Lets Make A Retro Game

Episode 9 – Enemy Movement

In the Intellivision game of Astro Smash, that we are basing our game on, the main enemies are Asteroids that fall down the screen, exploding when they hit the ground.
Points are awarded to the player, when one is destroyed and points are removed if one reaches the planet surface.
In this episode we are going to introduce some simple enemy movement, this will involve:

- Using a random number generator to determine when an asteroid should appear and at what speed.
- Moving any visible asteroids towards the bottom of the screen.
- Detecting when an asteroid reaches the bottom of the screen.

Step One – Asteroid Generation

Our first step is to determine when new asteroids should appear at the top of the screen, to do this we use a random number generator, along with our current difficulty level to work out whether one appears and another random value to work out how far across the screen it will appear.
The supplied template code includes a random number function that uses a time counter, which starts when the computer/console first starts.
Randomness is introduced in the amount of time between starting up and when the user presses a button on the controller to start the game.
This initial time value is called the Seed of our random number sequence of numbers. Yes, that's right, computer based random numbers are mostly based on a sequence of numbers that are generated from a set of calculations.
As computer chips have become more complex, random number generation has been able to produce larger sets of random numbers, but for our 8-bit game we don't need anything too complex, just random enough to introduce some uncertainty in enemy behaviour to the player.

Random Number Functions

There are two functions defined in the supplied template library as follows:
SEED_RANDOM – The random number Seed value in the register HL is stored in memory as a 32-bit number.
RND – This returns the next number (from 0 to 255) in the random number sequenced based on the current Seed value.

Setup Code

In our current code there are already two parts of our random number setup code, just after the START: tag as follows:

```
START:

    …

    ; Seed random numbers with a fixed number (nothing else to use?)
    LD HL,1967
    CALL SEED_RANDOM
```

This sets the seed at a known value, rather than just being zero, for any random numbers needed during our title screen and setup phase.
Then just after the MAIN_SCREEN: tag, we set the Seed to the amount of time that has passed since the game code started and when the player presses the fire button as follows:

```
MAIN_SCREEN:
    ; Read joysticks to clear any false reads
    CALL JOYTST

    LD HL,(TIME)
    LD (SEED),HL
    RR H
    RL L
    LD (SEED+2),HL
```

Now before we go too far we need some new areas of Ram set aside, so at the end of our source file we need to add two variables as follows:

```
ORG RAMSTART

LEVEL:        DS 1
LIVES:        DS 1
SCORE:        DS 3
LASTSCORE:    DS 3
ENEMYDATA:    DS 20
ANIMATE:      DS 1
```

These new areas of Ram need to be initialised to zero, so we have a known starting point as follows:

```
; Init Ram for a new game
INITRAM:
    LD A,3
    LD (LIVES),A
    LD HL,0
    LD (SCORE),HL
    LD (SCORE+1),HL
    LD A,1
    LD (LASTSCORE),A
    XOR A
    LD (LEVEL),A
    LD (ANIMATE),A
    ; initialise enemy data
    LD HL,ENEMYDATA
    LD (HL),A
    LD DE,ENEMYDATA+1
    LD BC,19
    LDIR
    RET
```

Generate Enemies

Now we have our random number generator setup, lets add a new function to our main game loop that will spawn new enemies. For the moment, it will just be asteroids, but later on we can add more enemy types.

Add the call to our new function SPAWN_ENEMIES after our other actions calls in our main loop as follows:

```
    ; Main game logic loop
MLOOP:
    ; check that a base tick has occurred
    ; ensures consistent movement speed between 50 & 60Hz systems
    LD      A,(TickTimer)
    CALL    TEST_SIGNAL
    OR      A
    JR Z,MLOOP2
    ; once per tick
    CALL MOVE_PLAYER
    CALL FIRE_PLAYER_BULLET
    CALL MOVE_PLAYER_BULLET
    CALL SPAWN_ENEMIES
MLOOP2:
    LD A,(QtrSecTimer)
    CALL TEST_SIGNAL
    OR A
    JR Z,MLOOP
```

```
    JR MLOOP
```

Now just after our other main loop functions add our SPAWN_ENEMIES function as follows:

```
; Spawn/create new enemies
SPAWN_ENEMIES:
    LD A,(LEVEL)
    INC A
    SLA A ; multiply by 4
    SLA A
    LD C,A
    CALL RND
    CP C
    RET NC
    ; see if there is an enemy object available
    LD HL,ENEMYDATA
    LD B,20
SE1:
    XOR A
    CP (HL)
    JR NZ,SE2
    ; enemy available
    PUSH HL
    ; calc our sprite memory position
    LD A,20
    SUB B
    SLA A
    SLA A
    LD C,A
    LD B,0
    LD HL,SPRTBL+12
    ADD HL,BC
    ; set Y to zero
    LD A,0
    LD (HL),A
    ; set X to a random value
    INC HL
    CALL RND
    LD (HL),A
    ; set pattern
    INC HL
    LD A,8
    LD (HL),A
    ; set colour
    INC HL
    LD A,0dh
    LD (HL),A
    POP DE
    LD A,1
    LD (DE),A
    RET
SE2:
    ; move to the next data position
    INC HL
    ; dec b and jump if non-zero
    DJNZ SE1
    RET
```

This is a bit more complex function than we have gone through previously, so I will step through what it does as follows:

1. Use the current level to work out a low number (that gets higher the higher our level of difficulty)
2. Call our random number generator, which will give us a number between 0 and 255, if that number is less than the number we worked out in step 1, we want to try and create a new enemy.
3. We can't have unlimited enemies on the screen, so I have set a limit of 20 (which should be plenty), and for each one we have a byte of memory set aside to control whether it is on the screen (and later on what type of enemy).
   In this step, we search our list of enemies until we find one that is not currently on screen.
4. We have used two sprites for our main ship, and one sprite for our player bullet, so the next 20 sprites we can use for the enemy objects.
   In this step, we work out the starting address of our local memory holding the definition of the sprite we want to use.
5. We set the Y position to 0 i.e. the top of the screen
6. We set the X position using our random number function i.e. 0-255 (we can adjust this a bit later if it doesn't quite suit).
7. We set our sprite starting pattern (16x16 sprites use four 8x8 patterns)
8. We set our sprite colour (0Dh – which is purple – no browns in most of the Z80 systems)
9. Lastly, we store a 1 in our enemy byte so we know it is in use.

Step Two – Enemy Movement

Our next step is to make the enemy objects that are currently on screen move, to start with we will just make them move straight down the screen.
After our call to SPAWN_ENEMIES add a new call to MOVE_ENEMIES as follows:

```
    CALL SPAWN_ENEMIES
    CALL MOVE_ENEMIES
```

Next add the new MOVE_ENEMIES function as follows:

```
; Move any active enemies
MOVE_ENEMIES:
    LD HL,ENEMYDATA
    LD B,20
ME1:
    XOR A
    CP (HL)
    JR Z,ME2
    ; found active enemy
    ; calc our sprite memory position (20-B) * 4
    PUSH BC
    LD A,20
    SUB B
    SLA A
    SLA A
    LD C,A
    LD B,0
    LD IX,SPRTBL+12
    ADD IX,BC
    POP BC
    ; get our enemy data
    LD A,(HL)
    ; Note: not used for the moment

    LD A,(ANIMATE)
    CP 0
    JR NZ,ME5
    ; animate our sprite pattern (hardwired for the moment)
    LD A,(IX+2) ; get current pattern
    ADD A,4     ; add four
    CP 16
    JR NZ,ME4   ; if we have reached 16 we need to move back to the
original pattern
    LD A,8
ME4:
    LD (IX+2),A ; store the value back
ME5:
    ; get current Y position
    LD A,(IX+0)
    ; fixed increase for now
    INC A
    CP 150
    JR C,ME3
    ; enemy has reached the bottom of the screen
    ; decrease score

    ; explosion?

    ; clear enemy data
    XOR A
    LD (HL),A
```

```
    ; clear sprite
    LD A,209
ME3:
    LD (IX+0),A
ME2:
    INC HL
    DJNZ ME1
    ; adjust our animation timing
    LD A,(ANIMATE)
    DEC A
    JP P,ME6
    LD A,2
ME6:
    LD (ANIMATE),A

    RET
```

Another bit of fairly complicated code, so I will step through it as follows:

1. We setup a loop, so we can work through all of our 20 enemy objects
2. First, we check that an enemy object is on screen by checking that our enemy byte value is not zero
3. Next, we need to work out the sprite memory position so we can look at and adjust the shape and position of the sprite (can you think of a way this could be made more efficient?)
4. Now we get our enemy data byte – in our example this will always be 1, so we don't use it for the moment.
5. I have included a step that changes the sprites pattern each frame, thus producing a slight animation effect.
6. Next, we move the enemy down the screen, simply by adding 1 to the Y value – another thing we can make more complex later.
7. Next, we need to detect when enemies have reached the bottom of the screen, so they can be removed.
   Later on we can add an explosion effect and sound etc.
8. We use the DJNZ command to keep going around our loop until we have processed all of the enemy objects.
9. Next, we adjust our animation byte so we know which pattern to use next time for our enemies – this is probably a bit fast at the moment.

Summary

We have covered a fair bit of ground in this episode, now we can:

- move our ship,
- fire bullets and
- we have our 1st type of enemy being generated and moving on the screen.

Next episode we will cover some basic collision detection, between the players bullet and the enemy objects, and show how to add scoring to our game.