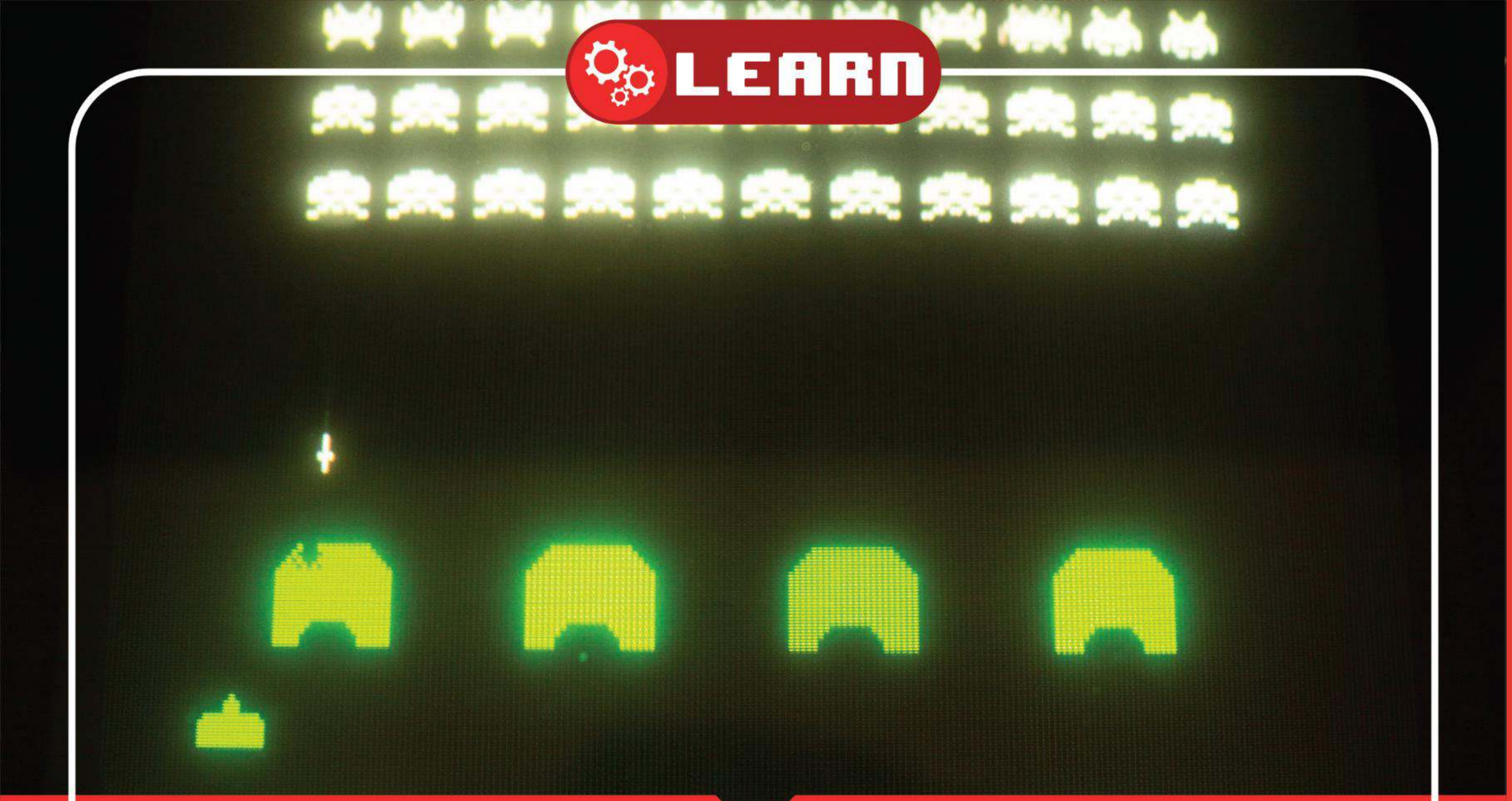
LEARN



CODING SPACE INVADERS PART 2

With 8BitCADE

Compatible with:

8BitCADE XL

AND

8BitCADE

Contents

Tutorial by 8BitCADE adapted from the amazing work by Xtronical

The original guide can be followed online at

https://www.xtronical.com/projects/space-invaders/

Lesson Number: 1

Lesson Title: Plotting Invaders on a Display

Code: Full Code for Lesson

System: Arduboy + Project ABE

Prerequisites to completing this tutorial

1. Know how to write code in either **Arduino IDE** or **Project ABE online Emulator**
2. Know how to draw pixel art by watching **Pixel Art Tutorial Space Invader**
3. Know how to convert pixel art into hexadecimal code by watching **Converting Pixel Art into Hexadecimal Code**
4. **Understand For loop, Switch Case, Arrays, Data Types**

Step 1 CREATING ALL THE CONSTANTS

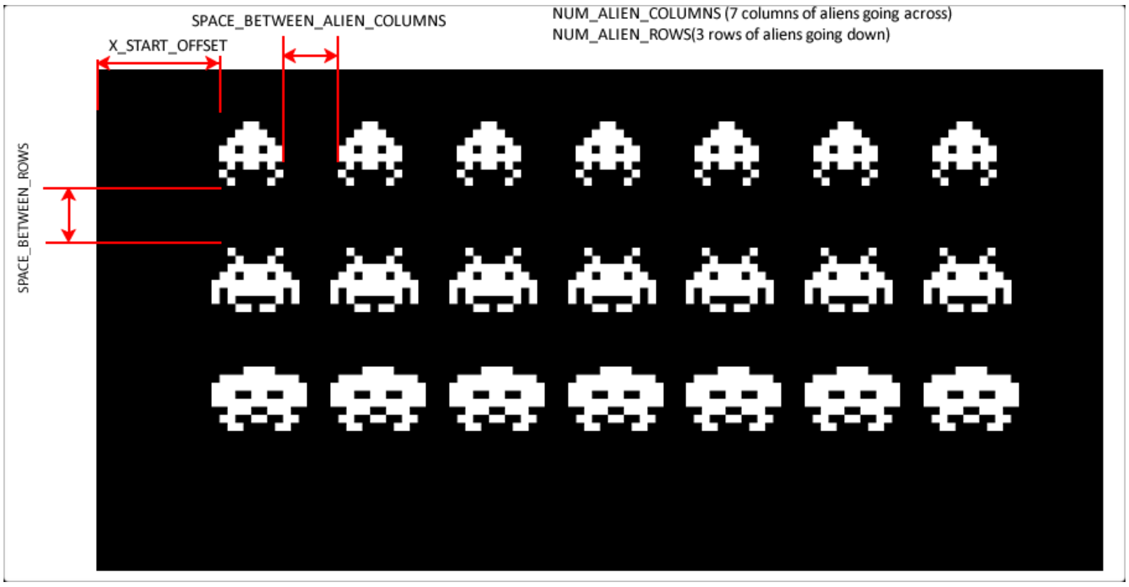
Type the following code into line 1-16

```
1.  /* Delete code from lesson 1 int XPos = 0; // integer for the initial position of the sprite + All of Void Loop
2.    www.pixilart.com for sprite generation
3.    http://javl.github.io/image2cpp/ for image conversion
4.    This code displays 3 types of space invader
5.  */
6.  //libraries for graphics/display
7.  #include <Arduboy2.h>
8.  Arduboy2 arduboy;
9.  //Sprite settings see graphic for explanation. We have 3 aliens
10. #define NUM_ALIEN_COLUMNS 7                //Columns of aliens going across
11. #define NUM_ALIEN_ROWS 3                   //Rows of aliens going down
12. #define SPACE_BETWEEN_ALIEN_COLUMNS 5      //Space in pixels between each alien in the columns
13. #define SPACE_BETWEEN_ROWS 9               //Space in pixels between each alien in the rows
14. #define LARGEST_ALIEN_WIDTH 11             //Size in pixels of the largest allien width
15. #define X_START_OFFSET 6                   //Position of the first alien from in X,from the 0 position
16.
```

Explanation (line 10-15)

Lines 10-15 define some constants that we will use in our code. Defined constants in arduino don't take up any program memory space on the chip. #define has two parameters, the constantName and value. All capital letters and underscores between words help identify the constants. Note that these are not really constants in a true C programming sense, they are as you can see "definitions" created with the define keyword. Throughout this tutorial the word constant will refer to these rather than define. The constants are fairly straight forward to explain, here is a diagram explaining the alien settings in particular.

See Next Page For Breakdown


SPACE_BETWEEN_ALIEN_COLUMNS
X_START_OFFSET
NUM_ALIEN_COLUMNS (7 columns of aliens going across)
NUM_ALIEN_ROWS(3 rows of aliens going down)
SPACE_BETWEEN_ROWS

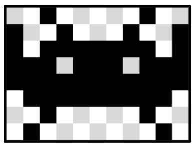
The screen size is 128 pixels wide by 64 pixels high. 0,0, position is top left.

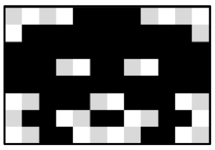
The constant: **NUM_ALIEN_ROWS** is used to set the number of rows = 3
The constant : **NUM_ALIEN_COLUMNS** is used to set the number of columns = 7
The constant: **SPACE_BETWEEN_ALIEN_COLUMNS** to set the spacing between columns of aliens = 5 (pixels)
The constant: **SPACE_BETWEEN_ROWS** to set the spacing between columns of aliens = 9 (pixels)
The constant: **LARGEST_ALIEN_WIDTH** is the width of the largest alien = 11 (pixels)
The constant : **X_START_OFFSET** position of the first alien in X axis = 6

Step 2 CREATING Graphics

Type the following code into line 17-31

```
17. // Sprites
18. static const unsigned char PROGMEM InvaderTopGfx [] = {
19.   //8x8
20.   0x98, 0x5c, 0xb6, 0x5f, 0x5f, 0xb6, 0x5c, 0x98
21. };
22.
23. static const unsigned char PROGMEM InvaderMiddleGfx [] = {
24.   // 11, 8,                                   // width, height,pixel size of image
25.   0x1e, 0xb8, 0x7d, 0x36, 0x3c, 0x3c, 0x3c, 0x36, 0x7d, 0xb8, 0x1e
26. };
27.
28. static const unsigned char PROGMEM InvaderBottomGfx [] = {
29.   //12x8
30.   0x1c, 0x5e, 0xfe, 0xb6, 0x37, 0x5f, 0x5f, 0x37, 0xb6, 0xfe, 0x5e, 0x1c
31. };
```



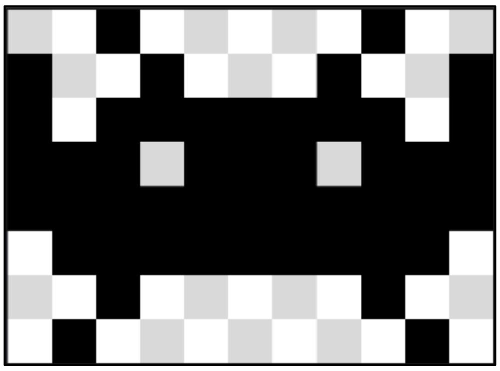


Explanation (line 17-31)

As explained in previous tutorials, graphics are simply definitions of where we want to show a pixel (or not). For simple black and white images a binary "1" would mean a white pixel and a 0 would mean black. In fact the graphics library used will plot a white pixel where there is a "1" defined and where there is a "0" it will not do anything – effectively leaving whatever is already on the screen there. So a white pixel already on screen would remain on screen if it had already been plotted previously. This can be a very useful behaviour for more advanced games but for Space Invaders it really doesn't matter.

Look below at the binary pattern for one of the invaders;

```
 static const unsigned char PROGMEM
 InvaderMiddleGfx []=
 {
   B00100000,B10000000,
   B00010001,B00000000,
   B00111111,B10000000,
   B01101110,B11000000,
   B11111111,B11100000,
   B10111111,B10100000,
   B10100000,B10100000,
   B00011011,B00000000
 };
```



You can see from the binary pattern of 000's and 111's that a 0 represents a black (or no pixel) and a 1 represents a white (or switched on pixel). An important note, although the invader is physically only 11 pixels wide we must **define up to a full byte**. 8 bits fits in the first byte across and then we only use a further (maximum) 3 bits of the second byte but we must pad it out after the last pixel (last "1") with 0's otherwise the image will not look right. Why? Well the compiler if it sees this line; **B10000000** it will **correctly** see it as the value 128 (which is the decimal equivalent of this byte). However if we put **B1** and no trailing 0's it will see it as the decimal value "1" and this is what will be stored, but what does decimal "1" look like when stored as a byte?

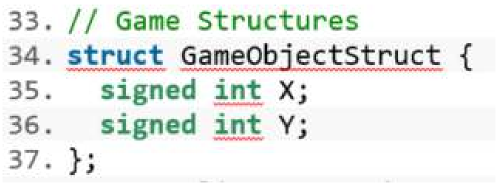
Like this:
B00000001
Which is dramatically different to what we actually intended. Why must it store as a byte (8 bits)? Well that is the fundamental unit of memory from basically the massive growth in micro-computers from the late 1970's to the early 80's. And it's this value that has kind of stuck as the basic unit of memory. So all 8 bits of a byte are always used to store a value. So a **B1** is the number 1 and the number 1 as an 8bit byte is **00000001**. So if you intend it to be the left most bit that is a 1 then you must explicitly put in the remaining 0's to give **B10000000.**

For this project the image was created in www.pixilart.com and then converted into hexadecimal code using http://javl.github.io/image2cpp/ for image conversion.

Like this:
0x98

Step 3 GAME STRUCTURES

Type the following code into line 1-6



```
33. // Game Structures
34. struct GameObjectStruct {
35.   signed int X;
36.   signed int Y;
37. };
```

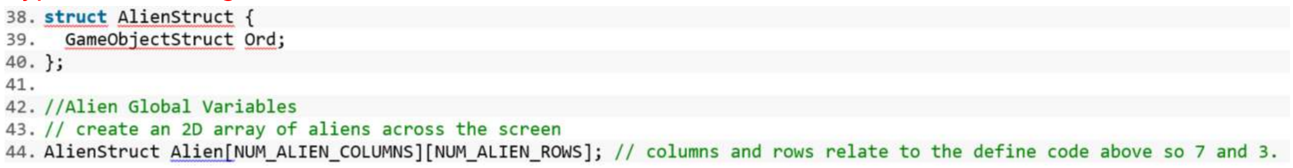
Explanation (line 33-37)

In this tutorial we introduce the concept of structures. It enables the programmer to create a **variable** that structures a selected set of data, often other variables which might be related, in simple terms, these are just convenient ways of **grouping related data together** to make the program more readable/maintainable and easier to write in the first place. Struct has a variable name and members (other variables/data). Line 35 and 36 defines two variables X and Y, which stores the X and Y position of a character and are members of **struct**.

Note: You might ask why a "signed int", why not a byte as the screen is only 128 pixel positions across and 64 pixel positions down. Initially it was like that in the first draft of the game, but as the program developed there was a requirement to have some objects partially or wholly off screen, the mothership for example and missiles and bombs etc. This meant we needed signed numbers so we can store negatives and numbers above 127 (last pixel position of the screen, goes 0-127). This meant that bytes were no longer an option despite taking up half the space as an int. An unsigned byte can only store 0-255, a signed byte only -128 to +127.

Step 4 ALIEN STRUCTURE & ARRAY

Type the following code into line 38-44



```
38. struct AlienStruct {
39.   GameObjectStruct Ord;
40. };
41.
42. //Alien Global Variables
43. // create an 2D array of aliens across the screen
44. AlienStruct Alien[NUM_ALIEN_COLUMNS][NUM_ALIEN_ROWS]; // columns and rows relate to the define code above so 7 and 3.
```

Explanation (line 38-44)

Line 38 – 40: **struct AlienStruct {** with **struct GameObjectStruct** Ord{ links **this structure** to **the previous structure** line 34 and its members. This structure as presented, consists of just the AlienStruct at the moment. As the game develops, additional struct code will be added to and linked to **GameObjectStruct and its members**.

At present the code up to this point would only enable us to display a **single Invader, perhaps on each row** but we need several on screen all at once. What is needed as an **array** of aliens and so we use an array to **store** many of these aliens.
Line 44: **AlienStruct Alien[NUM_ALIEN_COLUMNS][NUM_ALIEN_ROWS];** This creates NUM_ALIEN_COLUMNS across by NUM_ALIEN_ROWS down and in our case that's 7 columns across by 3 rows down. At present though the X and Y values could contain random 0's or at best all be set to 0, which is not what we want, we need a routine **to initialise the aliens to their starting positions on screen.**

Step 5 INITIALISE ALIENS TO STARTING POINTS ON SCREEN

Type the following code into line 77-84

```
77. void InitAliens(int YStart) {
78.   for (int across = 0; across < NUM_ALIEN_COLUMNS; across++) { // plots all 21 aliens using the array by ploting each alien across repeatedly
79.     for (int down = 0; down < 3; down++) {                   //plots each alien down repeatedly.....we add down to centralise the aliens
80.       Alien[across][down].Ord.X = X_START_OFFSET + (across * (LARGEST_ALIEN_WIDTH + SPACE_BETWEEN_ALIEN_COLUMNS)) - down; // this spaces each alien in the array on x
   axis get in correct position using #defineOFFSETs, -down helps centrilse them
81.       Alien[across][down].Ord.Y = YStart + (down * SPACE_BETWEEN_ROWS); // calculation to space aliens on Y axis
82.     }
83.   }
84. }
```

Explanation (line 77-84)

Line77: **void** InitAliens(**int** YStart) { A **void function** simply means that it runs some code (like any other **function**). The function must have a name, InitAliens and declares a local variable, **int** YStart, which can only be used in this function. This function is used in Void setup, **Line 49: InitAliens(0);** // See voidInitAliens. Initialises the aliens and sets up their X/Y Co-ordinates.
Line 78 is a **for statement**, which declares an integer across = 0. Then 0 is compared to **Num_ALIEN_COLUMNS** (which is define in line 10 as 7). across++ increases across by 1 until it is equal to **Num_ALIEN_COLUMNS.** The same approach is used in **line 79** but instead of using a predefined value the value 3 is included, to help centralise the graphics.
The above code essentially give the across (X) and down (Y) position for each individual Alien.

Line 80 **Alien[across][down]** is an array used to locate each individual Alien, **.Ord.X** give the Alien its X co-ordinate which is calculated from the following variables (defined in constant line 10-15) = **X_START_OFFSET + (across * (LARGEST_ALIEN_WIDTH + SPACE_BETWEEN_ALIEN_COLUMNS)) - down;**
Line 81 **Alien[across][down]** is an array used to locate each individual Alien **.Ord.Y** give the Alien its Y co-ordinate which is calculated from the following variables (defined in Function (line 49 and defined constant line 10-15) = = YStart + (down * SPACE_BETWEEN_ROWS);

All the above code does is set the original positions on screen for all the invaders using a combination of the constants defined earlier in the code, some new variables and an array.

Step 6 ALIEN WIDTH ARRAY

Type the following code into line 1-6

```
45. byte AlienWidth[] = {8, 11, 12}; // top middle and bottom widths of the graphics
46.
```

Explanation (line 1-6)

The AlienWidth Array Linked to Update Display Code for Spacing the Bitmaps
All Invaders have a height of 8, so this can be fixed in code. However their width varies on their type but every row is the same type. We could have stored within the alien structure a variable that stated the width of this particular alien/Invader and normally I would do it that way. However with an eye on the limited variable storage we have on the Arduino processor, we can see that each row of Invaders is of the same type, so we only need to store the width information for an invader on a per row basis. So this 3 item array is populated with the 3 width values for each row saving around 18 bytes for our particular number of aliens. If you had more columns you would save more. This array was defined and populated here:

Step 7 VOID SETUP & LOOP

Type the following code into line 47-54

```
47. void setup() { // put your setup code here, to run once:
48.   arduboy.begin();
49.   InitAliens(0); // See voidInitAliens. initialises the aliens and sets up their X/Y Co-ordinates
50. }
51.
52. void loop() { // put your main code here, to run repeatedly:
53.   UpdateDisplay(); // a function to constantly update the display
54. }
```

Explanation (line 47-54)

Line 48 arduboy.begin(). Is used to initialize the arduboy library, you must call its *begin()* function. This is usually done at the start of the sketch's *setup()* function.
Line 49 **InitAliens(0)** is a function to initialise/organise the aliens and set up their X/Y Co-ordinates. InitAliens is the function name and 0 in the brackets is a variable value. Segmenting code into functions allows a programmer to create modular pieces of code that perform a defined task and then return to the area of code from which the function was "called".
Line 53 **UpdateDisplay() {** is a function which is used to constantly update the display by drawing the aliens in their respective positions, updating them frame by frame..

Step 8 DISPLAYING

Type the following code into line 56 -74

```
56. void UpdateDisplay() {
57.   arduboy.clear();// clears the display every time
58.   for (int across = 0; across < NUM_ALIEN_COLUMNS; across++)
59.   {
60.     for (int down = 0; down < NUM_ALIEN_ROWS; down++)
61.     {
62.       switch (down)
63.       {
64.         case 0:// top row
65.           arduboy.drawBitmap(Alien[across][down].Ord.X, Alien[across][down].Ord.Y, InvaderTopGfx, AlienWidth[down], 8, WHITE); // goes to Alien width [down]for 8,11,
    12, the 8 is height.
66.           break;
67.         case 1: // middle row
68.           arduboy.drawBitmap(Alien[across][down].Ord.X, Alien[across][down].Ord.Y, InvaderMiddleGfx, AlienWidth[down], 8, WHITE);
69.           break;
70.         default: // bottom row
71.           arduboy.drawBitmap(Alien[across][down].Ord.X, Alien[across][down].Ord.Y, InvaderBottomGfx, AlienWidth[down], 8, WHITE);
72.       }
73.     }
74.   }
```

Explanation (line 1-6)

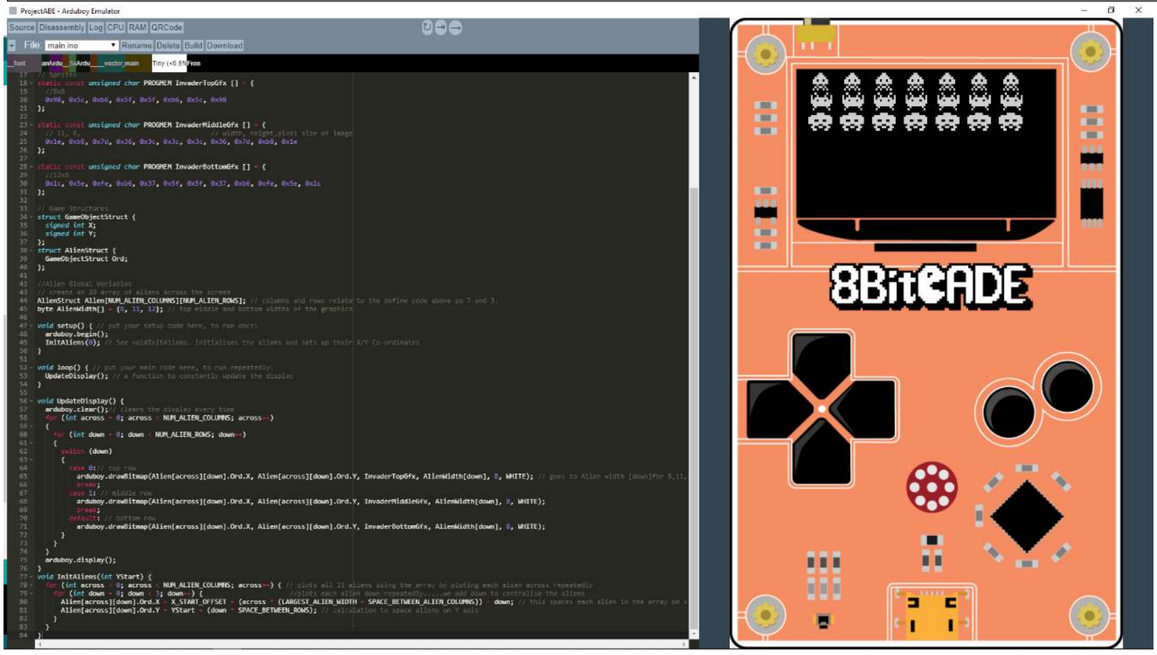
Line 56: Void UpdateDisplay() { is the function with its accompanying code, within the curly brackets to execute.
Arduboy.clear, clears the screen each loop.
Line 58 – 60 The first **for statement** correctly plots the invaders, incrementing the position until the condition in the statement is false, that 0 is the same as NUM_ALIEN_COLUMNS or the next line ROWS.
Line 62 Introduces a Switch Case Statement. Switch Case statements allow you to choose between several discrete options based on information received. Line 62 uses a Switch Statement but only with 3 options and they are switched between them in order. Line 62 is a switch statement (down) means move down along the switch statement in order, executing the code, case by case (Case 0. Case 1, Default).
Line 65-71 **arduboy.drawBitmap**, a function to draw in the arduboy library, the variable **Alien,** using the spacing **across and down** on the OLED screen starting at co-ordinates **Ord.X** and **Ord.Y,** the **graphic InvaderTopGfx,** which is variable **AlienWidth (wide) and 8, high,** coloured **white.**

Showing them on the screen is a relatively simple matter of just looping through the alien array and plotting them at their XY coordinates. The code above achieves this and is called in the main Arduino loop. The code plots invaders from left to right and then top to bottom, so the top left Invader is plotted first then the next one along until we reach the right most Invader. We then move down to the next row and start from the left again. Within the loop a decision is made as to which invader to actually plot. We have three rows and a different Invader per row so all we do within the plotting part of the code is check which row we are plotting and plot the appropriate Invader. The **switch** statement does this.

Final Code

```
/* Delete code from lesson 1 int XPos = 0; // integer for the initial position of the sprite + All of Void Loop
  www.pixilart.com for sprite generation
  http://javl.github.io/image2cpp/ for image conversion
  This code displays 3 types of space invader
*/
//libraries for graphics/display
#include <Arduboy2.h>
Arduboy2 arduboy;
//Sprite settings see graphic for explanation. We have 3 aliens
#define NUM_ALIEN_COLUMNS 7               //Columns of aliens going across
#define NUM_ALIEN_ROWS 3                  //Rows of aliens going down
#define SPACE_BETWEEN_ALIEN_COLUMNS 5     //Space in pixels between each alien in the columns
#define SPACE_BETWEEN_ROWS 9              //Space in pixels between each alien in the rows
#define LARGEST_ALIEN_WIDTH 11            //Size in pixels of the largest allien width
#define X_START_OFFSET 6                  //Position of the first alien from in X,from the 0 position

// Sprites
static const unsigned char PROGMEM InvaderTopGfx [] = {
  //8x8
  0x98, 0x5c, 0xb6, 0x5f, 0x5f, 0xb6, 0x5c, 0x98
};

static const unsigned char PROGMEM InvaderMiddleGfx [] = {
  // 11, 8,                                 // width, height,pixel size of image
  0x1e, 0xb8, 0x7d, 0x36, 0x3c, 0x3c, 0x3c, 0x36, 0x7d, 0xb8, 0x1e
};

static const unsigned char PROGMEM InvaderBottomGfx [] = {
  //12x8
  0x1c, 0x5e, 0xfe, 0xb6, 0x37, 0x5f, 0x5f, 0x37, 0xb6, 0xfe, 0x5e, 0x1c
};

// Game Structures
struct GameObjectStruct {
  signed int X;
  signed int Y;
};
struct AlienStruct {
  GameObjectStruct Ord;
};

//Alien Global Variables
// create an 2D array of aliens across the screen
AlienStruct Alien[NUM_ALIEN_COLUMNS][NUM_ALIEN_ROWS]; // columns and rows relate to the define code above so 7 and 3.
byte AlienWidth[] = {8, 11, 12}; // top middle and bottom widths of the graphics

void setup() { // put your setup code here, to run once:
  arduboy.begin();
  InitAliens(0); // See voidInitAliens. initialises the aliens and sets up their X/Y Co-ordinates
}
```

```cpp

void loop() { // put your main code here, to run repeatedly:
  UpdateDisplay(); // a function to constantly update the display
}

void UpdateDisplay() {
  arduboy.clear();// clears the display every time
  for (int across = 0; across < NUM_ALIEN_COLUMNS; across++)
  {
    for (int down = 0; down < NUM_ALIEN_ROWS; down++)
    {
      switch (down)
      {
        case 0:// top row
          arduboy.drawBitmap(Alien[across][down].Ord.X, Alien[across][down].Ord.Y, InvaderTopGfx, AlienWidth[down], 8, WHITE); // goes to Alien width [down]for 8,11,12, the 8 is height.
          break;
        case 1: // middle row
          arduboy.drawBitmap(Alien[across][down].Ord.X, Alien[across][down].Ord.Y, InvaderMiddleGfx, AlienWidth[down], 8, WHITE);
          break;
        default: // bottom row
          arduboy.drawBitmap(Alien[across][down].Ord.X, Alien[across][down].Ord.Y, InvaderBottomGfx, AlienWidth[down], 8, WHITE);
      }
    }
  }
  arduboy.display();
}
void InitAliens(int YStart) {
  for (int across = 0; across < NUM_ALIEN_COLUMNS; across++) { // plots all 21 aliens using the array by ploting each alien across repeatedly
    for (int down = 0; down < 3; down++) {                      //plots each alien down repeatedly.....we add down to centralise the aliens
      Alien[across][down].Ord.X = X_START_OFFSET + (across * (LARGEST_ALIEN_WIDTH + SPACE_BETWEEN_ALIEN_COLUMNS)) - down; // this spaces each alien in the array on x axis get in correct position using #defineOFFSETs, -down helps centrilse them
      Alien[across][down].Ord.Y = YStart + (down * SPACE_BETWEEN_ROWS); // calculation to space aliens on Y axis
    }
  }
}
```



```cpp
// Sprites
static const unsigned char PROGMEM InvaderTopGfx [] = {
  //8x8
  0x98, 0x5c, 0xb6, 0x5f, 0x5f, 0xb6, 0x5c, 0x98
};

static const unsigned char PROGMEM InvaderMiddleGfx [] = {
  // 11, 8,                         // width, height,pixel size of image
  0x1e, 0xb8, 0x7d, 0x36, 0x3c, 0x3c, 0x3c, 0x36, 0x7d, 0xb8, 0x1e
};

static const unsigned char PROGMEM InvaderBottomGfx [] = {
  //12x8
  0x1c, 0x5e, 0xfe, 0xb6, 0x37, 0x5f, 0x5f, 0x37, 0xb6, 0xfe, 0x5e, 0x1c
};

// Game Structures
struct GameObjectStruct {
  signed int X;
  signed int Y;
};
struct AlienStruct {
  GameObjectStruct Ord;
};

//Alien Global Variables
// create an 2D array of aliens across the screen
AlienStruct Alien[NUM_ALIEN_COLUMNS][NUM_ALIEN_ROWS]; // columns and rows relate to the define code above so 7 and 3.
byte AlienWidth[] = {8, 11, 12}; // top middle and bottom widths of the graphics

void setup() { // put your setup code here, to run once:
  arduboy.begin();
  InitAliens(0); // See voidInitAliens. initialises the aliens and sets up their X/Y Co-ordinates
}

void loop() { // put your main code here, to run repeatedly:
  UpdateDisplay(); // a function to constantly update the display
}

void UpdateDisplay() {
  arduboy.clear();// clears the display every time
  for (int across = 0; across < NUM_ALIEN_COLUMNS; across++)
  {
    for (int down = 0; down < NUM_ALIEN_ROWS; down++)
    {
      switch (down)
      {
        case 0:// top row
          arduboy.drawBitmap(Alien[across][down].Ord.X, Alien[across][down].Ord.Y, InvaderTopGfx, AlienWidth[down], 8, WHITE); // goes to Alien width [down]for 8,11,
          break;
        case 1: // middle row
          arduboy.drawBitmap(Alien[across][down].Ord.X, Alien[across][down].Ord.Y, InvaderMiddleGfx, AlienWidth[down], 8, WHITE);
          break;
        default: // bottom row
          arduboy.drawBitmap(Alien[across][down].Ord.X, Alien[across][down].Ord.Y, InvaderBottomGfx, AlienWidth[down], 8, WHITE);
      }
    }
  }
  arduboy.display();
}
void InitAliens(int YStart) {
  for (int across = 0; across < NUM_ALIEN_COLUMNS; across++) { // plots all 21 aliens using the array by ploting each alien across repeatedly
    for (int down = 0; down < 3; down++) {                      //plots each alien down repeatedly.....we add down to centralise the aliens
      Alien[across][down].Ord.X = X_START_OFFSET + (across * (LARGEST_ALIEN_WIDTH + SPACE_BETWEEN_ALIEN_COLUMNS)) - down; // this spaces each alien in the array on x
      Alien[across][down].Ord.Y = YStart + (down * SPACE_BETWEEN_ROWS); // calculation to space aliens on Y axis
    }
  }
}
```