

Coding for Game Development

Section Six

XNA Game Programming

Finite State Machine Design

1/17

Finite State Machine

- A Finite State Machine models behaviour by breaking it down into a small (finite) number of different states.
- The machine remains in each state for a relatively long period of time.
- State transitions are triggered by a particular series of events (inputs).
- Transitions between states are instantaneous.
- The machine can generate output when a transition occurs.
- (For the nerds, this form of Finite State Machine is called a "Mealy" machine)

2/17

State Diagram

- A state diagram has two major components:

• States:  State Transitions: 

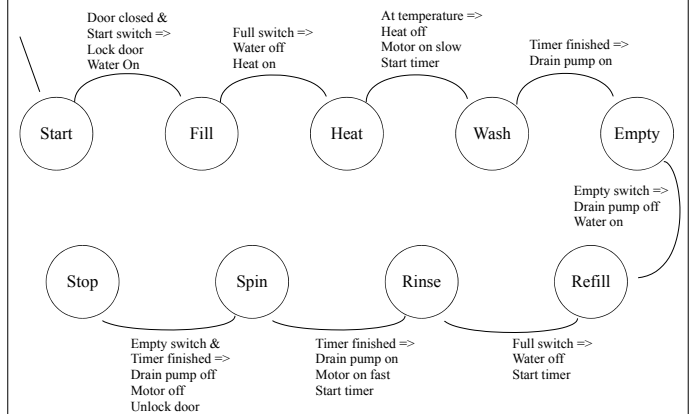
- Each transition may be labelled in the following format:

Input(s) which trigger transition (boolean) =>
Actions which occur during transition

- The trigger inputs may be omitted, in which case when the machine enters this state it will immediately move on to the next state.

3/17

Washing Machine FSM



4/17

Implementing a FSM

- Once we have a state diagram, implementing it in code is really easy.

```
const int START = 1,  
       FILL = 2,  
       HEAT = 3,  
       WASH = 4,  
       EMPTY = 5,  
       REFILL = 6,  
       RINSE = 7,  
       SPIN = 8,  
       STOP = 9;
```

(Or you can use an enum...)

(A real **switch** statement requires a **break** after every **case**, and a **default** case option)

```
int state = START;  
while (state != STOP)  
{  
    switch (state) {  
        case START : ...  
        case FILL : ...  
        case HEAT : ...  
        case WASH : ...  
        case REFILL : ...  
        case RINSE : ...  
        case SPIN : ...  
        case STOP : ...  
    }  
}
```

5/17

Case Actions

- Within each case clause, we implement the code for transitions to the next state.

```
case START :  
    if (doorClosed == true && startSwitch == true)  
    {  
        LockDoor(); // Lock the door  
        WaterOn(); // Turn on water  
        state = FILL; // Change to state: FILL  
    }  
    break;
```

```
case HEAT :  
    if (temperature > 40.0)  
    {  
        Heat(false);  
        MotorOn(0.5);  
        StartTimer();  
        state = WASH;  
    }  
    break;
```

```
case WASH :  
    if (timer > 10) //10 secs  
    {  
        DrainPumpOn();  
        state = EMPTY;  
    }  
    break;
```

6/17

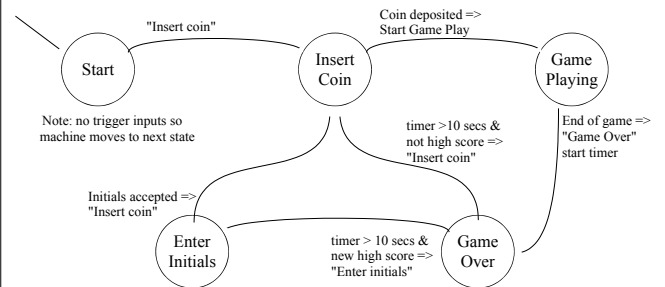
Spinning FSM

- If we implement the FSM as described, then it rapidly repeats the **while** loop waiting for a change of input.
- This situation is sometimes called *Spinning* (not to be confused with the SPIN state of the Washing Machine)
- When the FSM is spinning, it tends to eat up all the available processor time, so we should find a way to avoid this.
- In a real washing machine controller, we might put in a sleep so that it only read the inputs once per second.
- With an XNA game, we can use the game loop to process the FSM every 1/60th second...

7/17

Relevance to Gaming

- Finite State Machines might seem to be a long way from games programming, yet they can easily be applied to game design.



8/17

Implementing FSM in Update()

`int state;`
`int timer;` Declared as fields

```

protected override void Update(GameTime gameTime)
{
    // Allows the game to exit
    if (... == ButtonState.Pressed)
        this.Exit();

    // TODO: Add your update logic here
    switch (state) {
    case START : ...
    case INSERT_COIN : ...
    case GAME_PLAYING : ...
    case GAME_OVER : ...
    case ACCEPT_INITIALS : ...
    }

    base.Update(gameTime);
}
  
```

Boilerplate

Executed 60 times per second

9/17

Implementing FSM Case Actions

```

case START:
... Starting initialisation ...
state = INSERT_COIN;
break;
  
```

```

case GAME_PLAYING:
... Perform game actions ...

if (... game now over ...) {
    timer = 0;
    state = GAME_OVER;
}
break;
  
```

```

case INSERT_COIN:
if (coinInserted == true)
{
    state = GAME_PLAYING;
}
break;
  
```

```

case GAME_OVER:
timer++;
if (timer > 600) // 10 secs
{
    if (score > highScore)
        state = ENTER_INITIALS;
    else
        state = INSERT_COIN;
}
break;
  
```

10/17

Implementing FSM in Draw()

- Many states have associated messages:

```

protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    // TODO: Add your drawing code here
    spriteBatch.Begin();

    switch (state) {
    case START : ...
    case INSERT_COIN : ...
    case GAME_PLAYING : ...
    case GAME_OVER : ...
    case ACCEPT_INITIALS : ...
    }

    spriteBatch.End();
    base.Draw(gameTime);
}
  
```

case INSERT_COIN: spriteBatch.DrawString(...)

case GAME_OVER: spriteBatch.DrawString("GAME OVER", ...); break;

11/17

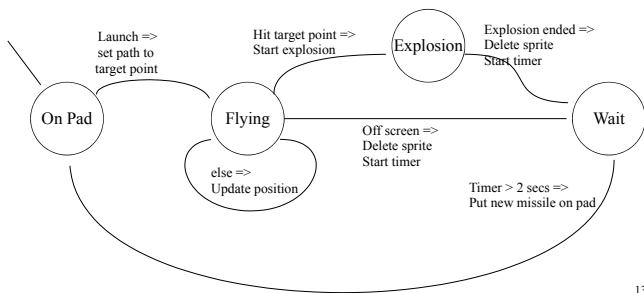
Sprite Lifetime FSM

- Consider a "Missile Attack" game.
- The user controls an interceptor missile (sprite) which is fired to shoot down incoming warheads (sprites).
- The missile sits on the launch pad until the user clicks the mouse to set a target position.
- The missile flies to the target position and explodes (possibly destroying an incoming warhead).
- If the warhead is destroyed then the score is increased by 10 points.
- If any warhead reaches the ground then it explodes and the game finishes.

12/17

Missile Lifetime FSM

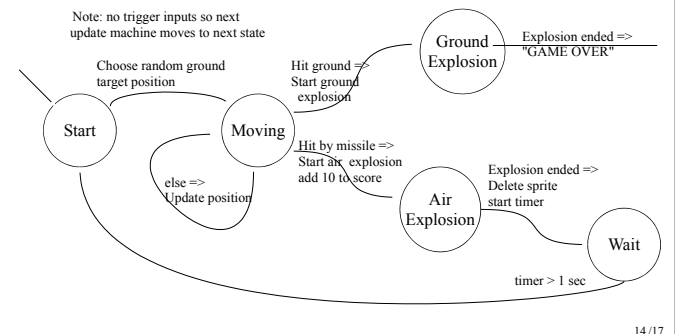
- We can model the missile sprite lifetime using this FSM.
- States are updated every $1/60^{\text{th}}$ of a second in **Update()**.



13 / 17

Warhead Sprite FSM

- The FSM for an incoming warhead looks like this:



14 / 17

Combining FSMs

- We have three Finite State Machines which we need to combine into one working application.
- Each FSM requires its own separate state variable.
- Each FSM is updated within the **Update()** method every $1/60^{\text{th}}$ second.
- The updates to missile and warhead states only take place when the gameState is GAME_PLAYING.
- The warhead FSM can trigger the game FSM to move to GAME_OVER when the game has finished.

15 / 17

Combining the Multiple FSMs

- A naïve approach is to create three state fields and try to code each FSM in the **Update()** method, using nested switches within switches.
- For a very simple program, this naïve approach can be made to work, but it gets very messy and cannot be recommended.
- A professional approach is to encapsulate sprites and sprite behaviour into separate classes with object oriented programming.
- This is what we shall be looking at in subsequent weeks.

16 / 17

17 / 17