Конечный автомат: теория и реализация

Конечный автомат — это некоторая абстрактная модель, содержащая конечное число состояний чего-либо. Используется для представления и управления потоком выполнения каких-либо команд. Конечный автомат идеально подходит для реализации искусственного интеллекта в играх, получая аккуратное решение без написания громоздкого и сложного кода. Рассмотрим теорию, а также узнаем, как использовать простой и основанный на стеке конечный автомат.

Что такое конечный автомат?

Конечный автомат (или попросту FSM — Finite-state machine) это модель вычислений, основанная на гипотетической машине состояний. В один момент времени только одно состояние может быть активным. Следовательно, для выполнения каких-либо действий машина должна менять свое состояние.

Конечные автоматы обычно используются для организации и представления потока выполнения чего-либо. Это особенно полезно при реализации ИИ в играх. Например, для написания «мозга» врага: каждое состояние представляет собой какое-то действие (напасть, уклониться и т. д.).

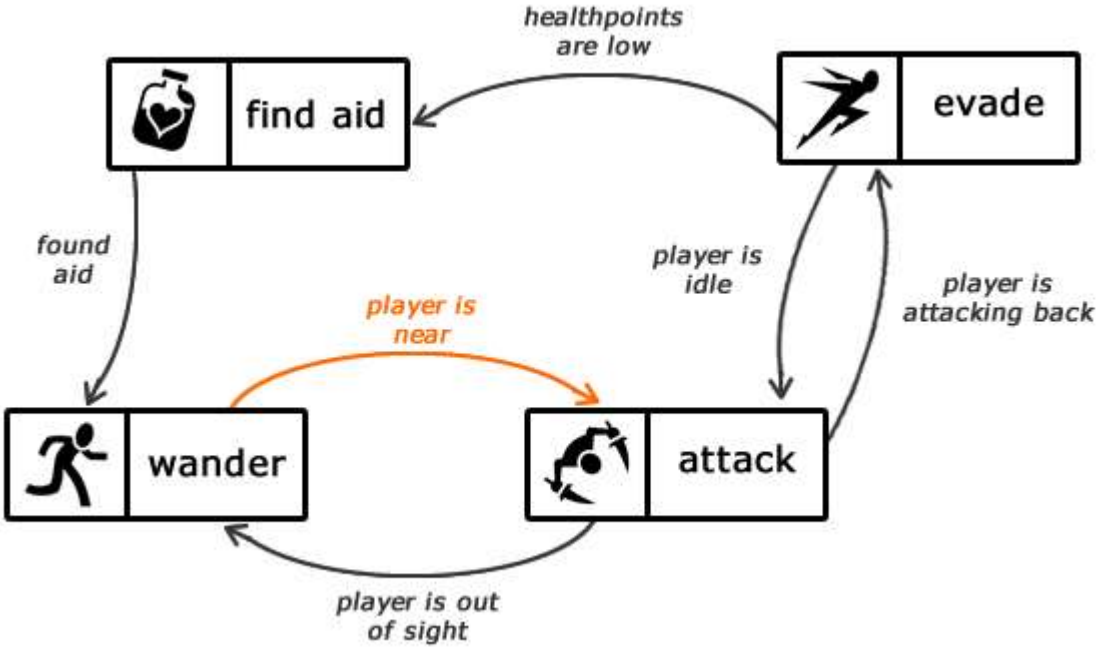

Рисунок 1. Описание состояний автомата

Конечный автомат можно представить в виде графа, вершины которого являются состояниями, а ребра – переходы между ними. Каждое ребро имеет метку, информирующую о том, когда должен произойти переход. Например, на изображении выше видно, что автомат сменит состояние «wander» на состояние «attack» при условии, что игрок находится рядом.

Планирование состояний и их переходов

Реализация конечного автомата начинается с выявления его состояний и переходов между ними. Представьте себе конечный автомат, описывающий действия муравья, несущего листья в муравейник:

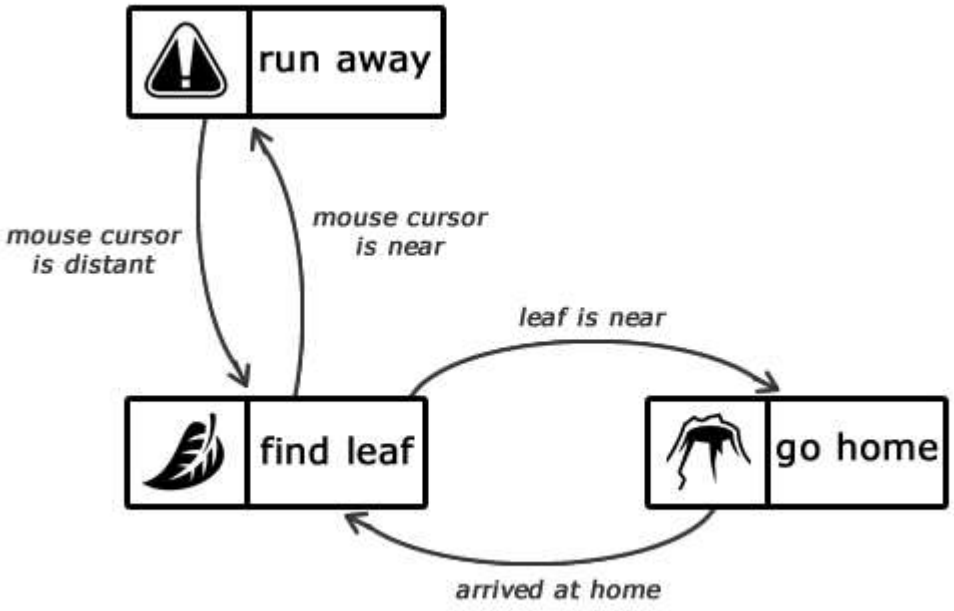

Рисунок 2. Описание состояний интеллекта муравья

Отправной точкой является состояние «find leaf», которое остается активным до тех пор, пока муравей не найдет лист. Когда это произойдет, то состояние сменится на «go home». Это же состояние останется активным, пока наш муравей не доберется до муравейника. После этого состояние вновь меняется на «find leaf».

Обратите внимание на то, что при направлении домой или из дома муравей не будет бояться курсора мыши. Почему? А потому что нет соответствующего перехода.

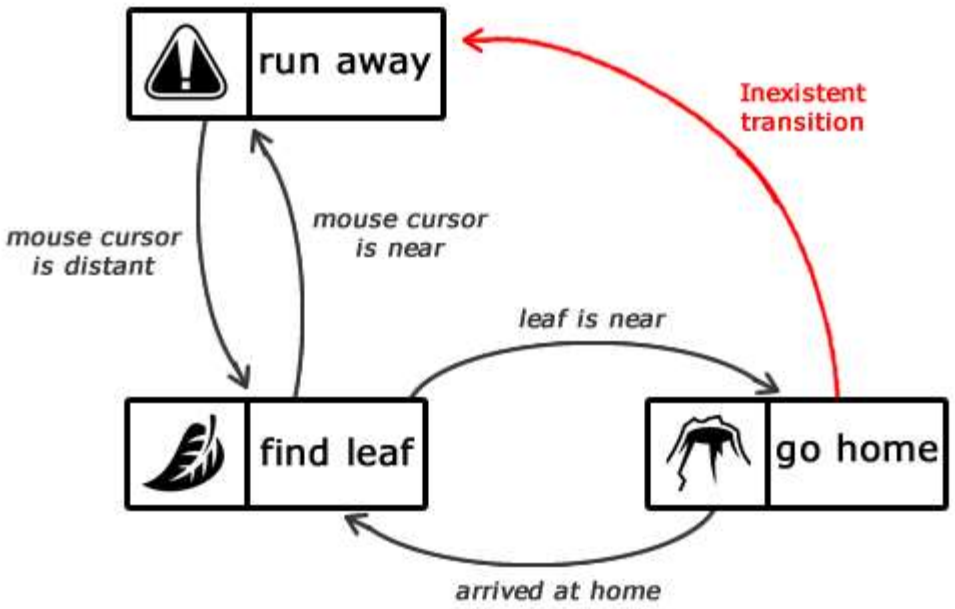

Рисунок 3. Описание состояний интеллекта муравья. Обратите внимание на отсутствие перехода между «run away» и «go home»

Реализация простого конечного автомата

Конечный автомат можно реализовать при помощи одного класса. Назовем его FSM. Идея состоит в том, чтобы реализовать каждое состояние как метод или функцию. Также будем использовать свойство activeState для определения активного состояния.

```
public class FSM {
    private var activeState :Function; // указатель на активное
состояние автомата

    public function FSM() {
    }

    public function setState(state :Function) :void {
        activeState = state;
    }

    public function update() :void {
        if (activeState != null) {
            activeState();
        }
    }
}
```

Всякое состояние есть функция. Причем такая, что она будет вызываться при каждом обновлении кадра игры. Как уже говорилось, в activeState будет храниться указатель на функцию активного состояния.

Метод update() класса FSM должен вызываться каждый кадр игры. А он, в свою очередь, будет вызывать функцию того состояния, которое в данный момент является активным.

Метод setState() будет задавать новое активное состояние. Более того, каждая функция, определяющая какое-то состояние автомата, не обязательно должна принадлежать классу FSM – это делает наш класс более универсальным.

Использование конечного автомата

Давайте реализуем ИИ муравья. Выше мы уже показывали набор его состояний и переходов между ними. Проиллюстрируем их еще раз, но в этот раз сосредоточимся на коде.

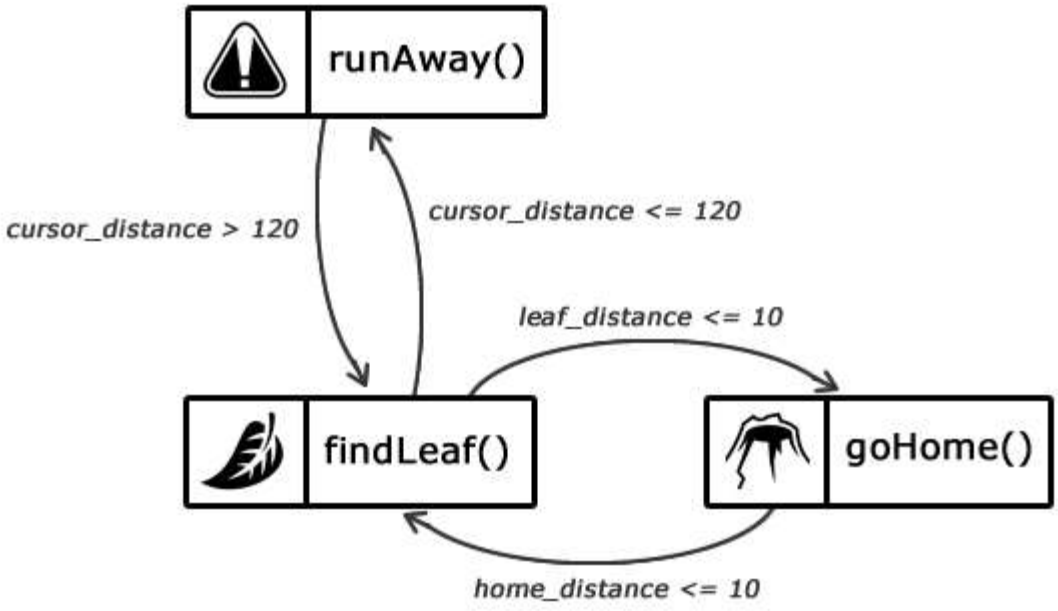

Рисунок 4. Описание состояний интеллекта муравья, сосредоточенное на коде

Наш муравей представлен классом Ant, в котором есть поле brain. Это как раз экземпляр класса FSM.

```
public class Ant
{
    public var position   :Vector3D;
    public var velocity   :Vector3D;
```

```
    public var brain      :FSM;

    public function Ant(posX :Number, posY :Number) {
        position    = new Vector3D(posX, posY);
        velocity    = new Vector3D( -1, -1);
        brain       = new FSM();

        // Начинаем с поиска листка.
        brain.setState(findLeaf);
    }

    /**
     * Состояние "findLeaf".
     * Заставляет муравья искать листья.
     */
    public function findLeaf() :void {
    }

    /**
     * Состояние "goHome".
     * Заставляет муравья идти в муравейник.
     */
    public function goHome() :void {
    }

    /**
     * Состояние "runAway".
     * Заставляет муравья убегать от курсора мыши.
     */
    public function runAway() :void {
    }

    public function update():void {
        // Обновление конечного автомата. Эта функция будет
вызывать
        // функцию активного состояния: findLeaf(), goHome() или
runAway().
        brain.update();

        // Применение скорости для движения муравья.
        moveBasedOnVelocity();
    }

    (...)
}
```

Класс Ant также содержит свойства velocity и position. Эти переменные будут использоваться для расчета движения с помощью метода Эйлера. Функция update() вызывается при каждом обновлении кадра игры.

Для понимания кода мы опустим реализацию метода moveBasedOnVelocity(). Если хотите узнать поподробнее на тему движения, прочитайте серию статей Understanding Steering Behaviors.

Ниже приводится реализация каждого из методов, начиная с findLeaf() – состояния, ответственного за поиск листьев.

```
public function findLeaf() :void {
    // Перемещает муравья к листу.
    velocity = new Vector3D(Game.instance.leaf.x - position.x,
Game.instance.leaf.y - position.y);

    if (distance(Game.instance.leaf, this) <= 10) {
        // Муравей только что подобрал листок, время
        // возвращаться домой!
        brain.setState(goHome);
    }

    if (distance(Game.mouse, this) <= MOUSE_THREAT_RADIUS) {
        // Курсор мыши находится рядом. Бежим!
        // Меняем состояние автомата на runAway()
        brain.setState(runAway);
    }
}
```

Состояние goHome() — используется для того, чтобы муравей отправился домой.

```
public function goHome() :void {
    // Перемещает муравья к дому
    velocity = new Vector3D(Game.instance.home.x - position.x,
Game.instance.home.y - position.y);

    if (distance(Game.instance.home, this) <= 10) {
        // Муравей уже дома. Пора искать новый лист.
        brain.setState(findLeaf);
    }
}
```

И, наконец, состояние runAway() – используется при уворачивании от курсора мыши.

```
public function runAway() :void {
    // Перемещает муравья подальше от курсора
```

```
        velocity = new Vector3D(position.x - Game.mouse.x,
    position.y - Game.mouse.y);

        // Курсор все еще рядом?
        if (distance(Game.mouse, this) > MOUSE_THREAT_RADIUS) {
            // Нет, уже далеко. Пора возвращаться к поискам
    листочков.
            brain.setState(findLeaf);
        }
    }
```

Улучшение FSM: автомат, основанный на стеке

Представьте себе, что муравью на пути домой также нужно убегать от курсора мыши. Вот так будут выглядеть состояния FSM:

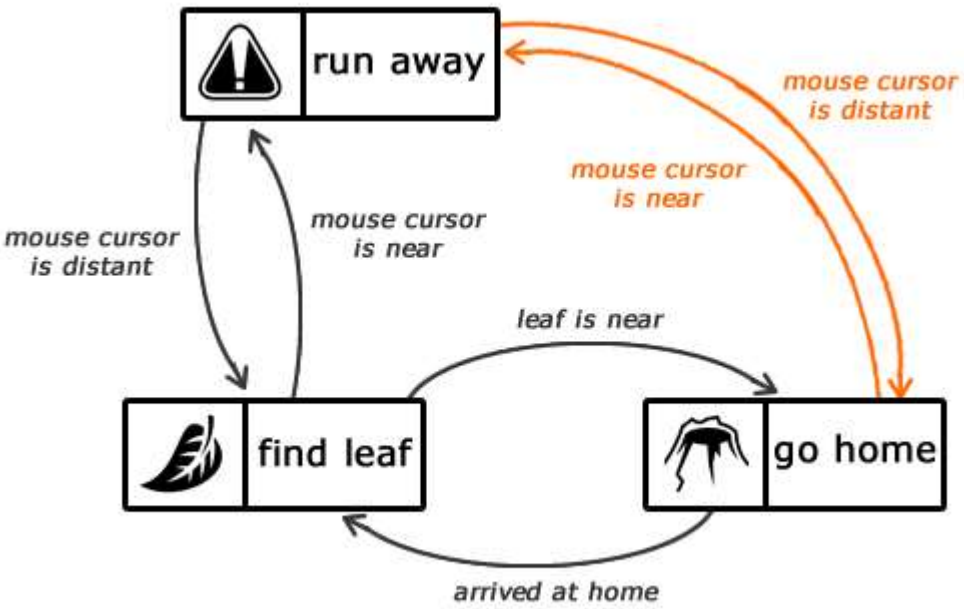

Рисунок 5. Обновленное описание состояний интеллекта муравья

Кажется, что изменение тривиальное. Нет, такое изменение создает нам проблему. Представьте, что текущее состояние это «run away». Если курсор мыши отдаляется от муравья, что он должен делать: идти домой или искать лист?

Решением такой проблемы является конечный автомат, основанный на стеке. В отличие от простого FSM, который мы реализовали выше, данный вид FSM использует стек для управления состояниями. В верхней части стека находится активное состояние, а переходы возникают при добавлении/удалении состояний из стека.

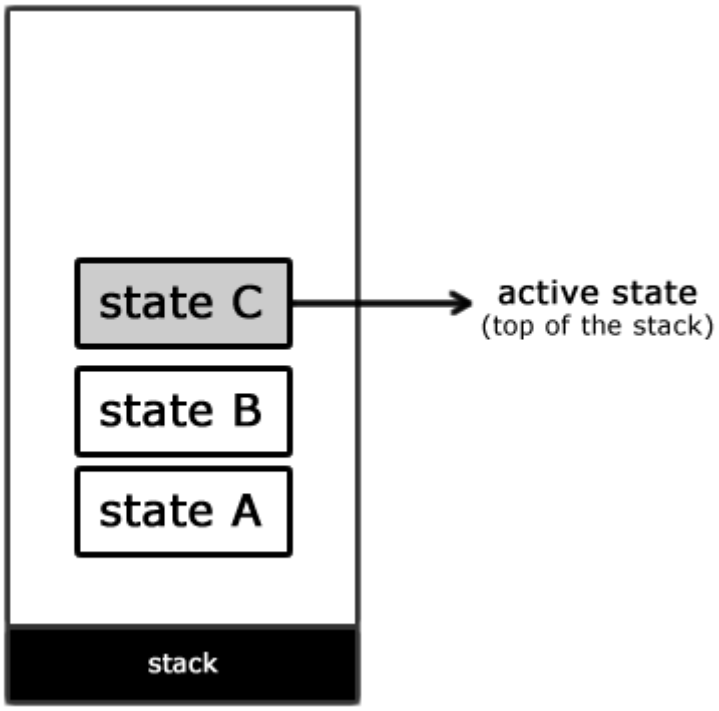

Рисунок 6. Конечный автомат, основанный на стеке

Реализация FSM, основанного на стеке

Такой конечный автомат может быть реализован так же, как и простой. Отличием будет использование массива указателей на необходимые состояния. Свойство activeState нам уже не понадобится, т.к. вершина стека уже будет указывать на активное состояние.

```
public class StackFSM {
    private var stack :Array;

    public function StackFSM() {
        this.stack = new Array();
    }

    public function update() :void {
        var currentStateFunction :Function = getCurrentState();

        if (currentStateFunction != null) {
            currentStateFunction();
        }
    }

    public function popState() :Function {
        return stack.pop();
    }
```

```
    public function pushState(state :Function) :void {
        if (getCurrentState() != state) {
            stack.push(state);
        }
    }

    public function getCurrentState() :Function {
        return stack.length > 0 ? stack[stack.length - 1] :
null;
    }
}
```

Обратите внимание, что метод setState() был заменен на pushState() (добавление нового состояния в вершину стека) и popState() (удаление состояния на вершине стека).

Использование FSM, основанного на стеке

Важно отметить, что при использовании конечного автомата на основе стека каждое состояние несет ответственность за свое удаление из стека при отсутствии необходимости в нем. Например, состояние attack() само должно удалять себя из стека в том случае, если враг был уже уничтожен.

```
public class Ant {
    (...)
    public var brain :StackFSM;

    public function Ant(posX :Number, posY :Number) {
        (...)
        brain = new StackFSM();

        // Начинаем с поиска листка
        brain.pushState(findLeaf);

        (...)
    }

    /**
     * Состояние "findLeaf".
     * Заставляет муравья искать листья.
     */
    public function findLeaf() :void {
        // Перемещает муравья к листу.
        velocity = new Vector3D(Game.instance.leaf.x -
position.x, Game.instance.leaf.y - position.y);
```

```
        if (distance(Game.instance.leaf, this) <= 10) {
            //Муравей только что подобрал листок, время
            // возвращаться домой!
            brain.popState(); // removes "findLeaf" from the
stack.
            brain.pushState(goHome); // push "goHome" state,
making it the active state.
        }

        if (distance(Game.mouse, this) <= MOUSE_THREAT_RADIUS) {
            // Курсор мыши рядом. Надо бежать!
            // Состояние "runAway" добавляется перед "findLeaf",
что означает,
            // что состояние "findLeaf" вновь будет активным при
завершении состояния "runAway".
            brain.pushState(runAway);
        }
    }

    /**
     * Состояние "goHome".
     * Заставляет муравья идти в муравейник.
     */
    public function goHome() :void {
        // Перемещает муравья к дому
        velocity = new Vector3D(Game.instance.home.x -
position.x, Game.instance.home.y - position.y);

        if (distance(Game.instance.home, this) <= 10) {
            // Муравей уже дома. Пора искать новый лист.
            brain.popState(); // removes "goHome" from the
stack.
            brain.pushState(findLeaf); // push "findLeaf" state,
making it the active state
        }

        if (distance(Game.mouse, this) <= MOUSE_THREAT_RADIUS) {
            // Курсор мыши рядом. Надо бежать!
            // Состояние "runAway" добавляется перед "goHome",
что означает,
            // что состояние "goHome" вновь будет активным при
завершении состояния "runAway".
            brain.pushState(runAway);
        }
    }
```

```
    /**
     * Состояние "runAway".
     * Заставляет муравья убегать от курсора мыши.
     */
    public function runAway() :void {
        // Перемещает муравья подальше от курсора
        velocity = new Vector3D(position.x - Game.mouse.x,
position.y - Game.mouse.y);

        // Курсор все еще рядом?
        if (distance(Game.mouse, this) > MOUSE_THREAT_RADIUS) {
            // Нет, уже далеко. Пора возвращаться к поискам
листочков.
            brain.popState();
        }
    }
    (...)
}
```

Вывод

Конечные автоматы, безусловно, полезны для реализации логики искусственного интеллекта в играх. Они могут быть легко представлены в виде графа, что позволяет разработчику увидеть все возможные варианты.

Реализация конечного автомата с функциями-состояниями является простым, но в то же время мощным методом. Даже более сложные переплетения состояний могут быть реализованы при помощи FSM.