Digital DI Ideation

Finite-state machine for turn-based combat in video games

Studio	**Game Extended 2019**
Written by	Dominic Lüönd, Digital Ideation, 3rd semester, 2019
Lecturers	Dragica Kahlina, Sebastian Hollstein

Lucerne University of Applied Sciences and Arts

HOCHSCHULE LUZERN

dominic.lueoend@stud.hslu.ch

1. Abstract 3
2. Keywords 3
3. Introduction 3
4. Categorization of finite-state machines 3
   Hard-coded finite-state machines 3
   Class-based finite-state machines 4
   Hierarchical finite-state machines 4
   Unity ScriptableObjects finite-state machines 4
5. Implementation for turn-based Combat 5
6. Technical Demo 5
7. Conclusion 6
8. Bibliography 7

1. Abstract

In this paper I describe a modular reusable solution for utilizing finite-state machines in turn-based games. This was achieved through literature research and the development of a prototype.

2. Keywords

finite-state machine, finite-state automaton, artificial intelligence, turn-based games, turn-based combat.

3. Introduction

I have been working on a turn-based RPG for several months now. I realized early on, that a solution to manage states was needed. After diving head-first into development, it became obvious, that my system was lacking structure and thus made progress very difficult. For this reason, I decided to redo this in the studio-module *Studio Game Extended 2019,* which took place for eight days.

16.10	17.10	18.10	23.10	24.10	25.10	26.10	30.11	31.11
Research	Research Konzeption	Definition States 'Competitor' Analysis Prototype	Implementation	Implementation	Implementation	Implementation	Preperation for Presentation	Presentation

4. Categorization of finite-state machines

There are multiple approaches on how to implement a finite-state machine. In Chapter 5.3 of *Artificial Intelligence for Games, Millington, I., Funge, J. (2009)* there are three primary ways to do this. Through research I came across *Lecture 03 – Finite State Machines, Edirlei Soares de Lima (2018)* which brought a fourth solution to my attention. In this solution *ScriptableObjects* in Unity are utilized to achieve the desired state-management.

Hard-coded finite-state machines

Hard-coded state machines are easy to write but are exceptionally difficult to maintain, since complex finite-state machines require thousands of lines of code.

There certainly are rare situations where the speed of this solution makes it a formidable option, though the complexity must be relatively low.

```
public class HardCodedStateMachine :
MonoBehaviour {
    public State state;

    private void Update() {
        switch (state) {
            case State.OutOfCombat:
                break;
            case State.PlayerChoice:
                break;
            case State.EnemyChoice:
                break;
            case State.EndCombat:
                break;
            default:
                break;
        }
    }
}
public enum State {
    OutOfCombat, PlayerChoice,
EnemyChoice, EndCombat }
```

Figure 1 - Hard coded state machine example

Class-based finite-state machines

A class-based approach increases the flexibility of the finite-state machine but reduces its performance due to and increased number of method calls.

As is visible in Figure 2, we work with independent state-objects, each with their own logic. Those instantiations of the 'state'-class are then "linked" through transitions, which in turn require a condition to be met, before the current state changes.

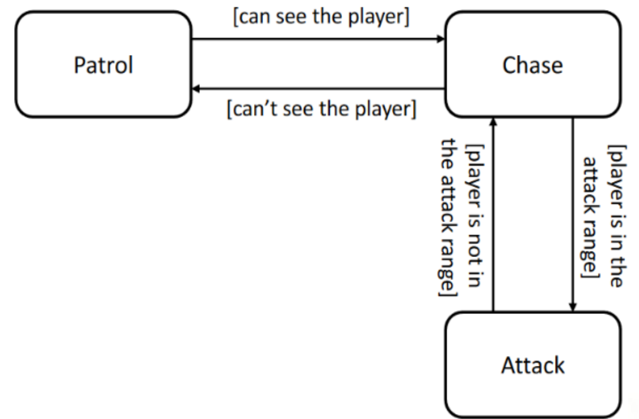

Figure 2 - Class-based state machine visualization

Hierarchical finite-state machines

The hierarchical approach to finite-state machines reduces the complexity of the finite-state machine by breaking it down into multiple smaller state machines. Other than that, this approach is very similar to the class-based method.

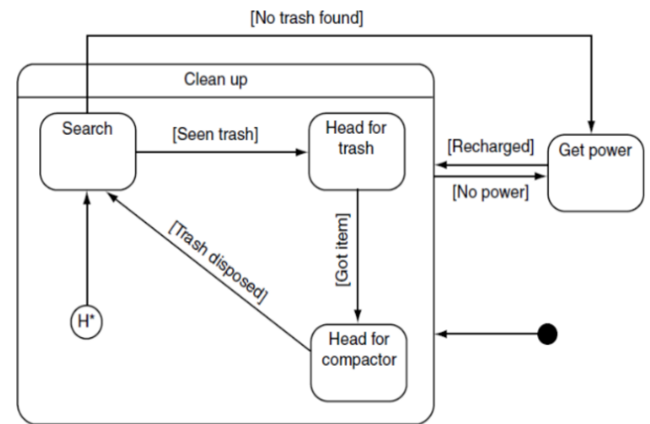

Figure 3 - Hierarchical finite-state machine example

Unity ScriptableObjects finite-state machines

In Unity, a *ScriptableObject* is a class that allows you to store data and execute code independent from script instances. Once a *ScriptableObject*-derived class has been defined, it is possible to use the *CreateAssetMenu* attribute to easily create custom assets of the class directly in the editor.

Figure 4 - Newly created AssetMenu directly in Unity Editor

5. Implementation for turn-based Combat

My specific implementation started with defining the states which I require. I came up with the following: *OutOfCombat, StartCombat, PlayerTurn, EnemyTurn, TargetSelection, EndCombat*. I am not entirely sure if this is sufficient or if some could even be left out (like *EndCombat* and instead transition directly to *OutOfCombat*) but for now this seems like a reasonable approach.

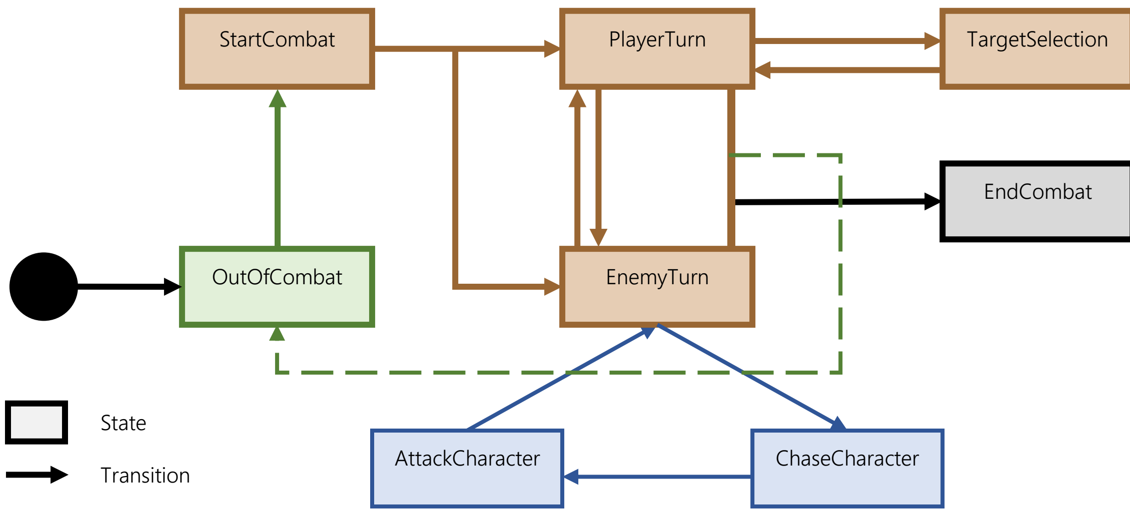

Figure 5 - Visualization of my finite-state machine

A possible extension of this implementation are the blue states and transitions which would make the enemy behavior more clearly separated.

6. Technical Demo

As a tool for my presentation as well as further work on the system, I created a small demo featuring a player, enemies and obstacles made from geometric primitives.

The player as well as the enemies are *Characters* and possess attributes such as *Action Points*, which they require during turn-based combat to perform actions.

While the current *State* is set to *OutOfCombat*, the player can move around without consuming *Action Points*. Once he is near enemies though, the finite-state machine will *SetupCombat* and thus create a turn order, aswell as restrict unpermitted movement of the player. According to this turn order, the player and enemies take turns utilizing their *Action Points* until either side has no more combatants.

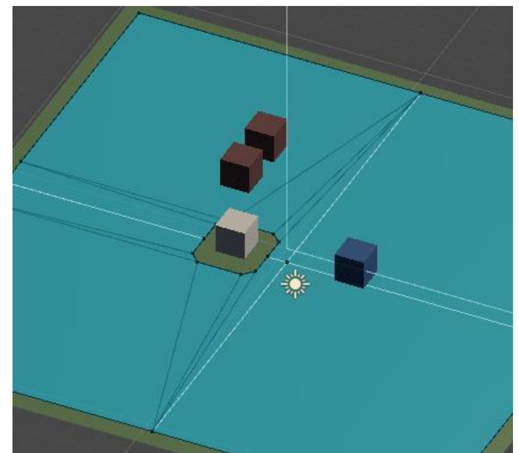

Figure 6 - Navmesh and playarea

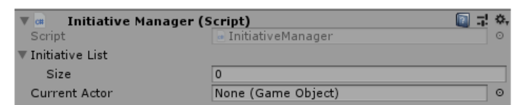

Figure 7 - Editor during OutOfCombat

Figure 8 - Editor after assigning turn order

7. Conclusion

The two main concepts I've grasped are firstly; the importance of research prior to development. I spent less time overall on this project than I did on my previous attempt and achieved a lot more. And secondly; detailed conceptual work prior to development tremendously reduce the amounts of standstills during development.

These insights were very important for me personally and for my progress as an aspiring game developer.

8. Bibliography

Artificial Intelligence for Games, Millington, I., Funge, J. (2009)

Game Programming Gems 1, Mark DeLoura (2000)

Game Programming Gems 5, Kim Pallister (2005)

Lecture 03 – Finite State Machines, Edirlei Soares de Lima (2018)
http://edirlei.3dgb.com.br/aulas/game-ai/GAME_AI_Lecture_03_Finite_State_Machines_2018.pdf

Figure 1 - Hard coded state machine example 3
Figure 2 - Class-based state machine visualization 4
Figure 3 - Hierarchical finite-state machine example 4
Figure 4 - Newly created AssetMenu directly in Unity Editor 4
Figure 5 - Visualization of my finite-state machine 5
Figure 6 - Navmesh and playarea 5
Figure 7 - Editor during OutOfCombat 5
Figure 8 - Editor after assigning turn order 5