Coding Tetris

With 8BitCADE

Compatible with:

8BitCADE XL

Contents

Written by the 8BitCADE Team
Support@8bitcade.com
Version 1
© 2020 8BitCADE Limited

Creating Tetris

In this tutorial, we bring together all of what we have learned to create Tetris on the 8BitCADE XL.

The below tutorials must be completed to fully understand this tutorial:

- Arduino Basics
- Arduinio Arrays
- Arduino Functions

The below tutorials are advised, but not necessary, to be completed beforehand:

- 8Bit Etch-a-Sketch Project

Everything can be found in the fundamentals section on the LEARN page – don't worry! These will not take long – it is just important for you to understand these before we move into coding! You will be up and running in no time!

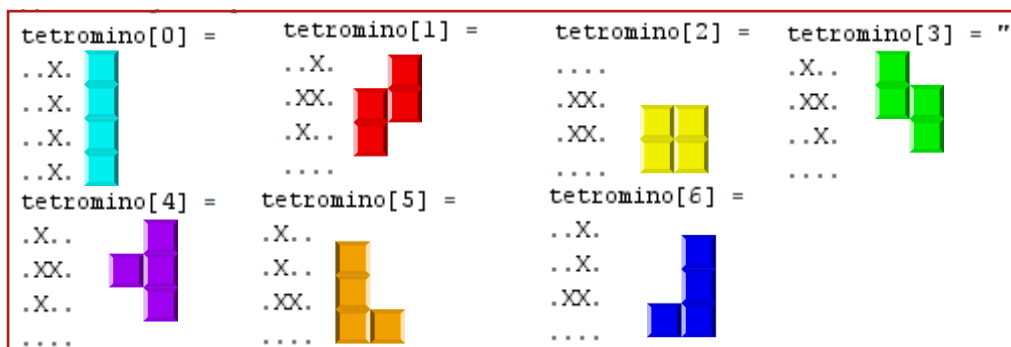Your first task is to create a new file called "Tetris_Tutorial"

Game Assets

The first thing we are going to need when creating any game Is some game assets. In Tetris, this means we need the various shapes – known as tetrominos.

To create these, we will first start by creating an array of 7 strings – one element for each piece. To create the pieces, we will use dots (.) to represent empty space and X's (X) to represent a part of the shape.

```
1  String tetromino[7];
2  void setup() {
3
4    //Creating the pieces
5    tetromino[0] = "..X...X...X...X."; // Tetronimos 4x4
6    tetromino[1] = "..X..XX..X......";
7    tetromino[2] = ".....XX..XX.....";
8    tetromino[3] = ".X...XX...X.....";
9    tetromino[4] = ".X...XX..X......";
10   tetromino[5] = ".X...X...XX.....";
11   tetromino[6] = "..X...X..XX.....";
12 }
```

Now this might seem quite confusing, however, if I was to break out each string into a 4x4 grid, you will see how the X's and dots come together to create the tetrominos:


```
tetromino[0] =          tetromino[1] =          tetromino[2] =          tetromino[3] = "
..X.                    ..X.                    ....                    .X..
..X.                    .XX.                    .XX.                    .XX.
..X.                    .X..                    .XX.                    ..X.
..X.                    ....                    ....                    ....
tetromino[4] =          tetromino[5] =          tetromino[6] =
.X..                    .X..                    ..X.
.XX.                    .X..                    ..X.
.X..                    .XX.                    .XX.
....                    ....                    ....
```

However, we write them inline as its easier and looks nicer! *Be sure to copy this bit of code in the void setup on lines 1 to 12.*

Using 1 Dimensional Arrays VA 2 Dimensional Arrays

When creating the assets, we have an option of creating them with a 2-dimensional array, or with a single dimensional array. In this section, we will go over why we will be using a single dimensional array.

If you want to check out the below explanation in video format, you can check out @javidx9 brilliant explanation from **3:13 to 7:23.**

https://youtu.be/8OK8_tHeCIA?t=193

We can create a grid using a 2-dimensional array that will look something like this:
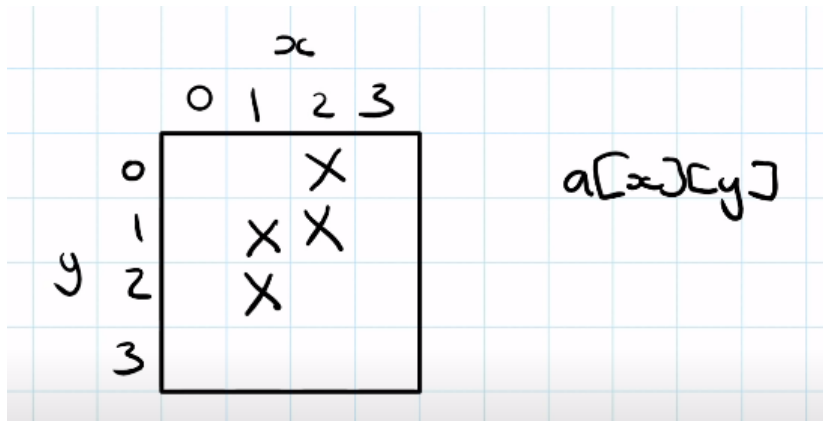
However, using this, we would have to create an asset for every single shape – and every single shape variation.

When we create this array in memory, what's actually happening is we get a contiguous block of memory. This means the memory blocks (e.g. memory block a[0][0], the first block, is next to the next block, a[1][0] and so forth) are assigned one after another.

If you look at the diagram on the left, you can see how we can now fill in the blank grid with numbers. This shows how each block is assigned one after another.

If we want to access memory block 10, in a two-dimensional array we would use a[2][2] as it is 2 columns across and 2 rows down.

Whereas in a one-dimensional array, we can use a formula to get the position using X and Y coordinates, as if the one-dimensional array was in a grid-like fashion.

$$i = y \times w + x$$

$$10 = 2 \times 4 + 2$$

4
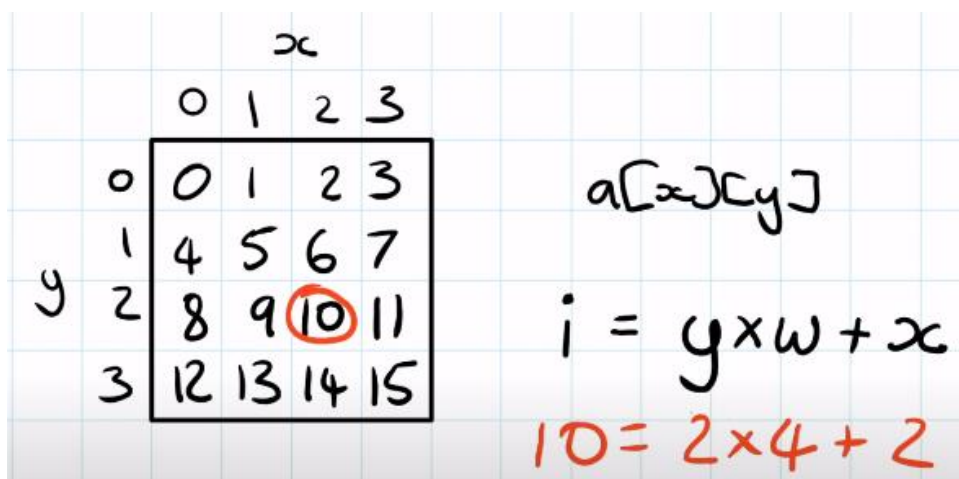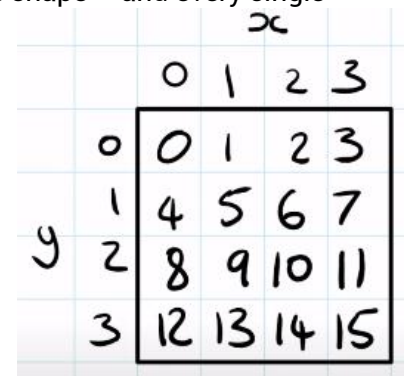
The formula is:

Index = (number of rows down) * (width of the array) + (number of columns across)

I = Y * w + x

Meaning we can take our x and y coordinates and calculate where it exists in our one-dimensional array.

Similar to what we would do if we had a two-dimensional array (using each dimension as an axis – x and y)

$$i = y \times w + x$$
$$10 = 2 \times 4 + 2$$

So why do we use this? Why don't we just use a two-dimensional array?

Consider when we need to rotate a Tetris piece:

$$0°) \; i = y \times w + x \qquad\qquad 90°) \; i = 12 + y - (x \times 4)$$

Here we can see what happens when we rotate a Tetris Piece. All the memory block values shift. We can also see, by comparing both grids, that we can find patterns and create an equation from it.

Compare both grids, notice that when x and y are both 0, the first index value is 12 in grid 2. Another pattern is when y increases, so do the index values by 1. Therefore we just add y to the 12 value. E.g. when the y coordinate is 1 in grid 2, the index position becomes 13, y = 2 then the index value is 14 etc.

Now we need to focus on the change in x in grid 2. We can see that it decreases in multiples of 4. When x = 1 the index value is reduced by 4 (12 − 8), when x = 2 the index value is reduced by 4 again (8 − 4) etc.

With these two rules we can create a formula:

I = 12 + y − (x * 4)

This formula rotates our one-dimensional array by 90 degrees and gives us the appropriate indexes for this rotation.

For a 180 degree rotation the formula is as so: i = 15 − (y * 4) − x

And for a 270 degree rotation, the formula is: i = 3 + y(x∗4)

Using these formulas allows us to only use one asset for each piece, using the index value to emulate a rotation for each piece.

Implementing Rotation

Now that we have the formulas, we need to implement this in code.

```
20  int Rotate(int px, int py, int r)
21  {
22      int pi = 0;
23      switch (r % 4)
24      {
25          case 0: // 0 degrees
26              pi = py * 4 + px;
27              break;
28          case 1: // 90 degrees
29              pi = 12 + py - (px * 4);
30              break;
31          case 2: // 180 degrees
32              pi = 15 - (py * 4) - px;
33              break;
34          case 3: // 270 degrees
35              pi = 3 - py + (px * 4);
36              break;
37      }
38      return pi;
39  }
```

Here we can see we have created a function that returns an integer (hence specifying the function as int) that takes in 3 parameters:

Piece X coordinate = **px**

Piece Y coordinate = **py**

And a value that represents how much we want to rotate the piece by = r. This is a value from 0 to 3

Using a switch command we can rotate the piece depending on the r value.

- 0 represents no change/0 degrees
- 1 represents a 90-degree turn
- 2 represents a 180-degree turn
- 3 represents a 270-degree turn

In the switch case condition, we use the modulo or MOD arithmetic operator to divide the r value by 4 and get the reminder.

If you are unfamiliar with modular (%), this operator returns the remainder of the division. E.g. 9 % 4 would be 1 as 4 fits into 8 2 times reminder 1. The modular operator disregards the 2 and simply takes the remainder. E.g. 19 % 8 = 16 r 3 therefore would return 3.

Using modular means the r value could be 23 (meaning we have clicked the rotation button 23 times) but as we are using modular, the r value would be remainder 3 (23 % 4 = 5 r 3) meaning a 270-degree turn. This allows us to add 1 to the r-value each time we want to rotate the Tetris piece – and not worry about resetting it to 0/having to deal with boundary checks when it reaches 3 etc.

Go ahead and copy this function below the void loop

Creating the board

One last asset we need to create is the board. To create the board we are going to need to declare some variables first.

```
3  //Playing Field
4  int nFieldWidth = 12;
5  int nFieldHeight = 18;
6  unsigned char pField[216];
```

nFieldWidth declared the width of the playing field

nFieldHeight declares the height of the playing field

pField is an empty array of unsigned chars that will be used to store the fields data.

Below is a preview of what the pField will store:

```
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000001009
900000001009
900000001009
900000001009
999999999999
```

```
6  unsigned char
```

The reason we are using unsigned chars for this array is to save memory space. A char, stores values from -128 to 127 using 1 byte of data. Signed chars store values from 0 to 255 in 1 byte of data.

When you think of chars you usually think of characters such as 'A' or 'a' but keep in mind that every character is represented as a denary value in either ASCII, extended ASCII or Unicode etc.

The reason we do not use integers, in this case, is because integers use 2 to 4 bytes of data to store a much larger range of values. As we will only be using values from 0 to 9, we are saving memory space by using chars.

This is common in 8bit programming to save memory and make the program more efficient.

The array will have 1 element for 1 pixel/brick for our game. The width * height of the field brings us to (12*18=) 216 elements/bricks.

Now that we have our variables and pField array declared. We want to fill the empty array however we want to fill It in a particular way to create a board. If the element is on the bottom, left or right edge, then we want to set it to a 9 as it is a boundary if it isn't then we want to set the element to 0 as it is empty space. The array will look like the previous picture

To do this we will use nested for loops to loop through the x and y values to set each element in the array to either 0 or 9:

```
19    for (int x = 0; x < nFieldWidth; x++) // Board Boundary
20      for (int y = 0; y < nFieldHeight; y++)
21        pField[y * nFieldWidth + x] = (x == 0 || x == nFieldWidth - 1 || y == nFieldHeight - 1) ? 9 : 0;
```

The first for statement loops through the width and will run the next for statement 12 times. This next for statement iterates through the height, running **line 21** 18 times, however as these are nested for loops, **line 21** will run (12*18=) **216 times** meaning each **element** in the **array will be assigned a value.**

Let's take a look at line 21:

```
pFieldLy * nFieldWidth + x] = (x == 0 || x == nFieldWidth - 1 || y == nFieldHeight - 1) ? 9 : 0;
```

This line can actually be expanded into the below code:

```
if (x == 0 || x == nFieldWidth - 1 || y == nFieldHeight - 1){
    pField[y * nFieldWidth + x] = 9
}
else{
    pField[y * nFieldWidth + x] = 0
}
```

```
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
999999999999
```

The above code reads as:

If the x value is equal to 0 = For the left side of border of the field

OR the x value is equal to 11 (nFieldWidth - 1) = For the right side of the border of the field

OR the y value is equal to 17 (nFieldHeight - 1) = For the bottom border of the field

Then set the value to 9. If none of these is true then set the value to 0

This will create something that looks like the image to the left. Where 9 represents the game border.

Above is an example of a ternary operator. It is used to replace if/else statements that take the below form:

```
if(condition) {
    var = X;
} else {
    var = Y;
}
```

Where the condition decides if the variable will be one value or another. In our case, it's either 0 or 9. We can write it in one line by using the below format:

Variable = (Condition) ? Option1 : Option 2;

If the condition is true, Option1 is assigned to the Variable. If the condition is false, then Option2 is assigned to the Variable.

Let's take another look at line 21 and identify what our variable is:

```
pFieldLy * nFieldWidth + x] = (x == 0 || x == nFieldWidth - 1 || y == nFieldHeight - 1) ? 9 : 0;
```

The variable part of the code is:

*pField[y * nFieldWidth + x]* Here we are using the formula we discussed earlier to get the index position in the one-dimensional array from two coordinates.

Go ahead and copy this into your void setup()

Lets start outputting some things

Now that we have a board, its worth outputting it to make sure everything is how it should be. We can use the Arduino Serial Monitor to do this however we are going to need to add some extra code.

To setup the serial monitor we need to add:

```
 8 void setup() {
 9    Serial.begin(9600);
10
11    //Creating the pieces
12    tetromino[0] = "..X...X...X...X.'
```

This will allow us to open up Serial monitor and view the data being sent from the Arduino to your computer. And prepares both your Arduino and computer to exchange data at a rate of 9600 bits per second.

Place this at the start of void setup()

And we need to create a function that can interrupt and output the field:

```
58 void OutputField() {
59    for (int i = 0; i < nFieldWidth * nFieldHeight; i++) {
60       if (i % nFieldWidth == nFieldWidth - 1) {
61          Serial.println(pField[i]);
62       }
63       else {
64          Serial.print(pField[i]);
65       }
66    }
67    Serial.println("----");
68 }
```

Add this underneath the void loop()

Here we loop through the entire pField array (nFieldWidth * nFieldHeight) and output the data. However, if we were to just output the data using Serial.print(), it would be on the same line, so we need to use the Serial.println() to print the next data on a new line every time we reach the end of a row.

To do this we use an if statement to see if the current index (i) has reached the end of the row (nFieldWidth).

Here we use i modular nFieldWidth to see if the remainder is 11 (nFieldWidth – 1). If the remainder is 11 then the next print needs to be on a new line as all 12 elements have been printed (as arrays are indexed from 0) and therefore the next set of elements are in the next row.

To output the data on the next row we use Serial.println(). This moves the cursor to the next line and then outputs the data. Whereas Serial.print() just outputs the data wherever the cursor is (inline).

[Line 67] Serial.println("----"); is used to help us differentiate between each Field

Game Loops

These are the most important aspect of a game engine. This sequences everything that is going on in your game. Tetris will not have an overly complex game loop. It will have aspects such as updating the game logic, handling time, getting the user input and displaying it to the screen.

Let's create our game loop:

Firstly, we need to add a simple bool variable to store if the game is over or not:

```
6  unsigned char pField[216];
7
8  bool bGameOver = false;
9
10 void setup() {
11    Serial.begin(9600);
```

And then we need to add a simple while loop that will only run if bGameOver is false:

```
34 void loop() {
35   while (!bGameOver) // Main Loop
36   {
37
38   }
39
40 }
```

This means, if bGameOver is True, then the game loop won't run and the game will be over as the variable is inverted meaning the condition will return false. However if the variable is False (the game isn't over), then the variable Is inverted and the condition returns true meaning the while loop will iterate.

Be sure to add these blocks of code. Use the line numbers as a reference.

Drawing the Frame

Before we start implementing features, we want to see what we are working with. To do this we will add some code that will print the game field to the serial monitor. For now, we will be using this to watch our game evolve. Using Serial Monitor means we can focus on the game engine and then the graphics engine afterwards.

Currently, we are outputting the field. The field is the numerical representation that will be used to show the boundary and 'locked-in' pieces – e.g. pieces that have been previously placed. This won't show actively moving pieces (aka the tetrominos that the player is currently controlling).

Now we are going to output the Frame. This will look more like the final version of the game and will be used in conjunction with the graphics engine.

The first thing we need to add is the translation of the pField to the output. This will convert all values in the pField into the various characters:

Firstly we must define the output array with 216 elements

```
6  unsigned char pField[216];
7  char output[216];
8
```

Now we can jump into the game loop and start iterating through the pField to convert each character:

```
26    while (!bGameOver) // Main Loop
27    {
28      // Draw Field
29      for (int x = 0; x < nFieldWidth; x++)
30      {
31        for (int y = 0; y < nFieldHeight; y++)
32        {
33          output[(y * nFieldWidth) + x] = " ABCDEFG=#"[pField[(y * nFieldWidth) + x]];
34        }
35      }
36
37      OutputField();
```

The first two for statements should be familiar from when we create the pField array. These simply loop through the x and y coordinates meaning line 33 will run 216 times.

```
output[(y * nFieldWidth) + x] = " ABCDEFG=#"[pField[(y * nFieldWidth) + x]];
```

Here we assign output[] with the index position, using the formula we are familiar with, to either " ABCDEFG=#" depending on the index position of the string.

> For those that aren't familiar with this, you can treat a string like an array of characters and access them individually like an array. E.g.
>
> "Hello"[1] returns 'e' as the letter with index position 1 is the second character. In this case, it is an e.

Here we are converting the pField value into a character. **pField[(y * nFieldWidth) + x]** also uses the formula we are familiar with and gets the index position of the array from the two coordinates generated from the for statements. The value returned, will be used as the index position for the string.

In this case, the 9's that we defined as the boundary are being converted to a '#' – for a more defined outline. And the 0's are being set as ' ' empty space.

If we had tetromino pieces such as 1, 2,3 etc then they would be converted to A,B,C etc

Go ahead and add this code to the game loop.

The next thing we need to add is a very similar function to OutputField(). But this time we want to output the 'output' function.

```
83
84  void OutputFrame() {
85    for (int i = 0; i < nFieldWidth * nFieldHeight; i++) {
86      if (i % nFieldWidth == nFieldWidth - 1) {
87        Serial.println(output[i]);
88      }
89      else {
90        Serial.print(output[i]);
91        //Serial.print( (uint8_t) output[i] );
92      }
93    }
94
95    Serial.println("----");
96  }
```

[Line 85] As explained before, the for statements loops 216 times, accessing all the elements in the array.

[Line 86] The if statements checks if it is the end of the row – if it then it outputs the data on a new line [Line 87]. If it isn't then output the data in line [Line 90].
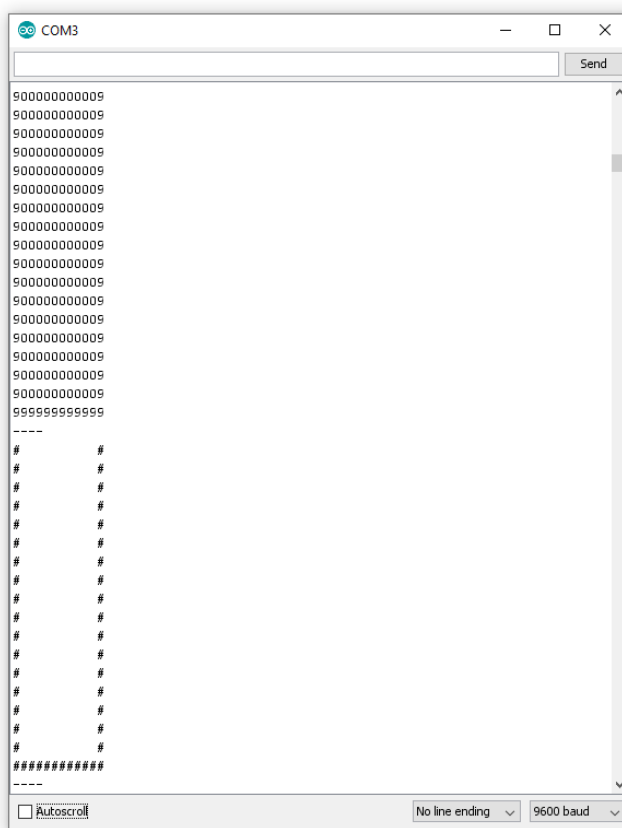
[Line 95] Then, once the whole frame has been outputted, print "----" to differentiate between the next print.

Add this function after the function OutputField()

Now we need to call the function in the game loop.

```
38        OutputField();
39        OutputFrame();
40    }
41 }
```

Let's upload this and take a look at the serial monitor

Now we can see both the field and frame being outputted.

The difference between the two will be showcased more clearly when we start adding in the tetrominos!

4 stages of Game Dev

The 4 stages of most computer games are as follow:

1.   Game timing
2.   Input
3.   Game Logic
4.   Render Output

These loop to create the game.

Let's go ahead and begin to tackle some of these stages in the game loop:

Collision

Tetris is quite simple when it comes to collision detection. This is because Tetris is made up of blocks and the blocks move one whole block at a time. Meaning before we move the piece, we want to check if the piece can be moved. If that current block is occupied – then we cannot move there.

To do this, before we make the final move, we see if the move is possible by checking ahead.
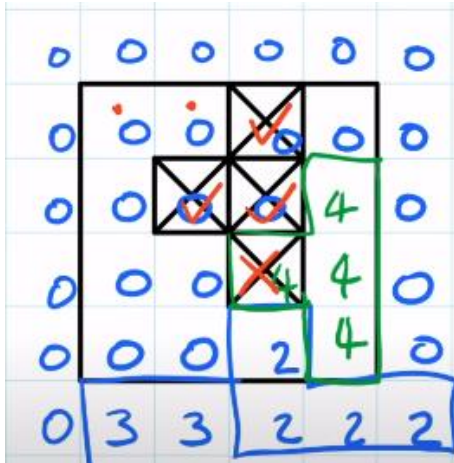
Here you can see how we would check this. Remember when we used dots and x's to create the tetrominos? Well, as dots represent empty space, we ignore them, however, if an X is overlapping with a value that is anything other then 0, then we return that the movement is invalid as there is a collision.

We also need to check it isn't going out of bounds.

As we are going to be checking this a lot, we are going to make a function called "DoesPieceFit"

```
43 bool DoesPieceFit(int nTetromino, int nRotation, int nPosX, int nPosY)
44 {
45   // All Field cells >0 are occupied
46   for (int px = 0; px < 4; px++)
47     for (int py = 0; py < 4; py++)
48     {
49       // Get index into piece
50       int pi = Rotate(px, py, nRotation);
51
52       // Get index into field
53       int fi = (nPosY + py) * nFieldWidth + (nPosX + px);
54
55       // Check that test is in bounds. Note out of bounds does
56       // not necessarily mean a fail, as the long vertical piece
57       // can have cells that lie outside the boundary, so we'll
58       // just ignore them
59       if (nPosX + px >= 0 && nPosX + px < nFieldWidth)
60       {
61         if (nPosY + py >= 0 && nPosY + py < nFieldHeight)
62         {
63           // In Bounds so do collision check
64           if (tetromino[nTetromino][pi] != '.' && pField[fi] != 0)
65             return false; // fail on first hit
66         }
67       }
68     }
69
70   return true;
71 }
```

Add this block of code after the void loop()

The first thing to notice is the parameters. We are going to need to know the ID of the Tetromino (0 to 6), what rotation it is in and where it is on the board (the x and y position).

Also note on line 246 that we, by default, will return true. Aka, we are only looking for situations where the piece cannot fit due to collision.

[Lines 46 & 47] The first two for statements iterate the tetrominos. This means we are checking every single block inside the tetrominos we created before (with the X's and dots).

LEARN

[Line 50] Next we get the index position of the piece (called pi) from the rotate function we created earlier. We do this to take into account any rotations that have occurred as the rotate function will return the updated index position of the piece in its rotated form.

[Line 53] We get the index position of the field (stored in fi) – this will be the block that the piece is 'overlapping' with. This follows the formula from beforehand: y * width + x
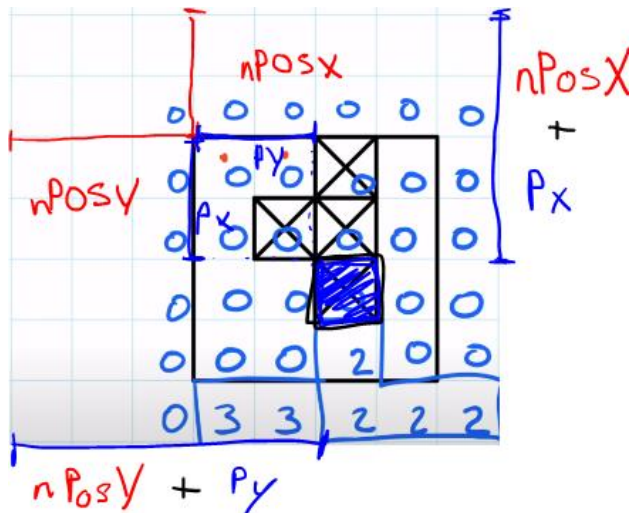
To get the y coordinate of the tetrominos piece in the field, we must add the nPosY, which will give us the position of the top left corner of the tetromino. We then add the py value to get the current position of the block in question in the tetromino.

We then multiply this with the width to follow the form of our formula.

And use a similar calculation to find the x value. Add nPosX to get the top left corner of the tetrominos, then add pX to get the tetrominos coordinate.

This will return the index position of where the block in question would be on the field.

[Lines 59 to 61] Check to ensure that we do not go out of bounds with our checks.

If the total of nPosx + px is equal to or greater than 0, then we are in range. If it is also less ten the nFieldWidth, then we are within range.

The next if statement does a similar check but with the y coordinates:

If the total of nPosY + py is equal to or greater than 0, then we are in range. If it is also less than nFieldHeight, then we are in range.

[Line 64] This is the actual collision check. Here we take the tetromino and check the value of its index position:

tetrominos[nTetromino][pi] will return the char at that index position, either 'X' or '.'

e.g. if nTetromino was 0, it would return '…X….X….X….X.'[pi] In which we are the indexing the string. It might look like a 2-dimensional array but the second index is for the string position.

If it is == to 'X' and the pField value is not equal to 0, if it is not empty space, then return false as there is a collision.

And that's it! Be sure to add this block of code after the void loop()

Testing Collision

Now that we have added collision, let's test it! We need to create and render our first tetromino piece.

Firstly, we have to add some variables for the tetrominos to be created:

```
 8 |
 9 int nCurrentPiece = 0;
10 int nCurrentRotation = 0;
11 int nCurrentX = nFieldWidth / 2;
12 int nCurrentY = 0;
13
14 void setup() {
```

nCurrentPiece = an integer value between 0 and 6 that defines what tetromino piece is currently active.

nCurrentRotation = an integer that stores the current rotation, this will be used with the rotate function

nCurrentX = nFieldWidth/2 = forces the start piece to be in the center.

nCurrentY = 0 = Sets the current y value of the active tetromino to 0 (at the top).

Be sure to add this code before the void setup()

We need to draw the active tetromino separate to the field as it isn't yet apart of the field. Anything apart of the field is 'locked in' and will be used to detect for a collision.

To do this, we use the output array (frame) that we created earlier. This is the key difference between the 'output' and the 'pField' array as the output will have the active and user-controlled tetromino piece.

```
44
45      // Draw Current Piece
46      for (int px = 0; px < 4; px++) {
47        for (int py = 0; py < 4; py++) {
48          if (tetromino[nCurrentPiece][Rotate(px, py, nCurrentRotation)] != '.')
49          {
50            output[(nCurrentY + py)*nFieldWidth + (nCurrentX + px)] = nCurrentPiece + 65;
51          }
52        }
53      }
54
55      OutputField();
```

Be sure to add this code in before the game loop, but after the 'Draw Field' loop. See line numbers for reference.

[Lines 46 & 47] These two for statements loop through the tetromino array. Each for statement loops through 4 times each as the tetromino pieces are 4 by 4.

[Lines 48] This if statement checks if the current tetromino piece is an X, if it Is, then it will run lines 50. Let's take a more detailed look at how we do this:

tetromino[nCurrentPiece][Rotate(px,py,nCurrentRotation] = Gets the current tetromino. To get it we need two variables: The current piece (nCurrentPiece, a value between 0 and 6) and the rotation of this piece, using the Rotate function with the px and py variables from the for loop and the current rotation variable. This Rotate function returns the index value.

… != '.' If this value is not equal to '.' (a dot) then run line 50,

[Lines 50] Sets the element with the same x and y position in the output array, to a character. This depends on what piece is active, e.g. piece 0 which is the blue long straight piece will set the block to 'A'. Let's take a look at how we do this.
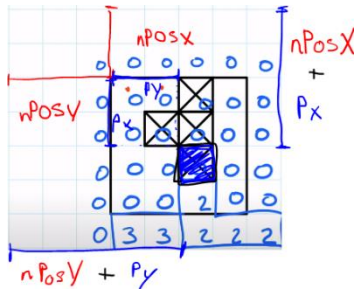
output[(nCurrentY + py) * nFieldWidth + (nCurrentX + px)] = Returns the index position of the piece that needs to be returned. This is similar to how we got the index value in the collision function:

Where nPosX/nPosY (in the diagram) are equal to nCurrentX/nCurrentY in our code

These are the coordinates of the top-left point of the tetromino.

The py and px are the coordinates of the block we are checking/displaying. By adding these together we can get the overall coordinates of the active block

nCurrentPiece + 65 = nCurrentPiece will be an integer value from 0 to 6. For example, if the nCurrentPiece is 0, 65 + 0 = 65. By adding 65 to it, we get the ASCII value for A. If the value is 1, then when we add 65 to the value and get 66, the ASCII value for B. This allows us to differentiate between the pieces and later use this in our graphics engine to render various colours for the various pieces.

As we used the two for loops, we are essentially going through every single block in the tetromino and comparing it. If it is an X then we set the element/block in the output array as a certain character.

Now lets go ahead and upload this to our 8BitCADE XL and see what we get in Serial:
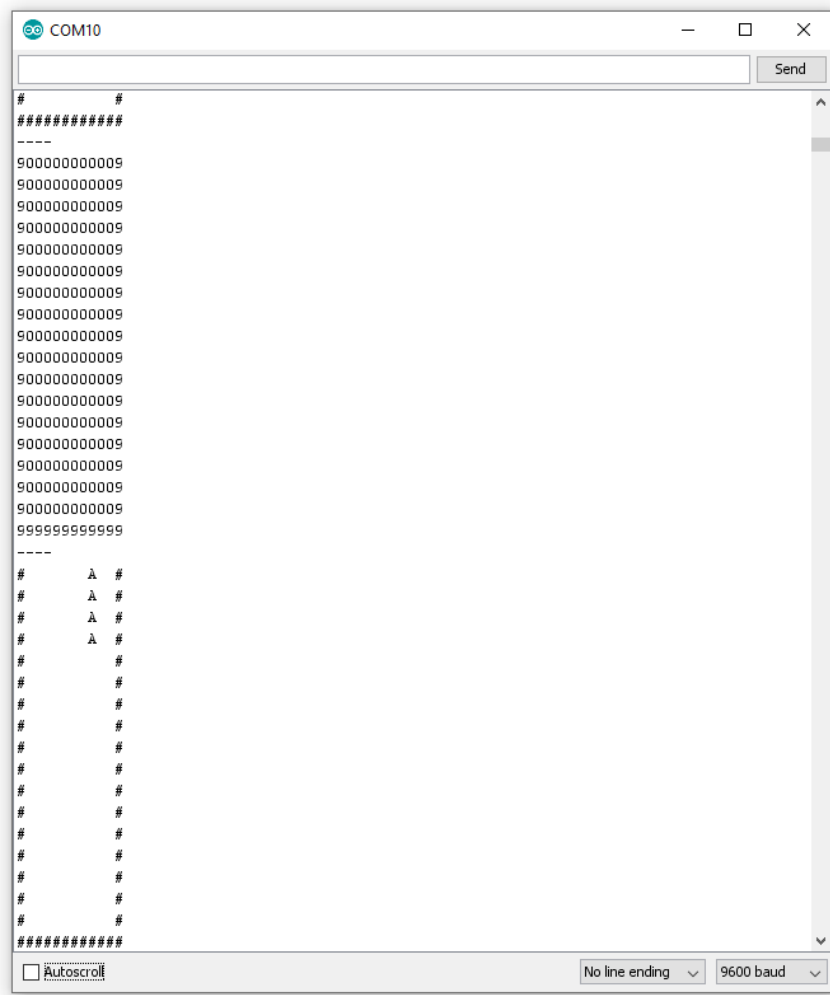
Here you can see the clear difference between the Field (first output [A]) and the Frame/output array (second output [B]).

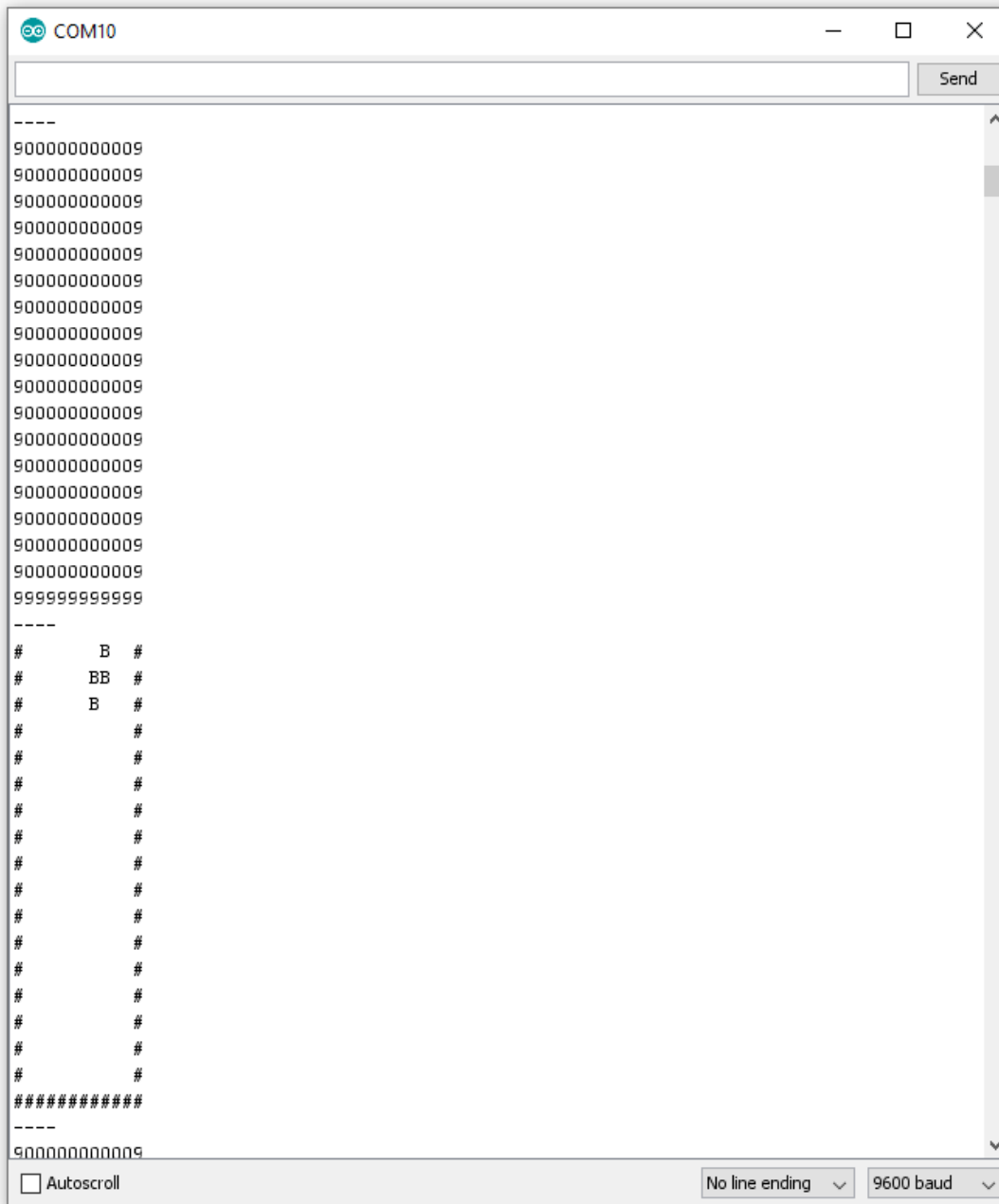The frame holds the active piece.

The field holds everything that is 'set'/'locked in'. The frame is also a lot more like the final game version.

We could actually play Tetris in the serial monitor if we wanted to!

Before we move on, we just want to do a quick check to make sure everything is in place. Go ahead and change the nCurrentPiece variable to say 1.

```
 9  int nCurrentPiece = 1;
10  int nCurrentRotation = 0;
11  int nCurrentX = nFieldWidth / 2;
12  int nCurrentY = 0;
```

This should change the active piece to a diagonal piece.

```
tetromino[1] = "..X..XX..X......";
```

Go ahead and upload this and check the serial monitor:

```
COM10                                          —    □    ×
[                                              ] [ Send ]
----
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
999999999999
----
#         B   #
#         BB  #
#         B   #
#             #
#             #
#             #
#             #
#             #
#             #
#             #
#             #
#             #
#             #
#             #
#             #
#             #
#             #
############
----
900000000009
☐ Autoscroll            [No line ending ∨] [9600 baud ∨]
```

Here you can see two changes:

1. The shape has changed
2. The values representing the piece have changed.

If your serial monitor does not look like this, please take a read back in the tutorial/check the final code at the bottom of the tutorial to see if you have any logical errors.

17

Go ahead and change the value back from 1 to 0.

If we want to properly test the collosion we are going to need to add two aspecsts of the game loop:

- Game Timing
- User Input

Lets go ahead and add game timing!

Game Timing

When playing tetris, the tetrominos move, based on the user input or simply being pushed down, after a given period of time. This is due to the implementation of game time. This allows us to control the speed of the game meaning we can also control the difficulty of the game.

In our game we have a common tick. Where the game progresses/moves on the tick. This is why, when we paly Tetris, we get that lagged effect where the game progresses after a set period of time.

We will setup the variables and counting now and make some adjustments later.

```
14  int nSpeed = 10;
15  int nSpeedCount = 0;
16  bool bForceDown = false;
```

nSpeed = Game speed. This will be compared against the nSpeedCount.

nSpeedCount = will increase every game loop by 1.

vforceDown = Dictates when we are moving the tetromino (giving us the lage effect). If this is True then the game progresses, if this is false then the game does not. We will be using this more later on in the tutorial when we implement the 'force down' feature that forces the tetromino down the playing field.

Add these variables above the void setup()

```
35  void loop() {
36      while (!bGameOver) // Main Loop
37      {
38          // Timing ==========================================
39          nSpeedCount++;
40          delay(nSpeed);
41          bForceDown = (nSpeedCount == nSpeed);
42          // Input ===========================================
43
44          // Draw Field
```

Every game loop we will be adding one to the nSpeedCount. When this reaches the nSpeed, aka 10, then we set the bForceDown variable to True. Later in the tutorial we will use an if statement with this variable. This variable dictates if we are going to move the tetromino or not.

[Line 39] Adds 1 to the nSpeedCount. This can be written as nSpeedCount = nSpeedCount + 1

[Line 40] Adds a simple delay using the nSpeed, otherwise the game is too fast. The difficulty increases when this delay is decreases.

[Line 41] Here we have another 'inline if statement'. We are using a comparative operator '==' that checks if nSpeedCount is equal to nSpeed.

(nSpeedCount == nSpeed) will return true if they are equal to each other. Aka if nSpeedCount is equal to 10. If not, then this statement will return false

As the variable bForcedown is a Boolean, we are assigning the result of this comparison to the variable.

Add this at the start of your game loop. See line numbers for a general guide, these may vary to what you have!

User Input

When playing tetris, we give the user 4 basic controls. Left, right, down and rotate. Now that we have a tetromino on the playing field, we need to give the user some sort of control over the piece.

To do this we are going to create a function called CheckControls();

Before we create this function, we are going to need to define the variables we are going to use:

```
30 float lastHoldTime = 0;
31 bool bKey[4] = {0, 0, 0, 0};
32 bool bRotateHold = true;
```

[Line 30] Creates a float to hold the 'lastHoldTime' of a button. We will use this to reduce button debounce (we will look into this in a bit). This is a float as it will hold milliseconds. 1 second is 1000 milliseconds therefore we need a data type that has a large range as this will hold the last time a button was pressed. If the user doesn't press a button for 5 seconds then the float will be 5000!

[Line 31] bKey[] Is an array that will hold the final button outputs. Its Boolean as a button can either be on or off.

[Line 32] bRotateHold. To stop the tetromino from violently spinning when we press the button, we need to detect when the button is being held. We will use an if statement to only rotate the tetromino once if the button is being held. This way the user has more control over the rotations – one press = one rotation, regardless of how long you hold the button down for.

Be sure to add this bit of code above the void setup()

Using Arduboy

Now we is a good time to add the Arduboy2 Library that will allow us to use the buttons. Lets add this library to the top of our code

```
1 #include <Arduboy2.h>
2 Arduboy2 aboy;
```

Next we need to initialize it in void setup()

```
37 void setup() {
38   Serial.begin(9600);
39
40   aboy.boot();
41   aboy.clear();
42
43   pinMode(PIN SPEAKER 1, OUTP
```

[Line 40] Initializes the aboy library

[Line 41] Clears the screen/screen buffer and sets he cursor to position 0,0. We will get into this more later.

Now that we have defined the variables and setup Arduboy2 Lib, we can jump into creating the function:

```
void CheckControls() []
  bKey[0] = (aboy.pressed(RIGHT_BUTTON) && (millis() - lastHoldTime) > 100) ? 1 : 0;
  bKey[1] = (aboy.pressed(LEFT_BUTTON) && (millis() - lastHoldTime) > 100) ? 1 : 0;
  bKey[2] = (aboy.pressed(DOWN_BUTTON) && (millis() - lastHoldTime) > 100) ? 1 : 0;
  bKey[3] = aboy.pressed(A_BUTTON);

  //Simple Buffer
  int Sum = 0;
  for (int i = 0; i < 4; i++) {
    Sum = Sum + bKey[i];
  }
  if (Sum != 0) {
    lastHoldTime = millis();
  }

  // Handle player movement
  nCurrentX += (bKey[0] && DoesPieceFit(nCurrentPiece, nCurrentRotation, nCurrentX + 1, nCurrentY)) ? 1 : 0;
  nCurrentX -= (bKey[1] && DoesPieceFit(nCurrentPiece, nCurrentRotation, nCurrentX - 1, nCurrentY)) ? 1 : 0;
  nCurrentY += (bKey[2] && DoesPieceFit(nCurrentPiece, nCurrentRotation, nCurrentX, nCurrentY + 1)) ? 1 : 0;

  // Rotate, but latch to stop wild spinning
  if (bKey[3])
  {
    nCurrentRotation += (bRotateHold && DoesPieceFit(nCurrentPiece, nCurrentRotation + 1, nCurrentX, nCurrentY)) ? 1 : 0;
    bRotateHold = false;
  }
  else
    bRotateHold = true;
}
```

As this is quite a long function with many pieces, we will go through it section but section. The first part is to get the user inputs:

```
bKey[0] = (aboy.pressed(RIGHT_BUTTON) && (millis() - lastHoldTime) > 100) ? 1 : 0;
bKey[1] = (aboy.pressed(LEFT_BUTTON) && (millis() - lastHoldTime) > 100) ? 1 : 0;
bKey[2] = (aboy.pressed(DOWN_BUTTON) && (millis() - lastHoldTime) > 100) ? 1 : 0;
bKey[3] = aboy.pressed(A_BUTTON);
```

Look familiar? We are using an inline if statement or ternary operator again!

Here we are assigning each element in the array bKey to the result of a user input. Element 0 in the array stores the right button result, element 1 stores the left button result, etc.

So what does this part do?

```
bKey[0] = (aboy.pressed(RIGHT_BUTTON) && (millis() - lastHoldTime) > 100) ? 1 : 0;
```

This stops button debounce.

Debounce: What is it

When we click a button once, currently it registers as 5 or 6 clicks. This is called ButtonDebounce.

The reason the Arduino reads it as 5 or 6 presses is because when we click the button down, the Arduino is fast enough to read the signal multiple times. To remove this "bounce", and read one press as a single press, we need to add "button debounce"

To achieve this, we Ignore all inputs for 100 milliseconds after we register a click.
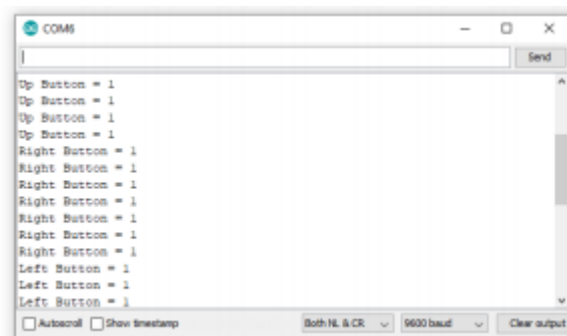
If we take a look at our line of code again, we can see that we are using millis() to detect when

Gets RIGHT_BUTTON button input

```
bKey[0] = (aboy.pressed(RIGHT_BUTTON) && (millis() - lastHoldTime) > 100) ? 1 : 0;
```

This acts as a delay that doesn't interrupt the program – the delay(); command stops the program for the specified amount of time. This ignores all inputs for 100 milliseconds, removing the debounce. Current time (millis()) minus last time pressed (lastHoldTime()) must be greater than 100. Aka ignore all inputs that have been pressed in the last 100 milliseconds.

The millis() function will returns the amount of milliseconds that have passed from when the program started. We can use this as a time stamp.

Note that we use the '!' NOT logic operator to inverse the result of the digitalRead(). We do this as we set the pinmode earlier as INPUT_PULLUP meaning when the butoton is pressed, the value is 0. By adding the '!' / not operator, we are inverting the result back meaning that when the button is pressed, 0 will be inverted to 1.

So when the button is pressed, and when we have waited 100 miliseconds from the last button press, set bKey[0] to 1. If not, set it to 0.

This line can also be written as:

```
if(aboy.pressed(RIGHT_BUTTON) && (millis() - lastHoldTime) > 100) {
    bKey[0] = 1;
}
else{
    bKey[0] = 0;
}
```

Note that we are going to need to update the lastHoldTime when a button is pressed. To do this, we have to take a look at the next section of code

```
176    //Simple Buffer
177    int Sum = 0;
178    for (int i = 0; i < 4; i++) {
179        Sum = Sum + bKey[i];
180    }
181    if (Sum != 0) {
182        lastHoldTime = millis();
183    }
```

Here we create a variable called Sum.

We use a for statement to loop through the bKey[] array. Every iteration, we add the bKey[] value to the sum. If a button is pressed, the sum will be 1, if a no buttons are pressed, then the value will be 0.

We then use a simple if statement to check the sum, if it is not equal to 0, then update the lastHoldTime with the current time

(millis())

Notice how the rotate key/a button, does not have the debounce code. This is because we want the down key to be much quicker then the rest of the buttons as the player is forcing the part

LEARN

down. When we upload the code and test out the controls, you will see the difference between the left, right and rotate buttons compared to the down button – it is much faster.

```
bKey[3] = aboy.pressed(A_BUTTON);
```

Now that we know what buttons the user has pressed; we need to actually move the tetromino. To do this we need to edit the tetrominos position etc.

```
185    // Handle player movement
186    nCurrentX += (bKey[0] && DoesPieceFit(nCurrentPiece, nCurrentRotation, nCurrentX + 1, nCurrentY)) ? 1 : 0;
187    nCurrentX -= (bKey[1] && DoesPieceFit(nCurrentPiece, nCurrentRotation, nCurrentX - 1, nCurrentY)) ? 1 : 0;
188    nCurrentY += (bKey[2] && DoesPieceFit(nCurrentPiece, nCurrentRotation, nCurrentX, nCurrentY + 1)) ? 1 : 0;
```

Here you can see that we also have a tertiary operator. The overall statement is saying:

If the button has been pressed and the piece fits the position we are moving it to (no collision), add 1 to the current x or y position.

Let's break down these lines:

```
186    nCurrentX += (bKey[0] && DoesPieceFit(nCurrentPiece, nCurrentRotation, nCurrentX + 1, nCurrentY)) ? 1 : 0;
```

Can be written as:

```
if (bKey[0] && DoesPieceFit(nCurrentPiece, nCurrentRotation, nCurrentX + 1, nCurrentY)) {
  nCurrentX += 1
}
else {
  nCurrentX += 0
}
```

bKey[0] returns either 1 or 0. 1 if the button has been pressed or 0 if it hasn't. For conditions, 1 can also be classed as True.

Next we called our collision detection function 'DoesPieceFit()'. This function requires the below parameters:

nCurrentPiece = The current tetromino. A value from 0 to 6

nCurrentRotation = The current rotation of the tetromino so we know how to index the tetromino

nCurrentX + 1 = Is where the tetromino is trying to move. As this is for the bKey[0] which is the right button, we are trying to shift the tetromino one block to the right therefore we add one to the current x position to get the 'future' position/where we want to move it

nCurrentY = As there is no change in the y coordinates, we do not add or subtract the coordinate.

This will return either true or false.

As we are using an and operator between the two statements, if both statements are true, then 1 is returned. This means the line would be:

nCurrentX += 1;

However, if the key hasn't been pressed or the piece does not fit, then the part will not move there as we are adding 0 to the nCurrentX. The line would read as:

nCurrentX += 0;

This format follows through for the next 2 statements. However:

```
185    // Handle player movement
186    nCurrentX += (bKey[0] && DoesPieceFit(nCurrentPiece, nCurrentRotation, nCurrentX + 1, nCurrentY)) ? 1 : 0;
187    nCurrentX -= (bKey[1] && DoesPieceFit(nCurrentPiece, nCurrentRotation, nCurrentX - 1, nCurrentY)) ? 1 : 0;
188    nCurrentY += (bKey[2] && DoesPieceFit(nCurrentPiece, nCurrentRotation, nCurrentX, nCurrentY + 1)) ? 1 : 0;
```

[Line 187] As it is gor bKey[1] or the Left button, we use '-=' instead of '+=' to minus 1 from the coordinate if the piece can fit there.

[Line 188] Here we are editing the y coordinates. It's the same principal as the above lines however we edit the Y coordinates. If the down button is pressed and the piece does not collide with the boundaries/other pieces, then move the piece down (add 1 to the coordinate).

The final part of the controls Is to rotate the piece:

```
190    // Rotate, but latch to stop wild spinning
191    if (bKey[3])
192    {
193      nCurrentRotation += (bRotateHold && DoesPieceFit(nCurrentPiece, nCurrentRotation + 1, nCurrentX, nCurrentY)) ? 1 : 0;
194      bRotateHold = false;
195    }
196    else
197      bRotateHold = true;
198 }
```

[Line 191] If the rotation button, or y button, has been pressed then:

1. [Line 193] Check if the rotated piece does not collide with any boundaries/pieces.

```
193      nCurrentRotation += (bRotateHold && DoesPieceFit(nCurrentPiece, nCurrentRotation + 1, nCurrentX, nCurrentY)) ? 1 : 0;
```

We use the "DoesPieceFit()" function to see, when we rotate the tetromino, if there are any collisions.

This is similar to what we just looked at with the nCurrentX and nCurrentY variables however instead of adjusting the coordinates we are adjusting the rotation, seeing if there are any collisions and then returning the result.

If we look at the parameters, we are adding 1 to the nCurrentRotation. If, when we rotate it in that orientation, there are no collisions, we return true.

2. Check if bRotateHold is true, if the button is being held down then this will be false (due to line 194) and means, due to the and statement, it wont rotate. Lets look at this in more detail below:

Dealing with debounce/overspinning the Tetris piece.

To ensure the user gets full control over the tetromino piece, it makes sense to have one button press per rotation. To do this we use the bRotateHold variable.

When the button is released, the bKey[3] is false therefore the 'else' section of the if statement runs and the bRotateHold is set to true again, allowing for another rotation to take place. If the button was being held down then:

1. On the first iteration, the nCurrentRotation would add 1 (assuming there is no collision) and the bRotationHold would be set to false
2. On the second iteration, with the button still being held down, even if the piece can be rotated with no collision it will not rotate as the bRotateHold variable is set to false. The only way the piece can be rotated is if the bRotateHold is True. The only way to do that, is if the button is released and the else statement is ran meaning the bRotateHold is set back to True.

Therefore, going back to line 193,

```
193      nCurrentRotation += (bRotateHold && DoesPieceFit(nCurrentPiece, nCurrentRotation + 1, nCurrentX, nCurrentY)) ? 1 : 0;
```

If both conditions are true, then the statement reads:

nCurrentRotation += 1;

Meaning the piece is rotated

If any condition is false. AKA the piece does not fit or the button is being held down, the statements readS:

nCurrentRotation += 0;

There is no change meaning the piece isn't rotated.

Add this function underneath your void loop() and make sure to call it in your game loop like so:

```
55      // Timing ==========================================
56      nSpeedCount++;
57      bForceDown = (nSpeedCount == nSpeed);
58      // Input ===========================================
59      CheckControls();
60
61      // Draw Field
62      for (int x = 0; x < nFieldWidth; x++)
```

Now go ahead and run the code. You should be able to see your tetromino moving in the serial monitor as you press the buttons on your console. Your console, at the moment, is acting like a controller!

Here you can see some screenshots from me just moving around the start piece and rotating it.

Note that: As we are printing every single element multiple times, it is very slow. Which is why you have to press a button for awhile before it moves. When we add the graphics engine and begin to render it on screen, you will get a responsive, Tetris like, game! Hang in there!

Adding the force down feature

If you have played Tetris, you know that the piece gets forced down after a set time period. Currently, our pieces are static and aren't forced down. Lets go ahead and add this feature:

24

```
52  void loop() {
53    while (!bGameOver) // Main Loop
54    {
55      // Timing =================================================
56      nSpeedCount++;
57      bForceDown = (nSpeedCount == nSpeed);
58      // Input ==================================================
59      CheckControls();
60
61      // Force the piece down the playfield if it's time
62      if (bForceDown)
63      {
64        // Test if piece can be moved down
65        if (DoesPieceFit(nCurrentPiece, nCurrentRotation, nCurrentX, nCurrentY + 1)) {
66          nCurrentY++; // It can, so do it!
67        }
68        else
69        {
70          // It can't! Lock the piece in place
71
72          // Check for lines
73
74          // Pick New Piece
75
76          // If piece does not fit straight away, game over!
77        }
78      }
79
```

Here we can see the game starting to come together, lets take a look at what we need to add:

[Line 62] This is that if statement I mentioned beforehand. If the piece is meant to fall down, then run the code, if it is not meant to fall down, skip over this and go straight to drawing the field/current piece. This is because we want the piece to fall down at a certain time. To get this time, we go back to the code we wrote before hand on lines 56 and 57.

[Line 56] Each game loop we add 1 to the nSpeedCount.

[Line 57] When the game loop has ran nSpeed amount of times (10) then we can set bForceDown to true. We do this by using the equal to operator that compares the two values and returns if it is true or not.

If nSpeedCount is equal to nSpeed, aka if the game loop has iterated 10 times, the statement will read like this:

bForceDown = True

If nSpeedcount is not equal to nSpeed, aka if the game loop has not tierated 10 times, the statement will read like this:

bForceDown = False

By doing this, we are forcing the tetromino down every 10 game loops.

Going back to Line 62], now, if the game loop has ran 10 times we can run this if statement and move the piece down. However, we need to firstly check if the piece can actually be moved down:

```
64        // Test if piece can be moved down
65        if (DoesPieceFit(nCurrentPiece, nCurrentRotation, nCurrentX, nCurrentY + 1)) {
66          nCurrentY++; // It can, so do it!
67        }
```

Using the DoesPieceFit() function, we simply check if the piece can be moved down. We do this by adding 1 to the nCurrentY to see if, if the piece were in that position, would it collide with any boundaries/any other pieces? If it doesn't collide, then we can go ahead and make that change to the nCurrentY in line 66.

If it doesn't then we need to do a whole bunch of checks

```
68        else
69        {
70          // It can't! Lock the piece in place
71
72          // Check for lines
73
74          // Pick New Piece
75
76          // If piece does not fit straight away, game over!
77        }
```

- If the piece cant be moved down anymore, that means it has reached the bottom and is now ready to be 'locked in'.
- This means we also need to check to see if any lines have been formed.
- As the piece is now locked in and is not 'active'. We need to drop in a new piece for the user to control.
- If this new piece does not fit straight away, then the game is over.

For those that haven't played Tetris before:

- When you align various tetrominos to fill a row, its called a line. This line then disappears and you get 100 points. The aim is to get as many lines as possible as you get the most points from creating lines and also to clear the board.
- In Tetris, you lose when a piece cannot be placed anymore, aka you stack all the pieces to the top.

So lets get started filling in the blanks in our code. Lets start with line 76 "If the piece does not fit straight away, game over"

```
76        // If piece does not fit straight away, game over!
77        bGameOver = !DoesPieceFit(nCurrentPiece, nCurrentRotation, nCurrentX, nCurrentY);
78        }
```

Quite simply, we use the varaible that dictates if the game loop runs or not – bGameOver. As menitioned earlier, if this varaible is true, then the game loop will not run.

Here we check if the piece fits by using our "DoesPieceFit" function. However, note that we inverse the result. This means, if the piece does not fit, the return of false will be flipped to return true. Therefore, if the piece does not fit, bGameOver = True therefore the game loop will not run and the game finishes.

Go ahead and add this line into your code.

Next we are going to 'pick the next piece' on line 74:

26

```
72        // Check for lines
73
74        // Pick New Piece
75        nCurrentX = nFieldWidth / 2;
76        nCurrentY = 0;
77        nCurrentRotation = 0;
78        nCurrentPiece = rand() % 7;
79
80        // If piece does not fit straight away, game over!
81        bGameOver = !DoesPieceFit(nCurrentPiece, nCurrentRotation, nCurrentX, nCurrentY);
82      }
83    }
```

[Line 75] We reset the nCurrentX coordinate back to the center

[Line 76] We reset the nCurrentY coordinate back to 0 meaning the new shape is produced at the top,

[Line 77] We reset the nCurrentRotation to 0 so the shape is in its start position

[Line 78] We set nCurrentPiece to a random number between 0 and 6. We can set the range using modulus 7 (meaning from 0 to 7). The range is exclusive meaning the maximum range (7) is not counted (the max is one less, in this case it is 6).

This is pseudo random, meaning each time we run the program, the 'random' variables will be the same each time. E.g. if the randomly generated numbers are 0,5,2,3,4,6,0,0,1,6,3 and we then restart the Arduino, the same sequence of numbers will be returned each time.

Add this set of code to line 74

Next we need to lock our piece into the field.

```
68        else
69        {
70          // It can't! Lock the piece in place
71          for (int px = 0; px < 4; px++)
72            for (int py = 0; py < 4; py++)
73              if (tetromino[nCurrentPiece][Rotate(px, py, nCurrentRotation)] != '.')
74                pField[(nCurrentY + py) * nFieldWidth + (nCurrentX + px)] = nCurrentPiece + 1
75
76          // Check for lines
77
78          // Pick New Piece
79          nCurrentX = nFieldWidth / 2;
80          nCurrentY = 0;
81          nCurrentRotation = 0;
82          nCurrentPiece = rand() % 7;
83
84          // If piece does not fit straight away, game over!
85          bGameOver = !DoesPieceFit(nCurrentPiece, nCurrentRotation, nCurrentX, nCurrentY);
86        }
87      }
```

[Lines 71 & 72] These two for statements allow us to iterate through the array element by element.

[Line 73] Checks the tetromino shape with the current position. This returns a string in which we then index the string (**[Rotate(px,py,nCurrentRotation)]**) Think of this statement as:

For example:

"..X..XX..X......"**[Rotate(px,py,nCurrentRotation)]** which will return the value at the index position.

In this condition, we are comparing that value with 'X'. If it is equal to X, then set then run line 74. If not, don't, as this would be empty space.

[Line 74]

```
pField[(nCurrentY + py) * nFieldWidth + (nCurrentX + px)] = nCurrentPiece + 1
```

Here we get the position of the current 'block'/element we compared earlier in the tetromino, from the field. We are then assigning it the nCurrentpiece + 1. We add oneas the index of the tetrominos starts from 0, however currently in our field, we have 0 set as empty space, therefore we add one to offset that. If tetromino 0 was being 'locked in'/ 'set' and we didn't add 1, then the piece would blend in with the empty space.

This also comes in handy when assigning the output. Remember this line?

```
93        for (int y = 0; y < nFieldHeight; y++)
94        {
95          output[(y * nFieldWidth) + x] = " ABCDEFG=#"[pField[(y * nFieldWidth) + x]];
96        }
97      }
98
```

Well, the index 0 of the string is empty space. Index 1, however, is A which represents the first tetromino (as each letter represents a different tetromino).

One last thing, we need to set the speed back to 0 now that we have foced the tetromino down 1 block:

```
60
61      // Force the piece down the playfield if it's time
62      if (bForceDown)
63      {
64        //Set speed back to 0
65        nSpeedCount = 0;
66
67        // Test if piece can be moved down
68        if (DoesPieceFit(nCurrentPiece, nCurrentRotation, nCurrentX, nCurrentY + 1)) {
69          nCurrentY++; // It can, so do it!
70        }
71        else
72        {
73          // It can't! Lock the piece in place
74          for (int px = 0; px < 4; px++)
75            for (int py = 0; py < 4; py++)
76              if (tetromino[nCurrentPiece][Rotate(px, py, nCurrentRotation)] != '.')
77                pField[(nCurrentY + py) * nFieldWidth + (nCurrentX + px)] = nCurrentPiece + 1;
78
79          // Check for lines
80
81          // Pick New Piece
82          nCurrentX = nFieldWidth / 2;
83          nCurrentY = 0;
84          nCurrentRotation = 0;
85          nCurrentPiece = rand() % 7;
86
87          // If piece does not fit straight away, game over!
88          bGameOver = !DoesPieceFit(nCurrentPiece, nCurrentRotation, nCurrentX, nCurrentY);
89        }
90      }
```

Go ahead and copy the above code and lets upload and open serial monitor and see what happens!

```
COM10                                              —   □   ×

[                                            ]  [ Send ]
----
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000000009
900000001009
900000001009
900000001009
900000001009
999999999999
----
#        À  #
#        À  #
#        À  #
#        À  #
#           #
#           #
#           #
#           #
#           #
#           #
#           #
#           #
#           #
#        À  #
#        À  #
#        À  #
#        À  #
############
----
900000000009

☐ Autoscroll              [ No line ending ∨ ] [ 9600 baud ∨ ]
```

You should be able to move your piece around, but also let it fall every game tick.

You should also notice that when it reaches the bottom, not only does it stoo and produce a new piece at the top, the pField array changes with the 'locked in' part!

We are almost ready to move to the graphics engine however we hae a couple things left.

One major feature being what happens when we get a full line! Here you can only lose – you get no points and the lines don't disappear. Lets fix that.

Checking for lines

Before we start, we need to declare an array that will store the y coordinate of the line so we know which row is a line:

```
17
18 int Lines[4] = {0, 0, 0, 0};
19
```

Be sure to add this before the void setup()

```
78
79            // Check for lines
80            for (int py = 0; py < 4; py++)
81              if (nCurrentY + py < nFieldHeight - 1)
82              {
83                bool bLine = true;
84                for (int px = 1; px < nFieldWidth - 1; px++)
85                  bLine &= (pField[(nCurrentY + py) * nFieldWidth + px]) != 0;
86
87                if (bLine)
88                {
89                  // Remove Line, set to =
90                  for (int px = 1; px < nFieldWidth - 1; px++) {
91                    pField[(nCurrentY + py) * nFieldWidth + px] = 8;
92                  }
93
94                  //Array push back
95                  for (int h = 0; h < 4; h++) {
96                    if (Lines[h] == 0) {
97                      Lines[h] = nCurrentY + py;
98                      break;
99                    }
100                  }
101                }
102              }
103
```

[Line 80] The first thing to note is, to be optimal, we only need to check the rows where the last tetromino was. This for loop takes the 4 rows from the tetromino and translates them to the field, by adding nCurrentY on line 81.

[Line 81] This is boundary check that ensures that we are not checking things beyond the boundary of the Field.

[Line 83] The first thing we do is assume there are lines, by setting bLine to true. This means we are trying to prove that there are no lines by checking if there is any empty space. If there is any empty space, then that row cannot be a line.

[Line 84] This for loop checks every element/brick in the column (apart from the two sides, hence starting from 1 (int px = 1) and minusing 1 from nFieldWidth (nFieldWidth -1) meaning we take out the first and last bricks/elements as these are boundaries and we do not want to check them.

[Line 85] bLine &= (pField[(nCurrentY + py) * nFieldWidth + px]) != 0;

Here we are checking if the element/brick in the field is empty or not if it is empty, we set bLine to false.

We firstly use our formula to find the index ((nCurrentY + py) * nFieldWidth + px) to get the element in pField to compare. The (!=) NOT EQUAL sign will then compare this value with 0, if it is 0, then it will return False, if it is anything other then 0, then it will return True.

Lets take a closer at the operator in front of bLine:

&= is a conditional operator that both assigns and compares the value. E.g.

> If VarT is true and VarF is false:
>
> VarT &= VarF will assign VarT as false as it is an AND condition.
>
> VarT &= VarT2 will assign VarT to true as true AND true returns True.

In our case, bLine is by default true. Therefore, we are comparing the result of the comparison with the bLine variable. If the condition returns false, where the current element is not empty, then:

bLine (**TRUE**) **&=** (pField[(nCurrentY + py) * nFieldWidth + px]) != 0 (**FALSE**);

Here we are comparing true and false, as it is an and statement and requires both sides to be true, the bLine var is set to false.

If it is anything other then 0 and returns true, then bLine is set to true.

Now if we have reached this point in the code, and bLine is still true, then it must be a complete line. We will then go through and set all the values to '=' signs to get an animated line.

```
87              if (bLine)
88              {
89                // Remove Line, set to =
90                for (int px = 1; px < nFieldWidth - 1; px++) {
91                  pField[(nCurrentY + py) * nFieldWidth + px] = 8;
92                }
93
94                //Array push back
95                for (int h = 0; h < 4; h++) {
96                  if (Lines[h] == 0) {
97                    Lines[h] = nCurrentY + py;
98                    break;
99                  }
100               }
101             }
```

[Line 87] Check if bLine is true, if it is true, then we have a line and must remove it.

[Line 90 to 91] This for statement iterates through the x coordinates in the line – excluding the boundary elements/bricks. This allows for line 91 to convert each pField element (getting the index position from the formula) into an 8, which if we look at our string [" ABCDEFG=#"](from when we convert pfield into the output) this is a '=' sign which, when the whole row is converted creates a line.

[Line 95 to 98] Iterates through the Lines array that we created earlier, and stores the column/y coordinate of the lines. This means we can, in a later date, remove the lines. This means the user will see the line, wait a little, then it will be removed. Creating a rewarding animation.

[Line 95] For statement iterates through the loop

[Line 96] If the current index position in the array is 0 (empty) then [Line 97] store the column number that has the line (nCurrenty+py).

[Line 97] As we are doing one line per loop, we can use the break command to break out of the for loop, meaning we are not setting any other element. This means, if there are multiple lines, they wont overwrite the data in the array, they will go to the next element in the array.

Be sure to add this section of code after the 'Lock Piece in place'. See lines for reference.

Removing Lines

Now that we have replaced the line with '=', we need to remove the line. *Be sure to add the below section of code after the //Draw Current Piece section of code*

```
126      // Draw Current Piece
127      for (int px = 0; px < 4; px++) {
128        for (int py = 0; py < 4; py++) {
129          if (tetromino[nCurrentPiece][Rotate(px, py, nCurrentRotation)] != '.')
130          {
131            output[(nCurrentY + py)*nFieldWidth + (nCurrentX + px)] = nCurrentPiece + 65;
132          }
133        }
134      }
135
136      //Remove Lines
137      int sum = 0;
138      for (int i = 0; i < 4; i++) {
139        sum = sum + Lines[i];
140      }
141
142      if (sum != 0)
143      {
144        // Display Frame (cheekily to draw lines)
145        OutputFrame();
146        for (int i = 0; i < 4; i++)
147          if (Lines[i] != 0) {
148            for (int px = 1; px < nFieldWidth - 1; px++)
149            {
150              for (int py = Lines[i]; py > 0; py--) {
151                pField[py * nFieldWidth + px] = pField[(py - 1) * nFieldWidth + px];
152              }
153              pField[px] = 0;
154            }
155          }
156          else {
157            break;
158          }
159        }
160
161        for (int g = 0; g < 4; g++) {
162          Lines[g] = 0;
163        }
164      }
165
166      //OutputField();
167      OutputFrame();
168    }
169 }
```

```
136    //Remove Lines
137    int sum = 0;
138    for (int i = 0; i < 4; i++) {
139      sum = sum + Lines[i];
140    }
141
142    if (sum != 0)
143    {
144      // Display Frame (cheekily to draw lines)
145      OutputFrame();
146      for (int i = 0; i < 4; i++)
147        if (Lines[i] != 0) {
148          for (int px = 1; px < nFieldWidth - 1; px++)
149          {
150            for (int py = Lines[i]; py > 0; py--) {
151              pField[py * nFieldWidth + px] = pField[(py - 1) * nFieldWidth + px];
152            }
153            pField[px] = 0;
154          }
155        }
156        else {
157          break;
158        }
159      }
160
161      for (int g = 0; g < 4; g++) {
162        Lines[g] = 0;
163      }
164    }
```

[Line 142] The first thing we need to do is check if there are any lines. If there are then we need to remove them. To check this, we are going to add up all the elements in the array, if it is greater then 0, then there are lines in the array meaning we need to remove them

[Line 137] Firstly we create the sum variable. Note that each game loop this will be recreated and set to 0.

[Line 138] The for statement then iterates through the array (4 times as there are 4 elements in the array).

[Line 139] We then add the elements in the line array to the sum each loop.

Now we are ready to check the sum on line 142. If there are no lines in the array then this will be false and wont run the code. However, if there is then it will run the code to remove the lines:

[Line 145] We firstly want to output the frame to draw in the '='/line before we remove the lines.

[Line 146] We then loop through the lines array, 4 times. This is so we can get the elements in the array.

[Line 147 & 156-157] If the element is not empty (0 is empty) then there is a line to be removed, if not then we have removed all of the lines and should break out of the loop.

To actually remove the lines we must:

[Line 148] Iterate through each element/brick using its x coordinate. We ignore the boundaries by starting the iteration at 1 and ending it before the boundary (nField -1).

[Line 150] Here we have a for loop that counts down. We start at the row with the line (int py = Lines[i]) and iterate until the y coordinate is 1 (py > 0) – minusing 1 each time (py--). We stop at 1 as if we stopped at row 0, we would have an error as row 0 cannot copy/move down the data

from the row above itself as there is now row above itself. To fill in the top row, we use line 153 to set the top row to all 0's. Overal, this means we are going up the playing field, a row at a time.

[Line 151] pField[py * nFieldWidth + px] = pField[(py - 1) * nFieldWidth + px];

As we are going up the board, row by row, we assign the current pField element with the pField element from above/the next row. This means the whole board gets shifted down as we iterate up the rows. Meaning the line is removed AND all the pieces are moved down.

[Line 153] As we have moved all the values down, the top values will not have a value. Therefore, we are assigning each top value to 0 (as empty) each time we loop through the cloumns (y coordinates)

[Lines 161 & 162] Iterate through the Lines array and resets all the values to 0. This gets It ready for the next game loop.

That was a lot of code! Before we move on, check your code to makre sure your void loop() looks similar to our void loop() below:

```
void loop() {
  while (!bGameOver) // Main Loop
  {
    // Timing =================================================
    nSpeedCount++;
    bForceDown = (nSpeedCount == nSpeed);
    // Input =================================================
    CheckControls();

    // Force the piece down the playfield if it's time
    if (bForceDown)
    {

      //Set speed back to 0
      nSpeedCount = 0;

      // Test if piece can be moved down
      if (DoesPieceFit(nCurrentPiece, nCurrentRotation, nCurrentX, nCurrentY + 1)) {
        nCurrentY++; // It can, so do it!
      }
      else
      {
        // It can't! Lock the piece in place
        for (int px = 0; px < 4; px++)
          for (int py = 0; py < 4; py++)
            if (tetromino[nCurrentPiece][Rotate(px, py, nCurrentRotation)] != '.')
              pField[(nCurrentY + py) * nFieldWidth + (nCurrentX + px)] = nCurrentPiece
+ 1;

        // Check for lines
        for (int py = 0; py < 4; py++)
          if (nCurrentY + py < nFieldHeight - 1)
          {
            bool bLine = true;
            for (int px = 1; px < nFieldWidth - 1; px++)
              bLine &= (pField[(nCurrentY + py) * nFieldWidth + px]) != 0;

            if (bLine)
            {
              // Remove Line, set to =
              for (int px = 1; px < nFieldWidth - 1; px++) {
                pField[(nCurrentY + py) * nFieldWidth + px] = 8;
              }

              //Array push back
```

34

```cpp
            for (int h = 0; h < 4; h++) {
              if (Lines[h] == 0) {
                Lines[h] = nCurrentY + py;
                break;
              }
            }
          }
        }
      }

      // Pick New Piece
      nCurrentX = nFieldWidth / 2;
      nCurrentY = 0;
      nCurrentRotation = 0;
      nCurrentPiece = rand() % 7;

      // If piece does not fit straight away, game over!
      bGameOver = !DoesPieceFit(nCurrentPiece, nCurrentRotation, nCurrentX, nCurrentY);
    }
  }


  // Draw Field
  for (int x = 0; x < nFieldWidth; x++)
  {
    for (int y = 0; y < nFieldHeight; y++)
    {
      output[(y * nFieldWidth) + x] = " ABCDEFG=#"[pField[(y * nFieldWidth) + x]];
    }
  }

  // Draw Current Piece
  for (int px = 0; px < 4; px++) {
    for (int py = 0; py < 4; py++) {
      if (tetromino[nCurrentPiece][Rotate(px, py, nCurrentRotation)] != '.')
      {
        output[(nCurrentY + py)*nFieldWidth + (nCurrentX + px)] = nCurrentPiece + 65;

      }
    }
  }

  //Remove Lines
  int sum = 0;
  for (int i = 0; i < 4; i++) {
    sum = sum + Lines[i];
  }

  if (sum != 0)
  {
    // Display Frame (cheekily to draw lines)
    OutputFrame();
    for (int i = 0; i < 4; i++) {
      if (Lines[i] != 0) {
        for (int px = 1; px < nFieldWidth - 1; px++)
        {
          for (int py = Lines[i]; py > 0; py--) {
            pField[py * nFieldWidth + px] = pField[(py - 1) * nFieldWidth + px];
          }
          pField[px] = 0;
        }
      }
      else {
        break;
      }
    }
```

35

```
        for (int g = 0; g < 4; g++) {
            Lines[g] = 0;
        }
    }

    //OutputField();
    OutputFrame();
    }
}
```

Adding Increasing Difficulty

Now that we have a fully working Tetris game, we need it to get harder. Let's increase the difficulty as time goes on:

Firstly we need to count how many game ticks have taken place. We are going to need a variable to count this:

```
16  bool bForceDown = false;
17
18  int Lines[4] = {0, 0, 0, 0};
19
20  int nTickCount = 0;
21
22  //Controls
23  #define Y_Button 7  //  Rotate
24  #define Left_Button  A2
```

nTickCount will keep a count, each game loop, of how many pieces have been placed.

```
64
65      if (bForceDown)
66      {
67
68          //Set speed back to 0
69          nSpeedCount = 0;
70
71          // Update difficulty every 50 ticks
72          nTickCount++;
73          if (nTickCount % 50 == 0)
74              if (nSpeed >= 10) nSpeed--;
75
76
77          // Test if piece can be moved down
78          if (DoesPieceFit(nCurrentPiece, nCurren
79
80
```

[Line 72] Each game loop, the nPieceCount will increase by 1

[Line 73] If the nTickCount is divisable by 50 with no remander, then we need to reduce the speed.

[Line 74] If the nspeed is greater than or equal to 10, then reduce the nSpeed by 1. This means the game wont reduce the difficulty if the speed is 9. Meaning the game is till playable but much faster.

Copy the above code. The section highlighted should go after the nSpeedCount = 0. See line numbers for a rough reference,

36

Adding Score

Now that we have a fully working tetris game, we are missing one key feature. A score!

Firstly we need a varaile to keep track of the score:

```
19
20  int nTickCount = 0;
21
22  int nScore = 0;
23
24  //Controls
```

nScore will keep the score.

We need to add the score in two places:

- +25 for every piece placed.
- +100 for every line formed

```
110                break;
111            }
112          }
113        }
114      }
115
116      nScore += 25;
117
118      //Add score for each line
119      for (int i = 0; i < 4; i++) {
120        if (Lines[i] != 0) {
121          nScore += 100;
122        }
123      }
124
125      // Pick New Piece
126      nCurrentX = nFieldWidth / 2;
127      nCurrentY = 0;
128      nCurrentRotation = 0;
129      nCurrentPiece = rand() % 7;
130
131      // If piece does not fit straight awa
132      bGameOver = !DoesPieceFit(nCurrentPie
133    }
134  }
135
```

If we are picking a new piece, that means that a piece has been placed. Therefore, on line 116, we can add +25 to the score as a piece has been locked down/placed.

The next bit of code, lines 118 to 123, loops through the lines array. For every element that isn't equal to 0 (aka for every line formed), the score gets +100.

Add these lines of code before the 'pick a new piece' section of code. See the line numbers for reference.

Finally, we need to display the score in serial:

```
180
181      for (int g = 0; g < 4; g++) {
182        Lines[g] = 0;
183      }
184    }
185    Serial.print( "Score:");
186    Serial.println(nScore);
187    //OutputField();
188    OutputFrame();
189  }                                    ← End of game loop
190  Serial.print( "Game Over!! Score:");
191  Serial.println(nScore);
192  }                                    ← End of void loop()
193
194  bool DoesPieceFit(int nTetromino, int nRotation, int nPosX, int nF
195  {
196    // All Field cells >0 are occupied
```

[Lines 185 & 186] Output the score every game loop

We also need a "Game Over" indication and to tell the player there final score. Lets add that:

[Lines 190 & 191] Output a Game Over message and the final score.

Be sure to add these lines of code to the end of the game loop/end of the void loop().

Graphics Engine

At this point, you can upload your code and check out the serial monitor to play Tetris in the serial monitor. (however, its quite hard as the field is constantly being outputted). The next section will focus on creating the graphics engine for our game. We will focus on rendering the Tetris game on the onboard screen on your 8BitCADE XL. Let's get started:

Render()

We are going to render the graphics in two functions.

> RenderFrame() = Will render the frame/boundary walls upon screen setup. This is because we don't need to constantly update the boundary, and if we do then we get a flickering effect that doesn't look good.

> Render() = This will render everything inside of the field. The tetrominos, lines, etc.

Lets tackle the Render() function first:

```
349  //Graphics Engine
350  void Render() {
351    int x = 0;
352    int y = 0;
353    for (int a = 0; a < (nFieldWidth) * (nFieldHeight ); a++) {
354      if (x % nFieldWidth == nFieldWidth - 1) {
355        y = y + BrickSize;
356      }
357      x = a % nFieldWidth;
358      switch (output[a]) {
359        case ' ':
360          aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
361          break;
362        case 'A':
363          aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
364          aboy.drawRect(x * BrickSize + 46, y, BrickSize, BrickSize, WHITE);
365          break;
366        case 'B':
367          aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
368          aboy.drawRect(x * BrickSize + 46, y, BrickSize, BrickSize, WHITE);
369          break;
370        case 'C':
371          aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
372          aboy.drawRect(x * BrickSize + 46, y, BrickSize, BrickSize, WHITE);
373          break;
374        case 'D':
375          aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
376          aboy.drawRect(x * BrickSize + 46, y, BrickSize, BrickSize, WHITE);
377          break;
378        case 'E':
379          aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
380          aboy.drawRect(x * BrickSize + 46, y, BrickSize, BrickSize, WHITE);
381          break;
382        case 'F':
383          aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
384          aboy.drawRect(x * BrickSize + 46, y, BrickSize, BrickSize, WHITE);
385          break;
386        case 'G':
387          aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
388          aboy.drawRect(x * BrickSize + 46, y, BrickSize, BrickSize, WHITE);
389          break;
390        case '=':
391          aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
392          aboy.drawLine(x * BrickSize + 46, y + BrickSize / 2, x * BrickSize + BrickSize + 46, y + BrickSize / 2, WHITE);
393          break;
394      }
395    }
396    aboy.display();
397  }
```

It looks long and complicated but in reality its just a large switch case! To summarise, we are simply looping through the output array and outputting, to the screen, a certain block depending on what the value is in the array. If its empty, output a black brick, if it's a tetromino, output a brick with a certain colour etc.

We need to firstly add the BrickSize variable to the top of our code:

```
5   int BrickSize = 3;
```

We set this to 3 as we want each block to be 3 pixels wide/high.

Lets take a look at the parts of the code leading up to the switch case:

```
350  void Render() {
351      int x = 0;
352      int y = 0;
353      for (int a = 0; a < (nFieldWidth) * (nFieldHeight ); a++) {
354          if (x % nFieldWidth == nFieldWidth - 1) {
355              y = y + BrickSize;
356          }
357          x = a % nFieldWidth;
358          switch (output[a]) {
```

[Lines 351 & 352] Firstly, in the function, we declare two local variables called x and y. These will be used to store the brick that we are going to render to the screen.

[Line 353] Is a for statement that will loop through all the bricks/elements in the output array.

[Lines 354] Similar to what we did before when we outputted the data to serial, when we reach the end of a row, we need to output the data on a new line/row. To do this, we use this if statement.

If the X coordinate divided by the field width (x % nFieldWidth) has a remainder of 11, (nFieldWidth -1) then [Line 355] we need to add 3 (bricksize) to the y coordinate to move it down a row (as each row is one brick and one brick is 3 pixels down).

We use a remainder of 11 as the array is indexed from 0 meaning 12 elements/bricks have been outputted. 12 bricks makes up one row as 12 is the width. (index positions 0 to 11 make up row 1, 12 – 23 make up row 2 etc) therefore if 23 divided by x is 11 (which it is) then add 3 (BrickSize) to the y coordinate, meaning the next data is offset to the next row.

[Line 357] X needs to constantly be reset when it reaches 11, back to 0, to print the new elements/bricks in the new row.

Here we use the 'a' variable from the for statement and divide it by the nFieldWidth. This means every time we shift the y value, we are also resetting the x value back to 0. E.g.

a = 23 meaning we need to shift the y value down to start a new row. The x value currently is:

x = 23 % nFieldWidth (12) meaning x is set to 11 (as 11 is the remainder).

The loop iterates once more and the a variable is now 24. This means the new data is being outputted on a new row and the x value needs to be reset to 0.

x = 24 % nFieldWidth (12) meaning x is set to 0 and so forth.

Now lets take a look at the switch case:

```
358    switch (output[a]) {
359      case ' ':
360        aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
361        break;
362      case 'A':
363        aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
364        aboy.drawRect(x * BrickSize + 46, y, BrickSize, BrickSize, WHITE);
365        break;
366      case 'B':
367        aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
368        aboy.drawRect(x * BrickSize + 46, y, BrickSize, BrickSize, WHITE);
369        break;
370      case 'C':
371        aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
372        aboy.drawRect(x * BrickSize + 46, y, BrickSize, BrickSize, WHITE);
373        break;
374      case 'D':
375        aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
376        aboy.drawRect(x * BrickSize + 46, y, BrickSize, BrickSize, WHITE);
377        break;
378      case 'E':
379        aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
380        aboy.drawRect(x * BrickSize + 46, y, BrickSize, BrickSize, WHITE);
381        break;
382      case 'F':
383        aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
384        aboy.drawRect(x * BrickSize + 46, y, BrickSize, BrickSize, WHITE);
385        break;
386      case 'G':
387        aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
388        aboy.drawRect(x * BrickSize + 46, y, BrickSize, BrickSize, WHITE);
389        break;
390      case '=':
391        aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
392        aboy.drawLine(x * BrickSize + 46, y + BrickSize / 2, x * BrickSize + BrickSize + 46, y + BrickSize / 2, WHITE);
393        break;
394    }
395  }
396  aboy.display();
397 }
```

Here we use the a variable from the for loop, as this will iterate all the way through the array, to get the index value of the output. We then compare this value with each character from the string [" ABCDEFG=#"]. The only thing we do not compare is the boundary '#' as we do not want to overwrite it every loop as it will cause flicker. That is where the other function comes in handy.

Switch + case statement: Similar to a bunch of If statements, a switch and case statement compares a value with the case value, aka if ValueToBeCompared was 2, then the 3rd case down would run as that is case 2; and ValueToBeCompared == 2, therefore a value would be returned and the loop would break. If no value is matched, then the default block is ran.

If the break statements are not used, then any case statement that matches the switch variable, will run. This includes the default block of code. See the example on the right.

```
1  switch (ValueToBeCompared) {
2  case 0:
3    //Do something like return a value
4    return [aValue];
5    break;
6  case 1:
7    //Do something like return a value
8    return [aValue];
9    break;
10 case 2:
11    //Do something like return a value
12    return [aValue];
13    break;
14 case 3:
15    //Do something like return a value
16    return [aValue];
17    break;
18 default: // if none of the cases match
19    //Do something like return a value
20    return 0;
21 }
```

```
358    switch (output[a]) {
359      case ' ':
360        aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
361        break;
```

The first case statement we use is for empty space. If the output value is equal to ' ' then we should output a blank brick. This means each time we iterate and a piece moves, the old piece on the screen is covered with a black brick.

To output to the screen, we call the fillRect function. This requires 5 parameters:

aboy.fillRect(xCoordinate, yCoordinate, Width, Height, BLACK/WHITE)

In our case, our x coordinate needs to be multiplied by the brickSize to translate it into the appropriate position as each brick is 3 pixels wide and our current x variable is from 0 to 11. We also add 46 pixels (+ 46) to centralize the board on the screen.

Our y coordinate doesn't need to be adjusted as that increases by 3 each time we finish a row.

As we want to render a square, the width and length can be set to BrickSize (or 3 by 3)

Finally, we must choose either BLACK or WHITE. Here, as they are classed as empty, we will set them to the background colour, BLACK. We use the keyword BLACK that is built into the Arduboy Library.

```
362      case 'A':
363        aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
364        aboy.drawRect(x * BrickSize + 46, y, BrickSize, BrickSize, WHITE);
365        break;
366      case 'B':
367        aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
368        aboy.drawRect(x * BrickSize + 46, y, BrickSize, BrickSize, WHITE);
369        break;
370      case 'C':
371        aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
372        aboy.drawRect(x * BrickSize + 46, y, BrickSize, BrickSize, WHITE);
373        break;
374      case 'D':
375        aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
376        aboy.drawRect(x * BrickSize + 46, y, BrickSize, BrickSize, WHITE);
377        break;
378      case 'E':
379        aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
380        aboy.drawRect(x * BrickSize + 46, y, BrickSize, BrickSize, WHITE);
381        break;
382      case 'F':
383        aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
384        aboy.drawRect(x * BrickSize + 46, y, BrickSize, BrickSize, WHITE);
385        break;
386      case 'G':
387        aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
388        aboy.drawRect(x * BrickSize + 46, y, BrickSize, BrickSize, WHITE);
389        break;
```

Here we can see the case statements for each tetromino piece.

To get a grid like affect, we use the 'aboy.drawRect()' Command that will draw just the outline of the rectangle. It requires the same parameters as the fillRect command but just does the outline.

Here we are firstly drawing a solid BLACK rectangle (to remove any previous bricks), and then drawing a WHITE outline around it. When you upload this to the screen, you'll notice how each brick can now be identified.

The final case statement is for any lines:

```
390      case '=':
391        aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
392        aboy.drawLine(x * BrickSize + 46, y + BrickSize / 2, x * BrickSize + BrickSize + 46, y + BrickSize / 2, WHITE);
393        break;
```

[Line 391] We firstly set the background of the brick to black. This removes any tetromino that was there before hand.

[Line 392] We then use a new function called 'aboy.drawLine()' which requires the below parameters:

aboy.drawLine(x0, y0, x1, y1, BLACK/WHITE);

Note x0 & y0 would be the start point, and x1 & y1 would be the endpoint. For us:

The start x coordinate is the same: x * BrickSize + 46

LEARN

The start y coordinate is : y + BrickSize / 2 as we want to go to the y coordinate, then draw the line in the middle of the brick.

The end x coordinate is: x * BrickSize + BrickSize + 46. We simply just add the Bricksize to the x coordinate to get to the 'other side' of the brick to draw a straight line that will be 2 pixels (BrickSize) wide.

The end y coordinate will be the same: y + BrickSize / 2 to keep the line straight.

As we do this for each brick in the row when a line is detected, this will remove any colours behind it by making the background black. We then draw a horizontal white line over it to showcase to the user that they have formed a line.

The final, most important line is:

```
396    aboy.display();
397 }
```
Which outputs all of this to the screen.

This is because, when we run a command like aboy.fillRect();
We are placing that in the screen buffer. aboy.display() displays the screen buffer to the screen.

Be sure to add this function at the end of your code. And call it in the below places:

```
217     Serial.print( "Score:");
218     Serial.println(nScore);
219
220     Render();
221     //OutputField();
222     //OutputFrame();
223   }
224   Serial.print( "Game Over!! Score:");
225   Serial.println(nScore);
226 }
227
228 bool DoesPieceFit(int nTetromino, int nRotation, int nPosX, int nPosY)
229 {
```

We need to call the render function at the end of the game loop.

We also comment out the serial outputs as if we don't, it will slow down our game by a lot!

```
192     }
193
194     if (sum != 0)
195     {
196       // Display Frame (cheekily to draw lines)
197       //OutputFrame();
198       Render();
199       for (int i = 0; i < 4; i++) {
200         if (Lines[i] != 0) {
201           for (int px = 1; px < nFieldWidth - 1; px++)
202           {
203             for (int py = Lines[i]; py > 0; py--) {
204               pField[py * nFieldWidth + px] = pField[(py - 1) * nFiel
205             }
206             pField[px] = 0;
207           }
208         }
209       else {
```

The next place is before we remove the lines (where we had the OutputFrame() function).

Simply comment out the OutputFrame() function and add the Render(); function.

This means we are rendering in the line, removing the line then rerendering the screen at the end of the game loop. This creates a cool 'line removal' animation

Be sure to call the function in these two places. Use the line numbers as a reference.

Now that we have completed the first render function, we need to draw the boundary/frame.

```
404  void RenderFrame() {
405    int x = 0;
406    int y = 0;
407    for (int b = 0; b <= (nFieldWidth) * (nFieldHeight); b++) {
408      if (x % nFieldWidth == nFieldWidth - 1) {
409        y = y + BrickSize;
410      }
411      x = b % nFieldWidth;
412      if (pField[b] == 9) {
413        aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, WHITE);
414      }
415    }
416    aboy.display();
417  }
```

Lines 405 to 411 should look familiar to the first 7 lines in the Render() function. This code prepares both the x and y coordinates to output the bricks on the display.

[Line 412] Here we are checking the pField display this time, not the output.

Why? Lets take a look at where we will call the function:

```
81    tetromino[5] = ".X...X...XX.....";
82    tetromino[6] = "..X...X..XX.....";
83
84    for (int x = 0; x < nFieldWidth; x++) // Board E
85      for (int y = 0; y < nFieldHeight; y++)
86        pField[y * nFieldWidth + x] = (x == 0 || x =
87
88    RenderFrame();
89  }
90
91  void loop() {
```

We call it before the output array has been created/updated, in the void setup, right after the pField array has been 'setup' with the boundary. Therefore the only array, at that moment in time, with the boundary, is the pField.

Therefore, we aren't checking for the char '#' we are checking for the number 9. – as 9 represents the boundary.

Therefore, if the pField[b] value is 9, then we need to print a white square to the screen:

To create a white box, we use the drawRect() function from before hand and set the colour to WHITE on line 413. The parameters of the x and y coordinates are also the same from the last function.

Be sure to add the above function to the bottom of your code and call it in the void setup(), after we fill in the pField array (see line numbers/photos for reference).

Now we have a fully working tetris game! Click upload and have a play!

You should notice a couple things are missing that we are going to add:

1. You can't see your score.
2. There is no indication that the game is over.
3. There is no pre-screen.

Lets go ahead and add these things:

Showing the Score

To do this, we are going to need some functions to draw text to the screen. As we are going to be using this a lot for the pre-screen, outputting the score and indicating that the game is over, we are going to make this a function.

To do this, we are going to create a function that draws text to a specified coordinate. Then another that will draw text in a centred position.

```
420  void aboy_drawString(String text, int x, int y, int size, uint16_t color, uint16_t BackgroundColour) {
421    aboy.setCursor(x, y);
422    aboy.setTextColor(color);
423    aboy.setTextSize(size);
424    aboy.setTextWrap(true);
425    aboy.println(text);
426    aboy.display();
427  }
428
429  void aboy_drawCenteredString(String text, int y, int size, uint16_t color) {
430    int len = (text.length()) * 6 * size;
431    int left = (128 - len ) / 2;
432    if (left < 0) {
433      left = 0;
434    }
435
436    aboy_drawString( text, left, y, size, color, BLACK);
437  }
```

The first function 'aboy_drawString()' is a easy way to print text to the screen in one line.

It takes all the parameters required to print the text to the screen:

- String Text = The message that we want to display
- Int x = Is the x coordinate
- Int y = is the y coordinate
- Int size = how big the text is going to be
- Unint16_t color = takes a 16 bit colour variable that will be used for the text colour
- Unint16_t BackgroundColour = Takes a 16 bit colour variable that will be used for the background colour.

With these variables we have to then actually print the text to the screen:

The first thing we do is set the cursor position. This will tell the Arduino where we want the string to be printed, see the grid below. The setCursor command will set the cursor to the top left of the text.

Next, we set the text colour, we can set both the colour of the text itself (the A in the picture) and the background (the greyed out 6x8 area). You can use aboy.setTextColor(BLACK/WHITE) to set the text colour to either BLACK or WHITE

Next, we have the Text size. This will multiply the default text size by the specified scale factor. The default size is 1 and would produce the text below (5 pixels wide by 7 pixels in height). Note that this is an integer value so values such as 1.5 cannot be used. A factor of 2 would produce text with 10 pixels in width and 14 pixels in height.
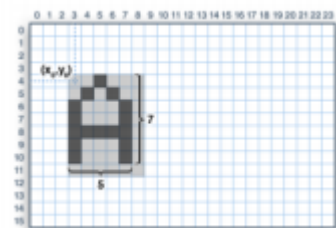
The text wrap is how the text interacts with meeting the edge of the screen, with the text wrap, it will simply move on to a new line. You can, however, turn this off by setting this to false, meaning the text CAN print off-screen.

Finally, we have the print command, aboy.println(). This will print the text to screen buffer under the above 'settings' or setup.

We then use aboy.display() on the next line to display this on the screen.

Now that we have the basic function, we want another function that will automatically print centered text depending on how large the string is:

```
428  void Tft_drawCenteredString(String text, int y, int size, uint16_t color) {
429    int len = (text.length()) * 6 * size;
430    int left = (128 - len ) / 2;
431    if (left < 0) {
432      left = 0;
433    }
434
435    Tft_drawString( text, left, y, size, color, BLACK);
436  }
```

Here we require 4 parameters:

- String text = the message we want to output
- Int y = the y coordinate that we want to output to the screen – notice there are no x coordinates as the text will be cantered.
- Int size = the size of the text.
- Unt16_t color = the colour of the text

Now that we have the parameters, we need to do something with them to output a centred message:

[Line 429] defined the length of the text and assigns it to len. We do this by getting the amount of characters in the 'text' string by using the .length() function. We multiple it by 6 to get how many pixels wide it is. We also multiply it by the text size. As if we wanted larger text, then there would be more pixels. Overall, this gets the length of the string in pixels:

e.g. the word "Hello" at font size 1 would be:

4 * 6 * 1 = 24 pixels

[Line 430] Next we find out the x coordinate of the text by finding out how far 'left' the text needs to be. We store this in the variable called 'left'.

To get this value, we minus the overall width of the screen with the length of the string. The returning value is then divided by 2 to get how many pixels there are either side of the string.

e.g.

(128 – 24 (for "Hello")) = 104 / 2 = 52. Therefore, to print the text "Hello" centred, we need 52 pixels either side. We store this in the variable called 'left' that will be passed through, in line 435, as the x coordinate.

[Lines 431 to 433] Simply checks if the 'left' variable is less then 0, if it is then It means the string is longer then the screen and, to not get any errors, we set the left coordinate to 0.

[Line 435] Here we are calling the earlier function we created. We are passing through the text, x coordinate as 'left', the y coordinate, the size, the color, and finally we assume the background colour is black.

Make sure to add these function to the end of the whole code (see line numbers for reference).

We now need to call the cantered string function to print out the score.

```
204
205        for (int g = 0; g < 4; g++) {
206          Lines[g] = 0;
207        }
208     }
209     //Serial.print( "Score:");
210     //Serial.println(nScore);
211
212     aboy_drawCenteredString(String(nScore), 56, 1, WHITE);
213
214     Render();
215     //OutputField();
216     //OutputFrame();
217   }
218   // Oh Dear = GAME OVER
219   Serial.print( "Game Over!! Score:");
```

[Lne 212] Here we are converting the integer 'nScore' To a string using the String() function.

We then pass in the y coordinate, 150, to display it underneath the Tetris game field. As we don't want it to big, we use text size 1 and white text.

Add this code in underneath where we serial.print() the score and upload the code to see the score go up!

Game Over screen

Now that we have these two functions, we can easily create a game over screen:

```
214        Render();
215        //OutputField();
216        //OutputFrame();
217     }
218     // Oh Dear = GAME OVER
219     Serial.print( "Game Over!! Score:");
220     Serial.println(nScore);
221
222     aboy_drawCenteredString("GAME OVER", 26, 2, WHITE);
223     aboy_drawCenteredString(String(nScore), 56, 1, WHITE);
224  }
225
226  bool DoesPieceFit(int nTetromino, int nRotation, int nPosX, int nPosY)
```

Here, we are displaying:

[Line 231] The Text "GAME OVER" in the centre of the screen. As the text prints with a black background, It covers some the game field meaning we an read the text. This shows to the gamer that the game is over and is placed after the game loop in the void loop.

[Line 232] We output the final score using the same line as before.

Go ahead and insert this code where we 'serial.print()' the game screen and play/lose the game to see it pop up!

LEARN

Creating a Pre-Screen

A pre-screen shows the game name/creator. For us, we want to show the words Tetris and your name!

To make it easier to implement, we are going to create it in a function:

```
432  void PreScreen() {
433    aboy_drawCenteredString("Tetris", 26, 2, WHITE);
434    aboy_drawCenteredString("By Jack Daly", 56, 1, WHITE);
435
436    delay(1000);
437  }
```

The function is called 'PreScreen' and is just an easy way to implement a Pre-Screen.

[Line 439] We firstly print "Tetris" in large writing in the centre of the screen.

[Line 443] Here we are printing your name to the screen, go ahead and change this to your name! We set the y value to 100 so it fits underneath the title.

Finally we add a 1 second delay so the player can read the pre-screen and move onto playing the game.

Now that we have created our own pre-screen, we need to call it in the void setup():

```
37  void setup() {
38    Serial.begin(9600);
39
40    aboy.boot();
41    aboy.clear();
42
43    //Output Pre Screen
44    PreScreen();
45    aboy.clear();
46
47    //Creating the pieces
```

Here we can see, we call the pre-screen, then clear the screen. This prepares it for when we render the frame as the background will be black and it removes any past text that was on the screen.

Think of it as 'cleaning the screen'.

Last touch - Sound

The game is pretty much complete, but it is to quiet... Lets add some basic sound to the game.

```
33
34  //Buzzer
35  #define SPEAKER PIN_SPEAKER_1//A1
36
37  void setup() {
38    Serial.begin(9600);
39
40    aboy.boot();
41    aboy.clear();
42
43    pinMode(PIN_SPEAKER_1, OUTPUT);
44    pinMode(PIN_SPEAKER_2, OUTPUT);
45
```

The first thing we need to do is define the speaker pin. We do this by using the Arduboy2 Lib keyword "**PIN_SPEAKER_1**".

To make the buzzer work, we need to set both PIN_SPEAKER_1 AND PIN_SPEAKER_2 to an OUTPUT. To let Arduino know we are sending data to the

If you get sick of the buzzer, you can simply set this to 0 and the buzzer will be gone.

Go ahead and add this above/in the void setup()

We are going to be adding core sounds in certain area of our code:

The first is to play a quick tune when we place down a piece.

```
151                  break;
152            }
153          }
154        }
155      }
156
157      //Play Tune when piece is placed
158      tone(SPEAKER, 1000, 40);
159      delay(40);
160      tone(SPEAKER, 1250, 20);
161      delay(20);
162      |
163      nScore += 25;
164
165      //Add score for each line
166      for (int i = 0; i < 4; i++) {
167        if (Lines[i] != 0) {
168          nScore += 100;
```

To use the buzzer we require two functions:

tone(speakerPin, Frequency, **delay**);

delay(**delay**);

The first function creates the actual sound.

This function requires 3 parameters: The pin of the buzzer, the frequency that we want the buzzer to play at and the delay, how long the tone will play for.

The second function, the delay, stops the program from running the second tone command and lets the first tone command play its tone. Note that the tone will stop playing after the given delay. Aka, the tone on line 158 will play for 40 milliseconds. To stop the next tone from playing, we pause for 40 seconds while the first tone plays.

The thing to note here is the frequency. The frequency varies from about 123 to 4978. You can see the '8BitCADE Colour: Sound' Tutorial for a more in depth breakdown of all the notes available.

Here we are using 1000, which is a lower tone, in the first tone command [Line 158] and then 1250, which is a bit higher, in the second tone function [Line 160]. This creates a cool sound that tells the user that their piece has been locked down.

Go ahead and add these lines to your code, use the line numbers for a reference. These lines of code should be placed above the nScore += 25;

The next tune to play is when the user gets a line:

```
210      {
211        // Display Frame (cheekily to draw line
212        //OutputFrame();
213        Render();
214
215        for (int i = 0; i < 1000 ; i += 100)
216        {
217          tone(SPEAKER, i, 50);
218          delay( 50);
219        }
220
221        for (int i = 0; i < 4; i++) {
222          if (Lines[i] != 0) {
```

Here, when a line is detected, we play a small tune.

We use a for loop that's starts from 0, ends at 1000, and increases in 100's, to play an increasing frequency.

We used 1000 and 100 instead of 10 and 1 so we could use the i value as the tone frequency.

Using the i as the frequency, allows for the tone to change after each iteration of the loop to create a tune that has increasing tones.

We use a small delay of 50 to not get in the way of the gameplay but to give us enough time to play the tune.

Be sure to add this small section of code in after the first Render(). Use the line numbers as a reference.

```
441  void PreScreen() {
442    aboy_drawCenteredString("Tetris", 26, 2, WHITE);
443    aboy_drawCenteredString("By Jack Daly", 56, 1, WHITE);
444    for (int i = 0; i < 7; i++) {
445      tone(SPEAKER, i * 100, 10 * i);
446      delay(10 * i);
447    }
448    tone(SPEAKER, 750, 40);
449    delay(40);
450
451    tone(SPEAKER, 500, 80);
452    delay(80);
453    delay(1000);
454  }
```

Finally we will adjust the pre-screen to add a nice tune:

The first thing to note is the tune inside the for loop on lines 444 and 446. Here we are taking the i value from the loop and:

i * 100 = multiplying it by 100 to give us a varying frequency after each iteration.

10 * i = Multiplying it by 10 to get a varying delay after each iteration.

The for statements will iterate 7 times to produce a varying toned sound.

After the for statement, we play two extra sounds, one that is higher than the last frequency in the loop, on line 471, and one that is lower then the last frequency player, on line 474.

Be sure to update your new pre-screen and upload your code to hear your new sound effects!

Final Code

Congratulations for completing this course! Now you have Tetris on your 8BitCADE XL. Now is your chance to play around with it and see what you can add. Why not adjust the splash screen, the sounds or see what else you can do! The world is yours!

Thank you for following along with this tutorial. If you have any programming questions, we strongly advise that you check out the Arduboy community. If you have any 8BitCADE related issues, please email us at 8BitCADE@support.com.

Check out our other tutorials at 8bitcade.com/learn

Tetris_Tutorial.ino

```
#include <Arduboy2.h>
Arduboy2 aboy;
int BrickSize = 3;

String tetromino[7];
bool bGameOver = false;

int nFieldWidth = 12;
int nFieldHeight = 18;
unsigned char pField[216];
char output[216];

int nCurrentPiece = 0;
int nCurrentRotation = 0;
int nCurrentX = nFieldWidth / 2;
int nCurrentY = 0;

int nSpeed = 10;
```

```cpp
int nSpeedCount = 0;
bool bForceDown = false;

int Lines[4] = {0, 0, 0, 0};

int nTickCount = 0;

int nScore = 0;

float lastHoldTime = 0;
bool bKey[4] = {0, 0, 0, 0};
bool bRotateHold = true;

//Buzzer
#define SPEAKER PIN_SPEAKER_1//A1

void setup() {
  Serial.begin(9600);

  aboy.boot();
  aboy.clear();

  pinMode(PIN_SPEAKER_1, OUTPUT);
  pinMode(PIN_SPEAKER_2, OUTPUT);

  //Output Pre Screen
  PreScreen();
  aboy.clear();

  //Creating the pieces
  tetromino[0] = "..X...X...X...X."; // Tetronimos 4x4
  tetromino[1] = "..X..XX..X......";
  tetromino[2] = ".....XX..XX.....";
  tetromino[3] = ".X...XX...X.....";
  tetromino[4] = ".X...XX..X......";
  tetromino[5] = ".X...X...XX.....";
  tetromino[6] = "..X...X..XX.....";

  for (int x = 0; x < nFieldWidth; x++) // Board Boundary
    for (int y = 0; y < nFieldHeight; y++)
      pField[y * nFieldWidth + x] = (x == 0 || x == nFieldWidth - 1 || y == nFieldHeight - 1) ? 9 : 0;

  RenderFrame();
}

void loop() {
  while (!bGameOver) // Main Loop
  {
    // Timing ================================================
    nSpeedCount++;
    delay(nSpeed);
    bForceDown = (nSpeedCount == nSpeed);
    // Input ================================================
    CheckControls();

    // Force the piece down the playfield if it's time
    if (bForceDown)
    {

      //Set speed back to 0
      nSpeedCount = 0;

      // Update difficulty every 50 ticks
      nTickCount++;
      if (nTickCount % 50 == 0)
```

```
          if (nSpeed >= 10) nSpeed--;


      // Test if piece can be moved down
      if (DoesPieceFit(nCurrentPiece, nCurrentRotation, nCurrentX, nCurrentY + 1)) {
        nCurrentY++; // It can, so do it!
      }
      else
      {
        // It can't! Lock the piece in place
        for (int px = 0; px < 4; px++)
          for (int py = 0; py < 4; py++)
            if (tetromino[nCurrentPiece][Rotate(px, py, nCurrentRotation)] != '.')
              pField[(nCurrentY + py) * nFieldWidth + (nCurrentX + px)] = nCurrentPiece
+ 1;

        // Check for lines
        for (int py = 0; py < 4; py++)
          if (nCurrentY + py < nFieldHeight - 1)
          {
            bool bLine = true;
            for (int px = 1; px < nFieldWidth - 1; px++)
              bLine &= (pField[(nCurrentY + py) * nFieldWidth + px]) != 0;

            if (bLine)
            {
              // Remove Line, set to =
              for (int px = 1; px < nFieldWidth - 1; px++) {
                pField[(nCurrentY + py) * nFieldWidth + px] = 8;
              }

              //Array push back
              for (int h = 0; h < 4; h++) {
                if (Lines[h] == 0) {
                  Lines[h] = nCurrentY + py;
                  break;
                }
              }
            }
          }

        //Play Tune when piece is placed
        tone(SPEAKER, 1000, 40);
        delay(40);
        tone(SPEAKER, 1250, 20);
        delay(20);

        nScore += 25;

        //Add score for each line
        for (int i = 0; i < 4; i++) {
          if (Lines[i] != 0) {
            nScore += 100;
          }
        }

        // Pick New Piece
        nCurrentX = nFieldWidth / 2;
        nCurrentY = 0;
        nCurrentRotation = 0;
        nCurrentPiece = rand() % 7;

        // If piece does not fit straight away, game over!
        bGameOver = !DoesPieceFit(nCurrentPiece, nCurrentRotation, nCurrentX, nCurrentY
);
      }
```

```
      }


      // Draw Field
      for (int x = 0; x < nFieldWidth; x++)
      {
        for (int y = 0; y < nFieldHeight; y++)
        {
          output[(y * nFieldWidth) + x] = " ABCDEFG=#"[pField[(y * nFieldWidth) + x]];
        }
      }

      // Draw Current Piece
      for (int px = 0; px < 4; px++) {
        for (int py = 0; py < 4; py++) {
          if (tetromino[nCurrentPiece][Rotate(px, py, nCurrentRotation)] != '.')
          {
            output[(nCurrentY + py)*nFieldWidth + (nCurrentX + px)] = nCurrentPiece + 65;

          }
        }
      }

      //Remove Lines
      int sum = 0;
      for (int i = 0; i < 4; i++) {
        sum = sum + Lines[i];
      }

      if (sum != 0)
      {
        // Display Frame (cheekily to draw lines)
        //OutputFrame();
        Render();

        for (int i = 0; i < 1000 ; i += 100)
        {
          tone(SPEAKER, i, 50);
          delay( 50);
        }

        for (int i = 0; i < 4; i++) {
          if (Lines[i] != 0) {
            for (int px = 1; px < nFieldWidth - 1; px++)
            {
              for (int py = Lines[i]; py > 0; py--) {
                pField[py * nFieldWidth + px] = pField[(py - 1) * nFieldWidth + px];
              }
              pField[px] = 0;
            }
          }
          else {
            break;
          }
        }

        for (int g = 0; g < 4; g++) {
          Lines[g] = 0;
        }
      }
      //Serial.print( "Score:");
      //Serial.println(nScore);
      aboy_drawCenteredString(String(nScore), 56, 1, WHITE);

      Render();
      //OutputField();
```

```
        //OutputFrame();
    }
    // Oh Dear = GAME OVER
    Serial.print( "Game Over!! Score:");
    Serial.println(nScore);

    aboy_drawCenteredString("GAME OVER", 26, 2, WHITE);
    aboy_drawCenteredString(String(nScore), 56, 1, WHITE);
}

bool DoesPieceFit(int nTetromino, int nRotation, int nPosX, int nPosY)
{
    // All Field cells >0 are occupied
    for (int px = 0; px < 4; px++)
      for (int py = 0; py < 4; py++)
      {
          // Get index into piece
          int pi = Rotate(px, py, nRotation);

          // Get index into field
          int fi = (nPosY + py) * nFieldWidth + (nPosX + px);

          // Check that test is in bounds. Note out of bounds does
          // not necessarily mean a fail, as the long vertical piece
          // can have cells that lie outside the boundary, so we'll
          // just ignore them
          if (nPosX + px >= 0 && nPosX + px < nFieldWidth)
          {
            if (nPosY + py >= 0 && nPosY + py < nFieldHeight)
            {
                // In Bounds so do collision check
                if (tetromino[nTetromino][pi] != '.' && pField[fi] != 0)
                  return false; // fail on first hit
            }
          }
      }

    return true;
}

int Rotate(int px, int py, int r)
{
    int pi = 0;
    switch (r % 4)
    {
      case 0: // 0 degrees       // 0  1  2  3
        pi = py * 4 + px;        // 4  5  6  7
        break;                   // 8  9 10 11
      //12 13 14 15

      case 1: // 90 degrees      //12  8  4  0
        pi = 12 + py - (px * 4); //13  9  5  1
        break;                   //14 10  6  2
      //15 11  7  3

      case 2: // 180 degrees     //15 14 13 12
        pi = 15 - (py * 4) - px; //11 10  9  8
        break;                   // 7  6  5  4
      // 3  2  1  0

      case 3: // 270 degrees     // 3  7 11 15
        pi = 3 - py + (px * 4);  // 2  6 10 14
        break;                   // 1  5  9 13
    }                            // 0  4  8 12

    return pi;
```

```cpp
}


void OutputField() {
  for (int i = 0; i < nFieldWidth * nFieldHeight; i++) {
    if (i % nFieldWidth == nFieldWidth - 1) {
      Serial.println(pField[i]);
    }
    else {
      Serial.print(pField[i]);
    }
  }
  Serial.println("----");
}

void OutputFrame() {
  for (int i = 0; i < nFieldWidth * nFieldHeight; i++) {
    if (i % nFieldWidth == nFieldWidth - 1) {
      Serial.println(output[i]);
    }
    else {
      Serial.print(output[i]);
      //Serial.print( (uint8_t) output[i] );
    }
  }

  Serial.println("----");
}

void CheckControls() {
  bKey[0] = (aboy.pressed(RIGHT_BUTTON) && (millis() - lastHoldTime) > 100) ? 1 : 0;
  bKey[1] = (aboy.pressed(LEFT_BUTTON) && (millis() - lastHoldTime) > 100) ? 1 : 0;
  bKey[2] = (aboy.pressed(DOWN_BUTTON) && (millis() - lastHoldTime) > 100) ? 1 : 0;
  bKey[3] = aboy.pressed(A_BUTTON);

  if(aboy.pressed(RIGHT_BUTTON) && (millis() - lastHoldTime) > 100){
    bKey[0] = 1;
  }
  else{
    bKey[0] = 0;
  }

  //Simple Buffer
  int Sum = 0;
  for (int i = 0; i < 4; i++) {
    Sum = Sum + bKey[i];
  }
  if (Sum != 0) {
    lastHoldTime = millis();
  }

  // Handle player movement
  nCurrentX += (bKey[0] && DoesPieceFit(nCurrentPiece, nCurrentRotation, nCurrentX + 1, nCurrentY)) ? 1 : 0;
  nCurrentX -
= (bKey[1] && DoesPieceFit(nCurrentPiece, nCurrentRotation, nCurrentX - 1, nCurrentY)) ? 1 : 0;
  nCurrentY += (bKey[2] && DoesPieceFit(nCurrentPiece, nCurrentRotation, nCurrentX, nCurrentY + 1)) ? 1 : 0;

  // Rotate, but latch to stop wild spinning
  if (bKey[3])
  {
    nCurrentRotation += (bRotateHold && DoesPieceFit(nCurrentPiece, nCurrentRotation + 1, nCurrentX, nCurrentY)) ? 1 : 0;
```

```
      bRotateHold = false;
    }
    else
      bRotateHold = true;
}

//Graphics Engine
void Render() {
  int x = 0;
  int y = 0;
  for (int a = 0; a < (nFieldWidth) * (nFieldHeight ); a++) {
    if (x % nFieldWidth == nFieldWidth - 1) {
      y = y + BrickSize;
    }
    x = a % nFieldWidth;
    switch (output[a]) {
      case ' ':
        aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
        break;
      case 'A':
        aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
        aboy.drawRect(x * BrickSize + 46, y, BrickSize, BrickSize, WHITE);
        break;
      case 'B':
        aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
        aboy.drawRect(x * BrickSize + 46, y, BrickSize, BrickSize, WHITE);
        break;
      case 'C':
        aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
        aboy.drawRect(x * BrickSize + 46, y, BrickSize, BrickSize, WHITE);
        break;
      case 'D':
        aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
        aboy.drawRect(x * BrickSize + 46, y, BrickSize, BrickSize, WHITE);
        break;
      case 'E':
        aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
        aboy.drawRect(x * BrickSize + 46, y, BrickSize, BrickSize, WHITE);
        break;
      case 'F':
        aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
        aboy.drawRect(x * BrickSize + 46, y, BrickSize, BrickSize, WHITE);
        break;
      case 'G':
        aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
        aboy.drawRect(x * BrickSize + 46, y, BrickSize, BrickSize, WHITE);
        break;
      case '=':
        aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, BLACK);
        aboy.drawLine(x * BrickSize + 46, y + BrickSize / 2, x * BrickSize + BrickSize
+ 46, y + BrickSize / 2, WHITE);
        break;
    }
  }
  aboy.display();
}

void RenderFrame() {
  int x = 0;
  int y = 0;
  for (int b = 0; b <= (nFieldWidth) * (nFieldHeight); b++) {
    if (x % nFieldWidth == nFieldWidth - 1) {
      y = y + BrickSize;
    }
    x = b % nFieldWidth;
    if (pField[b] == 9) {
```

```
      aboy.fillRect(x * BrickSize + 46, y, BrickSize, BrickSize, WHITE);
    }
  }
  aboy.display();
}

void aboy_drawString(String text, int x, int y, int size, uint16_t color, uint16_t Back
groundColour) {
  aboy.setCursor(x, y);
  aboy.setTextColor(color);
  aboy.setTextSize(size);
  aboy.setTextWrap(true);
  aboy.println(text);
  aboy.display();
}

void aboy_drawCenteredString(String text, int y, int size, uint16_t color) {
  int len = (text.length()) * 6 * size;
  int left = (128 - len ) / 2;
  if (left < 0) {
    left = 0;
  }

  aboy_drawString( text, left, y, size, color, BLACK);
}

void PreScreen() {
  aboy_drawCenteredString("Tetris", 26, 2, WHITE);
  aboy_drawCenteredString("By Jack Daly", 56, 1, WHITE);
  for (int i = 0; i < 7; i++) {
    tone(SPEAKER, i * 100, 10 * i);
    delay(10 * i);
  }
  tone(SPEAKER, 750, 40);
  delay(40);

  tone(SPEAKER, 500, 80);
  delay(80);
  delay(1000);
}
```