행동 선택 알고리즘

개요

Behavior Selection Algorithms

An Overview

Abstract

4.1 소개

게임용 인공지능 시스템을 제작하는 것은 콘솔 게이머들이 구매로부터 더 많은 것을 요구함에 따라 점점 더 복잡해졌고, 동시에 모바일 플랫폼용 소형 게임이 등장함에 따라 AI 프로그래머는 짧은 프레임 시간 내에 최적의 동작을 구현하는 방법을 아는 것이 중요해졌다.

강력한 구축성을 가진 시스템에서 실행되는 복잡한 게임에서도, NPC는 플레이어가 단순히 지나치거나 사냥할 수 있는 동물에서부터 플레이어와 몇 시간 동안이나 상호작용을 하는 완전한 동반자 캐릭터에 이르기까지 다양한 종류가 있다. 이러한 예시로, A.I는 감지-생각-행동 주기를 따르게 되는데, 그 주기 중 "생각" 부분이 잘못 정의되어있는 것이다. A.I는 다양한 알고리즘을 선택할 수 있으며 각 알고리즘은 적합한 그 용도가 정해져 있다. 따라서 최신 콘솔에서 인간 캐릭터를 구현한 최적의 선택이 웹 기반 보드게임에서는 적대적 플레이어를 만드는 데 적합하지 않을 수 있다.

해당 글에서는 업계에서 가장 인기 있고 검증된 의사 결정 알고리즘 중 일부를 제시하고 선택에 대한 개요를 제공해 각 알고리즘이 어느 시기에 사용하기 적합한지에 대해 보여줄 것이다. 이것은 포괄적인 소스는 아니지만, A.I 프로그래머가 이용할 수 있는 다양한 알고리즘 선택들에 대한 좋은 소개가 되기를 바란다.

4.2 유한 상태 기계

FSM(유한 상태 기계)은 오늘날 게임 A.I 프로그래밍에 사용되는 가장 일반적인 행동 모델링 알고리즘이다. FSM은 개념적으로 간단하고 빠르게 코딩할 수 있어 오버헤드가 거의 없는 강력하고 유연한 A.I 구조를 구현한다. FSM은 직관적이고 시각화가 쉬워 프로그래밍

기술 수준이 낮은 팀원과 의사소통 하는 데에 용이하다. 모든 게임 A.I 프로그래머는 FSM으로 작업하는 데에 익숙해야 하며 FSM의 장단점을 알고 있어야 한다.

FSM은 NPC의 전체적인 A.I를 '상태'라고 하는 작은 개별 조각으로 나눈다. 각 상태는 특정 동작 또는 내부 구성을 나타내며 한 번에 하나의 상태만 '활성'으로 간주된다. 각 상태는 특정 조건이 충족될 때마다 새로운 활성 상태로 전환해주는 지정 링크인 '전환'에 의해 연결된다.

FSM의 매력적인 특징은 스케치하고 시각화하기 쉽다는 것이다. 둥근 상자는 각 상태를 나타내며, 두 상자를 연결하는 화살표는 상태 간의 전환을 나타낸다. 전환 화살표의 레이블은 해당 전환이 실행되는데 필요한 조건이며, 실선 원은 초기 상태, FSM이 처음 실행될 때 들어갈 상태를 나타낸다. 예를 들어 그림 4.1과 같이 NPC가 성을 지키는 FSM을 설계한다고 가정해보자.

우리의 경비원 NPC는 순찰 상태에서 시작하여 경로를 따라 성에서 자신의 담당 구역을 감시하고 있다. 만약 소음이 들릴 경우, 순찰을 중지하고 잠시 소음을 조사하러 이동한 후 다시 순찰 상태로 돌아가며, 언제든 그가 적을 보면 공격 상태에 들어가 위협에 맞설 것이다. 적을 발견하고 공격하는 동안 체력이 너무 낮아지면 하루라도 더 살기 위해 도망칠 것이고, 적을 물리쳤을 경우, 다시 순찰 상태로 돌아간다. 해당 알고리즘을 구현한 많은 FSM이 있지만, 알고리즘의 구현 예를 살펴보는 것이 도움이 될 것이다. 첫 번째는 각각의 구체적인 상태(Attack, Patrol 등)가 확장할 FSM 상태 클래스이다.

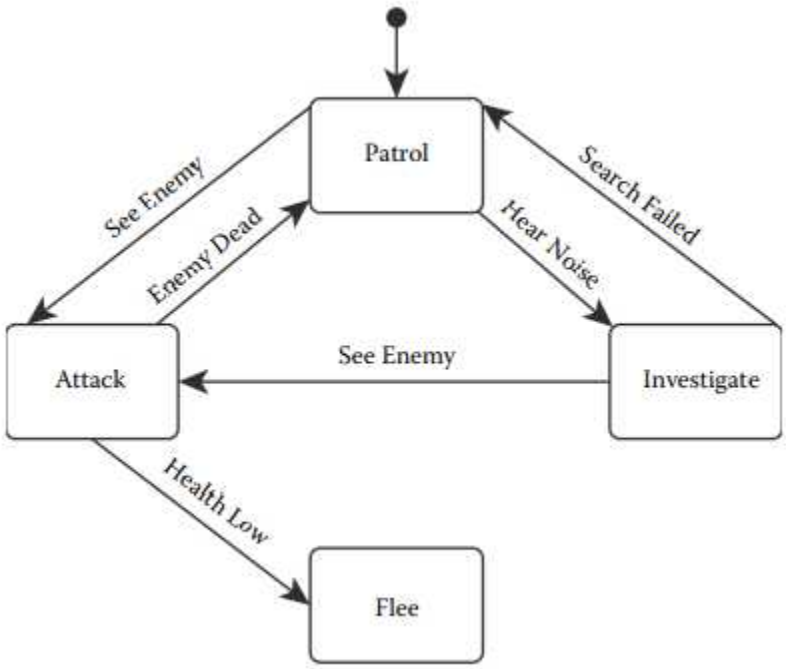

Figure 4.1. This FSM diagram represents the behavior of a guard NPC.

```
class FSMState
{
virtual void onEnter();
virtual void onUpdate();
virtual void onExit();
list<FSMTransition> transitions;
}
```

각 FSMState는 상태가 시작될 때, 상태가 종료될 때, 그리고 각 틱에서 상태가 활성이고 전환이 실행되지 않을 때, 총 세 가지의 서로 다른 시간에 동작을 실행할 기회가 있다. 또한 각 상태는 해당 상태에서 벗어날 수 있는 모든 전환을 나타내는 FSMTransition 개체 목록을 저장할 책임이 있다.

```
class FSMTransition
{
virtual bool isValid();
virtual FSMState* getNextState();
virtual void onTransition();
}
```

그래프의 각 전환은 FSMTransition에서 확장된다. isValid() 함수는 해당 전환의 조건이 충족되면 true로 간주하고, getNextState()는 해당 조건이 유효할 때 전환할 상태를 반환한다. onTransition() 함수는 FSMState의 onEnter()와 마찬가지로 전환이 실행될 때 필요한 동작 로직을 실행할 수 있는 마지막 기회이다. 마지막으로 FinitestateMachine 클래스는 다음과 같다.

```
class FiniteStateMachine
{
void update();
list<FSMState> states;
FSMState* initialState;
FSMState* activeState;
}
```

FiniteStateMachine 클래스에는 초기 상태 및 현재 활성 상태 뿐만 아니라 FSM의 모든 상태 목록이 포함된다. 또한, 각 틱이라고 부르는 중앙 update() 함수가 포함되어 있으며 다음과 같이 행동 알고리즘을 실행한다.

- isValid()가 true를 반환하거나 더 이상 전환이 없을 때까지 activeState.transtitions의 각 전환에서 isValid()를 호출한다.
- 유효한 전환이 발견되면 다음을 수행한다.
- activeState.onExit() 호출
- activeState를 validTransition.getNextState()로 설정
- activeState.onEnter() 호출
- 유효한 전환이 없으면 activeState.onUpdate()를 호출합니다.

해당 구조를 갖추면 원하는 A.I 동작을 생성하기 위해 전환을 설정하고 onEnter(), onUpdate(), onExit(), onTransition() 함수를 채우는 것을 고려해야 한다. 이러한 특유의 구현은 전적으로 설계에 따라 다르다. 예를 들어, 공격 상태가 50파트 II에서 “저기, 저놈을 잡아!”라는 대화를 촉발한다고 가정해보자. 구조적인 onEnter() 및 onUpdate()를 사용하여 주기적으로 전술적 위치를 선택하고, 엄폐물로 이동 및 적을 향해 발포하는 등의 작업을 수행하며 공격과 순찰 사이의 전환은 onTransition()에서 “위협 제거!”라는 추가적인 대화를 트리거할 수 있다.

FSM 코딩을 시작하기 전에 그림 4.1과 같이 몇 가지 다이어그램을 스케치하는 것은 알고리즘 동작의 로직과 상호 연결 방법을 정의하는 데 도움이 될 수 있다. 이제 다양한 상태와 상태의 전환이 이해가 된다면 코드 작성을 시작해보자. FSM은 범용적이고 강력하지만, 기본 로직을 개발하는 데 필요한 생각만큼만 작동한다.

4.3 계층적 유한 상태 기계

FSM은 유용한 도구이지만 한 가지 약점이 있다. NPC의 FSM에 두 번째, 세 번째 또는 네 번째 상태를 추가하는 것은 구조적으로 간단한 일이며, 필요한 몇 가지 기존의 필수 상태에 전환을 연결하기만 하면 된다. 그러나 개발이 막바지에 이르렀을 때 FSM이 이미 10, 20, 30개의 기존 상태를 갖고 있어 복잡한 상황이라면, 새 상태를 기존 구조에 맞추는 것은 매우 어렵고 오류가 발생하기 쉽다.

또한, 상황적 행동 재사용과 같이 FSM이 제대로 처리할 수 없는 몇 가지 일반적인 패턴도 있다. 이에 대한 예를 보여주기 위해 그림 4.2는 건물의 금고를 지키는 야간 경비원 NPC를 보여준다.

이 NPC는 현관문과 금고 사이를 계속 순찰한다. 경비원이 휴대 전화에 응답하고 잠시 대화를 나눈 후 순찰 상태로 돌아갈 수 있는 ‘대화’라는 새 상태가 추가된다고 가정하자. 만약 전화가 왔을 때 경비원이 문에 있었더라면, 대화가 끝난 후 경비원이 다시 문부터 순찰을 재개하길 원하고, 마찬가지로 전화가 왔을 때, 안전 순찰 중이라면 대화에서 전환될 시 안전 순찰로 돌아가야 한다.

상태를 호출 후 다시 전환할 상태를 알아야 하므로 그림 4.3과 같이 동작을 재사용할 때마다 새 대화 상태를 만들어야 한다.

이 간단한 예에서도 원하는 결과를 얻기 위해 두 가지의 대화 동작이 필요하고 더 복잡한 FSM에서는 더 많은 동작이 필요할 수 있다. 동작을 재사용할 때마다 이러한 방식으로 상태를 추가하는 것은 이상적이지 않다. 이는 폭발적인 상태의 증가와 그래프 복잡성으로 이어져 기존 FSM을 이해하기 어렵게 만들고 새로운 상태를 추가하기 더 어려워지거나 오류가

발생하기 쉬워진다.

다행히도 이러한 구조적 문제 중 일부를 완화할 수 있는 기술이 있는데, 이것을 계층적 유한 상태 기계(HFSM)이라고 부른다. HFSM에서 각 개별 상태는 전체 상태 기계 자체가 될 수 있다. 이 기술은 하나의 상태 기계를 계층 구조로 배열된 여러 상태 기계로 효과적으로 분리한다.

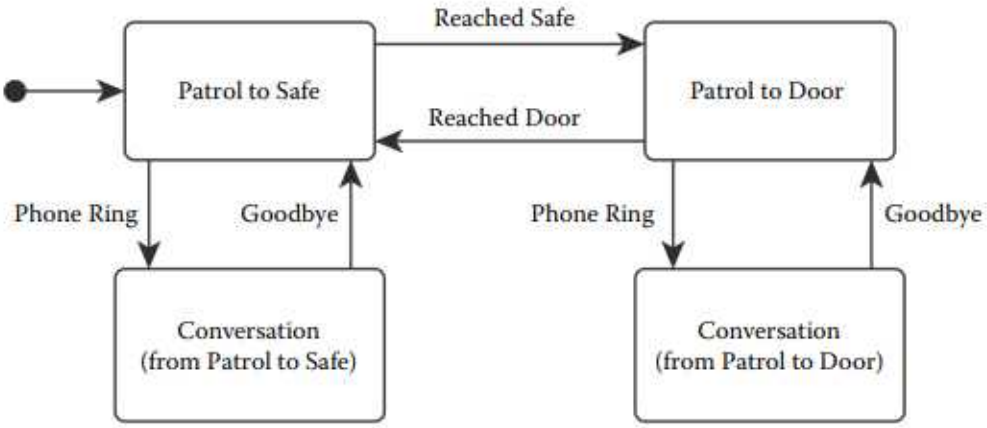

Figure 4.3. Our night watchman FSM requires multiple instances of the Conversation state.

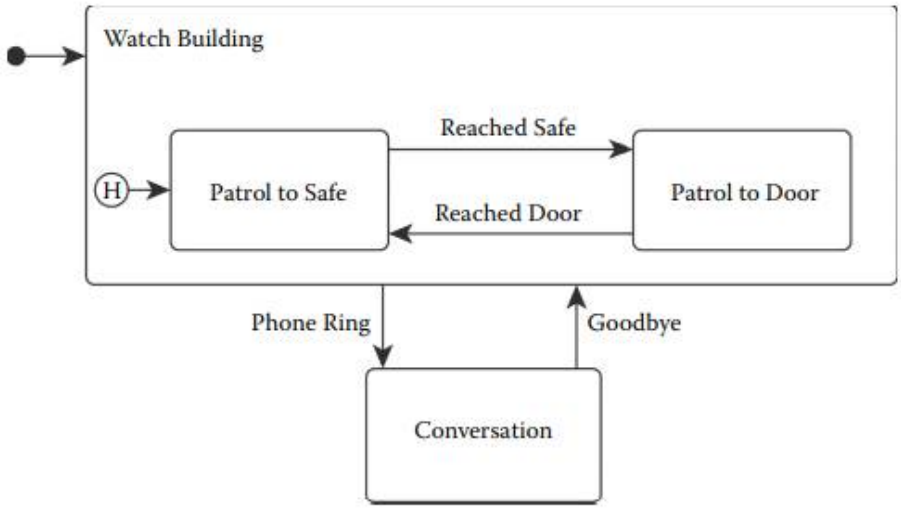

Figure 4.4. An HFSM solves the problem of duplicate Conversation states.

야간 경비원의 예로 돌아가서, 두 개의 순찰 상태를 Watch Building이라는 상태 시스템에 포함 시킨다면, 그림 4.4와 같이 하나의 '대화' 상태로 처리할 수 있다.

이것이 작동하는 이유는 HFSM 구조가 FSM에 없는 추가적인 현상을 추가하기 때문이다. 표준 FSM을 사용하면 상태 기계가 초기 상태에서 시작한다고 가정할 수 있지만 HFSM의 중첩 상태 기계에서는 그렇지 않다. 그림 4.4에서는 '기록 상태'를 가리키는 동그라미 부분 'H'에 주목한다. 처음 중첩된 Watch Building 상태 시스템으로 들어가면 기록 상태는 초기 상태를 나타내지만, 그 이후부터는 해당 상태 기계의 가장 최근 활성 상태를 나타낸다.

해당 예제 HFSM은 초기 상태로 Patrol to Safe를 선택하는 Watch Building(이전처럼 실선 원과 화살표로 표시됨)에서 시작한다. NPC가 금고에 도착하여 Patrol to Door로 전환하면 기록 상태가 Patrol to Door로 전환된다. 이 시점에서 NPC의 전화가 울리면 HFSM은 Patrol to Door and Watch Building을 종료하고 Conversation 상태로 전환한다. 대화가 종료되면 HFSM은 Patrol to Safe(초기 상태)가 아니라 Patrol to Door(기록 상태)에서 재개되는 Watch Building으로 다시 전환된다.

보다시피, 이 설정은 상태를 복제할 필요 없이 설계 목표를 달성하게 된다. 일반적으로 HFSM은 상태 layout에 대해 훨씬 더 많은 구조적 제어를 제공하므로 더 크고 복잡한 동작을 더 작고 단순한 조각으로 나눌 수 있다.

HFSM을 업데이트하는 알고리즘은 FSM을 업데이트하는 것과 유사하며, 중첩 상태 기계로 인해 재귀적 복잡성이 추가된다. 의사 코드 구현은 상당히 복잡해 해당 문서의 범위를 벗어나므로, 확실한 세부 구현은 Ian Millington and John Fungge의 게임용 인공지능 [Millington and Fungge 09]에서 섹션 5.3.9를 참조하라.

FSM과 HFSM은 게임 A.I 프로그래머가 일반적으로 직면하는 다양한 문제를 해결하는 데 매우 유용한 알고리즘이다. 논의된 바와 같이 FSM을 사용하는 데에는 많은 장점이 있지만 몇 가지 단점도 있다. FSM의 주요 단점 중 하나는 원하는 동작이 해당 구조에 적합하지 않을 수 있다는 것이다. HFSM은 일부 경우에서 이러한 압박을 완화하는 데 도움이 될 수 있지만, 전부는 하지 못한다. 예를 들어, FSM이 '전환 과부하'를 겪어 모든 상태를 다른 모든 상태에 연결하거나 HFSM이 도움이 되지 않는 등의 경우, 다른 알고리즘을 사용하는 것이 더 나은 선택이 될 수 있다. 이 문서에 기재된 기술을 검토하고 자신의 문제를 생각해 본 후 작업에 가장 적합한 도구를 선택하라.

REFERENCE

[1] M. Dawe, S. Gargolinski, L. Dicken, T. Humphreys, and D. Mark, *Behavior Selection Algorithms An Overview,* pp. 47-52, 2013.