CMSC 425: Lecture 20
Artificial Intelligence for Games: Decision Making
Tuesday, Apr 23, 2013

Reading: The material on Behavior Trees has been taken from a nice lecture by Alex Champandard, "Behavior Trees for Next-Gen Game AI," which appears on aigamedev.com (visit: http://aigamedev.com/insider/-presentations/behavior-trees/).

Decision Making: So far, we have discussed how to design AI systems for simple path planning. Path planning is a relatively well defined problem. Most application of AI in game programming is significantly more involved. In particular, we wish to consider ways to model interesting non-player characters (NPCs).

Designing general purpose AI systems that are capable of modeling interesting behaviors is a difficult task. On the one hand, we would our AI system to be general enough to provide a game designer the ability to specify the subtle nuances that make a character interesting. On the other hand we would like the AI system to be easy to use and powerful enough that relatively simple behaviors can be designed with ease. It would be nice to have a library of different behaviors and different mechanisms for combining these bahaviors in intersting ways.

Today we will discuss a number of different methods, ranging from fairly limited to more complex. In particular, we will focus on three different approaches, ranging from simple to more sophisticated.

- Rule-based systems
- Finite state machines
- Behavior trees

Rule-based Systms: The very first computer games used rule-based systems to control the behaviors of the NPCS. A *rule-based system* is one that stores no (or very little) state information, and the behavior of an NPC is a simple function of the present conditions it finds itself in.

As an example, consider planning the motion of one of the ghosts in the game *Pac-Man*.[1] Let's ignore for now the fact that ghosts have two states, depending on whether they are chaising the Pac-Man or they are being chased. Even when they are chasing the Pac-Man, ghosts alternate between two states, depending on whether they are wandering or chasing. Let's consider a simple example of how to use a rule-based system to define wandering behavior (see Fig. 1). This simple system prefers to go ahead whenever possible, and if the way is blocked it selects (in order of decreasing preference) turning right, turning left, and reversing.

Of course, this is too simplistic to be useable in a game, but it illustrates some of the features of a ruled-based system. It is easy to implement (just a look-up table), easy to modify, and easy even for non-programmers to understand. There a number of ways that one could enhance this simple idea. For example, rather than just having a single action for a given set of events, there may be a number of possibilities and randomization is used to select the next option (perhaps with weighted probabilities to favor some actions over others).

[1] Our description is not accurate. There are resources on the Web that provide detailed descriptions of the Pac-Man ghost behaviors. For example, see http://gameinternals.com/post/2072558330/understanding-pac-man-ghost-behavior and http://donhodges.com/pacman_pinky_explanation.htm.

Ahead	**Right**	**Left**	**Action**
Open	–	–	Go ahead
Blocked	Open	–	Turn right
Blocked	Blocked	Open	Turn left
Blocked	Blocked	Blocked	Turn around

Fig. 1: A (ridiculously) simple wandering behavior for a ghost in Pac-Man.

Finite State Machines: The next step up in complexity from a rule-based system is to add a notion of *state* to add complexity to a character's behavior. For example, a character may behave more aggressively when it is healthy and less aggressively when it is injured. As another example, a designer may wish to have a character transition between various states (patrolling, chasing, fighting, retreating) in sequence or when triggered by game events. In each state, the characters behavior may be quite different.

A *finite state machine* (FSM) can be modeled as a directed graph, where each node of the graph corresponds to a state, and each directed edge corresponds to a event, that triggers a change of state and optionally some associated action. The associated actions may include things like starting an animation, playing a sound, or modifying the current game state.

As an example, consider the programming of an enemy combatant "bot" NPC in a first-person shooter that seeks the player's character and engages it in combat. Suppose that as the designer you decide to implement the following type of behavior:

- If you dont see an enemy, wander randomly
- When you see enemy, attack him
- When hear an enemy, chase him
- On dying, re-spawn

An example of a possible implementation of this using an FSM is shown in Fig. 3(a). For example, suppose that we are currently in the *Wander* state. If we see our enemy (that is, the player) we transition to the *Attack* state. If we hear a sound, we transition to the *Chase* state. (This would presumably be followed by querying the game database to determine where the sound came from and the AI system to compute a path.) If we die, we jump to the *Spawn* state, where we presumably wait until we have been revived and exit this state. Note that the "!" indicates that the specified event has not occurred (or that the specified condition is not satisfied).

FSMs are a popular method of defining behavior in games. The principal reasons that they are liked is that they are easy to implement, easy to design (if they are not too big), and they are easy to interpret. For example, based on the graphical layout of our FSM we can observe one interesting anomaly, there is no transition from *Attack* to *Chase*? Was this intentional? To remedy this, we could create an additional state, called *Attack-with-sound-heard*. While we are attacking, if we hear a sound, we could transition to this state. When we are done with the attack, if we are in this new state, we transition immediately to the *Chase* state.

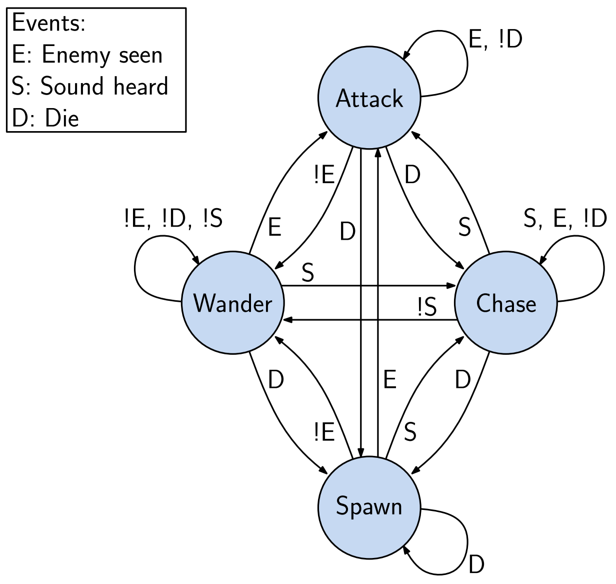

Fig. 2: Implementing an enemy combatant NPC for a FPS game.

How are FSMs implemented? Basically, they can be implemented using a two-dimensional array, where the row index is the current state and the column index (or indices in general) correspond to the events that may trigger a transition.

Note that the FSM we have showed is *deterministic*, meaning that there is only a single transition that can be applied at any time. More variation can be introduced by allowing multiple transitions per event, and then using randomization to select among them (again, possibly with weights so that some transitions are more likely than others).

The principal problem with FSMs is that the number of states can *explode* as the designer dreams up more complex behavior, thus requiring more states, more events, and hence the need to consider a potentially quadratic number of mappings from all possible states to all possible events. For example, suppose that you wanted to model multiple conditions simultaneously. A character might be *healthy/injured*, *wandering/chasing/attacking*, *aggressive/defensive/neutral*. If any combination of these qualities is possible, then we would require $2 \cdot 3 \cdot 3 = 18$ distinct states. This would also result in a number of repeated transitions. (For example, all 9 of the states in which the character is "healthy" would need to provide transitions to the corresponding "injured" states if something bad happens to us. Requiring this much redundancy can lead to errors, since a designer may update some of the transitions, but not the others.)

Hierarchical FSMs: One way to avoid the explosion of states and events is to design the FSM in a hierarchical manner. First, there are a number of high-level states, corresponding to very broad contexts of the character's behavior. Then within each high-level state, we could have many sub-states, which would be used for modeling more refined behaviors within this state. The resulting system is called a *hierarchical finite state machine* (HFSM). This could be implemented, for example, by storing the state on a stack, where the highest-level state descriptor is pushed first, then successively more local states.

Returning to our fighting bot example, each of the major states, *Wander*, *Attack*, *Chase*, and *Spawn*, could be further subdivided into lower level states. For example, we could design different types of chasing behavior (chase on foot, chase while riding a unicyle, chase while flying a hovercraft).

The process of looking up state transitions would proceed hierarchically as well. First, we would check whether the lowest level sub-state has any transition for handling the given event. If not, we could check its parent state in the stack, and so on, until we find a level of the FSM hierarchy where this event is to be handled.

The advantage of this hierarchical approach is that it naturally adds modularity to the design process. Because the number of local sub-states is likely to be fairly small, it simplifies the design of the FSM. In particular, we can store even a huge number of states because each sub-state level need only focus on the relatively few events that can cause transitions at this level.

Other Approaches to Hierarchical Decision Making: A good general, scalable approach is to design the AI decision system *hierarchically*. There are three classical methods for achieving a hierarchical AI structure:

Programming systems: Programming and scripting systems are very powerful (Turing complete). They are very good at describing sequential, conditional, and repetitive behaviors, since these constructs are common to all programming languages. The downside is that they are so general that it is hard to reuse components in other games or to compose components to form new behaviors.

Hierarchical Finite-State Machines (HFSM): As we have seen, these are easy to design for moderately sized systems. It is easy to generate a bunch of states and then consider the conditions under which one state leads to another. They have a nice modular structure, which makes it possible to copy FSMs from one game to another. However, FSMs and HFSMs can be clunky to deal with when designing "procedural" behaviors (do x, then y, then repeat z 20 times), which are more easily handled in a programming/scripting system.

Hierarchical Planners: AI planning systems are software systems that are used for searching among a complex state of possible actions to determine the best course of action. An example (which we will not discuss) is called a *hierarchical task network* (HTN). Planning-based systems are very powerful, and can be very useful when dealing with complex decision making. Because the planning process can take a lot of computational resources, HTN software systems tend to be rather slow. Also they do not adapt well to highly dynamic contexts, since any change in the system may require that the (time-consuming) planner be re-run from scratch. Planners are often overkill for many of the simple decisions that need to be made in typical games.

Behavior Trees: The question that this raises is whether there is a system that combines the strengths of these various systems. We would like a system that is more general the FSMs, more structured than programs, and lighter weight than planners. Behavior trees were developed by Geoff Dromey in the mid-2000s in the field of software engineering, which provides a modular way to define software in terms of actions and preconditions. They were first used in Halo 2 and were adopted by a number of other games such as Spore.

Following Alex Champandard's example, let us consider the modeling of a *guard dog* in an FPS game. The guard dog's range of behaviors can be defined hierarchically. At the topmost level, the dog has behaviors for major tasks, such as *patrolling*, *investigating*, and *attacking* (see Fig. 3(a)). Each of these high-level behaviors could then be broken down further into lower-level behaviors. For example, the patrol task may include a subtask for *moving*. The investigate task might include a subtask for *looking around*, and the attack task may include a subtask for *bite* (ouch!).

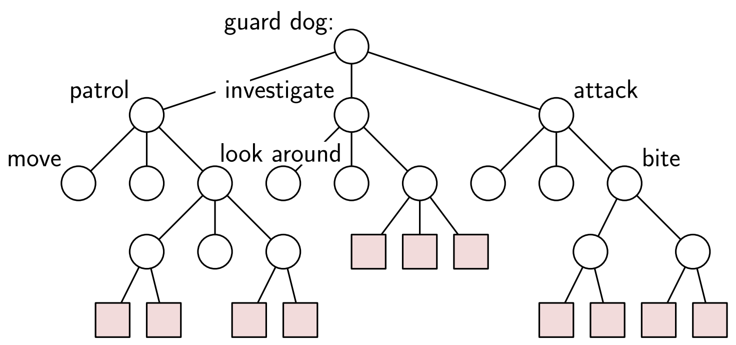

Fig. 3: Sample hierarchical structure of a guard-dog's behavior.

The leaves of the tree are where the AI system interacts with the game state. Leaves provide a way to gather information from the system through *conditions*, and a way to affect the progress of the game through *actions*. In the case of our guard dog, conditions might involve issues such as the dog's state (is the dog hungry or injured) or geometric queries (is there another dog nearby, and is there a line of sight to this dog?). Conditions are *read-only*. Actions make changes to the world state. This might involve performing an animation, playing a sound, picking up an object, or biting someone (which would presumably alter this other object's state). Conditions can be thought of as *filters* that indicate which actions are to be performed.

A *task* is a piece of code that models a latent computation. A task consists of a collection *conditions* that determine when the task is enabled and *actions*, which encode the execution of the task. Tasks can end either in *success* or *failure*.

Composing Tasks: *Composite tasks* provide a mechanism for composing a number of different tasks. There are two basic types of composite tasks, which form natural complements.

Sequences: A sequence task performs a series of tasks sequentially, one after the other (see Fig. 4(a)). As each child in the sequence succeeds, we proceed to the next one. Whenever a child task fails, we terminate the sequence and bail out (see Fig. 4(b)). If all succeed, the sequence returns success.

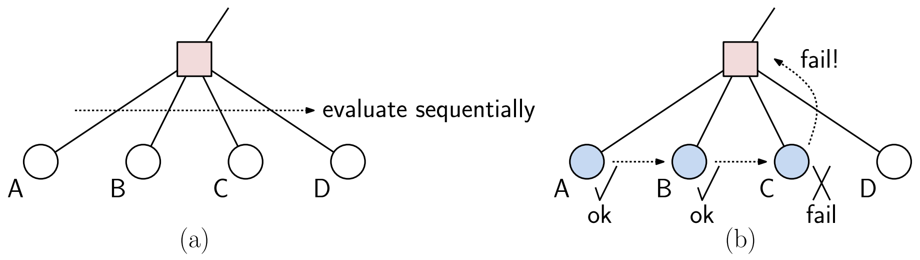

Fig. 4: Sequence: (a) structure and (b) semantics.

Selector: A selector task performs at most one of a collection of child tasks. A selector starts by selecting the first of its child tasks and attempts to execute it. If the child succeeds, then the selector terminates successfully. If the child fails, then it attempts to execute the next child, and so on, until one succeeds (see Fig. 5(b)). If none succeed, then the selector returns failure.

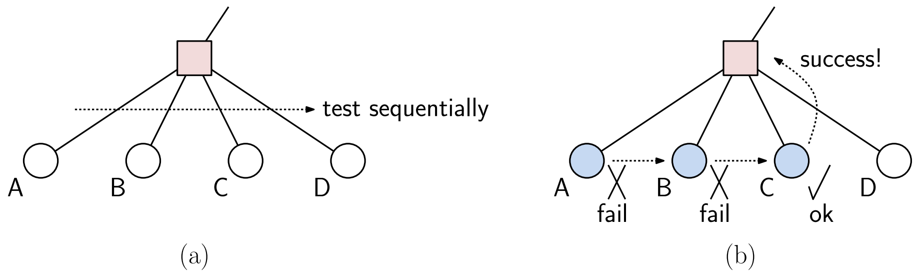

Fig. 5: Selector: (a) structure and (b) semantics.

Sequences and selectors provide some of the missing elements of FSMs, but they provide the natural structural interface offered by hierarchical finite state machines. Sequences and selectors can be combined to achieve sophisticated combinations of behaviors. For example, a behavior might involve a sequence of tasks, each of which is based on making a selection from a list of possible subtasks. Thus, they provide building blocks for constructing more complex behaviors.

From a software-engineering perspective, behavior trees give a programmer a more structured context in which to design behaviors. The behavior-tree structure forces the developer to think about the handling of success and failure, rather than doing so in an ad hoc manner, as would be the case when expressing behaviors using a scripting language. Note that the nodes of the tree, conditions and tasks, are simply links to bits of code that execute the desired test or perform the desired action. The behavior tree provides the structure within which to organize these modules.