you were paying attention, you may have noticed that this code was identical to the code that masks the bottom pixels when retrieving the second row of data. This code exists when retrieving the first piece of data to cater for the scenario where the overlap occurs within a single row of data.

Simple right?

A sample project can be found in my GitHub repository <https://github.com/filmote/CollideImage>

This method seems quite laborious and I welcome input to speed it up. You will see in the sample project that I perform a simple check using the standard Arduboy2 method before doing any pixel - level work to ensure that the code returns promptly if there is no chance of a collision. When a possibility of a collision is detected, the code processes the overlapping rectangle left to right and then top to bottom and will return on the first collision. If the collision is detected in the first comparison the code will again return quickly however if a collision is not detected until the last byte of data is compared, then the routine must process a lot of data.

This may sound like a significant drawback but need not be. If you are detecting the collisions as two objects move around the screen – say one or two pixels at a time – the overlapping area when two image rectangles collide will be quite small minimizing the amount of processing required.

The code sample provided can easily be extended to accept two images and their corresponding masks and the detection code altered to execute over the mask instead.