Lets Make A Retro Game

Episode 10 – Collision Detection

In this episode we are going to add some simple collision detection so that the bullets our ship is firing are able to hit and destroy the asteroids that are falling down the screen.

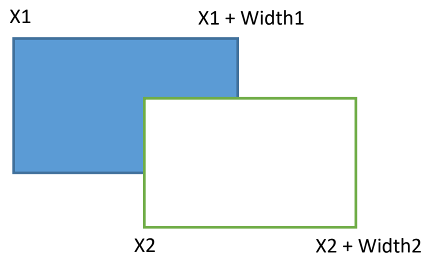
At this stage we only have one type of enemy, which is at a fixed size, but later we will have different sized asteroids as well as some other enemy types so we need to allow for different size enemy shapes.

Step One – Object Collision Function

Our first step is to write a function that we can call that works out whether two objects of different sizes have hit each other.

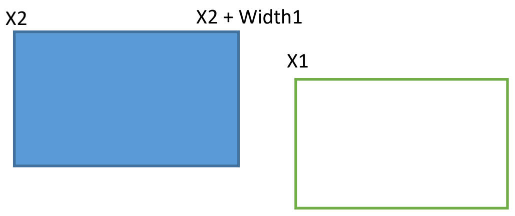
Now how do we check two objects have hit each other, well it requires a little bit of maths, not too complicated, but very important.

In interests of simplicity, and processing time, I always treat objects on 8-bit systems as rectangles. What we are trying to do is work out whether two rectangles intersect as follows:



So in this example 1st step is we see our 2nd object plus it's width (don't worry about X and Y at this stage, it's the same both ways, just with different values), is less than the position of the 1st object i.e. the 2nd object is completely to the left of the 1st object.

If it is they can't be hitting so we exit.



```
LD A,(IX+0)
SUB E
CP (IY+0)
JR NC,NOHIT
```

So the observant may say, hey you are subtracting the width of the 2nd object from the 1st one, not adding it to the position of the 2nd one? Well we are, just utilising a bit of a maths trick i.e.

X1 = X2 + width

Is the same as

X1 – width = X2

As we subtracted width from both sides.

Next, we see if the 1st object plus it's width, is less than the position of the 2nd object, just like the first step if this is true then they can't be hitting so we exit.

```
ADD A,E ; get our original value back
ADD A,L
```

```
      CP (IY+0)
      JR C,NOHIT
```

So after we have done that we have finished in one direction so the remainder of the code just repeats for the other coordinate and widths.

```
      LD A,(IX+1)
      SUB D
      CP (IY+1)
      JR NC,NOHIT
      ADD A,D ; get our original value back
      ADD A,H
      CP (IY+1)
      JR C,NOHIT
```

So if we make it through this section of code, the two objects have hit each other, so we set the carry flag and return.

```
      SCF
      RET
```

Otherwise we clear the carry flag and return.

```
NOHIT:
      XOR A
      RET
```

So our complete function is as follows:

```
; ==================================================
; Test whether two objects are colliding
; ==================================================
; IX+0 = 1st object Y
; IX+1 = 1st object X
; IY+0 = 2nd object Y
; IY+1 = 2nd object Y
; D = 2nd object width
; E = 2nd object height
; H = 1st object width
; L = 1st object height
; ==================================================
; Result: Carry flag set if two objects collide
; ==================================================
COLTST:
      LD A,(IX+0)
      SUB E
      CP (IY+0)
      JR NC,NOHIT
      ADD A,E ; get our original value back
      ADD A,L
      CP (IY+0)
      JR C,NOHIT
      LD A,(IX+1)
      SUB D
      CP (IY+1)
      JR NC,NOHIT
      ADD A,D ; get our original value back
      ADD A,H
```

```
        CP (IY+1)
        JR C,NOHIT
        SCF
        RET
NOHIT:
        XOR A
        RET
```

Step Two – Bullets Hitting Enemies

So now that we have a collision detection routine, let's put it to use and see if we can get our bullets hitting the falling meteors.

This requires a little planning, especially where we are planning a game where there can be quite a few objects moving on the screen, the last thing we want is for things to start slowing down.

So as the enemies will always be the largest number of things that will be on screen, it's best that we only loop around them once, and inside that loop check whether they are hitting any of our other objects.

Find the existing MOVE_ENEMIES: function the section of code where the meteors are removed as they hit the bottom of the screen i.e. the label ME3:.

```
    ; clear enemy data
    XOR A
    LD (HL),A
    ; clear sprite
    LD A,209
ME3:
    LD (IX+0),A

ME2:
```

For us to insert new code after the ME3: label we need to make sure the 'clear sprite' code just before skips directly to ME2:

```
    ; clear enemy data
    XOR A
    LD (HL),A
    ; clear sprite
    LD A,209
    LD (IX+0),A
    JR ME2
ME3:
    LD (IX+0),A
```

So we are repeating the store of the sprite Y value and jumping to ME2 to continue our loop.

Next we need to add some code to check whether a player bullet is on screen before we do a check for a collision.

```
ME3:
    LD (IX+0),A

    ; enemy object has been moved now do collision detection
    LD IY,SPRTBL+8 ; bullet y position
    LD A,(IY+0)
    CP 209 ; check that it is on screen
    JR Z,ME2

ME2:
```

Now we can setup and do our call to the collision test routine. As we need to use both HL and DE to pass our parameters, we save them to the stack using PUSH (remembering to POP them off when we are finished). And then call the COLTST routine.

```
ME3:
    LD (IX+0),A

    ; enemy object has been moved now do collision detection
    LD IY,SPRTBL+8 ; bullet y position
    LD A,(IY+0)
    CP 209 ; check that it is on screen
    JR Z,ME2
    PUSH HL ; save values so we can use the registers
    PUSH DE
    LD HL,0E0Eh ; set our meteor size at 14x14 - will change per enemy type
later
    LD DE,0208h ; set our bullet size at 2x8
    CALL COLTST
    POP DE
    POP HL
    JR NC,ME2
    ; we have a hit, for the moment just make both objects disappear
    LD A,209
    LD (IY+0),A
    LD (IX+0),A
    XOR A
    LD (HL),A ; deactive the enemy
    ; later we will:
    ; - increase the score
    ; - explosion sound
    ; - animate enemy

ME2:
```

So compiling and running that should now have our bullets being able to hit and remove the meteors.

Next Episode

Next time we will be adding more collision detection and adding some scoring.