4장. Behavior Selection Algorithm

작성일시	@2022년 7월 2일 오후 7:41
강의 번호	
유형	논문 번역
자료	
복습	☐
날짜	
Game A.I	

Behavior Selection Algorithms

An Overview

Abstract

This document explains the principles behind the various tools used to create game AI.

Keyword : FSM, HFSM, Behavior Trees, Utility Systems, GOAP, HTN

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 문서에서는 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(Finite-State Machines)

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.

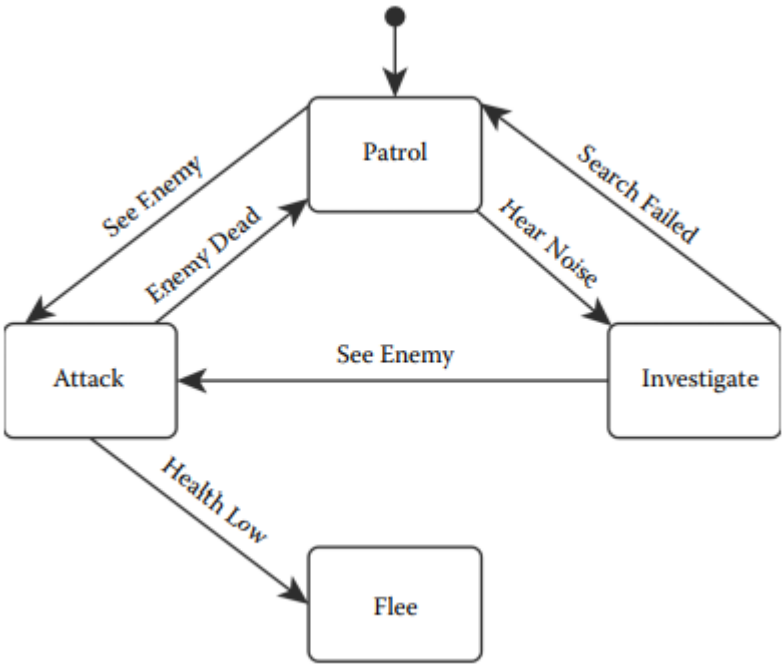

Figure 4.1. This FSM diagram represents the behavior of a guard NPC.

```
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 주

기적으로 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(Hierarchical Finite-State Machines)

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.

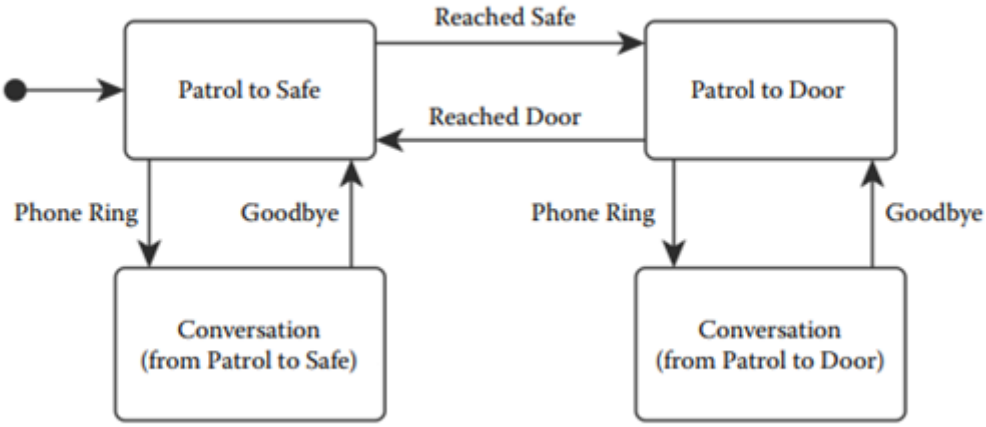

Figure 4.3. Our night watchman FSM requires multiple instances of the Conversation state.

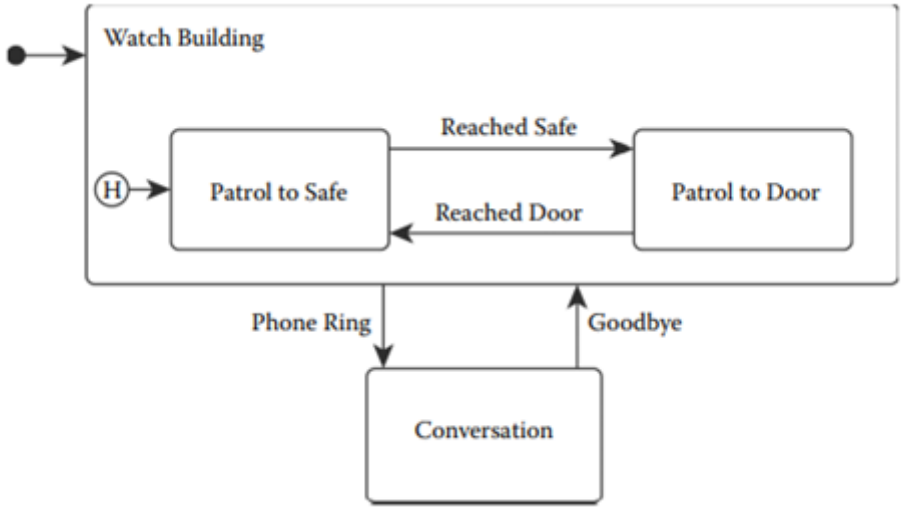

Figure 4.4. An HFSM solves the problem of duplicate Conversation states.

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.

4.4 행동 트리(Behavior Trees)

행동 트리는 일부 뿌리 노드에서 시작하여 NPC가 수행할 수 있는 개별 작업인 행동으로 구성된 데이터 구조를 말한다. 각 행동은 차례로 자식 행동을 가질 수 있으며, 이는 알고리즘에 나무에 나뭇가지가 뻗어있는 것과 같은 특성을 부여한다.

모든 행동은 에이전트가 행동을 실행할 조건인 전제 조건과 행동을 수행할 시 에이전트가 수행하는 실제 작업을 정의한다. 알고리즘은 트리의 뿌리에서 시작하여 행동의 전제 조건을 검사하여 차례로 어떤 행동을 수행할지 결정한다. 트리의 각 레벨에서 하나의 행동만 선택될 수 있고, 행동이 실행되면 그 행동의 자식 행동은 전제 조건이 계속 검사되지만, 같은 레벨인 형제 행동은 검사되지 않는다. 반대로, 행동의 전제 조건이 참이 아니라면, 알고리즘은 해당 행동의 자식 검사를 건너뛰고 다음 형제 행동의 검사로 이동한다. 트리의 끝에 도달하면 알고리즘이 실행할 가장 높은 우선순위 행동이 결정되고 각 행동이 차례로 실행된다.

행동 트리를 실행하는 알고리즘은 다음과 같다.

- 뿌리 노드를 현재 노드로 설정한다.

● 현재 노드가 존재하는 동안,

● 현재 노드의 전제 조건 실행

● 전제 조건이 참일 경우

- 실행 목록에 해당 노드를 추가
- 노드의 자식 노드를 현재 노드로 설정한다.

● 참이 아닐 경우

- 노드의 형제 노드를 현재 노드로 설정한다.

● 실행 목록의 모든 행동(노드)을 실행한다.

행동 트리의 진정한 강점은 단순함에서 나온다. 간단한 특성으로 인해 기본 알고리즘을 빠르게 구현할 수 있다. 트리는 상태가 존재하지 않기 때문에 알고리즘은 주어진 프레임에서 어떤 행동을 실행하기 위해 이전에 실행된 동작을 기억할 필요가 없다. 또한, 행동은 서로를 인식하지 못하도록 작성될 수 있으므로(그래야만 함), 캐릭터의 행동 트리에서 행동을 추가하거나 제거해도 트리의 나머지 실행에는 영향을 미치지 않는다. 이는 모든 상태가 다른 모든 상태에 대한 전환 기준을 알아야 하는 FSM의 일반적인 문제를 완화한다.

또 다른 행동 트리의 장점으로는 확장성이 있다. 설명된 대로 기본 알고리즘에서 시작하고 기능을 추가하는 것은 쉽다. 일반적인 기능의 추가는 행동이 처음 시작될 때와 완료될 때 실행되는 동작을 구현하는 on_start/on_finish 함수를 이용한다. 다양한 행동 선택자를 구현할 수도 있다. 예를 들어, 부모 행동은 실행할 자식 행동 중 하나를 선택하는 것 대신, 각 자식을 차례로 실행하거나 자식 중 하나를 무작위로 선택하여 실행하도록 지정할 수 있으며, 실제로 자식 행동은 유틸리티 시스템 유형 선택기(아래 참조)를 기반으로 실행할 수 있다. 행동이 특정 이벤트에 대한 응답으로 실행되도록 사전 조건을 작성할 수도 있어, 트리가 에이전트 자극에 반응할 수 있는 유연성을 제공한다. 또 다른 일반적인 확장 기능은 개별 동작을 비배타적 동작으로 지정하는 것인데, 이는 전제 조건이 실행되면 행동 트리가 해당 레벨에서 형제 노드의 전제 조건을 계속 검사해야 한다는 것을 의미한다.

행동 트리는 단순하고 강력하지만 항상 최선의 선택 알고리즘은 아니다. 트리는 행동을 선택할 때마다 뿌리로부터 실행되어야 하므로, 실행시간은 유한 상태 시스템의 실행 시간보다 일반적으로 더 길다. 또한 기본 구현에는 다수의 조건문이 있을 수 있으며, 대상 플랫폼에 따라 속도가 매우 느릴 수 있다. 반면에 트리 내의 가능한 모든 행동을 평가하는 것은 처리 능력이 제한적인 다른 행동에서는 느릴 수 있다. 두 접근법 모두 알고리즘의 유효한 구현이 될 수 있으므로, 프로그래머는 무엇이 최선의 선택인지 결정해야 할 것이다.

행동 자체가 상태 비저장이므로 메모리를 적용하는 것처럼 보이는 행동을 생성할 때는 주의를 기울여야 한다. 예를 들어, 전투에서 도망치는 시민을 상상해 보자. 전투 지역에서 멀리 떨어져 있으면 '도망' 행동의 실행이 중지될 수 있으며, 높은 우선 순위의 행동이 시민을 다시

전투 지역으로 데려가 시민이 두 행동 사이를 반복하게 만들 수 있다. 이러한 종류의 문제를 방지하기 위한 조치를 취할 수 있지만, 기존의 기획자들은 상황을 더 쉽게 다룰려는 경향이 있다.

4.5 유틸리티 시스템(Utility Systems)

AI 논리의 상당 부분과 컴퓨터 논리는 단순한 부울 질문에 기초하고 있다. 예를 들어, 에이전트가 “적을 볼 수 있습니까?”라고 물을 수 있거나 “나에게 탄약이 떨어졌나?” 와 같은 순전히 “예” 또는 “아니오” 의 질문들이다. 이전 아키텍처에서 보았듯이, 이러한 질문의 결과는 주로 단일 행동에 직접 매핑되는 것처럼, 부울 질문에서 나온 결정들은 종종 똑같이 양극화된다. 예를 들어,

```
if (CanSeeEnemy())
{
AttackEnemy();
}
if (OutOfAmmo())
{
Reload();
}
```

여러 기준이 결합된 경우에도 부울 방정식은 매우 불연속적인 결과 집합으로 이어지는 경향이 있다.

```
if (OutOfAmmo() && CanSeeEnemy())
{
Hide();
}
```

그러나 의사 결정 알고리즘의 많은 측면들이 그렇게 깔끔하지 않다. “예, 아니오"라는 대답이 적절하지 않은 곳에서 질문할 수 있는 수많은 질문들이 있다. 예를 들어, 우리는 적이 얼마나 멀리 있는지, 총알이 얼마나 남았는지, 배가 고픈지, 내가 얼마나 다쳤는지, 등 수많은 연속적인 값들을 고려하기를 원할 수 있다. 이에 따라, 이러한 연속적인 값은 단순히 작업을 수행할지 여부를 결정하는 것이 아니라 작업을 얼마나 수행하고자 하는지에 상응할 수 있다. 유틸리티 기반 시스템은 잠재적 행동의 선호도를 결정하기 위해 많은 고려사항을 측정, 가중치를 부여하고, 결합, 등급을 매기고, 순위를 매긴다. 위의 예를 지침으로 사용하여 공격, 재장전, 숨기기 등의 행동을 얼마나 원하는지(또는 얼마나 필요한지) 평가할 수 있다.

유틸리티 기술이 다른 아키텍처의 전환 논리를 보완하는 데 사용되는 동안, 유틸리티를 기초한 전체 의사 결정 엔진을 구축하는 것 또한 가능하다. 사실, 유틸리티 기반 AI를 구축하는 것이 다른 방법보다 훨씬 더 선호될 때가 있다. 여기에는 여러 가지 가능한 동작이 있는 게임

이 포함될 수 있으며, 하나의 ‘올바른’ 답이 없거나 선호되는 동작의 선택이 다수의 경쟁 입력에 기반할 수 있다. 이러한 경우, 우리는 단순히 유틸리티를 사용하여 무언가를 측정하거나 평가하는 것을 넘어서고 있다. 대신에, 우리는 그것을 실제 의사결정 메커니즘을 추진하는데도 사용하고 있다. 이를 표현하는 또 다른 방법은 유틸리티 기반 시스템이 "이것이 당신이 할 수 있는 한 가지 행동이다."라고 말하는 대신, "여기 당신이 하고 싶은 몇 가지 가능한 옵션이 있다."라고 제안하는 것이다. 해당 특성으로 인해 유틸리티 시스템은 부울 질문으로부터 자유로운 구현이 가능하다.

이에 대해 문서화가 잘 된 한 가지 예는 The Sims의 유틸리티 사용이다. 이러한 게임에서 에이전트(즉, 실제 "심스")는 자신의 환경에서 정보를 가져와 자신의 내부 상태와 결합하여 각 잠재적 행동에 대한 선호도에 도달한다. 예를 들어, 내가 "매우 배고프다"는 사실과 "불량한 음식"을 이용할 수 있다는 사실이 내가 "조금만 배고프다"는 표현보다 확실히 더 매력적일 것이다. 게다가, 비록 내가 "조금 배가 고프다"고 해도, ‘훌륭한’ 음식의 선호는 여전히 높은 우선 순위가 될 수 있다. ‘spectacular’, ‘rather’, ‘poor’ 및 ‘a little’ 설명자는 실제로 설정된 최소값과 최대값 사이의 숫자로 구현되어있다. (일반적으로 사용하는 방법은 0과 1 사이의 부동 소수점 숫자이다.)

새로운 행동을 후보들 중에서 선택해야 할 때(현재 행동이 완료되었거나 일종의 인터럽트 시스템을 통해) 몇 가지 방법이 사용된다. 예를 들어, 잠재적인 작업에 대한 점수를 정렬하여 "가장 적절한" 작업, 즉 가장 높은 점수를 받은 작업을 간단히 선택할 수 있다. 다른 방법은 가중치 무작위 선택을 시드하기 위해 점수를 사용하는 것이다. 이러한 가중 확률에 대해 난수를 캐스팅함으로써 가장 적절한 행동이 선택될 확률이 높아진다. 해당 방법들은 행동의 적합성이 높아지면 점수가 올라가고 선택될 확률도 높아지는 방식이다.

유틸리티 기반 아키텍처가 다른 아키텍처보다 선호될 수 있는 또 다른 예는 RPG이다. 주로 이러한 게임에서 에이전트들이 선택할 수 있는 옵션은 다양하며 상황에 따라 미묘하게 더 좋거나 더 나쁠 수 있다. 예를 들어, 적의 유형, 에이전트의 상태, 플레이어의 상태 등을 고려하여 어떤 무기, 주문, 아이템 또는 특정 행동을 취해야 하는지 선택하는 것은 복잡한 밸런스 작업이 될 수 있다.

유틸리티 아키텍처를 이끄는 또 다른 조타실은 경제적 의사결정 계층이 있는 모든 게임 시스템입니다. 예를 들어, 실시간 전략 게임에서 건설할 유닛이나 건물의 문제는 비용, 시간 및 많은 우선 순위 축(예: ‘공격’ 또는 ‘방어’)을 고려한 저글링 행위이다. 유틸리티 기반 아키텍처는 종종 변화하는 게임 상황에 더 잘 적응하는 특성이 있으므로, 혼란스럽지만 아무 일도 일어나지 않은 것처럼 그저 굴러만 가는 수많은 스크립트 모델보다 더 오류의 파악, 수정이 쉽다.

이러한 적응성의 주된 이유는 선호도 점수가 매우 역동적이기 때문이다. 게임 속 환경이 바뀌거나 에이전트 상태가 바뀌면 대부분의 행동에 대한 점수가 변경되고(전부는 아닐지라도), 행동 점수가 변경됨에 따라 "합리적인" 행동으로 선택될 가능성도 바뀌게 된다. 결과적으로 가중된 무작위 선택과 결합될 때 행동 점수의 감소와 흐름은 종종 매우 역동적인 신규 행동으로 이어진다.

하지만, 부울 전환 결정 논리를 사용하는 아키텍처와 달리 유틸리티 시스템은 종종 예측할 수 없습니다. 그러나, 선택은 주어진 상황과 맥락에서 행동이 얼마나 "이치에 맞는지"를 기반으로 하기 때문에 행동이 합리적으로 보이는 경향이 있어야 한다. 이러한 예측 불가능성에는 장점과 단점이 있다. 장점으로는 주어진 상황에서 발생할 수 있는 다양한 행동이 예측 가능한 로봇 if/the-기반 모델보다 훨씬 더 자연스러워 보이는 에이전트를 만들 수 있기 때문에 신뢰성을 향상시킬 수 있다. 이 방법은 많은 상황에서 바람직하지만 설계가 매우 특수한 시점에 특정 동작을 요구하는 경우 더 많은 스크립트로 작성된 작업으로 유틸리티 계산을 재정의해야 한다.

유틸리티 기반 아키텍처를 사용할 때의 또 다른 주의 사항으로, 얻을 수 있는 모든 섬세함과 반응성에는 종종 대가가 따른다는 것이다. 핵심 아키텍처는 설정이 비교적 간단하고 새로운 동작을 간단하게 추가할 수 있지만 조정하기가 다소 어려울 수 있다. 유틸리티 기반 시스템에서 행동이 따로 고립되어 있는 경우는 드물다. 대신, 관련된 수학적 모델에 적절한 행동이 "최고로 거품이 돋게" 하도록 장려할 것이라는 아이디어와 함께 다른 모든 잠재적 행동의 더미에 추가된다. 가장 합리적인 행동이 가장 적절할 때 빛나도록 장려하기 위해 모든 모델을 저글링하는 것이 요령이다. 이것은 종종 과학보다 예술에 가깝다. 따라서, 예술과 마찬가지로, 생성된 결과는 종종 단순한 과학만을 사용하여 생성된 결과보다 훨씬 더 매력적이다.

유틸리티 기반 시스템에 대한 자세한 내용은 이 책의 유틸리티 이론 소개[Graham 13] 및 책 Behavioral Mathematics for Game AI[Mark 09]를 참조하십시오.

4.6 목표 지향적 행동 플래너(Goal-Oriented Action Planners)

GOAP(Goal-Oriented Action Planning)는 2005년 모노리스의 제프 오킨이 게임 F.E.A.R.를 위해 개척한 기술이며, 그 이후 가장 최근에는 저스트 코즈 2와 데우스 엑스: 휴먼 레볼루션과 같은 타이틀에 많이 사용되었다. GOAP는 1970년대 초에 처음 개발된 AI에 대한 STRIPS(스탠퍼드 연구 문제 해결사) 접근 방식에서 파생되었다. 일반적으로 STRIPS(및 GOAP)는 AI 시스템이 플레이어에게 게임 세계가 어떻게 작동하는지에 대한 설명, 즉 가능한 행동의 목록, 각 행동을 사용하기 전의 요구 사항('전제 조건'이라고 함), 행동의 효과에 대한 설명을 제공함으로써 플레이어가 문제 해결에 대한 자신만의 접근 방식을 만들 수 있도록 한다. 그 다음, AI는 게임 세계의 초기 상태와 달성해야 할 객관적 사실들의 집합을 플레이어에게 표현한다. GOAP에서 이러한 목표들은 일반적으로 NPC가 달성하고자 하는 미리 결정된 목표들의 집합에서 선택되며, 우선순위나 상태 전이와 같은 몇몇 방법에 의해 선택된다. 그런 다음 계획 시스템은 통제 중인 에이전트가 현재 상태의 세계에서 목표를 충족시키기 위해 참이어야 하는 사실을 포함하는 상태로 변화시킬 수 있는 일련의 조치를 결정할 수 있다. 고전적인 계획에서, 이 일련의 조치는 목표 상태로 가기 위한 이상적인 경로일 것이고, 그 목표는 모든 객관적 사실을 포함하는, 모든 상태 중 가장 쉽게 도달할 수 있는 상태가 될 것이다.

GOAP는 ‘역방향 연쇄 검색’을 통해 작동하는데, 이것은 당신이 성취하고자 하는 목표에서 시작하여, 어떤 행동이 필요한지를 알아낸 다음, 파악한 행동의 전제 조건을 달성하기 위해 무엇이 일어나야 하는지를 알아낸다는 의미이다. 당신은 당신이 처음 시작한 상태에 도달할 때까지 해당 방식으로 거꾸로 작업을 계속하게 된다. 하지만 이것은 과학계에서 선호되지 않는 상당히 전통적인 접근 방식이며, 휴리스틱 검색, 가지치기 및 기타 속임수에 의존하는 ‘순방향 연쇄 검색’으로 대체되었다. 그러나 역방향 검색은 여전히 강력한 도구이며 비록 고급적이지는 않지만 현대적인 기술보다 훨씬 더 쉽게 이해하고 구현할 수 있다.

역방향 연쇄 검색은 다음과 같은 방식으로 작동한다.

- 미해결 목록에 해당 목표를 추가
- 각 미해결 사실에 대하여
- 해당 미해결 사실을 제거
- 사실에 영향을 미치는 행동 찾기
- 행동의 전제 조건이 충족되면,
    - 계획 목록에 해당 행동을 추가한다
    - 역방향으로 작업하여 현재 지원되는 연쇄 행동을 계획 목록에 추가한다.
- 충족되지 않으면,
    - 행동의 전제 조건을 미해결 사실로 추가

GOAP의 마지막 흥미로운 측면 중 하나는 계획 시스템에서는 무시되지만, 작업이 실행되기 위해 런타임에 충족되어야 하는 ‘컨텍스트 전제 조건’을 허용한다는 것이다. 이를 통해 상징적으로 쉽게 표현될 수 없는 세계의 특정 측면(예를 들어, 발사 전에 목표물에 대한 가시선 확보)을 우회할 수 있으며, 계획 중에 이용할 수 없는 정보에 접근함으로써 이러한 제약 조건을 충족할 수 있다. 따라서 GOAP가 생성하는 계획은 어느 정도 유연해질 수 있으며, GOAP에 요구되는 작업은 기본적인 실행 수준보다 전술적인 수준에서 더 많이 적용된다. 즉, 계획은 무엇을 해야 하는지 알려주지만 어떻게 해야 하는지는 알 수 없다. 예를 들어 사격을 시작하기 위한 조준선 설정 방법 등 세부 지시사항이 생략되어 보다 유연하게 처리할 수 있다.

다른 캐릭터를 사살하는 것이 목표인 전형적인 군인 NPC 캐릭터가 있다고 가정해보자. 이 NPC의 목표를 Target.Dead로 나타낼 수 있고, 대상을 사살하기 위해서는 캐릭터가 그를 쏴야할 것이다(기본적인 시스템에서). 사격의 전제 조건은 무기를 장비하는 것이며, 무기가 없다면 권총집에서 권총을 뽑아 캐릭터에게 무기를 줄 수 있는 행동이 필요하다. 물론 여기에는 캐릭터의 인벤토리에 사용할 수 있는 무기가 있다는 전제 조건이 있다. 그렇다면, 무기를 그린 다음 사용하는 간단한 계획을 만들었었는데, 만약 캐릭터가 무기를 가지고 있지 않

다면 어떻게 될까? 아마 우리는 무기를 얻을 수 있는 방법을 찾아야 할 것이다. 방법을 찾을 수 없는 경우, 역추적 검색을 통해 사격 행동에 대한 대안을 찾을 수 있으므로 근처에 타겟에 제공할 수 있는 사용가능한 무기가 있을 수 있다. 데드 이펙트나 목표물을 뛰어넘는 데 사용할 수 있는 차량도 마찬가지다. 두 경우 모두, 세상에서 할 수 있는 것에 대한 포괄적인 행동 선택권을 제공함으로써, 우리는 개발 중에 상상하고 창조해야 하는 것이 아니라, 역동적이고 흥미로운 행동들이 자연스럽게 나타나도록 하는 것을 캐릭터에 맡길 수 있다는 점은 분명하다고 할 수 있다.

마지막으로, 무기의 사거리에 최대치가 있는 게임을 생각해보자. 컨텍스트 전제 조건으로 대상이 해당 사거리 내에 있어야 한다고 말할 수 있어야 한다. 플래너는 이를 실현하기 위해 시간을 투자하지 않는다. 목표물이 어떻게 움직일지 등에 대한 추론을 수반하기 때문에 그럴 수 없기 때문이다. 대신에, 조건이 충족될 때까지 무기를 발사하지 않거나, 사정거리가 더 긴 다른 무기로 대체 전술을 사용할 것이다.

자동화된 계획을 기반으로 하는 NPC 제어에 대한 접근 방식에 대해 좋아해야 할 점이 많다. 디자이너가 행동으로 자체 조립되는 간단한 구성 요소를 만드는 데 집중할 수 있도록 하여 개발 과정을 간소화하고, 팀이 전혀 예상하지 못했던 '신기한' 솔루션도 가능하게 해 플레이어들이 훌륭한 일화를 종종 만들어주기도 한다. GOAP 자체는 자동화된 계획이 제공할 수 있는 것 중 가장 낮은 결실로 남아 있으며, 순수 과학적 관점에서, GOAP가 개발된 이후 최첨단 기술이 크게 발전했다. 즉, 올바르게 사용하면 여전히 매우 강력한 기술이 될 수 있으며, 특정 사용자 지정에 적합하고 적응력이 뛰어난 시작점을 제공한다.

성격 중심적인 지능관을 채택하는 이러한 종류의 접근 방식은 개발 팀으로부터 많은 권한 및 감독 통제를 제거한다는 점에 주목할 필요가 있다. 스스로 "생각"할 수 있는 캐릭터는 게임 세계에서 느슨한 대포가 될 수 있으며, 캐릭터의 목표를 달성하는 데는 유효하지만 몰입적이고 매력적인 경험을 만드는 광범위한 목표를 달성하지 못해 게임 흐름을 잠재적으로 방해할 수 있다. 예를 들어 병사가 편리한 곳에 배치되어 있는 거대한 총을 지나치는 경우가 있다.

지식 공학 기술과 표현 트릭을 사용하여 이러한 종류의 문제를 피할 수는 있지만 원하는 동작을 캐릭터의 결정 논리에 직접 주입할 수 있는 행동 트리같은 아키텍처처럼 간단하지 않다. 동시에 GOAP 접근 방식은 계층적 작업 네트워크를 기반으로 하는 접근 방식보다 훨씬 쉽게 설계할 수 있다. GOAP에서는 세계 내 개체의 역학을 설명하기만 하면 되기 때문이다.

GOAP 및 이와 유사한 기술들은 만능 솔루션이 아니지만 적절한 상황에서 플레이어가 완전히 참여할 수 있는 실감나는 행동과 몰입감을 주는 캐릭터를 만드는 데 강점이 있음을 입증할 수 있다.

4.7 계층적 작업 네트워크(Hierarchical Task Networks)

GOAP가 가장 잘 알려진 게임 플래너일지라도, 다른 유형의 플래너도 인기를 꽤 얻었다. 그러한 시스템 중 하나인 계층적 작업 네트워크(HTN)는 릴라 게임즈의 킬존 2 및 하이 문 스

튜디오의 트랜스포머: 사이버트론의 몰락과 같은 타이틀에 사용되었다. 다른 플래너와 마찬가지로 HTN은 NPC가 실행할 계획을 찾는 것을 목표로 하지만, 차이점은 그 계획을 찾는 방법이다.

HTN은 초기 세계 상태와 우리가 해결하고자 하는 문제를 나타내는 루트 작업으로 시작하여 작동한다. 이 높은 수준의 작업은 문제를 해결하기 위해 실행할 수 있는 작업 계획이 완성될 때까지 점점 더 작은 작업으로 분해된다. 각 상위 수준 작업은 여러 가지 방법으로 수행할 수 있으므로 현재 세계 상태를 사용하여 상위 수준 작업을 어떤 작은 작업 집합으로 분해해야 하는지 결정합니다. 이를 통해 여러 추상적 수준에서 의사 결정을 내릴 수 있다.

원하는 세계 상태에서 시작하여 현재 세계 상태에 도달할 때까지 뒤로 이동하는 GOAP와 같은 역방향 플래너와 달리 HTN은 순방향 플래너이다. 즉, 현재 세계 상태에서 시작하여 원하는 솔루션을 향해 작업해 나간다. 플래너는 세계 상태를 시작으로 여러 유형의 기본 요소로 작업하고, 세계 상태는 문제 공간의 상태를 나타낸다. 게임 용어의 예는 NPC의 세계관과 그것에 대한 그의 견해일 수 있다. 이 세계 상태는 건강, 체력 , 적의 체력, 적의 사정거리 등과 같은 여러 속성으로 나뉜다. 해당 지식 표현을 통해 플래너가 무엇을 해야 할지 추론할 수 있다.

다음으로, 두 가지 다른 유형의 작업인 원시 작업과 복합 작업이 있다. 원시 작업은 문제를 해결하기 위해 수행할 수 있는 실행 가능한 작업으로, 게임 용어로 FireWeapon, Reload 및 MoveToCover가 될 수 있다. 이러한 작업은 FireWeapon 작업이 탄약을 사용하고 Reload 작업이 무기를 재충전하는 방식과 같이 세계 상태에 영향을 줄 수 있다. 복합 작업은 방법으로 설명되는 다양한 방식으로 수행할 수 있는 더 높은 수준의 작업이다. 방법은 복합 작업을 수행할 수 있는 작업 집합이며 방법을 사용할 수 있는 시기를 결정하는 전제 조건이다. 복합 작업을 통해 HTN은 게임 세계에 대해 추론하고 취해야 할 조치를 결정할 수 있다.

이제 복합 작업을 사용하여 HTN 도메인을 구축할 수 있다. 도메인은 특정 유형의 NPC로 행동하는 방법과 같이 문제를 해결하는 모든 방법을 나타내는 작업의 큰 계층이다. 다음 의사 코드는 계획이 어떻게 구축되는지 보여준다.

- 분해 목록에 루트 복합 작업 추가
- 분해 목록의 각 작업에 대해
- 작업 제거
- 작업이 복합적인 경우
    - 현재 세계 상태에 만족하는 복합 작업에서 방법 찾기
    - 메서드가 발견되면 해당 메서드의 작업을 분해 목록에 추가
    - 그렇지 않은 경우, 마지막으로 분해된 작업 이전 상태로 플래너를 복원

- 원시 작업인 경우
    - 작업의 효과를 현재 세계 상태에 적용
    - 최종 계획 목록에 작업 추가

언급했듯이, HTN 플래너는 매우 높은 수준의 루트 작업으로 시작하여 더 작은 작업으로 지속적으로 분해한다. 이 분해는 각 방법의 조건을 현재 세계 상태와 비교하여 각 복합 작업의 방법 집합으로 조정되고, 마침내 원시적인 작업을 발견하면 최종 계획에 추가한다. 각각의 기본 작업은 실행 가능한 단계이기 때문에 그 작업의 효과를 세계 상태에 적용할 수 있으며, 본질적으로 시간이 지남에 따라 앞으로 나아갈 수 있다. 그러다가 분해 목록이 비어있게 되면, 유효한 계획을 갖게 되거나 계획 없이 작업이 완전히 취소된다.

HTN이 어떻게 작동하는지 보여주기 위해 AI를 작성해야 하는 군인 NPC가 게임에 있다고 가정한다. 루트 복합 작업의 이름은 BeSoldierTask로 한다. 다음으로, 군인은 공격할 적이 있는지 없는지에 따라 다르게 행동해야 한다. 따라서 이러한 경우에 수행할 작업을 설명하려면 두 가지 방법이 필요하다. 적이 있는 경우, BeSoldierTask는 해당 조건이 필요한 방법을 사용하여 분해한다. 이 경우 메서드의 작업 이름은 AttackEnemyTask이다. 이 작업 방법은 군인이 공격할 수 있는 다양한 방법을 정의한다. 예를 들어, 군인은 소총 탄약이 있는 경우 엄폐 위치에서 사격할 수 있고, 화기에 사용할 탄약이 없으면 적에게 돌진하여 전투용 칼로 공격할 수 있다. 이것을 완성하면 AttackEnemyTask에 작업을 완료하는 두 가지 방법이 제공된다. 군인의 행동을 자세히 정의할수록 계층 구조가 더 많이 형성되고 정제된다. 이로 인해, 도메인의 구조는 한 사람이 다른 사람에게 어떤 행동을 설명하는 방식에 자연스럽게 매핑된다.

HTN은 계층 구조를 사용하여 동작을 설명하기 때문에 캐릭터에 대한 구축 및 추론이 자연스럽게 이루어지므로 디자이너가 HTN 도메인을 보다 쉽게 읽을 수 있어 프로그래밍과 디자인 간의 수월한 협업을 지원한다. 다른 플래너와 마찬가지로 AI가 실제로 수행하는 작업은 근사한 모듈식 원시 작업에 보관되므로 다양한 AI 캐릭터에서 재사용이 가능하다.

HTN은 그래프를 통한 검색이므로 그래프의 크기가 검색 시간에 영향을 주지만 검색 크기를 제어하는 두 가지 방법이 있다. 첫째, 메서드의 조건을 사용하여 계층 구조의 전체 분기를 제거할 수 있는데, 이것은 행동이 구축되면서 자연스럽게 발생한다. 둘째, 부분적인 계획이 있으면 계획이 실행될 때까지 복잡한 계산이 지연될 수 있다. 예를 들어 복합 작업 AttackEnemy를 고려해보자. 한 가지 방법에는 하위 작업 NavigateToEnemy 다음에 MeleeEnemy가 있을 수 있다. NavigateToEnemy에는 경로 계산이 필요하며, 이는 비용이 많이 들 뿐만 아니라 계획과 실행 사이에 변경될 수 있는 세계 상태의 영향을 받을 수 있다. 부분적인 계획을 활용하려면 이 두 가지 작업을 하위 작업이 있는 한 가지 방법이 아닌 각각의 두 가지 방법으로 나누게 된다. 적이 범위를 벗어나면 NavigateToEnemy, 범위 안에 있으면 MeleeEnemy가 된다. 이를 통해 적이 범위를 벗어날 때 NavigateToEnemy의 부분적인 계획만 형성할 수 있어 검색 시간이 단축된다.

한 가지 다른 주의 사항은 사용자가 HTN이 작동도록 네트워크를 구축해야 한다는 것이다. 이 점이 GOAP 스타일의 플래너와 비교하면 양날의 검인 셈이다. 이는 디자이너가 목표로 하는 행동을 표현하는 데 적절하지만 디자이너가 생각하지 못한 계획을 세울 수 있는 NPC의 능력이 제거되므로, 제작하는 게임에 따라 강점 또는 약점으로 작용할 수 있다.

4.8 결론

이처럼 다양한 행동 선택 알고리즘을 사용할 수 있는 상황에서 AI 프로그래머는 주어진 상황에 가장 적절한 도구를 적용하기 위해 각 도구에 대한 지식을 반드시 가져야 한다. 특정한 NPC에 가장 적합한 알고리즘은 게임, NPC의 지식 상태, 대상 플랫폼 등에 따라 달라질 수 있다. 해당 방법은 사용 가능한 모든 옵션에 대한 포괄적인 처리는 아니지만 옵션으로 시작할 위치에 대해 조금이라도 아는 것은 매우 중요하다 할 수 있다. 게임의 요구 사항에 대해 신중하게 생각하는 시간을 가짐으로써, 개발 시간과 제작 용이성 사이의 균형을 유지해 최고의 플레이 경험을 제공하는 AI 시스템을 제작할 수 있다.

REFERENCE

[1] M. Dawe, S. Gargolinski, L. Dicken, T. Humphreys, and D. Mark, *Behavior Selection Algorithms An Overview,* pp. 47-59, 2013.