Spike Outcome Report

Number: 14

Spike Title: Soldier on Patrol

Personal: Michael Williams (7668481)

**Goals:**

* Create a “soldier on patrol” simulation where an agent has two or more high-level FSM modes of behaviour and low-level FSM behaviour. The model must show (minimum)
  + High level "patrol" and "attack" modes
  + The "patrol" mode must use a FSM to control low-level states so that the agent will visit (seek/arrive?) a number of patrol-path way points.
  + The "attack" mode must use a FSM to control low-level fighting states. (Think “shooting”, “reloading” – the actual states and transition rules are up to you.)

**Technologies, Tools, and Resources used:**

* Python IDE
* Sample Lab Code
* Lecture Material

**Tasks undertaken:**

1. The first step undertaken in this spike was to work out what code we already had that we could recycle. From the spike description we identified that we would need 3 things:
   1. An agent following a path.
   2. An agent that could shoot.
   3. State machines to control these actions.

1. We started off using the code for the Agent Markmanship spike for the baseline. In order to make this fit for purpose we needed to add back in the code to allow the “shooter” to follow a path, as this was stripped out originally. We took this from Spike 6. We then modified the target to be stationary, have a random start position, take two hits to “die” and respawn in a random location.

Now that everything should be working, we tested our code before continuing. While it worked most of the time, there was some issues with the collision detection. This was due to the rudimentary implementation used previously. We added some extra code to improve this slightly and it worked much better:

**def** check\_for\_collision(self):  
 rightmost\_point = Point2D(self.pos.x + self.radius,self.pos.y)  
 leftmost\_point = Point2D(self.pos.x - self.radius,self.pos.y)  
 hit = **False  
  
  
 if** self.world.target.point\_inside(rightmost\_point):  
 hit = **True  
 elif** self.world.target.point\_inside(leftmost\_point):  
 hit = **True  
  
 if** hit:  
 self.world.bullet = **None** self.world.target.hit()

1. Now that we had the basic code working, the last aspect was automating it.
2. We started off by implementing the high level parent state machine using the first Lab as our pattern. It looked like this:

**class** StateMachineAgent(object):STATES = {**"ATTACK"**:0,  
 **"PATROL"**:1}  
  
 **def** \_\_init\_\_(self,shooter):  
 self.current\_state = self.STATES[**"PATROL"**]  
 self.shooter = shooter  
 self.sm\_attack = StateMachine\_Attacking(shooter)  
 self.sm\_patrol = StateMachine\_Patrol(shooter)  
 self.shoot\_threshold = 200  
  
 **def** change\_state(self):  
 **if** self.shooter.dist\_from\_target() < self.shoot\_threshold:  
 self.current\_state = self.STATES[**"ATTACK"**]  
 **else**:  
 self.current\_state = self.STATES[**"PATROL"**]  
  
 self.apply\_state()  
  
 **def** apply\_state(self):  
 **if** self.current\_state == self.STATES[**"PATROL"**]:  
 self.sm\_attack.wait = **True** self.sm\_patrol.stop = **False  
 elif** self.current\_state == self.STATES[**"ATTACK"**]:  
 self.sm\_attack.wait = **False** self.sm\_patrol.stop = **True** self.sm\_patrol.change\_state()  
 self.sm\_attack.change\_state()

1. We then made the two lower level state machines:
   1. The “Patrol” Mode

**class** StateMachine\_Patrol(object):  
 *#follow path  
 #if path finished randomise path* STATES = {**"FOLLOW\_PATH"**: 0,  
 **"RANDOMISE\_PATH"**: 1,  
 **"STOP"**:2}  
  
 **def** \_\_init\_\_(self,shooter):  
 self.shooter = shooter  
 self.stop = **True** self.current\_state = self.STATES[**"STOP"**]  
  
  
 **def** change\_state(self):  
 **if** self.stop:  
 self.current\_state = self.STATES[**"STOP"**]  
 **elif** self.current\_state == self.STATES[**"FOLLOW\_PATH"**]:  
 **if** self.shooter.path.is\_finished():  
 self.current\_state = self.STATES[**"RANDOMISE\_PATH"**]  
 **elif** self.current\_state == self.STATES[**"RANDOMISE\_PATH"**]:  
 self.current\_state = self.STATES[**"FOLLOW\_PATH"**]  
 **elif** self.current\_state == self.STATES[**"STOP"**]:  
 self.current\_state = self.STATES[**"FOLLOW\_PATH"**]  
  
 self.apply\_state()  
  
 **def** apply\_state(self):  
 **if** self.current\_state == self.STATES[**"STOP"**]:  
 self.shooter.mode = **"STOP"  
 elif** self.current\_state == self.STATES[**"FOLLOW\_PATH"**]:  
 self.shooter.mode = **"FOLLOW\_PATH"  
 elif** self.current\_state == self.STATES[**"RANDOMISE\_PATH"**]:  
 self.shooter.randomise\_path()

* 1. The “Attack” Mode

**class** StateMachine\_Attacking(object):  
 *#shoot  
 #reload* STATES = {**"SHOOT"**: 0,  
 **"RELOAD"**: 1,  
 **"WAIT"**:2}  
  
 **def** \_\_init\_\_(self,shooter):  
 self.shooter = shooter  
 self.wait = **True** self.current\_state = self.STATES[**"WAIT"**]  
  
  
 **def** change\_state(self):  
  
 **if** self.wait:  
 self.current\_state = self.STATES[**"WAIT"**]  
 **elif** self.current\_state == self.STATES[**"SHOOT"**]:  
 self.current\_state = self.STATES[**"RELOAD"**]  
 **elif** self.current\_state == self.STATES[**"RELOAD"**]:  
 **if not** self.shooter.world.bullet:  
 self.current\_state = self.STATES[**"SHOOT"**]  
 **elif** self.current\_state == self.STATES[**"WAIT"**]:  
 self.current\_state = self.STATES[**"SHOOT"**]  
  
 self.apply\_state()  
  
 **def** apply\_state(self):  
 **if** self.current\_state == self.STATES[**"WAIT"**]:  
 **pass  
 elif** self.current\_state == self.STATES[**"SHOOT"**]:  
 self.shooter.fire\_weapon()  
 **elif** self.current\_state == self.STATES[**"RELOAD"**]:  
 **pass**

1. We then added a simple line of code in the main program to call the change\_state method on the high level state machine each tick.

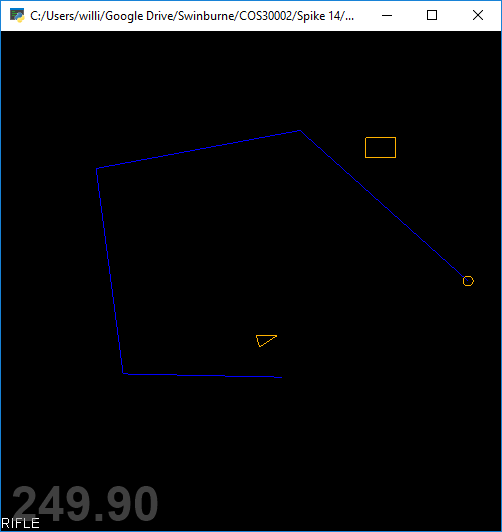
**What we found out:**

We found out that using multiple state machines in a hierarchical structure is an excellent way of dynamically controlling agent behaviour. Importantly, this is quite a beautiful solution as each state machine only has to care about managing itself and it children making this a very good solution in terms of object-oriented principles by reducing the coupling. This allows for a lot of control over bot behaviours, for example, each gun could have its own state machine which the bot knows how to interact with through an interface.

Overall, this is a very intuitive model to implement and in practice works extremely well.

**Screenshots**

Patrol (Follow Path)



Attack

