GPP Technical Report

홍익대학교 게임소프트웨어, B777035 한태욱

목차

1. 프로젝트 소개

Unity3D 엔진을 기반으로 한 기초적인 Defense Game의 기초적 요소를 구현한 어플리케이션이다. Defense 게임의 기본적 요소로는 적의 끊임없는 Re-Spawn, 플레이어의 방어 타워 설치, 적 입장에서 달성해야 할 Goal의 존재가 대표적으로 존재한다. 해당 프로젝트에서는 그에 더불어 RPG 형식의 컨트롤을 지원하는 플레이어 객체와, 플레이어의 커맨드로 하여금 게임을 재시작 할 수 있게 하는 기능을 추가했다.

이 어플리케이션의 간략한 조작법을 알아보자면, 다음과 같다.

조작키	조작 내용
Mouse RB	플레이어가 마우스 커서 위치로 이동한다.
P	플레이어가 포탑을 자신의 위치에 설치한다.
R	게임의 내용을 초기화하고, 지금까지의 게임 플레이 내용을 재생한다.

다음으로, 이 어플리케이션에서 구현된 게임 요소들은 다음과 같다.

요소	요소 내용
Enemy	끊임없이 Castle을 향해 나아가며, 일정 거리에 이르면 공격한다.
Player	사용자가 조작하는 객체다. 타워를 설치해 Enemy들을 막는다.
Castle	Enemy들이 파괴해야 할 객체이다. 파괴되면 Enemy의 이벤트가 활성화된다.
Spawner	Enemy 객체를 순서대로 스폰시킨다. 죽은 Enemy는 다시 가지고 있는다.
Tower	일정 감지범위 안의 Enemy 객체를 파괴하려고 하며, 포탄을 쏜다.

2. 구현된 기능 및 적용된 패턴

2-1. 각 객체의 행동 상태 제어 (State)

특히 이 프로젝트에서 State Pattern을 적용시킨 이 부분은 Enemy 객체를 구현할 때 특히 유용하게 사용했다. Enemy는 총 두 종류가 존재하며, Skeleton과 Creeper로 구분된다. Creeper와 Skeleton은 Enemy로부터 상속을 받아 동작하며, Enemy는 데미지를 받을 수 있고, 체력을 가질 수 있게끔 IDamageable 인터페이스와 HPController로부터 상속을 받는다. 그리고 Enemy, Creeper, Skeleton은 모두 EnemyState라는 namespace로 묶여서 관리받으며, Enemy의 기본적인 구현 함수들은 가상함수로 선언되어 그 하위 객체인 Creeper와 Skeleton에서 오버라이드해서 추가적으로 작성할 수 있게 해 주었다.

스테이트	스테이트 내용
MOVE	Castle을 찾아 이동한다.
HITTED	Tower가 쏘는 Bullet에 맞으면 경직에 걸리며(Creeper는 경직면역), 빨간색으로 변한다.
ATTACK	Castle과의 거리가 공격 사거리 이내라면 공격한다.

Enemy에서 우리는 Enemy가 게임 내에서 취해야 할 기본적인 상태들을 정의해준다. 일단 Castle을 공격하기 위해 다가가야 하는 만큼, MOVE라는 상태가 존재할 것이며, 이동중에 Tower에게 피격당했을 경우, 일정시간동안 경직에 걸리며 빨갛게 변하는 HITTED 상태도 존재해야 한다. 또, 자신의 공격거리가 닿는 거리에 Castle이 위치하고 있다면 공격을 해야 하므로, ATTACK 상태도 존재해야 한다. 이들은 enum형식으로 EnemyState라고 선언되었다.

또한, 나는 객체가 현재 상태와 다음에 이루어질 스테이트를 구분지어 관리하길 원했으므로, 아무것도 하지 않는 NONE 스테이트도 선언해 주었다. 왜냐하면 상황에 따라, 스테이트가 바뀔 때만 초기화 혹은 따로 설정을 해줘야 하는 요소들도 분명 존재할 수 있기 때문이다. 여기서는 피격 시에 Enemy객체의 색깔이 0.2초간 붉게 물들어야 했으므로, 객체의 색깔을 바꿔주는 부분과 스테이트에 진입한 후 시간을 기록해줄 state_timer 변수를 초기화할 목적으로 만들어줬다.

이로 인해 스테이트를 관리하게 되는 Update 함수 안의 구조는 총 세 부분으로 나뉘어 있게 되었으며, 바로 스테이트를 바꾸는 조건을 감지하는 부분, 스테아트가 바뀐 후 최초로 실행할 부분, 그리고 해당 스테이트 안에서 계속 실행할 부분으로 나뉘게 되었다. 이는 Enemy와 이를 상속받은 두 객체 Skeleton, Creeper 뿐만이 아닌, Player와 Tower 역시 그러하며, 사실상 프로젝트에서 액션을 취하는 대부분의 객체가 그렇게 관리되고 있다.

EnemyState에서 가장 먼저 구현된 것은 Skeleton이다. Enemy에서 기본적인 틀은 다 만들어두고 있지만, 스테이트가 어떻게 변환될지, 그리고 그 스테이트에서 어떤 Skeleton만의 고유한 액션을 취할지는 만들어지지 않았다. 특히, 공격 부분에서의 공격 딜레이나 공격 범위 등은 Enemy에서 정의된 것이 아니기에, 그것을 먼저 추가적으로 만져줘야 했다.

사실, Enemy안에서 스테이트의 흐름이 총 세 단계로 되어있다고 말했지만, Enemy 그 자체에서는 두 단계만 보여지고 있다. 가장 첫번째 단계인 스테이트 변환 조건 검사문은 그 하위 객체에서 구현해주어야 했다.

그렇게 구현된 Skeleton은 Castle과의 거리가 공격 범위 이내라면 화살을 쏘며 공격하고, 피격 시에 붉게 물들며 이동 및 공격에 제약을 받는다.

EnemyState의 두번째 객체는 Creeper이다. 역시 세부적인 구현 내용은 Skeleton과 유사하며, 마찬가지로 Castle과의 거리가 사거리 이내라면 폭탄을 던지며 공격하고, 피격 시에 붉게 물든다. 하지만 Creeper는 공격에 제약이 생기긴 하지만, 이동을 멈추진 않는다.

위 내용을 구현한 후에서야 부모객체의 Update를 호출해, Enemy의 기본적인 특징을 가진 액션을 수행할 수 있게끔 설계했다.

이후 구현한 Tower, Player도 State를 사용해 제어한것은 Enemy의 하위 두 객체와 매우 유사하며, 다만 이들은 어디로부터 상속받은 것이 아니고, 각자의 스테이트를 가지고 동작하한다. 따라서 구조적으로 더욱 간단하기에, 간략하게 각각의 스테이트 및 스테이트 내용을 제시한 후 넘어가도록 하겠다.

<Player의 스테이트>

스테이트	스테이트 내용
MOVE	마우스 커서 위치로 이동한다.
PLACEMENT	자신의 Transform에 Tower를 생성한다.
REPLAY	해당 스테이트가 호출된 시점까지의 게임 내용을 재생한다.

<Tower의 스테이트>

스테이트	스테이트 내용
IDLE	마우스 커서 위치로 이동한다.
SHOOT	자신의 Transform에 Tower를 생성한다.

다만, Player의 경우엔 스테이트의 상세한 내용들이 Player 안에서 진행되는 것이 아니라, Command로부터 액션 내용을 지시받아 진행되는 것이므로, 이는 차후 2-4에서 Command를 사용한 부분에 대해 설명하며 상세히 이야기하도록 하겠다.

2-2. 특정 조건 시 적들이 일괄적으로 취하는 이벤트(Observer)

이는 Castle에 이식되어 있는 요소이다. Castle은 이 어플리케이션에서 승패를 결정하는 주요 오브젝트이며, 역시 공격을 받아 파괴되어야 하기 때문에 IDamageable 인터페이스와 HPController의 상속을 받아 Enemy가 쏘아대는 공격에 맞아 피격 받을 수 있도록 설정했다.

Castle은 데미지를 받아 체력이 0 이하가 되면 가지고 있던 delegate로 선언된 타입으로 관리되는 이벤트들에 저장된 함수들을 실행하며, 이 함수들은 Enemy의 doFireworks()로, Castle을 함락시키고 Player가 실패한다면 Enemy가 위치했던 해당 좌표가 불꽃놀이를 하는 파티클을 생성시키는 내용을 지니고 있다.

Enemy들은 Start()에서 Castle의 myEvent에 접근하여, 자신의 doFireworks()를 등록시킨다.이는 Castle에서 가지고 있다가, 기회를 봐서 조건이 만족하면 myEvent를 Invoke시켜 실행시키는 기능을 하고 있다.

2-3. 적 스포너 (Object Pool, Singleton)

게임 어플리케이션에서 지속적으로 몬스터를 생성하고, 제거해 주는 것은 필수적이다. 다만, 객체 자체를 생성하고 파괴하는 것으로 이를 구현하는 것은 상당한 낭비다. 특히나, 디펜스 게임의 형태를 하고 있는 본 프로젝트에서는 더더욱 그렇다. 몬스터가 생성될 때마다 객체 자체를 새로 메모리를 할당하고 생성해주고, 객체를 제거할 때 할당했던 메모리를 해제하는 이 절차는, 몬스터가 많아지면 많아질 수록 메모리에 상당한 부담을 주게 된다.

이것을 해결하기 위한 것이 바로 오브젝트 풀이다. 오브젝트 풀에서 관리되는 객체는 월드에서 해당 오브젝트 풀에 오브젝트를 요청하면 꺼내지며, 사용이 끝나면 오브젝트는 다시 오브젝트 풀에 돌려주어지게 된다.

기존에 생각했던 스포너 타입은 말 그대로 Enemy를 필요할 때마다 Instanciate시키는 방식으로 설계하려 했었고, Enemy가 공격에 피격당해서 사망하게 될 때는 해당 Enemy를 Destroy시켰었다. 이는 파괴 후 가비지컬렉팅에 의해 일어나는 프레임 저하를 고려하지 않았던 방식이었다.

하지만 이후 오브젝트 풀을 접목시켜 새로운 형태로 재설계를 해주었다. 먼저, 생성할 Prefab을 담을 GameObject를 Spawner 안에 넣어주었다. Spawner는 이것을 담을 Queue를 안에 선언하여 가지고 있으며, 게임이 시작되고 Start()가 호출되면, Enemy를 Instaciate 해준 후에 Enemy를 자신의 Queue에 저장한다. 그 후에 Enemy는 SetActive(false)가 되어 필요로 할 때까지 Active되지 않도록 한다. 그리고 이것을 Spawner가 가지고 있을 Enemy의 총 갯수만큼 반복한 뒤, 적을 순차적으로 스폰시키는 함수 EnemySpawn()을 Coroutine을 사용하여 실행시켜 주었다.

EnemySpawn은 Spawner의 Queue 사이즈가 0이 아닐동안 계속 실행되며, 만약 사이즈가 0이 아니라면 자신이 가지고 있는 Enemy를 PopEnemy()시켜준다. 그 후에 Pop된 Enemy의 위치를 최초 조정해준 후, 게임시간으로 2초를 기다려준 뒤에 이를 반복한다. 즉, 2초마다 Enemy가 Spawner로부터 생성되어(사실은 Deactivate 되어있던 것이 Activate 되어서) 월드로 나오는 것이다.

하지만 한가지 문제가 더 있다. Spawner로부터 Enemy를 생성시키는 것은 만들었으나, Enemy가 죽은 뒤에 죽은 Enemy를 어떻게 Spawner에게 돌려주는냐가 문제다.

이것을 해결하기 위해, 나는 Spawner에서 생성된 Queue에 접근할 수 있도록 하기 위해, 그리고 코드 내에서 Spawner가 하나 더 구현되는 문제를 방지하기 위해 public static의 형태로 Spawner를 가리켰고, 이는 Start()에서 자기 자신을 가리키게끔 설정해주었다.

그 후에 Enemy의 하위 두 객체의 HPController로부터 상속받은 Die()를 오버라이드 하여 내용을 수정해 주었다. 죽은 후에, 미연의 오류를 방지하고자 간단한 초기화 작업을 실시한 뒤에 SetActive(false)를 해준 후, Spawner.instance.InsertEnemy(...)의 형태로 접근할 수 있도록 하여 다시 Spawner에 반환시켜주었다.

이렇게 반환된 죽은 Enemy는 Spawner의 Queue에 Insert되어 가지고 있다가, 웰드에서 Enemy가 필요해지면 Active되어 세상에 나온다. 이 때, Enemy의 OnEnable()에서 Enemy의 Start()를 호출해 나타나므로, 공격 상태나 체력 등등, 이전에 사망하기 전 가지고 있던 데이터를 제거한 채 다시 스폰되게 된다.

<Spawner의 함수 내용>

함수	함수 내용
InsertEnemy	죽은 Enemy를 자신의 Queue에 다시 Enqueue시킨다.
PopEnemy	가지고 있는 Enemy를 Queue에서 Dequeue시키고, Active시킨다.
EnemySpawn	2초마다 PopEnemy를 반복하며, 이는 Queue에 더이상 Enemy가 없을때 까지 반복된다.

2-4. 플레이어의 행동 관리 (Command, Singleton)

본격적으로 Player의 행동 관리를 말하기에 앞서서, 앞의 2-1에서 보여주었던 플레이어의 표를 잠시 보도록 하자.

<Player의 스테이트>

스테이트	스테이트 내용
MOVE	마우스 커서 위치로 이동한다.
PLACEMENT	자신의 Transform에 Tower를 생성한다.
REPLAY	해당 스테이트가 호출된 시점까지의 게임 내용을 재생한다.

엄밀히 말하자면, 이 표의 내용은 틀렸다. 정확히는, 스테이트의 결과가 저 내용이 나오는 것은 맞으나, 결코 스테이트가 저 행동을 취해주는 것은 아니다. 따라서, 알맞게 수정된 표를 보자면 다음과 같다.

<Player의 스테이트>

스테이트	스테이트 내용
MOVE	MouseRButton에 Move()커맨드를 할당하고, Excute() 한다.
PLACEMENT	KeyP에 PLACEMENT()커맨드를 할당하고, Excute() 한다.
REPLAY	게임 시간을 0으로 초기화하고, KeyR에 이미 할당된 REPLAY()를 Excute() 한다.

즉, 실제로 저 행동을 취해주는 것은 PlayerCommand 휘하의 세 커맨드, Move(), PLACEMENT(), REPLAY()일 뿐이다. 즉, 실제로 액션을 취하게 하는 부분은 커맨드가 담당하고 있다는 소리다.

<PlayerCommand의 자식 커맨드>

자식	자식 내용
MOVE	마우스 커서 위치로 이동한다.
PLACEMENT	Player의 Transform에 Tower를 생성한다.
REPLAY	Player의 initReplay()를 호출시켜, 게임을 리플레이 시킨다.

나는 Player가 어플리케이션에서 행한 행동 모두를 기록하기 위해, Command 형태로 Player의 행동을 통제하기로 마음먹었다. 따라서, 사실 2-1에서 소개했던 표는 PlayerCommand의 표였던 셈이다. PlayerCommand는 기본적으로 timeStamp와 mouseHoverPos를 가지고 있고, GetTimeStamp()를 통해서 timeStamp를 반환한다. PlayerCommand는 호출되자마자 호출된 시각과, 호출된 시점의 마우스 위치를 받아오게 된다. 하지만 PlayerCommand는 추상클래스이므로 직접 생성자를 호출할 수 없으며, 반드시 하위의 세 커맨드 Move(), PLACEMENT(), REPLAY()를 통해 생성자가 호출되어야 한다.

추가적으로, 나는 게임 내용이 리셋되고 처음부터 리플레이 되기를 원했다. 게임 내에서 리플레이를 저장하는 커맨드의 리스트는 단 한개뿐이고, 이는 Player 내부에서 리플레이 액션이 취해지기 때문에 Player 안에서 선언되므로, PlayerCommand 안에서도 접근이 가능해야 했다.

그 때문에, 이미 지나간 커맨드들을 기록해둘 PlayerCommad타입의 List, oldCommands를 생성해 관리하도록 했다. 또한, 만들어진 PlayerCommand의 하위 개체들도 만져줄 필요성이 있었다. 기존에는 그저 주어진 액션만 행하면 되는 코드였으나, 이번에는 실행 후 Player의 oldCommands에 자신을 Push()하도록 만져주었다.

<Replay기록을 위한 PlayerCommand의 자식 커맨드>

자식	자식 내용
MOVE	마우스 커서 위치로 이동한다. 그리고 Move()를 oldCommads에 Push()한다.
PLACEMENT	Player의 Transform에 Tower를 생성한다. 그리고 Placement()를 oldCommads에 Push()한다.
REPLAY	Player의 initReplay()를 호출시켜, 게임을 리플레이 시킨다. 리플레이는 굳이 Push()하지 않는다.

이렇게 해서, Player는 oldCommands를 통해 리플레이를 시킬 수 있는 권한을 가지게 되었다. 다만, 여전히 해야 할 일은 많다. 지금 이 상태로 리플레이를 시키면, 플레이어의 기록만 재생될 뿐, 다른 Enemy의 Skeleton, Creeper나 Castle의 체력, 그리고 이미 설치된 Tower 등등은 초기화되지 않고, 덧씌워지거나 기존에 하던 액션을 그대로 진행할 뿐이다.

따라서, 우리는 오류가 나지 않도록 이를 초기화 해주어야 한다. 일단 Castle의 경우, 다시 살아날 수 있도록 doReset()이라는 함수를 추가한 후, 본인의 Start()를 호출하게끔 해주었다.

Skeleton, 그리고 Creeper는 리플레이가 호출된 후에 일괄적으로 Die()를 강제로 선언시키고, Observer를 사용한 이벤트가 호출되었는지 검사하는 isFireworked를 false로 바꾸어주는 doReset()을 호출해주었다.

이미 호출되어 인스턴싱된 파티클의 경우, 일괄적으로 삭제시켜 주었으며, Player가 아니었다면 생성되지 않았을 Tower들도 Destroy시켜 월드를 비워주었다.

이렇게 모든 초기화 작업을 하는 함수가 바로 Player의 initReplay()이며, 이 안에서 트리거가 작동되면 doReplay()가 실행되며 리플레이가 진행된다.

리플레이를 진행하는 방법은 간단하다. PlayerCommand는 timeStamp와 mouseHoverPos를 가진다. 즉, 해당 커맨드가 언제 입력되었는지 안다는 소리다. 그렇다면, oldCommand를 탐색하며, 첫번째 인덱스에 저장된 커맨드의 시작 시간이 되면, 그 커맨드의 Do()를 호출해준 후 넘어가준다. 그리고 인덱스 값을 +1 시켜, 두번째 인덱스 값의 timeStamp와 currentTime을 비교하며 동작을 반복해준다.

그리고 마침내 리플레이를 누른 시간과 currentTime이 같아지면, 그때 리플레이 상태를 벗어난다.

3. 구현 후 느낀점

본격적으로 Programming Pattern을 사용해야겠다고 염두에 두고 만든 프로젝트로는 처음이었다. 이전에 Singleton이나 State Pattern은 알게 모르게 자주 써 왔지만, 그 개념이나 정확한 사용 예시 및 주의할 점까지 익힌 후에 제작해본 것으로는 낯선 경험이었다.

하지만 생각보다 낯선만큼 어렵지 않다는 걸 깨닫기도 했다. 계획 없이 무작정 구현했을 때 보다 오히려 후반부로 가면 갈수록 개발이 쉬워지는 듯한 느낌도 받았다. 하지만, 아직도 아쉬운 점이 있다면 역시 Observer 패턴을 좀 더 깊게 다뤄보지 못했다는 점이다.

잘만 가능하다면 좀 더 다양한 조건을 두고 다양한 객체로부터 이벤트를 생성시켜 볼 수도 있었을 것이고, 시간이 충분했다면 레벨디자인 요소까지 포함하여 도전과제가 달성되거나, Castle이나 Tower 객체가 레벨업 하는 등의 재밌는 이벤트도 가능했을 것이란 생각이 들었다.

이 수업을 한 학기동안 수강하며 느낀 것이지만, 이 수업은 적어도 3-1때 수강할 수 있어야 한다고 생각한다. 3-2가 되어서 졸업작품을 할 때, 많은 도움이 될 것 같기 때문이다. 만약 그때 이 수업을 들었더라면 내 졸업작품도 좀 더 퀄리티 높은 결과물이 될 수 있지 않았을까 하는 아쉬운 마음이 든다.