Multi-Threading in assembly

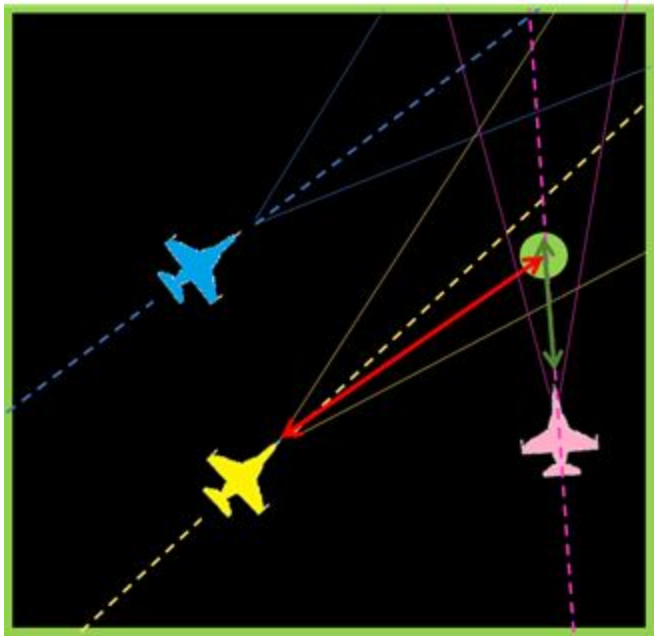
Written **entirely in assembly language**. No C code and no usage of C standard library functions other than those specifically noted below. The only C standard library code is the initialization code at "_start" which calls main( ), usage of printf and fprintf functions to print float point numbers and integers , sscanf in order to convert the command line arguments. malloc( ) and free( ) is also used, to handle dynamic memory allocation.

A simple user-mode co-routine ("multi-thread") simulation, using a co-routine mechanism. The goal is further proficiency in assembly language, especially as related to floating point, and understanding stack manipulations, as needed to create a co-routine (equivalently thread) management scheme.

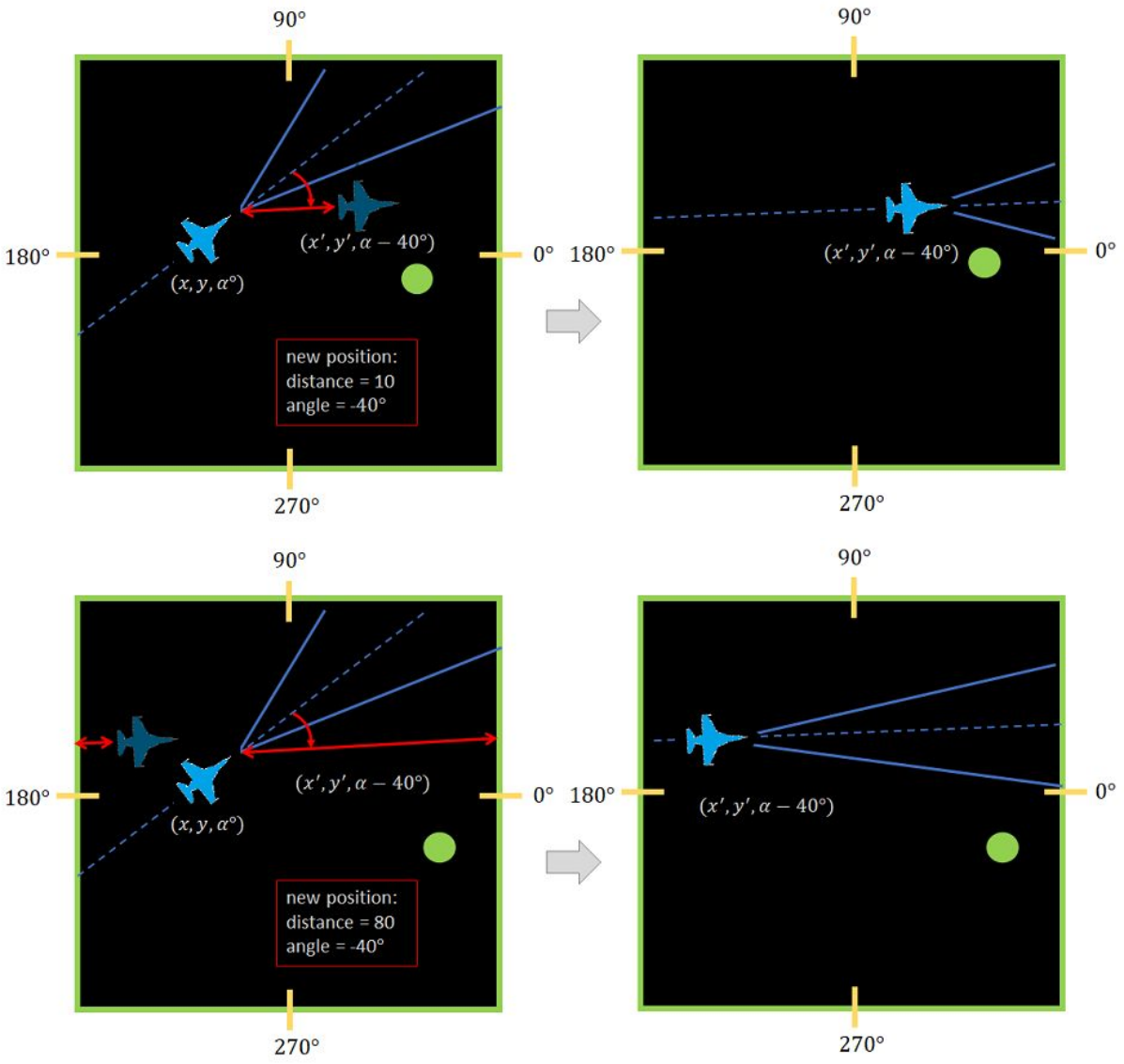
Program Goal

A multi-drones single-target game. Suppose a 100x100 game board. Suppose group of N drones which see the same target from different points of view and from different distance. Each drone tries to detect where is the target on the game board, in order to destroy it. Drones may destroy the target only if the target is in drone's field-of-view, and if the target is no more than some maximal distance from the drone. When the current target is destroyed, some new target appears on the game board in some randomly chosen place. The first drone that destroys T targets is the winner of the game. Each drone has three-dimensional position on the game board: coordinate x, coordinate y, and direction (angle from x-axis). Drones move randomly chosen distance in randomly chosen angle from their current place. After each movement, a drone calls **mayDestroy(…)** function with its new position on the board. **mayDestroy(…)** function returns TRUE if the caller drone may destroy the target, otherwise returns FALSE. If the current target is destroyed, new target is created at random position on the game board. Note that drones do not know the coordinates of the target on the board game. On the example below, the blue drone does not see the target, the yellow drone sees the target, but the target is very far from it, and the pink drone both sees the target and close enough to it. The pink drone can destroy the target.

Please note: the target is just a point, not a circle as it appears in the figures. The painting of a circle is just for the convenience of illustration. Also note that drones are illustrated as airplanes but they do not have all the appropriate functionality of airplanes.

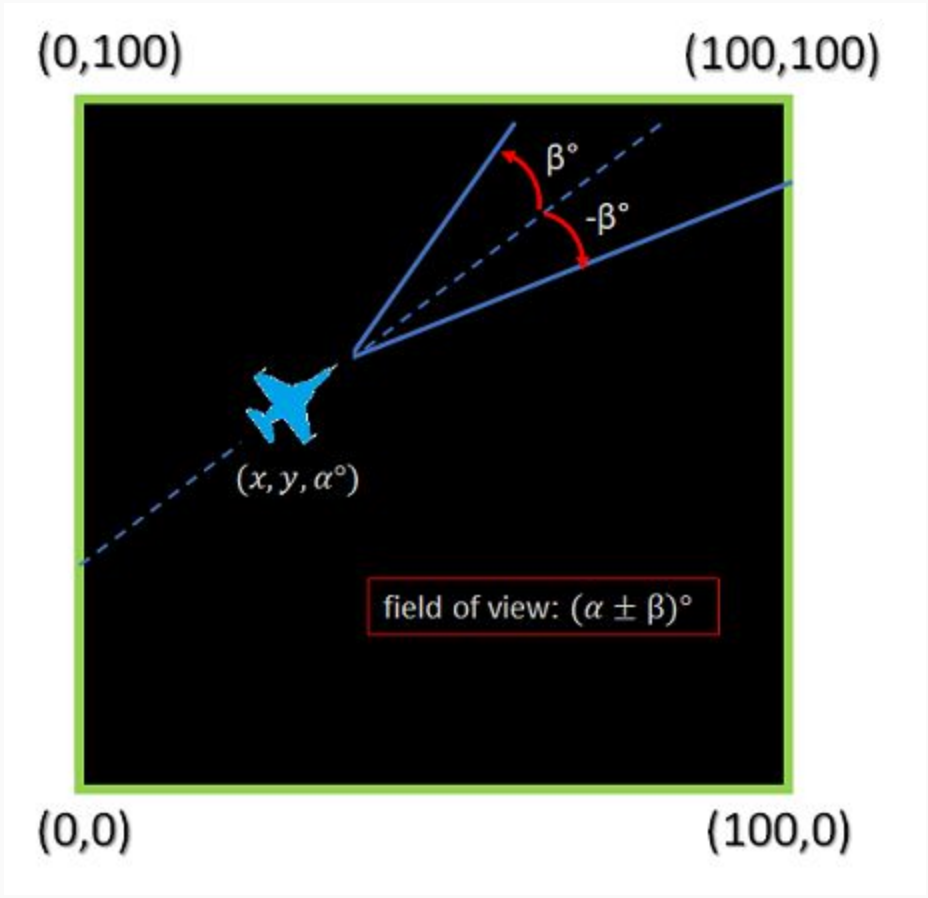


When a drone moves from its current position to a new position, it may happen that the distance move makes the drone cross the game field border. We treat drone motion as if on a torus. That is, when the next location is greater than the board width or height, or is negative in either axis (this requires checking 4 conditions), subtract or add the torus cycle (100 in this example) if needed. On the first figure below, we see a simple movement of the drone, and on the second figure we may see the movement that would move the drone out of the right border, and instead it is "wrapped around" to the left border of the game board.



How a field-of-view of a drone on the game board is calculated

Note that each drone has its current position defined by coordinates x, y, and an angle α: (x,y,α). We need to calculate a field-of-view of a drone in order to detect if a drone may destroy a target. The figure below shows the field-of-view of the drone. We define a field-of-view as two lines of angle ±β from the drone's current angle.



Targes is in drone's field-of-view - Detection

Let the drone be at coordinates (x1, y1) with the current angle alpha. Let the target be at coordinates (x2, y2). Let maximum detection distance be d, and let detection spread angle be beta.
Compute angle

```
gamma = arctan2(y2-y1, x2-x1)
```

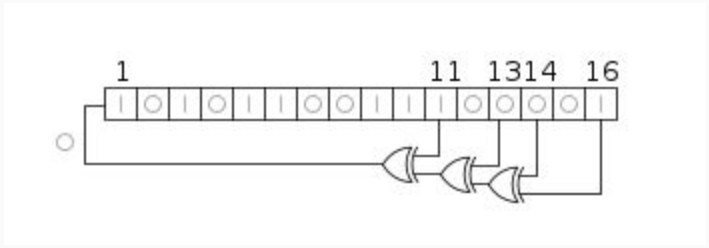
The drone may detect and destroy the target iff:

```
(abs(alpha-gamma) < beta) and sqrt((y2-y1)^2+(x2-x1)^2) < d
```

How to generate a pseudo-random number

Linear-feedback Shift Register method gets a random number in Assembly. A **linear-feedback shift register (LFSR)** is a shift register whose input bit is a linear function of its previous state. LFSR is a shift register whose input bit is driven by the XOR of some

bits of the overall shift register value. The initial value of the LFSR is called the seed, and because the operation of the register is deterministic, the stream of values produced by the register is completely determined by its current (or previous) state. We would use Fibonacci LFSR version. The figure below describes a 16-bit Fibonacci LFSR. The feedback tap numbers shown correspond to a primitive polynomial in the table, so the register cycles through the maximum number of 65535 states excluding the all-zeroes state. The state shown, 0xACE1 will be followed by 0x5670.



The bit positions that affect the next state are called the taps. In the diagram the taps are [16,14,13,11]. The rightmost bit of the LFSR is called the output bit. The taps are XOR'd sequentially with the output bit and then fed back into the leftmost bit. The sequence of bits in the rightmost position is called the output stream. The bits in the LFSR state that influence the input are called taps.

Note that the XOR function is as follows:

A	B	A **XOR** B
0	0	0
0	1	1
1	0	1
1	1	0