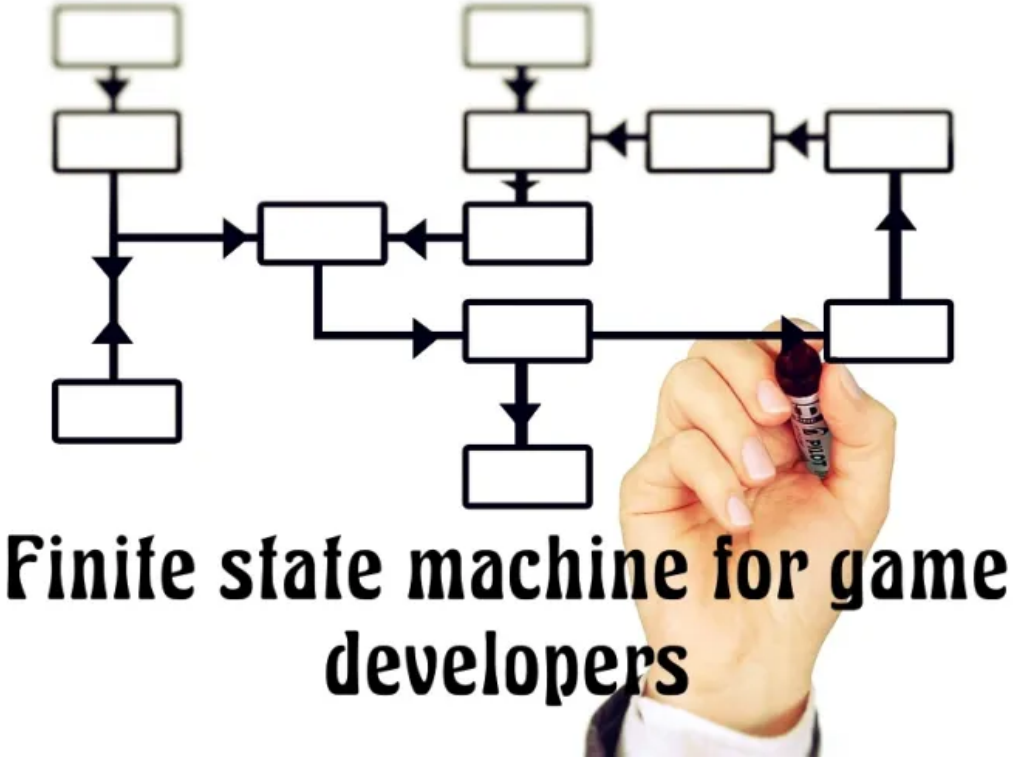
Finite State Machine For Game Developers



Finite state machine (FSM) for game developers:

If you want to develop games, sooner or later you will have to deal with a finite state machine. Let's start with the basics, what is a finite state machine?

If you want the complex answer, ***here****(https://en.wikipedia.org/wiki/Finite-state_machine)* is the link to wikipedia. But we all like to keep things simple, so our definition of a finite state machine is gonna be this:

"finite *State Machine is a structure that allows to define complex behaviors*"

That's it. Thanks to a finite state machine we can define complex behaviors and encapsulate them into mini single interactions which we will call state. Each state

should describe a very simple action. The interactions of many single state will lead to complex behavior.

Simple finite state machine example:

Let’s start with a very simple example: a button.

If we want to define a classic button we can define it’s behavior into two simple states: Pressed and Released. If we want to implement the logic of a button into a game, a simple boolean will be sufficient:

```
Bool isButtonPressed = false;
```

With the boolean *isButtonPressed,* we have defined two states. After that, we can define a class called Button that will contain the behavior of each state. It should look something like this:

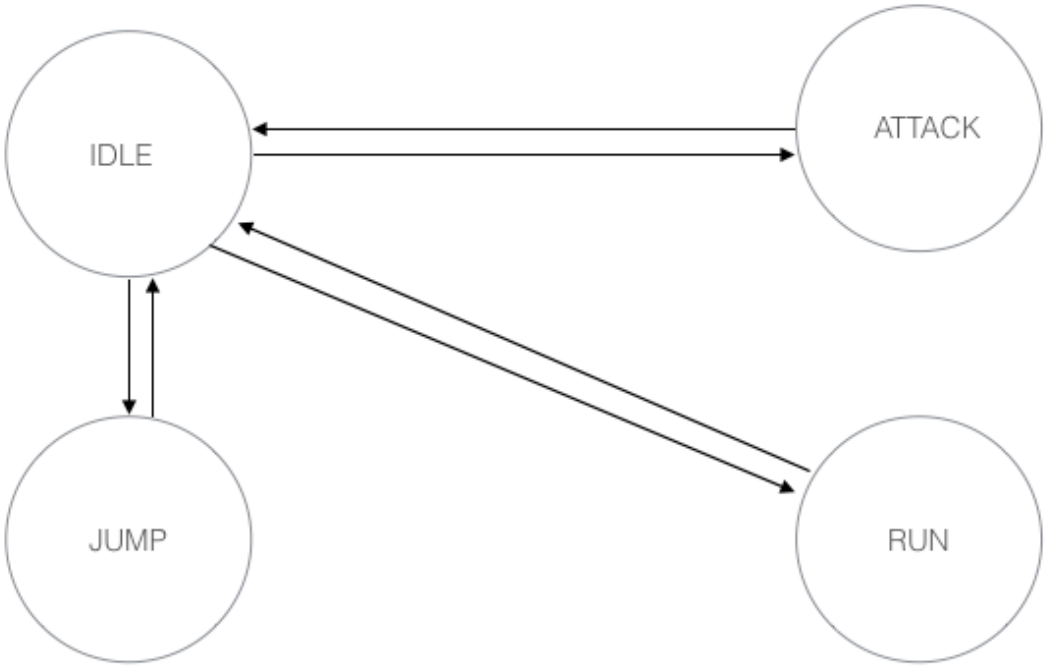
```
Public Button
{
   Bool isButtonPressed = false.

   While (true)
  {
     If (isButtonPressed) {
     //The button has been pressed do what you want
     }
      else {
        //The button has been released do what you
want
     }
  }
}
```

Very simple and straightforward.

Now let’s start to make things more fun. Imagine you have a player that can do these actions:

- Jump
- Move
- Attack
- Idle



As you can see, it’s a simple finite state machine with 4 states and we can be in one state at the time. But we have also added a new factor, we add the relation between states.

You may think that the solution is to simply use four booleans to define the four states. In theory you can do that, but in practice, it is strongly discouraged. The reason is simple:

1. You will end up with an incredible series of “If-Else” statements, because each time you have to check the current state you’re into, and then check which is the state that you want to reach, and based on that define the behavior.
2. This can easily land to bugs, since it’s easy to set by mistake two or more states to true, and by consequence we’ll be in both states at the same time.

Simple finite state machine implementation:

We can keep things simple by use an enum instead of a series of bool. For example:

```
Public PlayerBheavior {

        Enum PlayerState {
                Idle,
                Jump,
                Run,
                Attack
        }
}

PlayerState actualPlayerState = PlayerState.Idle;

Void Update() {
        Switch (actualPlayerState) {
        case PlayerState.Idle:
                //Do what you want in the idle state
        break;
        case PlayerState.Jump:
                //Do what you want in the Jump state
        break;
        case PlayerState.Run:
                //Do what you want in the Run state
        break;
        case PlayerState.Attack:
                //Do what you want in the Attack state
        break;
        }
}
```

By using this code, we solved the 2 problems that were listed before.

Medium complexity finite state machine:

But this solution doesn't take into account the fact that some states are reachable only from some other state. For example, we can only go from idle to run and vice versa, but we can't go from run to attack.

If we want to solve this problem we can simply add another variable called "nextState" and every time that we want to change the state, we have to check the currentState to be sure that the nextState that we want to reach is reachable from the currentState. But this will again lead to a series of If-Else statements and if we have a lot of states, the code will become unreadable very fast.

So how do we solve the problem? We need a solution to be sure that some states are reachable only from some pre assigned state.

I've designed a finite state machine which is a little bit more complicated than the one that we defined before, you can find it on *GitHub**(https://github.com/MarcoMig/Finite-State-Machine-FSM)* with all the instructions to use it. This finite state machine allows you to define the transition between the states. For example:

```
public class Program
    {
        static void Main(string[] args)
        {
            ChildFSM fsm = new ChildFSM();
            Console.WriteLine("Current State = " +
fsm.currentState);
            //MoveNext used to change the state
            Console.WriteLine("Command.Begin: Current
State = " + fsm.MoveNext(PlayerState.Run));
            //CanReachNext used to check if the next
state it's reachable
            Console.WriteLine("Invalid transition: " +
fsm.CanReachNext(PlayerState.Idle));
            Console.WriteLine("Previous State = " +
fsm.previous);
        }
```

```
}
```

As you can see with this new structure we can use a method like: CanReachNext(stateToReach). This method will make sure to move to the right state, and if you try to reach a state that is unreachable from the current one, the finite state machine won't change the state.

Complex Finite state machine:

Since now we've defined a finite state machine that allows us to define complex behavior, let's try to complicate things a little bit. The problem is the following: we want to define very complex behavior for each state. For example,every time I enter in on a state, I want to be able to track the execution of the state and then do something when I exit from that state, we can describe these behavior with the following methods:

- Void OnStateEnter(PlayerState state);
- Void OnStateExecution(PlayerState state);
- Void OnStateExit(PlayerState state);

Imagine defining this behavior for each state inside one single class. As you can imagine define this behavior inside a single class will dramatically increase the complexity, and our class will became unreadable very soon.

This solution is not good at all for complex behaviors. We'll end up with a lot of switch statements with complex and log methods for each case inside a single class. We need to find a solution that is more easy to use and more scalable.

Complex state machine definition:

Before to start, we need to find a way to make sure that each state is going to have the tree method listed before. The following solution expected to define a class for each state, thanks to this, we can define complex behavior in each class.

To define the methods to use in each class, we can use ***interface*** ***(http://stackoverflow.com/questions/2866987/what-is-the-definition-of-interface-in-object-oriented-programming)*** s:

```
public interface IState
{

    void OnStateExecution();

    void OnStateEnter ();
    void OnStateExit ();
}
```

Now we can use **classes** to define states:

```
public class AttackState: IState {

    Void OnStateEnter() {
                //Define you enter behavior here
    }

    Void OnStateExecution() {
        //Define your execution behavior here
    }

    Void OnStateExit() {
        //Define your exit behavior here
    }
}
```

In our player class we'll define a series of classes as states, for example:

```
public class Player {
        AttackState attackState = New AttackState();
        IdleState IdleState = New IdleState();
        Istate currentState = Idle;
}
```

Each time we change state we can do something like

```
Private void ChangeStateExampleMethos(Istate
nextState)
```

```
{
        currentState.OnStateExit();
        nextState.OnStateEnter();
        currentState = nextState;
}
```

Execute the state:

```
Void Update() {
        currentState.OnStateExecution();
}
```

I just gave you a little implementation idea, If you want to see them in action, you can have a look at this video directly from unity. You can take their version and adapt them to define your specific behavior. This finite state machine combined with the one with medium complexity allows you define every kind of complex behavior and still be sure to not reach the wrong state accidentally.

Inheritance in finite state machines:

But there is a problem again :(. Imagine that your player can Attack while jumping, running, and moving down. We need three different states for each one of these actions and as you can imagine we can end up with a really big state machine where most of the states have similar behaviors. For example, the Attack state will probably be very similar to the JumpAttack state where the only difference is the animation to run.

To avoid repeating the same code we can use inheritance and share the code between similar states.

Example

Void JumpAttackState : AttackState { //Defiene your behavior here}

Where to use finite state machines

1. When your behavior can be defined by a finite set of states.

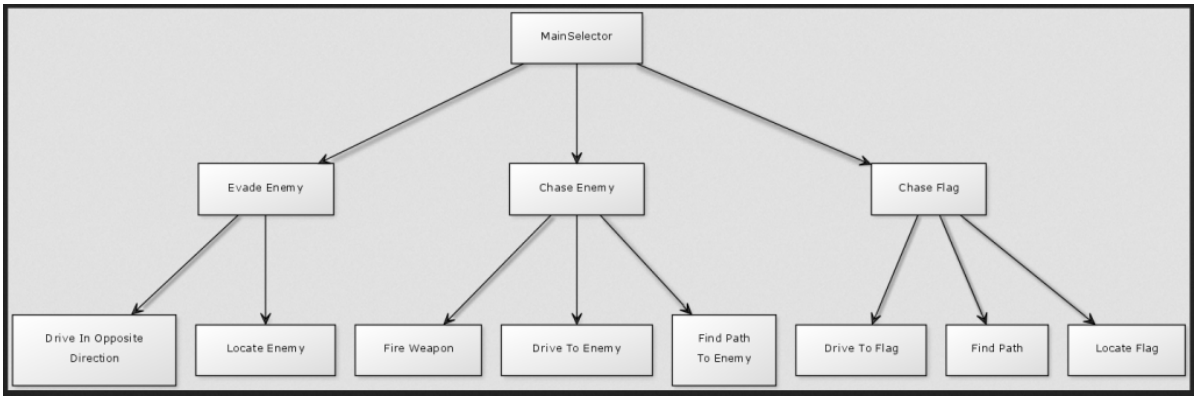
2. When each state can be reached by an external or internal trigger/Action.
3. Define behavior not too much complex.

Finite state machine alternatives

If you want to define a very complex AI that has a lot of similar states, there is the risk of ending up with an over complicated finite state machine with too much states.

The solution to this problem is to use behavior trees. These allow us to define very complex behavior and are used almost everywhere in the artificial intelligence field.

Example:



Each state inside the behavior tree can be a single state machine.