12

A Reusable, Light-Weight Finite-State Machine

David "Rez" Graham

12.1 Introduction

Finite-state machines are an architectural structure used to encapsulate the behavior of discrete states within a system and are commonly used in games for AI, animation, managing game states, and a variety of other tasks (Fu 2004).

For example, consider a guard character that can walk to different waypoints. When he or she sees the player, he or she starts shooting. The guard can be in one of the two states: patrol or attack. It is easy to expand on this behavior by adding more states or tweaking existing ones.

State machines are great for composing behaviors because each state can be parameterized and reused for multiple different characters. For example, you could have a state for attacking the player with each type of enemy determining the attack animations to run, how close they need to be, and so on. The biggest advantage of state machines is that they offer a lot of flexibility while still being extremely simple to implement and maintain. Due to this simplicity, this architecture is best suited for games with smaller AI needs.

Going back to the guard example, one common place for parameterization is the vision cone for the guard. How close does the player have to be before the guard sees him or her?

You could have different kinds of guards with different capabilities. For instance, an elite guard may have a bigger vision cone than a regular guard.

One decision you need to make when building a state machine is how to organize the states and where to put the transition logic. This chapter will discuss how to use transition objects to create encapsulated, reusable objects for managing transitions. This allows designers to compose the overall behavior of an enemy by selecting the appropriate states, choosing transitions for those states, and setting the necessary parameters.

The biggest difference between a traditional state machine and the one presented in this chapter is that this state machine can monitor and change states as necessary. In a traditional state machine, state changes come externally.

The techniques in this chapter were used in the platformer game *Drawn to Life: The Next Chapter* for the Wii, developed by Planet Moon Studios, to handle all of the enemy AI.

12.2 Finite-State Machine Architecture

The driver of the state machine is the `StateMachine` class. This class is owned by the intelligent `GameObject` and manages the behaviors for that object. It owns a list of every possible state the object can be in as well as a pointer to the current state the object is in right now.

The state machine is updated periodically (either every frame or on a cadence that makes sense from a performance point of view) and has an interface for setting the state. The states owned by the state machine are handled with a `State` base class. There are virtual functions for handling the entering and exiting of states, as well as a function to be called on periodic updates. All of the concrete states used by the system will inherit from this class and will override the appropriate methods.

The third major component of this architecture is the `StateTransition` class, which is also owned by the state machine. This class has a `ToTransition()` virtual function that returns a Boolean for whether or not the transition should occur. Much like states, every possible transition in the game is a subclass of `StateTransition`. Figure 12.1 shows a diagram of this architecture.

The states owned by the state machine are handled with a `State` base class, shown in Listing 12.1.

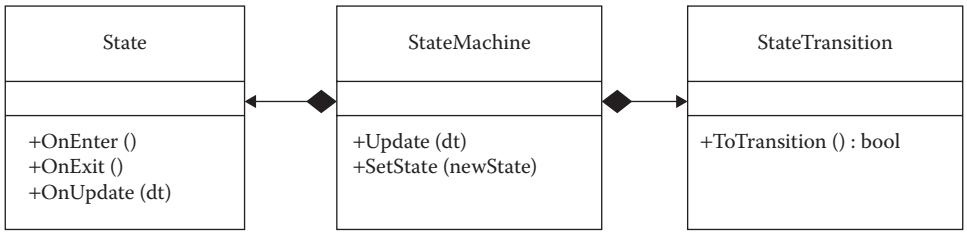

Figure 12.1

State machine architecture.

Listing 12.1. Base class for a single state.

```
class State
{
    GameObject* m_pOwner;

public:
    State(GameObject* pOwner)
        : m_pOwner(pOwner)
    { }
    virtual ~State() { }
    virtual void OnEnter() { }
    virtual void OnExit() { }
    virtual void OnUpdate(float deltaTime) { }

protected:
    GameObject* GetOwner() const { return m_pOwner; }
};
```

It is important to note that state instances live for the entire lifetime of the state machine, so it is important for `OnEnter()` or `OnExit()` to handle resetting the state.

12.3 State Transitions

The key to this architecture lies in the transition objects and the way that the state machine is updated. Every possible state inside the state machine is linked to a sorted list of transition/state pairs. Listing 12.2 shows a partial definition for the `StateMachine` class to illustrate this data structure.

Listing 12.2. `StateMachine` class.

```
class StateMachine
{
    typedef pair<Transition*, State*> TransitionStatePair;
    typedef vector<TransitionStatePair> Transitions;
    typedef map<State*, Transitions> TransitionMap;

    TransitionMap m_transitions;

    State* m_pCurrState;

public:
    void Update(float deltaTime);
};
```

The variable `m_transitions` holds the network of transitions for this state machine and defines how the game object will move from state to state. These are typically read from data and built at initialization time, which we will address later in this chapter.

During every update, the state machine checks to see if there are any transitions for this state. If there are, it walks through the list of those transitions and checks to see if it is time to move to another state. If it is, the transition occurs. Listing 12.3 shows the code for this update.

Having no transitions for a state is perfectly valid. For instance, you might not want to allow an enemy to transition out of the death state, or you might have a state that you want to be permanent until some outside force causes the state to manually change via a call to `StateMachine::SetState()`.

Listing 12.3. State machine update.

```
void StateMachine::Update(float deltaTime)
{
    // find the set of transitions for the current state
    auto it = m_transitions.find(m_pCurrState);
    if (it != m_transitions.end())
    {
        // loop through every transition for this state
        for (TransitionStatePair& transPair : it->second)
        {
            // check for transition
            if (transPair.first->ToTransition())
            {
                SetState(transPair.second);
                break;
            }
        }
    }

    // update the current state
    if (m_pCurrState)
        m_pCurrState->Update(deltaTime);
}
```

Using this technique, the states are completely decoupled from each other. Ideally, only the state machine cares about the states, and it never has to know which specific state a character is in.

The transition class itself is trivial and defined in Listing 12.4.

Listing 12.4. Transition class.

```
class Transition
{
    GameObject* m_pOwner;

public:
    Transition(GameObject* pOwner)
        :m_pOwner(pOwner)
    { }

    virtual bool ToTransition() const = 0;
};
```

Each `Transition` holds a reference to the owning game object and declares a pure virtual `ToTransition()` function, which is overridden in the subclass.

States will often need the ability to end naturally and transition into another state. A good example of this is a state that handles pathing to a location. It will naturally end once the game object reaches its destination and needs to move to another state.

One way of doing this is to define a transition that asks if the current state is finished. If it is, the transition occurs and moves to the new state. This keeps all the transition logic in the same place (the transition map) while still allowing natural transitions.

12.4 Hierarchies of State Machines and Metastates

One way to expand this state machine system is to add hierarchy to state machines through metastates. A metastate is a state that contains its own state machine, which has multiple internal states. The simplest way to manage this is by having a special `MetaState` subclass that internally has its own state machine. It would be tuned just like any other state machine and could reuse any existing states.

To illustrate this concept, let us go back to the guard example from the beginning of the chapter where we had a guard that could patrol and attack. Both of these are metastates. The patrol metastate has a transition that tests to see if the player is within the guard's vision cone. The attack metastate has a transition that tests to see if the player is outside of a large radius, which is when we consider the guard to have lost the player. So far, there is nothing special or different. As far as the root system is concerned, there are two states with simple transitions.

Inside of the attack metastate, there are two states. The initial state is an alert state that plays an alert animation. It transitions to the shoot state after a few seconds as long as the player remains in the vision cone. The shoot state will transition back to alert if the player steps out of the vision cone.

The important thing to note is that every active state machine is getting its update, which means that any state can transition at any moment. If the attack state suddenly transitions to the patrol state, it will immediately end whatever state the attack metastate was in. This allows you to treat that entire branch as self-contained.

The addition of metastates adds more complexity but also allows considerably more flexibility. You can create a metastate tree as deep as you wish. One interesting side effect with this kind of system is that it can make going from initial prototype to final implementation a little easier. For example, when implementing the above guard character, the attack and patrol states could start as regular states just to test the concepts and see how they feel in game. Once the character was locked in with animation and a full design, those states could be broken into metastates to manage the low-level nuts and bolts of the AI.

Although you do get a considerable amount of flexibility with a hierarchical approach, it is often not needed. For *Drawn to Life*, we did not find this complexity necessary at all.

12.5 Data-Driving the State Machine

One of the most important things to get right is how to drive this whole system from designer-authored data. Specific implementations are beyond the scope of this chapter and would not be useful since every project will have different needs. The original code on *Drawn to Life* used Lua tables for data. Similar solutions have been implemented in

Unity using C#, which used Unity's inspector and C# reflection for the data. These state machines and transition maps have also been built using XML.

Regardless of which solution you choose, there are several considerations for how to organize this data. The most important is to allow your states and transitions to be parameterized from data. Ideally, you should only have a handful of generic states and transitions, each of which can deliver different behaviors based on those parameters.

For example, consider the AI of a shopkeeper who needs to tend their shop. He or she might wipe down the counter, sweep the floor, tend to shelf of goods, or wait patiently at the shop counter. These all may seem different, but they really are not. At their core, each one paths to a specific target and runs an animation. That is it.

You could take this even further and have this generic state send an optional event to the target. This would allow you to reuse the same state for things like opening windows, restocking inventory, or even striking someone by sending a damage event. This same state could optionally change the internal state of the NPC, so our shopkeeper could eat when hungry. While eating, their hunger needs are fulfilled.

This is the power of data-driving this system. Designers will come up with dozens (or hundreds) of more cases for the simple state described above. Listing 12.5 shows an example of a data-driven `RunAnimationState`.

Listing 12.5. Generic state for running an animation.

```
class RunAnimationState : public State
{
    AnimationId m_animToRun;

public:
    RunAnimationState(GameObject* pOwner)
        :State(pOwner)
    { }

    // Function to load the state definition from XML, JSON,
    // or some other data system. This will fill all the
    // internal data members for this state.
    virtual bool LoadStateDef(StateDef* pStateDef)override;

    virtual void OnEnter() override
    {
        GetOwner()->RunAnimation(m_animToRun);
    }
};
```

12.6 Performance and Memory Improvements

The performance and memory footprint of this system can definitely be improved, and in this section, we will discuss a few of these improvements.

The simplest performance improvement is to time-slice the state machine update by limiting how often the state machine is updated, and how many game objects can update in a single frame. This can be done with an update list where every state machine is put

into a list, and the game only allows a certain amount of time to be dedicated to the AI updates. As long as the transition map stays static during a frame, there is no reason you could not time-slice in the middle of an update either.

Another performance improvement would be to remove state updates entirely and make the whole thing event driven. This is a bit trickier to implement, but it is worth it if the performance of this system is a huge concern. In this version, states have no `OnUpdate()` function, only an `OnEnter()` and `OnExit()`. `OnEnter()` will spin up a task if necessary and will wait for a notification of completion. For example, say you have a `GoToObject` state. The `OnEnter()` function would tell the pathing system to find a path and then do nothing until the path was complete.

This system works well if many of your states do not require an update, or you can easily fit it within an existing system. At the very least, it can help eliminate a few virtual function calls.

On the memory side of things, one big issue is that the state machine class proposed above can be very wasteful if it exists on multiple game objects. For example, say you have a dozen orc enemies, and they all have the same behavior. All twelve of those orcs would be duplicating a lot of data.

One solution here is to split `StateMachine` into two different classes, one for handling the static data that never change at run-time and the other to handle the volatile data that does. The class with the volatile data is given a reference to the static data class. This would not even require a change to the interface. This is effectively the Flyweight design pattern (Freeman-Hargis 2006).

The specific choices of what data to pull out of the run-time class will depend on your game. For example, on *Drawn to Life* the transition maps never changed during run-time, so pulling those into a data class would be perfectly fine.

Along these same lines, you could pull out all of the run-time data from each state instance and store it in a state blackboard. States would be parameterized through this blackboard, allowing you to have only a single instance for each state in your system. Transitions could work the same way; all run-time data would exist on a blackboard for that character so only one instance would be required per class.

12.7 Conclusion

This chapter has only touched on the surface of this kind of architecture. It is very simple and light-weight, yet flexible enough to allow a considerable amount of power and control. This architecture can be applied directly to your AI needs, but there are a few takeaways that can apply to any architecture.

The power of the transition objects to decouple the states from one another comes from a common programming design pattern called the strategy pattern (Gamma et al. 1997). The basic idea is to wrap up an algorithm into a self-contained class so that several such algorithms can be composed at run-time to change behavior. It is very good for data-driven systems since these objects can be instantiated with a factory and composed into a data structure (the transition map in our case) to determine the final behavior. You can even change these at run-time, significantly altering behavior by swapping a few objects.

This system can also be used in conjunction with other systems. For example, you might have a `UtilityTransition` class that scores its target state to see if it is something the

character wants to do. You could have a decision tree as your transition logic, using states only to manage their actions.

Either way, hopefully you see the power and flexibility of this kind of architecture.

References

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1997. *Design Patters: Elements of Reusable Object-Oriented Software.* Upper Saddle River, NJ: Addison-Wesley.

Freeman-Hargis, J. 2006. Using STL and patterns for game AI. In *AI Game Programming Wisdom 3*, ed. S. Rabin. Newton, MA: Charles River Media, pp. 13–28.

Fu, D. and Houlette, R. 2004. The ultimate guide to FSMs in games. In *AI Game Programming Wisdom 2*, ed. S. Rabin. Newton, MA: Charles River Media, pp. 283–302.