Exercise – Finite State Machines

Exercises:

1. Implement your own Finite State Machine. This should consist of:
   1. A Finite State Machine class that will handle state changing.
   2. A State base class containing pure virtual Enter, Update and Exit functions.
   3. A way to transition between States

Your State Machine will need the ability to:

1. Update the current state.
2. Transition to a new state if needed.

As an example, you could implement a State class that is an abstract base class with a pure virtual update / execute method and contains a collection of Transitions that link to other State objects.

The Transition class has a Condition and a target State, and a method for checking if the Condition has been met to trigger the Transition. The State class then has a method to query for the first triggered Transition that it has.

Finally a Finite State Machine class would contain the current state of the machine. It would update by first querying if any of the current state’s Transitions have triggered. If so it would exit the current state, enter the Transition’s target State, and update the Finite State Machine’s current State to be the target State. If no Transitions had triggered then it would simply update the current State.

The Finite State Machine could derived from the Behaviour class covered in previous lessons. The Finite State Machine’s update method would simply be the overloaded method from the Behaviour class.

1. Set up a project that contains 2 GameObjects, one to represent a Player and one to represent a Guard patrolling an area.

The Player object should simply be controlled by the player and have the ability to move around the screen. Alternatively you could use the mouse cursor location to represent the Player.

Using your Finite State Machine class, set up the following state machine with the specified States and Transitions and assign it to the Guard object:

* 1. **Patrol** State: The Guard should patrol between a sequence of locations looping continuously until a Transition is triggered. The locations could simply be a collection of structs containing two float variables, x and y, that the Guard should move to, and then move to the next in the sequence once it reaches the position.
     1. Transition to the **Attack** State if the Player is within a specified distance to the Guard
  2. **Attack** State: The Guard should chase the Player.
     1. Transition to the **Idle** State if the Player becomes out of range of the Guard.
  3. **Idle** State: The Guard should stand still**.**
     1. Transition to the **Attack** State if the Player comes back in to range.
     2. Transition to the **Patrol** State if a specified time has elapsed.

Close to Player

Time elapsed

Far from Player

Close to Player