State Machine

barodapride edited this page on 21 Sep 2016 · 7 revisions

Introduction

Since the early days of video games *finite state machines* (FSM) are a common instrument to imbue a game agent with the illusion of intelligence.

The following is a descriptive definition of FSM. A finite state machine is a device which has a finite number of states it can be in at any given time and can operate on input to either make transitions from one state to another or to cause an output or action to take place. A finite state machine can only be in one state at any moment in time.

The idea behind a FSM is to decompose an object's behavior into easily manageable states.

There are a number of ways of implementing finite state machines. A typical naive approach is to use a switch statement like the following.

```
public void runAway(int state) {
        switch (state) {
        case 0: // Wander
                wander();
                if (seeEnemy())
                        state = MathUtils.randomBoolean(0.8f) ? 1 : 0;
                if (isDead())
                        state = 3;
                break;
        case 1: // Attack
                attack();
                if (isDead())
                        state = 3;
                break;
        case 2: // Run away
                runAway();
                if (isDead())
                        state = 3;
                break;
        case 3: // Dead
                slowlyRot();
                break;
        }
}
```

The code above is a legitimate state machine, however it has some serious weakness:

- The state changes (also known as transitions) are poorly regulated.
- States are of type int and would be more robust and debuggable as classes or enums.
- The omission of a single `break` keyword would cause hard-to-find bugs.
- Redundant logic appears in multiple states.

- No way to tell that a state has been entered or exited.

The right solution to these issues is to provide a more structured approach. Having performance in mind we have chosen to implement FSMs through *embedded rules*, thus hard-coding the rules for the state transitions within the states themselves. This architecture is known as the *state design pattern* and provides an elegant and powerful way of implementing state-driven behavior with minimal overhead.

The state machine implementation provided by gdx-ai is mainly based on the approach described by Mat Buckland in his book *"Programming Game AI by Example"*. This same approach, with minor variations, has been supported by hundreds of articles over the years.

IMPORTANT NOTE:

- You don't have to restrict the use of state machines to agents. The state design pattern is also useful for structuring the main components of your game flow. For example, you could have a menu state, a save state, a paused state, an options state, a run state, etc.

The State Interface

The states of the FSM are encapsulated as objects and contain the logic required to facilitate state transitions. All state objects implement the `State` interface which defines the following methods:

- `enter(entity)` will execute when the state is entered
- `update(entity)` is called on the current state of the FSM on each update step
- `exit(entity)` will execute when the state is exited
- `onMessage(entity, telegram)` executes if the entity receives a message from the message dispatcher while it is in this state.

Actually, *enter* and *exit* methods are only called when the FSM changes state. When a state transition occurs, the StateMachine.changeState method first calls the `exit` method of the current state, then it assigns the new state to the current state, and finishes by calling the `enter` method of the new state (which is now the current state).

IMPORTANT NOTES

- **States as Java enumeration:** Conceptually speaking each concrete state should be implemented as a singleton object in order to ensure that there is only one instance of each state, which agents share. Using singletons makes the design more efficient because they remove the need to allocate and deallocate memory every time a state change is made. In real world, concrete states are typically implemented as a *Java enumeration*, since they are an easy and versatile way to implement early loading singletons as we'll see in the examples below. However, using singletons (or enums) has one drawback. Because they are shared between clients, singleton states are unable to make use of their own local, agent-specific data. For instance, if an agent uses a state that when entered should move it to an arbitrary position, the position cannot be stored in the state itself because the position may be different for each agent that is using the state. Instead, it would have to be stored somewhere externally and be accessed by the state through the agent. Anyways, it's worth noticing that the framework doesn't force you to use the singleton design. This is just the recommended approach you should stick to as long as there are not well-founded reasons to allocate a new state on each transition.
- **Global State:** Often, when designing finite state machines you'll end up with code that is duplicated in every state. When it happens it's convenient to create a *global state* that is called every time the FSM is updated. That way, all the logic for the FSM is contained within the states and not in the agent class that owns the FSM.
- **State Blip:** Occasionally it will be convenient for an agent to enter a state with the condition that when the state is exited, the agent returns to its previous state. This behavior is called a *state blip*. For instance, in the Far West example below the agent Elsa can visit the bathroom at any time, then she always returns to its prior state.

The StateMachine Interface

All the state related data and methods are encapsulated into a state machine object. This way an agent can own an instance of a FSM and delegate the management of current states, global states, and previous states to it. Also, a state machine can be explicitly delegated by its owner to handle the messages it receives. A FSM instance implements the following StateMachine interface.

```
public interface StateMachine<E, S extends State<E>> extends Telegraph {
        public void update();
        public void changeState(S newState);
        public boolean revertToPreviousState();
        public void setInitialState(S state);
        public void setGlobalState(S state);
        public S getCurrentState();
        public S getGlobalState();
        public boolean isInState(S state);
        public boolean handleMessage(Telegram telegram);
}
```

All an agent has to do is to own an instance of a StateMachine and implement a method to update the state machine to get full FSM functionality.

DefaultStateMachine

The `DefaultStateMachine` class provided by the framework is the default implementation of the `StateMachine` interface. The `handleMessage` method of the `DefaultStateMachine` first routes the telegram to the current state. If the current state does not deal with the message, it's routed to the global state (if any). Especially, the boolean value returned by the `onMessage` method of the State interface indicates whether or not the message has been handled successfully and enables the state machine to route the message accordingly. This technique is rather interesting. For instance, what if you want a global event response to the message MSG_DEAD in every state but the state STATE_DEAD? The solution is to override the message response MSG_DEAD within STATE_DEAD. Since messages are sent first to the current state you can consume the message by returning true so to prevent it from being sent to the global state.

StackStateMachine

The `StackStateMachine` implements a pushdown automaton. It actually inherits from `DefaultStateMachine` and mostly behaves the same. The only difference is the behavior of `revertToPreviousState()` . While the default implementation will always change back and forth between the same two states when it is called multiple times, the `StackStateMachine` will instead keep track of all past states, store them in a stack-like manner and is able to revert to those past states in a "last in, first out" (LIFO) order. This is especially useful when using a state machine for hierarchical menu structures.

Let's assume we have just a single `MenuScreen` to handle all menus. Each menu would be a State. For example `MainMenuState` , `OptionsMenuState` , `GraphicsOptionsMenuState` and `InputOptionsMenuState` . Usually the user would start with the main menu, then navigate to the options menu and choose the graphics options. When he is done, it is common to navigate this hierarchical menu structure backwards by just pressing ESC. With the stack implementation this can now easily be done by just calling `stateMachine.revertToPreviousState()` , whenever the ESC button is pressed. When there is no more "previous" state, the revert method will not change the state and return false.

A Simple Example

Imagine a Troll class that has member variables for attributes such as health, anger, stamina, etc., and public methods to query and adjust those values. A Troll can be given the functionality of a FSM by adding a member variable `stateMachine` .

```
public class Troll {

    // An instance of the state machine class
    public StateMachine<Troll, TrollState> stateMachine;

        public Troll() {
        stateMachine = new DefaultStateMachine<Troll, TrollState>(this, TrollState.S
    }
    public void update(float delta) {
        stateMachine.update();
    }

        /* OTHER METHODS OMITTED FOR CLARITY */

}
```

When the update method of a Troll is called, it in turn calls the update method of its FSM which in turn calls the update method of the current state. The current state may then use the Troll interface to query its owner, to adjust its owner's attributes, or to effect a state transition. In other words, how a Troll behaves when updated can be made completely dependent on the logic in its current state. This is best illustrated with an example, so let's create a couple of states to enable a troll to run away from enemies when it feels threatened and to sleep when it feels safe.

```
public enum TrollState implements State<Troll> {

        RUN_AWAY() {
                @Override
                public void update(Troll troll) {
                        if (troll.isSafe()) {
                                troll.stateMachine.changeState(SLEEP);
                        }
                        else {
                                troll.moveAwayFromEnemy();
                        }
                }
        },

        SLEEP() {
                @Override
                public void update(Troll troll) {
                        if (troll.isThreatened()) {
                                troll.stateMachine.changeState(RUN_AWAY);
                        }
                        else {
                                troll.snore();
                        }
                }
        };

        @Override
        public void enter(Troll troll) {
        }

        @Override
        public void exit(Troll troll) {
        }

        @Override
        public boolean onMessage(Troll troll, Telegram telegram) {
                // We don't use messaging in this example
                return false;
        }
}
```

As you can see, when updated, a troll will behave differently depending on its current state. Both states are encapsulated as a Java enumeration and both provide the rules effecting state transition.

A Complete Example with Messaging

As a practical example of how to create agents using finite state machines and messaging, we are going to look at a game environment set in the Far West. Well, actually, you'll have to use your imagination since the example has no graphics at all. Any state changes or output from state actions will be sent as text to the logging system. This simple approach demonstrates clearly the mechanism of a finite state machine without adding the code clutter of a more complex environment.

In our imaginary country in the Far West there are 2 characters, Bob the miner and his wife Elsa:

- Bob always moves between 4 locations: a gold mine, a bank where Bob can deposit any nuggets he finds, a saloon in which he can quench his thirst, and his home where he can sleep. Exactly where he goes, and what he does when he gets there, is determined by Bob's current state. He will change states depending on member variables like thirst, fatigue, and how much gold he has found hacking away down in the gold mine.
- Elsa doesn't do much; she's mainly busy with cleaning the shack, cooking food and emptying her bladder (she drinks way too much).

Of course Bob and Elsa will communicate in certain situations. They only have two messages they can use:

- *HI_HONEY_I_M_HOME* used by Bob to let Elsa know he's back at the shack.
- *STEW_READY* used by Elsa to let herself know when to take dinner out of the oven and for her to communicate to Bob that food is on the table.

To run the example you have to launch the StateMachineTest class from gdxAI tests. The other classes of the example are inside the package com.badlogic.gdx.tests.ai.fsm. Here is a sample of the output from the test program.

```
Bob: All mah fatigue has drained away. Time to find more gold!
Bob: Walkin' to the goldmine
Elsa: Walkin' to the can. Need to powda mah pretty li'lle nose
Elsa: Ahhhhhh! Sweet relief!
Elsa: Leavin' the Jon
Bob: Pickin' up a nugget
Elsa: Washin' the dishes
Bob: Pickin' up a nugget
Elsa: Makin' the bed
Bob: Pickin' up a nugget
Bob: Ah'm leavin' the goldmine with mah pockets full o' sweet gold
Bob: Goin' to the bank. Yes siree
Elsa: Makin' the bed
Bob: Depositing gold. Total savings now: 3
Bob: Leavin' the bank
Bob: Walkin' to the goldmine
Elsa: Walkin' to the can. Need to powda mah pretty li'lle nose
Elsa: Ahhhhhh! Sweet relief!
Elsa: Leavin' the Jon
Bob: Pickin' up a nugget
Bob: Ah'm leavin' the goldmine with mah pockets full o' sweet gold
Bob: Boy, ah sure is thusty! Walking to the saloon
Elsa: Makin' the bed
Bob: That's mighty fine sippin liquer
Bob: Leaving the saloon, feelin' good
Bob: Walkin' to the goldmine
Elsa: Washin' the dishes
Bob: Pickin' up a nugget
Elsa: Makin' the bed
Bob: Pickin' up a nugget
Bob: Ah'm leavin' the goldmine with mah pockets full o' sweet gold
Bob: Goin' to the bank. Yes siree
Elsa: Washin' the dishes
Bob: Depositing gold. Total savings now: 4
Bob: Leavin' the bank
Bob: Walkin' to the goldmine
Elsa: Makin' the bed
Bob: Pickin' up a nugget
Elsa: Makin' the bed
Bob: Pickin' up a nugget
Bob: Ah'm leavin' the goldmine with mah pockets full o' sweet gold
Bob: Boy, ah sure is thusty! Walking to the saloon
Elsa: Makin' the bed
Bob: That's mighty fine sippin liquer
Bob: Leaving the saloon, feelin' good
Bob: Walkin' to the goldmine
Elsa: Moppin' the floor
Bob: Pickin' up a nugget
Bob: Ah'm leavin' the goldmine with mah pockets full o' sweet gold
Bob: Goin' to the bank. Yes siree
Elsa: Washin' the dishes
```

```
Bob: Depositing gold. Total savings now: 5
Bob: WooHoo! Rich enough for now. Back home to mah li'lle lady
Bob: Leavin' the bank
Bob: Walkin' home
Message handled by Elsa at time: 11213152138
Elsa: Hi honey. Let me make you some of mah fine country stew
Elsa: Putting the stew in the oven
Elsa: Fussin' over food
Message received by Elsa at time: 11213889030
Elsa: StewReady! Lets eat
Message handled by Bob at time: 11214221965
Bob: Okay Hun, ahm a comin'!
Bob: Smells Reaaal goood Elsa!
Elsa: Puttin' the stew on the table
Bob: Tastes real good too!
Bob: Thankya li'lle lady. Ah better get back to whatever ah wuz doin'
Elsa: Washin' the dishes
Bob: ZZZZ...
Elsa: Moppin' the floor
Bob: ZZZZ...
Elsa: Makin' the bed
Bob: ZZZZ...
Elsa: Makin' the bed
Bob: ZZZZ...
Elsa: Makin' the bed
```

As you have seen, the use of messaging combined with state machines gives you the illusion of intelligence and the output of the program looks like the interactions of two real people. What's more, this is only a very simple example.

Divide and Conquer

The unbounded growth of states is one of the main problems plaguing state machines. Using a global state reduces this problem greatly, but the real solution is to be disciplined and not let too many states exist within a single FSM. This can be achieved by using 2 techniques:

- **Hierarchical State Machines**: Rather than combining all the logic into a single state machine, we can separate it into several by dividing an AI's tasks into independent chunks that can each become a self-contained state machine. For instance, your game agent may have the states *Explore*, *Combat*, and *Patrol*. In turn, the *Combat* state may own a state machine that manages the states required for combat such as *Dodge*, *ChaseEnemy*, and *Shoot*.
- **Simultaneous State Machines**: Another way to deal with limiting the size of state machines is to use several different state machines at the same time. For example, you can imagine an AI that has a master FSM to make global decisions and other FSMs that deal with movement, gunnery, or conversations.

Properly combining and structuring these two techniques is an extremely powerful way to limit the complexity of individual state machines.

Limitations of State Machines

Even with all the common techniques and extensions mentioned above, state machines are still pretty limited. The trend these days in game AI is more toward exciting things like behavior trees and planning systems. If complex AI is what you're interested in, all this chapter has done is whet your appetite. 😋

This doesn't mean finite state machines, pushdown automata, and other simple systems aren't useful at all. They're a good modeling tool for certain kinds of problems. Finite state machines are useful when:

- You have an entity whose behavior changes based on some internal state.
- That state can be rigidly divided into one of a relatively small number of distinct options.
- The entity responds to a series of inputs or events over time.

In games, they are most known for being used in AI, but they are also common in implementations of user input handling, navigating menu screens, parsing text and network protocols.