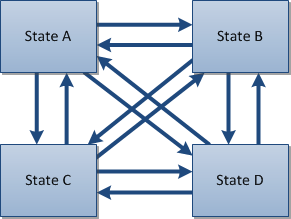
https://habr.com/ru/post/173187/

**Конечные автоматы.**



Базовой частью FSM является состояние. Это, то что агент AI делает в конкретный заданный момент времени.

Всё, о чем требуется знать агенту, содержится в коде для состояния, в котором он находится. К примеру, анимации, которые ему необходимо “проиграть” для конкретного состояния, заложены в теле кода этого состояния. Другая часть конечного автомата состояний – это необходимая “логика” определения того, что нужно делать дальше. Сюда можно также отнести переход в другое состояние — или наоборот, сохранение своего состояния.

Чаще всего конечные автоматы используют сложные механизмы срабатывания, которые связаны с игровой логикой и ситуацией.

Итак, для каждого состояния существует некий код, который отвечает за то, что делать, пока мы находимся в этом состоянии, и (что тоже важно) – какое существует условие перехода из него, и куда мы попадем в результате. И хотя некоторые из условий будут использовать одни и те же проверки, в итоге каждое состояние имеет свой собственный набор переходов в другие состояния, и эти переходы используются исключительно для него.

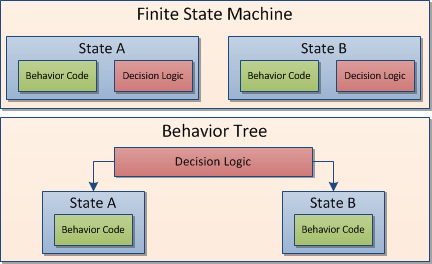
Минусы:

1. С добавлением новых состояний к конечному автомату число переходов начинает расти в разы быстрее. Количество переходов = [кол-во состояний] \* ([кол-во состояний] – 1).

2. Главный минус - количество труда, который приходится проделывать при добавлении каждого нового состояния к существующим. Для того, чтобы сделать состояние достижимым, вам придется пройтись по всем другим состояниям, которые теоретически могут переходить в новое.

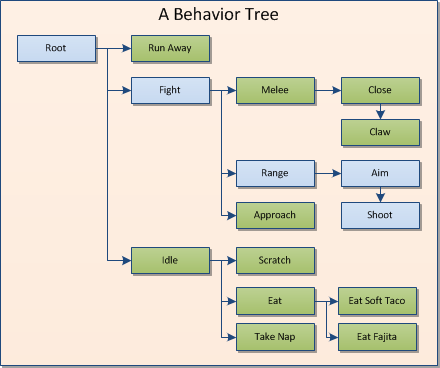
3. Любая логика, которая вовлечена в переход, также нуждается в пересмотре — для того, чтобы ее можно было совместить с той логикой, которая уже была здесь до этого.

**Behavior tree.**



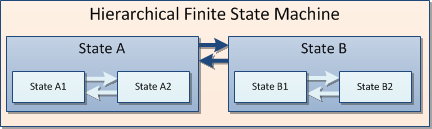
Логика принятия решений отделена от кода самого состояния. Обе они все еще находятся в коде AI, но они не связаны настолько тесно, что логика принятия решений находится прямо в коде конкретного состояния. Вместо этого она вынесена в отдельную самостоятельную архитектуру.

Такое решение позволяет ей запускать саму себя – непрерывно или по необходимости – где она выбирает, в каком состоянии агент должен находиться. Весь код состояния отвечает за действия, которые уникальны для этого состояния – анимация, изменение значений во внешнем мире и прочее.



Главным преимуществом такого подхода становится то, что вся логика принятия решений находится в одном месте. Мы можем сделать ее настолько сложной и запутанной, насколько захочется, — и нам не придется переживать при этом насчет обязательной синхронизации между различными состояниями. Если мы добавляем новое поведение, то мы добавляем код для его вызова в одно место вместо того, чтобы проходить по всем существующим состояниям. Если же нам надо отредактировать логику перехода для определенного поведения, то это тоже надо будет сделать всего в одном месте.

**Иерархический конечный автомат.**

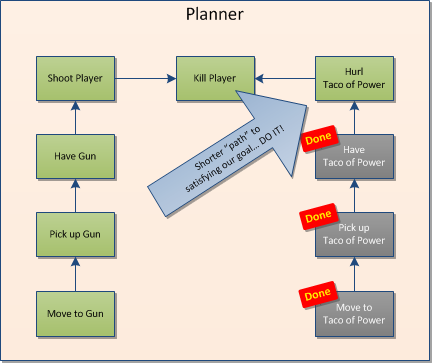


В иерархической FSM некоторые из состояний содержат другие состояния, что делает управление связанными состояниями удобнее.

В иерархическом конечном автомате (HFSM) есть несколько уровней состояний. Тогда состояния верхнего уровня будут заняты взаимодействиями только с состояниями своего уровня. С другой стороны, состояния нижнего уровня, находящиеся внутри родителя, могут переходить только друг в друга. Подобное многоуровневое разделение ответственности придает плоскому FSM дополнительную организацию и позволяет взять часть сложности под контроль.

**Планировщик.**

Как и в случае «behavior tree», толчком к использованию планировщика в качестве архитектуры служит отделение AI от кода, который «делает все остальное». Планировщик сравнивает свою ситуацию – состояние мира в определенный момент – с коллекцией индивидуальных атомарных действий, которые он может сделать. Следом он объединяет одну или несколько этих задач в последовательность («план»), так чтобы его текущая цель была достигнута.



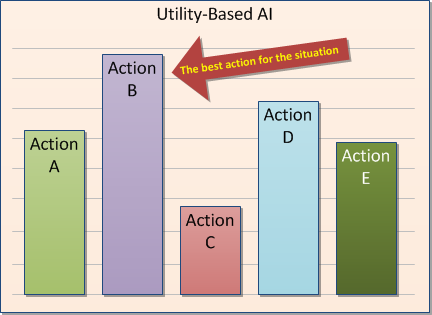
В отличие от других архитектур, которые “идут” от своего текущего состояния и смотрят “вперед”, планировщик работает в обратном направлении от своей цели. Например, если целью является “убить игрока”, то планировщик может выяснить, что подходящим методом для данной цели будет “пристрелить игрока”. Конечно же, для этого потребуется ружье. Если у агента не окажется ружья, то ему надо будет его найти. Если поблизости ничего такого нет, тогда ему придется пойти туда, где по его сведениям оно есть. Если он ничего такого не знает, то стоит начать поиски. Результатом поиска в обратном порядке станет план, который выполняется в прямом порядке.

Планирование расходится с FSM и BT в том, что, в отличие от них, в нем нет жесткого управления действиями. В этом и заключается разница между планировщиками – они находят решение для ситуаций, основываясь на том, что можно сделать и как эти возможные действия могут быть связаны между собой. Прелесть использования подобной структуры состоит в том, что при возникновении новых производных ситуаций она зачастую самостоятельно найдет способ действия.

**Метод оценки полезности (“utility-based”)**

Как и в случае планировщика, эта система не имеет предопределенных действий. Вместо этого, решения о потенциальных действиях принимаются на основании взвешивания факторов – почему это хорошо? почему это плохо? – и выбором самого подходящего решения.

Вместо того, чтобы строить план, как это сделал бы планировщик, подобная система просто выберет следующий возможный шаг.



Как и в случае с «behavior tree» или планировщиками, система с оценкой полезности действия полагается на реакции на события. После того, как действие выбрано, агент будет должен выполнить переход в это состояние, ну а на плечах системы лежит ответственность за выбор, какое состояние станет следующим. Однако, код принятия решений, как и в случае предыдущих архитектур, будет содержаться в одном месте. Это делает настройку и изменение системы заметно удобнее, да и преимущество планировщика в плане простоты добавления новых поведений тоже сохраняется. При добавлении поведения с уже распределенными весами AI автоматически принимает его во внимание и начинает использовать в необходимых ситуациях.

С другой стороны, недостаток подобной системы как раз в том, что не всегда возможно предсказать, что в конечном результате произойдет. В случае с деревом BT можно легко совершить его обход и найти ветви и узлы, которые будут активны в конкретной ситуации. В случае с нашей системой ее решения сложно представить в двоичном виде “да — нет”, и определение ее поведения становится задачей нетривиальной. Все это не к тому, что в подобной системе AI будет неконтролируемым или неконфигурируемым. На самом деле, такие системы, наоборот, предоставляют глубокий уровень контроль. Разница в том, что вместо того, чтобы под вашим чутким командованием выбирать, что стоит делать в конкретной ситуации, система сама предлагает возможные варианты – и эта способность может оказаться неплохим вариантом для AI игрового проекта. В этом плане система с оценкой полезности имеет общие черты с планировщиками – AI просто смотрит на доступные возможности и затем решает, какие из них подходят лучше всего.