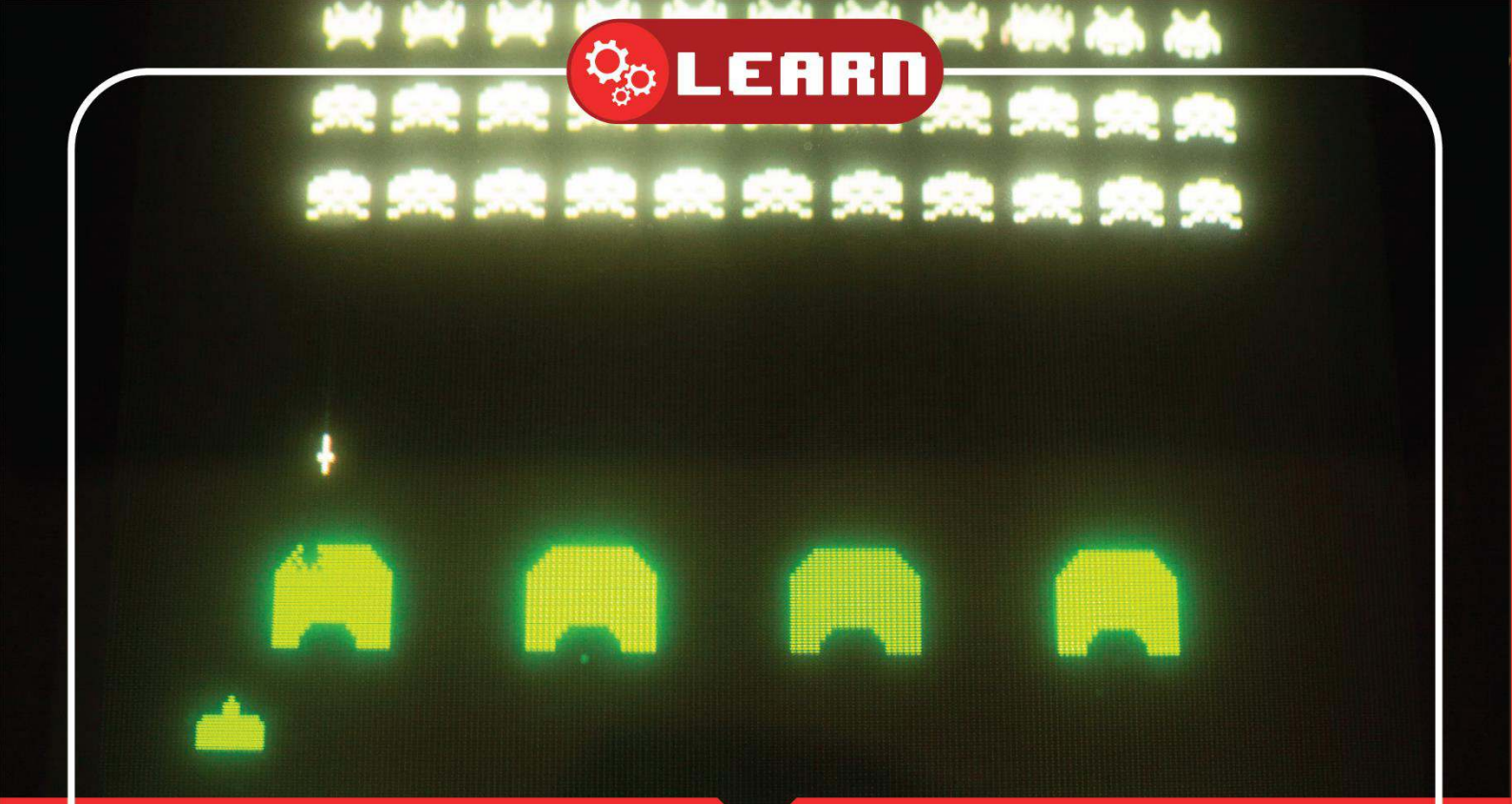
LEARN



CODING SPACE INVADERS PART 5

With 8BitCADE

Compatible with:

8BitCADE XL

AND

8BitCADE

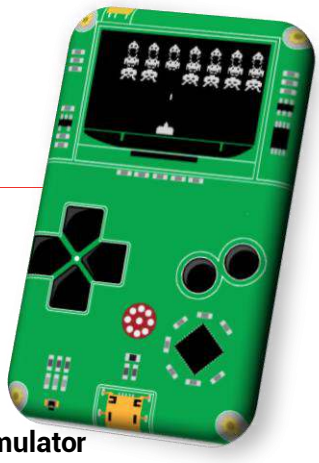
Contents

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: 5



Lesson Title: Adding Missiles and Collisions

Code: Full Code for Lesson

System: Arduboy + Project ABE

Prerequisites to completing this tutorial

1. Tutorials 1a,1b,1c,2,3,4
2. Know how to write code in either **Arduino IDE** or **Project ABE online Emulator**

Step 1 ADDING NEW CONSTANT UNDER ALIEN SETTINGS

Type the following code into line 22

```
22. #define INVADER_HEIGHT 8
```

Explanation (line 22)
Which as it says just defines the height of the invaders. All the Space invaders have the same height so this makes things super easy for us.

Step 2 ADDING NEW CONSTANT UNDER PLAYER SETTINGS

Type the following code into line 29-31

```
29. #define MISSILE_HEIGHT 4
30. #define MISSILE_WIDTH 1
31. #define MISSILE_SPEED 1
```

Explanation (line 29-31)
The lines of code define some properties of player missiles. The graphics height and width and also the speed that it travels at, the higher the number the faster. This is basically the number of pixels moved per game "click". A game "click" is just one cycle though all the physics code (remember the *physics* code is the code that implements all the movement/game logic/ collision detection etc.

Step 3 ADDING NEW STATUS TO GAME OBJECTS

Type the following code into line 34

```
34. #define DESTROYED 2
```

Explanation (line 34)
Line 34 adds a new status for game objects such as Invaders which allows us to flag if they have been destroyed and thus shouldn't be displayed etc.

Step 4 CREATING A GLOBAL OBJECT MISSILE

Type the following code into line 16-17

```
89. GameObjectStruct Missile;
```

Explanation (line 17-31)
Line 89 creates a global object for the one missile that the player can fire at a time:

Step 5 UPDATING PHYSICS FUNCTION TO INCLUDE MISSILES AND COLLISIONS

Type the following code into line 16-17

```
111.      void Physics()  {
112.        AlienControl();
113.        PlayerControl();
114.        MissileControl();
115.        CheckCollisions();
116.      }
```

Explanation (line 17-31)
The physics function has been updated to include two new aspects, the control of any fired missile and the collisions.

Step 6 UPDATE PLAYER CONTROL TO INCLUDE MISSILES

Type the following code into line 124-130

```
118.      void PlayerControl()  {
119.        // user input checks
120.        if ((digitalRead(RIGHT_BUT) == 0) & (Player.Ord.X + TANKGFX_WIDTH < SCREEN_WIDTH))
121.          Player.Ord.X += PLAYER_X_MOVE_AMOUNT;
122.        if ((digitalRead(LEFT_BUT) == 0) & (Player.Ord.X > 0))
123.          Player.Ord.X -= PLAYER_X_MOVE_AMOUNT;
124.        if ((digitalRead(FIRE_BUT) == 0) & (Missile.Status != ACTIVE))
125.        {
126.          Missile.X = Player.Ord.X + (6); // offset missile so its in the mideel of the tank
127.          Missile.Y = PLAYER_Y_START;
128.          Missile.Status = ACTIVE;
129.        }
130.      }
```

Explanation (line 124-130)
The firing is a player action, so quite logically this is controlled in the *PlayerControl* function. The first line is making the decision of **If Player has pressed fire AND there isn't an active missile** , if this condition is satisfied then we set up the firing of the missile. **Line 208** sets the missiles X position to the middle of the players tank (where the barrel is). **Line 209** sets the missiles Y to the players Y, and so the missile is ready to launch from the players tank barrel. We then mark the status of the missile to *Active*. This will be used by several routines, including this one as just noted at the start of the paragraph. If it isn't active for example then the display will not plot it to screen and if active then you are not allowed to fire any more missiles.

Step 7 ADDING MISSILE CONTROL

Type the following code into line 132-140

```
132.      void MissileControl()
133.      {
134.        if (Missile.Status == ACTIVE)
135.        {
136.          Missile.Y -= MISSILE_SPEED;
137.          if (Missile.Y + MISSILE_HEIGHT < 0)  // If off top of screen destroy so can be used again
138.            Missile.Status = DESTROYED;
139.        }
140.      }
```

Explanation (line 132-140)
This is a very simple piece of code which basically just changes the missiles position according to the speed constant mentioned earlier. It also checks if it's gone off the top of the screen(Y less than 0). If so it marks the missiles *status* as destroyed and then another can be fired by the player.

Step 8 CHECK COLLISIONS

Type the following code into line 16-17

```
171.      void CheckCollisions()
172.      {
173.        MissileAndAlienCollisions();
174.      }
175.
176.      void MissileAndAlienCollisions()
177.      {
178.        for (int across = 0; across < NUM_ALIEN_COLUMNS; across++)
179.        {
180.          for (int down = 0; down < NUM_ALIEN_ROWS; down++)
181.          {
182.            if (Alien[across][down].Ord.Status == ACTIVE)
183.            {
184.              if (Missile.Status == ACTIVE)
185.              {
186.                if (Collision(Missile, MISSILE_WIDTH, MISSILE_HEIGHT, Alien[across][down].Ord, AlienWidth[down], INVADER_HEIGHT))
187.                {
188.                  // missile hit
189.                  Alien[across][down].Ord.Status = DESTROYED;
190.                  Missile.Status = DESTROYED;
191.                }
192.              }
193.            }
194.          }
195.        }
196.      }
```

Explanation (line 17-31)

This goes through all the Invaders and for those that are still *Active* it checks if the missile and this Invader are in collision, to do that it uses this line;

if(Collision(Missile,MISSILE_WIDTH,MISSILE_HEIGHT,Alien[across][down].Ord,AlienWidth[down],INVADER_HEIGHT))

This is just calling a function called ***Collision*** which takes 6 parameters (arguments). The first three are for the first of the objects that may be in collision, the next three are for the second object that may be colliding with the first object. If they are colliding it returns true else false. If true then the above routine marks this particular Invader as destroyed and marks the missile as destroyed also. At this point I could have coded the routine to end as the missile can only be in collision with one object at a time, but I didn't. The game runs fast enough and it's not an issue but it could be an area of improvement if you wished to do this.

Step 9 COLLISIONS

Type the following code into line 16-17

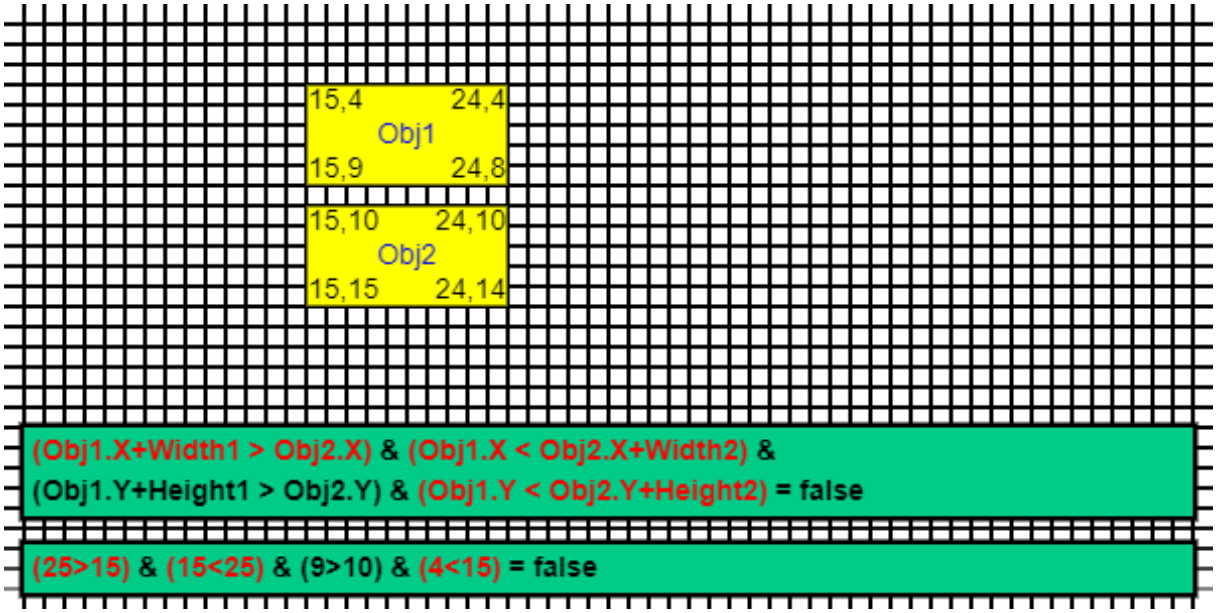
```
197.      bool Collision(GameObjectStruct Obj1, unsigned char Width1, unsigned char Height1, GameObjectStruct Obj2, unsigned char Width2, unsigned char Height2)
198.      {
199.        return ((Obj1.X + Width1 > Obj2.X) & (Obj1.X < Obj2.X + Width2) & (Obj1.Y + Height1 > Obj2.Y) & (Obj1.Y < Obj2.Y + Height2));
200.      }
```

Explanation (line 17-31)

The collision function is a single line of code that returns true if two objects are in collision or false otherwise. It takes just two coordinate structures for the objects in question. It also takes the width and height of the two objects in question. With this information it can be worked out if these two objects are overlapping (in collision).

Note: Usually we would have a structure that included the coordinates and the width/depth of the object all in one, however this elegance has been compromised in order to save memory on the memory challenged processor used on our Arduino's. To put this into perspective even some of those early 1980's home computers had more available memory than we have at our disposal with the Atmel processor we are using. However this processor was never designed to be the equivalent of a general purpose desktop computer! The code above is relatively simple (being only 1 line) but sometimes the code can be hard to imagine what it's doing in your head. So below I am using an example of this exact code. There are 4 conditions that must be met for a collision

to have taken place, when any one of these conditions are true it highlights in red. When they are all true you will see the all conditions in red and the only time this happens is when the two objects overlap. Underneath the code I've also shown that actual numbers for each of the four conditions. Have a play with the controls to move object 1 whilst looking at the code and the coordinates as they change.



Live link: https://www.xtronical.com/projects/space-invaders/parts-1-7/part-5-player-missiles-collisions/

Step 10 DISPLYING THE MISSILE

Type the following code into line 277-279

```
277.          //missile
278.          if (Missile.Status == ACTIVE)
279.            arduboy.drawBitmap(Missile.X, Missile.Y,  MissileGfx, MISSILE_WIDTH, MISSILE_HEIGHT, WHITE);
280.          arduboy.display();
281.        }
```

Explanation (line 277-279)
We have some very small additional code to our display routine which shouldn't need any explanation, the code is covered in previous tutorials.

Step 11 INITIALISING THE MISSILE

Type the following code into line 286

```
283.        void InitPlayer()  {
284.          Player.Ord.Y = PLAYER_Y_START;
285.          Player.Ord.X = PLAYER_X_START;
286.          Missile.Status = DESTROYED;
287.        }
```

Explanation (line 286)
The missile is set up prior to a game starting in the **InitPlayer** routine. All we do is simply mark the missiles status as currently destroyed.

Final Code

```
/*   www.pixilart.com for sprite generation
  http://javl.github.io/image2cpp/ for image conversion
*/
#include <Arduboy2.h>
Arduboy2 arduboy;
// DISPLAY SETTINGS
#define SCREEN_WIDTH 128
#define SCREEN_HEIGHT 64
// Input settings
#define FIRE_BUT 7
#define RIGHT_BUT A1
#define LEFT_BUT A2
// Alien Settings
#define NUM_ALIEN_COLUMNS 7
#define NUM_ALIEN_ROWS 3
#define X_START_OFFSET 6
#define SPACE_BETWEEN_ALIEN_COLUMNS 5
#define LARGEST_ALIEN_WIDTH 11
#define SPACE_BETWEEN_ROWS 9
#define INVADERS_DROP_BY 4            // pixel amount that invaders move down by
#define INVADERS_SPEED 20             // speed of movement, lower=faster.
#define INVADER_HEIGHT 8
// Player settingsc
#define TANKGFX_WIDTH 13
#define TANKGFX_HEIGHT 8
#define PLAYER_X_MOVE_AMOUNT 1
#define PLAYER_Y_START 56
#define PLAYER_X_START 0
#define MISSILE_HEIGHT 4
#define MISSILE_WIDTH 1
#define MISSILE_SPEED 1
// Status of a game object constants
#define ACTIVE 0
#define DESTROYED 2
// graphics
// aliens
const unsigned char InvaderTopGfx [] PROGMEM = {
  0x98, 0x5c, 0xb6, 0x5f, 0x5f, 0xb6, 0x5c, 0x98
};
const unsigned char InvaderTopGfx2 [] PROGMEM = {
  0x58, 0xbc, 0x16, 0x1f, 0x1f, 0x16, 0xbc, 0x58
};
const unsigned char PROGMEM InvaderMiddleGfx [] ={
  0x1e, 0xb8, 0x7d, 0x36, 0x3c, 0x3c, 0x3c, 0x36, 0x7d, 0xb8, 0x1e
};
const unsigned char PROGMEM InvaderMiddleGfx2 [] = {
  0x78, 0x18, 0x7d, 0xb6, 0xbc, 0x3c, 0xbc, 0xb6, 0x7d, 0x18, 0x78
};
const unsigned char PROGMEM InvaderBottomGfx [] = {
  0x1c, 0x5e, 0xfe, 0xb6, 0x37, 0x5f, 0x5f, 0x37, 0xb6, 0xfe, 0x5e, 0x1c
};
const unsigned char PROGMEM InvaderBottomGfx2 [] = {
  0x9c, 0xde, 0x7e, 0x36, 0x37, 0x5f, 0x5f, 0x37, 0x36, 0x7e, 0xde, 0x9c
};
// Player grafix
const unsigned char PROGMEM TankGfx [] = {
  0xf0, 0xf8, 0xf8, 0xf8, 0xf8, 0xfe, 0xff, 0xfe, 0xf8, 0xf8, 0xf8, 0xf8, 0xf0
};
static const unsigned char PROGMEM MissileGfx [] = {
  0x0f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};

```

```
63. // Game structures
64. struct GameObjectStruct  {
65.   // base object which most other objects will include
66.   signed int X;
67.   signed int Y;
68.   unsigned char Status;  //0 active, 1 exploding, 2 destroyed
69. };
70. struct AlienStruct  {
71.   GameObjectStruct Ord;
72. };
73. struct PlayerStruct  {
74.   GameObjectStruct Ord;
75. };
76. //alien global vars
77. //The array of aliens across the screen
78. AlienStruct  Alien[NUM_ALIEN_COLUMNS][NUM_ALIEN_ROWS];
79. // widths of aliens
80. // as aliens are the same type per row we do not need to store their graphic width
    per alien in the structure above
81. // that would take a byte per alien rather than just three entries here, 1 per row,
     saving significnt memory
82. byte AlienWidth[] = {8, 11, 12}; // top, middle ,bottom widths
83. char AlienXMoveAmount = 1; // norm is 2 , this is pixel movement in X
84. signed char InvadersMoveCounter;            // counts down, when 0 move invaders, s
    et according to how many aliens on screen
85. bool AnimationFrame = false; // two frames of animation, if true show one if false
    show the other
86.
87. // Player global variables
88. PlayerStruct Player;
89. GameObjectStruct Missile;
90.
91. void setup()
92. {
93.   arduboy.begin();
94.   arduboy.setFrameRate(60);
95.   InitAliens(0);
96.   InitPlayer();
97.   pinMode(RIGHT_BUT, INPUT_PULLUP);
98.   pinMode(LEFT_BUT, INPUT_PULLUP);
99.   pinMode(FIRE_BUT, INPUT_PULLUP);
100.       }
101.
102.       void loop()
103.       {
104.         if (!arduboy.nextFrame()) {
105.           return;
106.         }
107.         Physics();
108.         UpdateDisplay();
109.       }
110.
111.       void Physics()  {
112.         AlienControl();
113.         PlayerControl();
114.         MissileControl();
115.         CheckCollisions();
116.       }
117.
118.       void PlayerControl()  {
119.         // user input checks
120.         if ((digitalRead(RIGHT_BUT) == 0) & (Player.Ord.X + TANKGFX_WIDTH < SCREEN
    _WIDTH))
121.           Player.Ord.X += PLAYER_X_MOVE_AMOUNT;
122.         if ((digitalRead(LEFT_BUT) == 0) & (Player.Ord.X > 0))
123.           Player.Ord.X -= PLAYER_X_MOVE_AMOUNT;
```

```
124.         if ((digitalRead(FIRE_BUT) == 0) & (Missile.Status != ACTIVE))
125.         {
126.           Missile.X = Player.Ord.X + (6); // offset missile so its in the mideel o
    f the tank
127.           Missile.Y = PLAYER_Y_START;
128.           Missile.Status = ACTIVE;
129.         }
130.       }
131.
132.       void MissileControl()
133.       {
134.         if (Missile.Status == ACTIVE)
135.         {
136.           Missile.Y -= MISSILE_SPEED;
137.           if (Missile.Y + MISSILE_HEIGHT < 0)  // If off top of screen destroy so
    can be used again
138.             Missile.Status = DESTROYED;
139.         }
140.       }
141.
142.       void AlienControl()
143.       {
144.         if ((InvadersMoveCounter--) < 0)
145.         {
146.           bool Dropped = false;
147.           if ((RightMostPos() + AlienXMoveAmount >= SCREEN_WIDTH) | (LeftMostPos()
     + AlienXMoveAmount < 0)) // at edge of screen
148.           {
149.             AlienXMoveAmount = -
    AlienXMoveAmount;             // reverse direction
150.             Dropped = true;                                    // and indicate we a
    re dropping
151.           }
152.           // update the alien postions
153.           for (int Across = 0; Across < NUM_ALIEN_COLUMNS; Across++)
154.           {
155.             for (int Down = 0; Down < 3; Down++)
156.             {
157.               if (Alien[Across][Down].Ord.Status == ACTIVE)
158.               {
159.                 if (Dropped == false)
160.                   Alien[Across][Down].Ord.X += AlienXMoveAmount;
161.                 else
162.                   Alien[Across][Down].Ord.Y += INVADERS_DROP_BY;
163.               }
164.             }
165.           }
166.           InvadersMoveCounter = INVADERS_SPEED;
167.           AnimationFrame = !AnimationFrame; ///swap to other frame
168.         }
169.       }
170.
171.       void CheckCollisions()
172.       {
173.         MissileAndAlienCollisions();
174.       }
175.
176.       void MissileAndAlienCollisions()
177.       {
178.         for (int across = 0; across < NUM_ALIEN_COLUMNS; across++)
179.         {
180.           for (int down = 0; down < NUM_ALIEN_ROWS; down++)
181.           {
182.             if (Alien[across][down].Ord.Status == ACTIVE)
183.             {
184.               if (Missile.Status == ACTIVE)
```

```
185.               {
186.                 if (Collision(Missile, MISSILE_WIDTH, MISSILE_HEIGHT, Alien[across
    ][down].Ord, AlienWidth[down], INVADER_HEIGHT))
187.                 {
188.                   // missile hit
189.                   Alien[across][down].Ord.Status = DESTROYED;
190.                   Missile.Status = DESTROYED;
191.                 }
192.               }
193.             }
194.           }
195.         }
196.       }
197.       bool Collision(GameObjectStruct Obj1, unsigned char Width1, unsigned char He
    ight1, GameObjectStruct Obj2, unsigned char Width2, unsigned char Height2)
198.       {
199.         return ((Obj1.X + Width1 > Obj2.X) & (Obj1.X < Obj2.X + Width2) & (Obj1.Y
    + Height1 > Obj2.Y) & (Obj1.Y < Obj2.Y + Height2));
200.       }
201.       int RightMostPos()  {
202.         //returns x pos of right most alien
203.         int Across = NUM_ALIEN_COLUMNS - 1;
204.         int Down;
205.         int Largest = 0;
206.         int RightPos;
207.         while (Across >= 0) {
208.           Down = 0;
209.           while (Down < NUM_ALIEN_ROWS) {
210.             if (Alien[Across][Down].Ord.Status == ACTIVE)
211.             {
212.               // different aliens have different widths, add to x pos to get right
    pos
213.               RightPos = Alien[Across][Down].Ord.X + AlienWidth[Down];
214.               if (RightPos > Largest)
215.                 Largest = RightPos;
216.             }
217.             Down++;
218.           }
219.           if (Largest > 0) // we have found largest for this coloum
220.             return Largest;
221.           Across--;
222.         }
223.         return 0;  // should never get this far
224.       }
225.       int LeftMostPos()  {
226.         //returns x pos of left most alien
227.         int Across = 0;
228.         int Down;
229.         int Smallest = SCREEN_WIDTH * 2;
230.         while (Across < NUM_ALIEN_COLUMNS) {
231.           Down = 0;
232.           while (Down < 3) {
233.             if (Alien[Across][Down].Ord.Status == ACTIVE)
234.               if (Alien[Across][Down].Ord.X < Smallest)
235.                 Smallest = Alien[Across][Down].Ord.X;
236.             Down++;
237.           }
238.           if (Smallest < SCREEN_WIDTH * 2) // we have found smalest for this colou
    m
239.             return Smallest;
240.           Across++;
241.         }
242.         return 0;  // should nevr get this far
243.       }
244.
245.       void UpdateDisplay()
```

```
246.       {
247.         arduboy.clear();
248.         for (int across = 0; across < NUM_ALIEN_COLUMNS; across++)
249.         {
250.           for (int down = 0; down < NUM_ALIEN_ROWS; down++)
251.           {
252.             if (Alien[across][down].Ord.Status == ACTIVE) {
253.               switch (down)  {
254.                 case 0:
255.                   if (AnimationFrame)
256.                     arduboy.drawBitmap(Alien[across][down].Ord.X, Alien[across][do
    wn].Ord.Y,  InvaderTopGfx, AlienWidth[down], INVADER_HEIGHT, WHITE);
257.                   else
258.                     arduboy.drawBitmap(Alien[across][down].Ord.X, Alien[across][do
    wn].Ord.Y,  InvaderTopGfx2, AlienWidth[down], INVADER_HEIGHT, WHITE);
259.                   break;
260.                 case 1:
261.                   if (AnimationFrame)
262.                     arduboy.drawBitmap(Alien[across][down].Ord.X, Alien[across][do
    wn].Ord.Y,  InvaderMiddleGfx, AlienWidth[down], INVADER_HEIGHT, WHITE);
263.                   else
264.                     arduboy.drawBitmap(Alien[across][down].Ord.X, Alien[across][do
    wn].Ord.Y,  InvaderMiddleGfx2, AlienWidth[down], INVADER_HEIGHT, WHITE);
265.                   break;
266.                 default:
267.                   if (AnimationFrame)
268.                     arduboy.drawBitmap(Alien[across][down].Ord.X, Alien[across][do
    wn].Ord.Y,  InvaderBottomGfx, AlienWidth[down], INVADER_HEIGHT, WHITE);
269.                   else
270.                     arduboy.drawBitmap(Alien[across][down].Ord.X, Alien[across][do
    wn].Ord.Y,  InvaderBottomGfx2, AlienWidth[down], INVADER_HEIGHT, WHITE);
271.                }
272.              }
273.           }
274.         }
275.         // player
276.         arduboy.drawBitmap(Player.Ord.X, Player.Ord.Y,  TankGfx, TANKGFX_WIDTH, TA
    NKGFX_HEIGHT, WHITE);
277.         //missile
278.         if (Missile.Status == ACTIVE)
279.           arduboy.drawBitmap(Missile.X, Missile.Y,  MissileGfx, MISSILE_WIDTH, MIS
    SILE_HEIGHT, WHITE);
280.         arduboy.display();
281.       }
282.
283.       void InitPlayer()  {
284.         Player.Ord.Y = PLAYER_Y_START;
285.         Player.Ord.X = PLAYER_X_START;
286.         Missile.Status = DESTROYED;
287.       }
288.
289.       void InitAliens(int YStart)  {
290.         for (int across = 0; across < NUM_ALIEN_COLUMNS; across++)  {
291.           for (int down = 0; down < 3; down++)  {
292.             // we add down to centralise the aliens, just happens to be the right
    value we need per row!
293.             // we need to adjust a little as row zero should be 2, row 1 should be
     1 and bottom row 0
294.             Alien[across][down].Ord.X = X_START_OFFSET + (across * (LARGEST_ALIEN_
    WIDTH + SPACE_BETWEEN_ALIEN_COLUMNS)) - (AlienWidth[down] / 2);
295.             Alien[across][down].Ord.Y = YStart + (down * SPACE_BETWEEN_ROWS);
296.           }
297.         }
298.       }
```