프로젝트 기술보고서

게임제작특론 Take-Home Exam

게임소프트웨어전공 4학년 B893248 정해빈

1 프로젝트 기술보고서

홍익대학교 게임학부에서 재학 중 강의 및 게임 제작동아리(I2P) 활동을 통해 게임과 관련하여 여러 프로젝트를 진행했습니다. 게임을 구현하다 보면 이전에 구현한 프로그램보다 기술적으로 더 좋은 성능을 내는 새로운 프로그래밍 방법을 고민하게 됩니다. 최근에도 취업 준비를 하면서 포트폴리오로 내세울 작품들의 품질에 대한 고민이 있었습니다. 게임제작특론 강의를 통해 여러 디자인 패턴에 대해 깊이 있게 알 수 있었는데 디자인 패턴을 주먹구구식으로 구현한 프로젝트의 코드 품질을 개선하는 목적으로 적용하면 훨씬 좋은 품질로 게임이 완성될 것을 기대할 수 있었습니다.

본 기술보고서는 강의에서 언급한 여러 디자인 패턴을 이용하여 기존의 기능을 개선함으로써 얻을 수 있었던 효과와 왜 그 방법을 채택하였는지 고민하는 과정을 기록하는 목적이 있습니다. 두 가지 대표 작품을 선정하여 해당 프로젝트를 구현할 때 사용한 디자인 패턴 또는 개선할 경우 적용할 수 있는 디자인 패턴을 소개하고 해당 프로젝트의 리팩토링 과정을 통해 개선한 방법과 결과를 토대로 해당 디자인 패턴의 기술적 효과를 분석하는 내용을 담았습니다.

1.1 디자인 패턴

디자인 패턴이란 프로그래밍에서 반복적으로 일어나는 문제를 어떻게 풀어나가는지에 대한 방법입니다. 프로그래밍 경험이 누적될수록 이전에 구현했던 기능을 다시 사용하게 되곤 합니다. 해당 기능을 그대로 가져와서 쓰거나 그것을 현재 프로젝트에 알맞게 변형하여 사용하다 보면 재사용이 가능하도록 기능을 설계하고 이를 유지보수 및 최적화하는 역할을 수행하는 등 더 나은 프로그래밍 방식을 선택할 수 있게 됩니다. [GoF의 디자인 패턴]에서는 객체 지향적 디자인 패턴을 생성 패턴, 구조 패턴, 행동 패턴의 세 종류로 분류합니다. [1] 생성 패턴으로는 Singleton, Prototype, Builder 등이 있고 구조 패턴으로는 Flyweight, Adapter, Composite 등의 패턴이, 행동 패턴으로는 Command, Observer, State 등의 패턴이 있다고 합니다.

1.2 대표 작품

대표 작품으로 선정한 게임은 러닝 액션 게임 <Forest Dash>와 퍼즐 어드벤처 게임 <Hell's DIEter>입니다. 두 작품을 선택한 이유로 모두 4학년 1학기에 진행했던 프로젝트 중 가장 최근에 작업한 작품이며 제작 당시 구조와 완성도를 높이기 위한 고민을 많이 했기 때문에 대표작품으로 선택하였습니다. 두 게임은 공통적으로 Unity를 이용한 3D 게임이고 Unity의 기반이 되는 Component 시스템과 Prefab으로 사용되는 Prototype 패턴, 관리자 클래스에 적용한 Singleton 패턴이 적용되었습니다. 프로젝트에 대한 간결한 설명과 더불어 기존 방법의 문제점과 개선 과정, 개선 결과에 대해

기술하겠습니다.

1.3 Forest Dash(포레스트 대시)

<Forest Dash>는 2021년 4월부터 5월까지 기능성게임프로그래밍에서 제작한 러닝 액션 게임입니다. 제작에 사용된 모델 리소스는 Unity Asset Store에서 구할 수 있는 무료 asset을 이용했고 UI의 경우 Text Mesh, Text Mesh Pro, 저작권 문제로부터 자유로운 이미지 등 여러 종류의 리소스를 적극적으로 이용하여 구현하였습니다. 플랫폼은 PC로 구현되었지만, 최적화와 개선과정을 통해 모바일 환경(Android, IOS)에서도 원활하게 구동할 수 있게 만드는 것이 본 프로젝트의 최종 목표입니다. 해당 작품에서 개별적으로 사용한 디자인 패턴은 Object Pool 기법입니다. 기술보고서에서는 바닥 플랫폼을 생성 및 삭제하는 기존의 방법을 개선하는 목적으로 Object Pool을 적용한 내용을 작성했습니다.

1.4 Hell's DIEter(헬스 다이어터)

<Hell's DIEter>는 2021년 3월부터 5월까지 게임제작프로젝트(2)에서 제작한 퍼즐 어드벤처 게임입니다. 이 프로젝트는 게임그래픽디자인전공 이동건 학우와 협업하는 과정을 통해 2D Graphic 리소스인 UI 및 Title, Story image를 받아 게임 제작에 활용하였습니다. 3D 모델의 경우 소품에 사용된 책과 병 모델은 3Ds MAX를 이용하여 직접 제작했고 나머지 Asset은 Unity Asset Store에서 무료 asset을 이용했습니다. 해당 작품에서 사용한 디자인 패턴은 State 패턴으로 이를 이용하여 Player와 Slime의 애니메이션 상태 FSM을 개선하였습니다. 본 기술보고서에서는 Player의 애니메이션을 기준으로 작성하였습니다.

2 공통으로 사용한 디자인 패턴

2.1 Component Pattern(컴포넌트 패턴)

프로그래밍을 하는 과정에서 어떠한 기능을 가져와 사용해야 하는 경우, 주로 부모 클래스의 기능을 물려받아 사용할 수 있는 방법으로 상속을 사용했습니다. 그러나 상속의 단점으로 자식클래스가 부모 클래스의 기능에서 사용하지 않는 기능까지 물려받게 되는 점과 부모 클래스가 수정될 때 자식 클래스들까지 그 영향을 받게 되는 점이 있습니다. 게다가 상위 클래스의 구조가 복잡해지면 하위 클래스에 미치는 영향을 예측하기 어렵고 상위 클래스에 대한 이해가 복잡해지면서 사용하기 어렵게 되기도 합니다.

컴포넌트 패턴은 Unity 엔진에서 GameObject 클래스에 잘 녹아 있는 디자인 패턴입니다. 컴포넌트는 각자의 고유한 기능이나 성질을 가진 클래스로 분리한 것입니다. 미리 만들어 둔 클래스를 필요한 객체에 추가하면 됩니다. 즉, 컨테이너 역할을 하는 객체에 독립적인 기능을 추가는 방식입니다. GameObject의 경우 필요할 때에 GetComponet<T>() 메서드를 이용하여 decoupling 되어있는 클래스들을 불러와 사용합니다. 기본적으로 하나의 GameObject를 생성하면 해당 객체에는 위치와 회전, 크기를 나타내는 Transform이 기본 컴포넌트로 추가됩니다. 해당 객체가 EmptyObject로 생성한 것이 아니라면 점과 선, 면의 기능을 하는 MeshFilter, 객체가 화면에 그려지는 기능을 하는 MeshRenderer, 물리적인 충돌을 처리하는 Collider와 같은 기능들이 추가됩니다. 컴포넌트는 다양한 종류로 구성되어 있으며[2] 어떤 객체에 컴포넌트를 추가한다고 해서 다른 객체에 그 영향이 가지 않는 독립성을 가지기 때문

에 커플링 문제가 발생하지 않고 컴포넌트를 추가 및 제거하는 작업이 쉽습니다. Unity의 모든 GameObject는 컴포넌트를 추가하거나 삭제할 수 있고 각 GameObject마다 독립적으로 동작하기 때문에 유지보수의 편리성을 제공합니다.

2.2 Prototype Pattern(프로토타입 패턴)

프로토타입 패턴은 원본이 되는 객체로 생성할 객체의 종류를 명시하고 필요할 때마다 원본 객체를 복제하여 독립적인 객체를 생성하는 방식을 의미합니다. 즉, 복제하고 싶은 어떤 객체의 원본을 미리 만들어 두고 모양과 기능이 같은 다른 객체가 필요할 때 원본의 데이터를 재사용하여 새로운 객체로 생성(Spawn)할 수 있습니다. [3] Unity에서는 이를 프리팹(Prefab) 시스템으로 사용합니다. 프리팹은 중첩이 가능하기 때문에 프리팹을 계층별로 설정하여 각각 설정할 수 있고 프리팹을 수정한 내용은 일괄적으로 적용이 됩니다. 변경 내용을 모두 적용하고 싶지 않을 때는 Variant를 생성하여 원본과 유사한 프리팹을 생성하는 방법도 제공됩니다.

<Forest Dash>에서는 바닥으로 생성되는 플랫폼 블록을 프리팹으로 만들었습니다. 일반 평지에 속하는 Land_flat 블록의 Variant로 나무, 꽃, 버섯 등 환경 오브젝트가 붙은 Land_A부터 Land_D를 프리팹으로 생성함으로써 일반 평지 블록인 Land_flat 외에도 다양한 평지 블록이 등장할 수 있도록 했습니다. 이들을 생성하고 관리하는 문제는 이후 3.2에서 다루었습니다.

<Hell's DIEter>에서는 무게 저울 스테이지에서 저울에 올려야 하는 몬스터(슬라임)의 프리팹을 만들었습니다. 스테이지에 입장하면 프리팹을 이용하여 무작위 위치에 같은 종류의 슬라임을 생성하고 주위를 돌아다니도록 했습니다. 각각의 슬라임은 같은 기능을 수행하는 스크립트가 컴포넌트로 추가되어 있지만 개별적으로 동작하여 정해진 구역을 순찰합니다. 또한 특정 상태 조건을 만족할 때 이동속도와 애니메이션이 바뀌게 되는데 슬라임의 움직임을 표현하는 애니메이션은 본문 4에서 기술한 플레이어 애니메이션을 개선한 방법을 적용하였습니다. 슬라임뿐만 아니라 모든 스테이지에 배치된 벽, 바닥, 장애물, 소품도 각각 프리팹으로 만들어 적재적소에 배치했습니다.

2.3 Singleton Pattern(싱글톤 패턴)

싱글톤 패턴은 어떤 클래스에 대한 객체의 인스턴스를 한 개만 생성할 수 있도록 제한합니다. 처음 한 번의 생성이 발생하면 이후에는 해당 인스턴스를 재사용 하고, 한 번도 사용하지 않은 경우에는 인스턴스를 생성하지 않기 때문에 두 경우 모두 메모리가 낭비되는 것을 방지할 수 있습니다. 또한, 정적(static) 변수를 이용하여 구현되는 싱글톤 패턴은 전역적으로 접근이 가능한 접근점을 제공하기 때문에 다른 클래스 및 객체와 데이터 공유가 용이합니다. 즉, 싱글톤은 여러 클래스에서 동일한 데이터를 처리하는 객체에 접근할 때, 또는, 그 객체가 여러 개로 만들어질 필요가 없는 경우에 사용됩니다. 그런데 싱글톤을 남용할 경우 발생하는 문제들도 있습니다. 싱글톤으로 구현된 클래스는 전역적 접근점을 제공하다 보니 여러 종류의 다른 클래스로부터 참조를 받게 됩니다. 이는 Coupling의 문제로 이어지며 여러 클래스와 복잡하게 얽힌 상태로 싱글톤 인스턴스를 수정하게 될 경우 멀티스레딩 환경에서의 동기화 문제와 같은 부정적인 side-effect가 발생할 여지가 생기게 됩니다.

싱글톤 구현 과정에서 정적 변수를 사용하기

때문에 컴파일러가 프로그램이 실행되기 전에 정적변수를 초기화를 합니다. 따라서 런타임 중에 알 수 있는 정보를 이용할 수 없는 문제가 발생합니다. 이를 해결하기 위한 구현방식으로 Lazy Initialization을 적용하게 됩니다. Lazy initialization은 컴파일 시점에 인스턴스를 생성하지 않고, 인스턴스가 필요한 시점, 즉, GetInstance()와 같은 메서드나 프로퍼티가 호출되었을 때 런타임 중 성격이 결정되는 Dynamic Binding을 통해 인스턴스를 생성하는 방식입니다. 이를 이용하여 기능을 구현하는 방법은 여러 종류가 있으며 그 중 멀티스레딩 환경에서 안정적이고 일반적으로 사용하는 방법은 Lazy Holder 클래스를 이용하는 방법이 있습니다. Lazy Holder는 싱글톤 클래스 내부에 정적 멤버 클래스를 두는 방법으로, volatile과 synchronized 키워드를 사용하여 발생하는 성능 저하 문제를 보완하면서도 멀티스레딩 환경의 동시성 문제를 해결하는 효율적인 방법이기도 합니다. 이 정적 멤버 클래스는 내부에 변수가 없기 때문에 초기화 과정에서 제외됩니다. 이후 해당 싱글톤이 필요할 때 메서드가 호출될 때 초기화를 진행하는 방식으로 진행되는데 이는 런타임 중에 그 성격이 결정되는 Dynamic Binding의 특징을 이용하면서도 멀티스레딩 환경에서도 안정적인 성능을 보입니다. 다만 초기화를 진행할 때의 동기화 문제만을 방지해주기 때문에 멀티스레딩 환경에서 싱글톤 인스턴스 접근 시에 의해 발생할 수 있는 안정성 문제는 따로 구현해주어야 합니다.

싱글톤 패턴을 이용하여 구현하게 되는 대상은 주로 관리자(Manager)의 성향을 갖는 클래스입니다. 예를 들어, 게임의 진행을 관리하는 GameManager는 게임 내에서 단독으로 게임의 진행을 관리 및 감독하게 됩니다. 만약 시간이나 점수를 나타내는 UI에서 해당 정보가 필요하다면 GameManager는 전역적 접근점을 제공하기 때문에 해당 관리자에 접근하여 원하는 정보를 받아오면 되는 방식으로 동작하게 됩니다. 싱글톤 패턴은 상속이 가능하며 하위 클래스는 모두 싱글톤의 성격을 갖게 됩니다. 오디오 매니저와 같이 특정 컴포넌트를 가지고 있어야 하면서도 하나의 인스턴스만 존재하는 관리자 객체에는 모노싱글톤(MonoSingleton)을 이용하여 구현하기도 합니다.

<Forest Dash>에서는 GameManager, DataManger 클래스가 싱글톤을 이용하여 구현되었습니다. GameManager는 게임의 진행상태 및 주행거리를 기록하는 역할을 수행하며 DataManager의 경우 목표를 달성했는지를 확인하고 목표 달성 시에 다른 캐릭터나 탈 것을 제공할 수 있도록 하는 역할과 게임을 종료하고 다시 켰을 때에도 이 기록이 삭제되지 않도록 저장하고 관리하는 기능도 수행합니다.

<Hell's DIEter> 또한 SceneManager, GameManager, DataManager와 같은 싱글톤으로 구현된 관리자 객체를 두어 각각 씬 전환과 스테이지 통과 여부 및 진행도 확인, 저장 데이터 관리하도록 설계되었습니다. 특히 <Hell's DIEter> 구현 당시 GameManager의 역할이 커질수록 다른 클래스와의 커플링이 많아지면서 클래스간 결합도가 강해지는 문제가 발생했습니다. 이 문제를 해결하기 위해 GameManager 클래스에서 수행하던 역할의 일부를 SceneManager와 DataManager로 각각 분할하자 상당한 양의 커플링 문제를 해소할 수 있었습니다.

3 <Forest Dash>에서 사용한 디자인 패턴

3.1 기존 구현 방식의 문제점

러닝 액션 게임 장르인 <Forest Dash>는 기능성 게임프로그래밍 강의 중 러닝 게임 만들기 코드를 발전 시켜 제작한 프로젝트입니다. 기존 코드는 바닥 플랫폼을 Instantiate하고 플레이어가 지나쳐 간 블록은 카메라 밖으로 나갔을 경우 Destroy를 하는 방식으로 구현되어 있어 생성과 삭제가 아주 빈번하게 일어났습니다. Instantiate를 호출하면 메모리에 데이터가 할당되고 Destroy를 할 때마다 Garbage Collector(GC)가 발생하여 메모리 반환 작업이 일어나게 됩니다. GC의 동작은 C#과 Unity 엔진 각각에서 다른 차이점이 있습니다.

3.1.1 C#에서의 메모리 관리

C#의 메모리 중 Heap 메모리는 Native Heap과 Managed Heap으로 나뉩니다. [4] 또한, 85KB를 기준으로 Small Object Heap(SOH), Large Object Heap(LOH)으로 나뉘게 됩니다. 특히 LOH에서는 할당과 동시에 2세대로 분류하며 오버헤드가 커서 메모리 재배치 대상에서 제외됩니다. 메모리 할당이 필요한 경우 현재 필요한 메모리 크기만큼 사용 메모리 공간을 증가시킵니다. 할당한 메모리를 0세대로 지정하고 증가한 크기만큼 포인터를 이동합니다. 만약 할당을 위한 메모리가 충만하지 않을 경우 GC를 호출합니다. 메모리가 부족하다면 Heap 메모리를 현재 크기의 2배만큼 늘립니다. 메모리 공간을 확장해도 사용 가능한 메모리 공간이 부족하다면 애플리케이션을 강제로 종료합니다.

GC는 실제 메모리 공간이 부족하거나 Managed Heap에 할당된 개체에 사용될 메모리가 허용되는 임계 값을 넘었을 때 호출됩니다. Heap 메모리 상에서 사용하고 있는 객체의 참조 그래프를 가지고 있어 현재 사용하고 있지 않은 메모리를 해제하거나 사용하고 있는 메모리를 재배치하는 작업을 수행합니다. 메모리는 가장 최근에 할당된 0세대 메모리부터 GC가 호출되었음에도 살아남은 메모리는 최대 2세대까지 분류됩니다. 가장 최근에 생성된 객체일수록 메모리가 해제될 가능성이 높다는 전제하에 0세대 메모리부터 해제 대상이 됩니다.

3.1.2 Unity에서의 메모리 관리

Unity 엔진에서는 C#에서 사용되는 GC와는 다른 GC가 동작합니다. 기존 Unity의 GC에는 메모리 내 세대 구분이나 SOH/LOH, 메모리 재정렬 기능이 존재하지 않았습니다. Unity 엔진은 Boehm-Demers-Weiser의 알고리즘을 이용한 GC를 사용하며 이 GC는 stop the world방식을 사용하기 때문에 가비지 컬렉션을 수행할 때마다 실행중인 프로그램을 중지하고 완료되면 재개하는 방식으로 동작합니다. 이 때문에 GC spike가 발생하게 되며 게임에서는 프레임 드랍의 문제가 발생하게 됩니다. 이를 개선하고자 Unity에서는 점진적 가비지 컬렉션을 도입하게 되었습니다. 점진적 가비지 컬렉션은 GC의 작업을 여러 개로 분할하여 GC Spike발생 빈도를 낮추는 방법으로 Unity 2019.1. 알파 버전부터 지원되는 기능입니다. 정식으로 도입된 것이 아니 시범도입의 개념으로 지원되는 기능이기 때문에 점진적 가비지 컬렉션을 사용하고 싶다면 설정을 통해 사용해야 합니다. 다만 이 점진적 가비지 컬렉션은 분할되는 부분이 마킹 단계일 경우 계속해서 분할하고 메모리를 정리하면서 메모리 정리에 걸리는 시간이 늘어나는 경우도 있으니 주의해야 합니다. 또한, 이 기능을 사용하기 위해선 추가적인 코드를 구현해야 합니다.

3.2 개선 방법 : Object Pool

<Forest Dash>에서는 잦은 Instantiate와 Destroy로 인해 GC가 자주 호출되었습니다.

GC의 수집과정은 CPU에 상당한 부하를 주기 때문에 오브젝트 풀 기법을 사용했습니다. 오브젝트 풀은 오브젝트를 생성한 후 풀에 저장하고 해당 오브젝트가 필요할 때 풀에서 꺼내 사용하는 기법입니다. 만약 필요한 오브젝트가 모두 사용 중인 경우 새 오브젝트를 생성하고 꺼내는 방식을 취하기 때문에 약간의 Instantiate가 발생할 수는 있지만, Destroy를 최소화함으로써 GC의 호출 빈도를 낮출 수 있습니다. 꺼낸 오브젝트의 사용이 끝났다면 오브젝트 풀로 반납하여 다음에 사용할 때 다시 꺼내게 됩니다.

게임은 기존의 주어진 코드에 기능을 추가하는 방식으로 구현했기 때문에 기존 코드에서 Instantiate와 Destroy를 호출하는 대신 오브젝트 풀을 적용해보기로 했습니다. 구현을 시작하기 전에 기존 방식으로 게임을 진행할 경우 GC Spike가 얼마나 발생하는지를 확인하기 위해 Unity Profiler를 이용하여 그래프를 출력했습니다. [그림1] Instantiate와 Destroy를 할 때 GC Spike가 자주 발생함을 알 수 있었으며 특히 Spike가 발생했을 때 일시적으로 100FPS 미만으로 떨어졌습니다. 이러한 프레임 드롭이 직접 확인되는 정도가 아니었기 때문에 그동안 인지하지 못했습니다. 그러나 게임을 안드로이드 환경에서도 구동할 수 있게 추가적인 작업을 할 계획이 있었기 때문에 Profiler를 통해 발견한 문제를 개선해야만 했습니다.

오브젝트 풀을 사용하여 리팩토링을 진행하기에 앞서 오브젝트 풀이 무엇인지, 어떤 방식으로 동작하는지를 공부하고 작업에 들어갔습니다. 처음에는 오브젝트 풀을 하나만 두는 방식으로 구현했으나 플랫폼 블록의 종류에 맞게 블록이 출력되지 않는 문제가 발생했습니다. 풀에서 Slope에 해당하는 블록이 필요했지만, Queue에서 꺼낸 블록의 종류가 평지에 해당하는 Land_Flat의 블록이 나온 문제가 있었습니다. 정상적으로 수정하기 위해 바닥 플랫폼의 종류에 따라 오브젝트 풀을 생성하고 어떤 종류의 플랫폼이 필요할 때 해당 타입을 갖는 블록을 저장한 풀에서 블록을 꺼내 배치하는 방식으로 수정했습니다. 구현을 마치고 Profiler를 통해 GC Spike의 발생을 관찰한 결과 100프레임 미만으로 떨어지는 프레임 드롭을 발생시키면서 자주 발생하던 Spike들이 눈에 띄게 줄어들었고 이전보다 안정적으로 개선된 상태를 확인할 수 있었습니다. [그림2]

4 <Hell's DIEter>에서 사용한 디자인 패턴

4.1 기존 구현 방식의 문제점

플레이어와 몬스터에게 애니메이션을 세팅하는 과정에서 많은 상태를 나타내는 Flag를 사용하다 보니 조건문의 양이 매우 많아지면서 상태가 중복되는 경우가 발생했고 가독성이 좋지 않은 코드로 완성이 되었습니다. 또한, 상태의 변화에 대한 명확한 시퀀스가 제공되지 않아 버그 발생이 잦았고 기능 구현과 오류 수정에서 많은 시행착오를 겪었습니다.

4.2 개선 방법 1 : FSM

처음 기존 구현 방식을 개선하기 위해 사용했던 방법은 FSM입니다. FSM은 Flying Spaghetti Monster와 Finite State Machine으로 나뉘게 됩니다. 여기서는 유한 상태 기계를 의미하는 FSM을 다뤘습니다. FSM은 유한한 개수의 상태를 갖고 상태 간의 전이를 통해 상태를 바꿉니다. 컴퓨터 프로그램 및 논리 회로 설계에 사용되는 수학적 모델이기도 합니다. 유한 상태 기계는 크게 두 가지로, 하나는 유한 상태 오토마타로 하나의 출력만 내보내는 경우로 주로

NPC에서 많이 사용되고 다른 하나는 유한 상태 기계로 입력 시퀀스에 대응하는 상태로 전이하여 출력하는 경우로 플레이어 애니메이션과 같은 기능을 구현할 때 자주 사용됩니다.

FSM의 경우 상태 전환이 직관적이고 구현이 쉽기 때문에 수정 또한 용이한 장점이 있지만 하나의 전이 조건에 의해 상태가 변경되기 때문에 복잡한 조건으로 전이가 되는 상태를 표현하기 어려운 단점이 있었습니다. 구현이 간편하기 때문에 개선 방안으로 쉽게 떠올릴 수 있던 방법이었지만 적절하지 못한 판단이었습니다. 프로젝트에서 플레이어는 비행을 하기 위한 조건으로 점프 중이며 Space bar를 누르고 있고 연료의 양이 0보다 커야 합니다. 이 조건을 만족한 경우에는 제트팩을 이용하여 연료가 떨어지기 전까지 비행할 수 있는데 다양한 조건을 전제로 하므로 FSM으로 표현하기에는 어려움이 있었고 이를 보완할 수 있는 다른 방법을 찾아야 했습니다.

4.3 개선 방법 2 : State Pattern

FSM을 보완할 방법으로 상태 패턴을 적용해 보기로 했습니다. 상태 패턴은 해당 객체가 취할 수 있는 각 상태를 별도의 클래스로 만들고 현재 상태를 나타내는 객체에 적용하는 기법입니다. 즉, 내부 상태로 행동을 취하고 동적으로 행동을 교체할 수 있습니다. 여러 상태를 각각의 클래스로 캡슐화 하므로 나중에 수정해야 하는 상황이 발생하더라도 해당 상태 클래스만 변경하면 되는 장점이 있습니다. 상태에 대한 클래스를 각각 구현하기 때문에 FSM만큼이나 직관적이라고 생각되어 State패턴을 선택했습니다.

상태 패턴을 이용하여 애니메이션 전환을 구현하기 위해 아래의 준비과정을 거쳤습니다.

- PlayerAnimation.cs : PlayerControl에서 애니메이션과 관련된 내용을 분리
- IAnimState.cs : 상태에 해당하는 클래스들이 인터페이스로 받아 사용할 상태 인터페이스
- PIdle.cs, PWalk.cs, PJump.cs, PExercise.cs, PJetpack.cs : 각 상태에 대한 애니메이션 설정
- PlayerStateManager.cs : 플레이어 입력에 따른 상태 전이를 제어하는 클래스

우선 PlayerControl 클래스에서 움직임에 대한 애니메이션 제어 부분을 떼어내고 이 클래스를 PlayerAnimation 클래스로 명명했습니다. PlayerAnimation 클래스에서는 상태의 입력을 받는 메서드, 상태변수를 설정하는 setter, 상태를 받아오는 getter에 해당하는 메서드가 있습니다. 또한, interface키워드를 사용하여 IAnimState 인터페이스를 작성하고 상태와 관련된 클래스를 만들었습니다. Player가 취할 수 있는 상태는 총 5종류로 Idle, Walking, Jumping, Exercising, UsingJetpack이 있습니다. 각각의 상태 클래스는 IAnimState를 인터페이스로 받아오며, 플레이어가 특정 조건을 만족할 때 그에 따른 애니메이션 및 파티클 효과가 재생되도록 상태에 따른 처리를 설정했습니다.

상태 패턴을 이용하여 구현한 결과 특정 상태에 대한 수정 사항이 발생했을 때 다른 클래스에 영향을 주지 않고 해당 클래스 내에서만 수정할 수 있던 부분이 큰 장점으로 느껴졌습니다. 처음 구현하였던 Flag 변수를 남발하는 방식에 비하면 애니메이션을 취하는 클래스에서는 변수 남용을 방지할 수 있었고 FSM과 비교했을 때는 상태 전이 조건이 하나뿐인 한계를 극복할 수 있었습니다. 다만 상태 패턴을 사용

했을 때의 단점이자 아쉽게 느껴진 부분도 있었습니다. 어떤 상태에 대한 처리를 각각의 클래스로 구현했기 때문에 관리자 클래스가 필수적으로 필요했습니다. 상태 각각을 클래스로 만들기 때문에 상태의 수와 비례하여 클래스 자체의 수가 많아졌고 다른 방법에 비해 비교적 관리가 어렵게 느껴졌습니다. 특히 전환되는 상태가 단순한 경우, 클래스 생성이 오히려 부담스럽게 느껴졌습니다.

4.4 그 외 사용한 디자인 패턴 : Observer Pattern (with delegate)

<Hell's DIEter>의 Tutorial Stage에서는 NPC가 플레이어에게 게임의 기본 시스템을 숙지하고 적응할 수 있도록 간단한 미션을 제시합니다. 총 3가지 미션은 각각 1~3개의 세부 미션을 가지며 모두 순차적으로 진행해야 합니다. 따라서 플레이어가 해당 순서에 맞는 입력을 통하여 목표를 달성했는지를 확인하는 작업이 필요했고 주어진 미션을 수행했는지를 관찰하는 Observer를 Unity 엔진의 C# Delegate를 이용하여 TutorialQuest 클래스를 구현했습니다.

옵저버 패턴은 대상이 되는 객체의 상태 변화가 발생했을 때 그 객체에 관심을 갖는 다른 객체들에게 변화에 대한 내용을 전달하고 전달받은 객체들은 그 상태에 따라 각자의 처리를 진행하는 방식입니다. 즉, Subject와 Observer의 관계는 1:N의 관계를 갖습니다. Observer는 Subject를 관찰하는 객체이고 Subject는 Observer에게 상태 변화를 알리게 되는 대상이 됩니다. 옵저버 패턴을 사용하면 클래스 간의 커플링 없이도 특정 객체의 변화에 따라 다른 객체들이 영향을 받을 수 있게 되며 이는 응집도를 높이는 효과입니다.

delegate는 함수에 대해 참조를 하는 변수로 하나의 delegate로 여러 함수에 접근 및 인수로 전달하는 것이 가능하며 이벤트를 발생시키기도 합니다. 함수를 참조하면서도 여러 함수를 추가 및 삭제하여 호출할 수 있는 근거는 함수의 개수만큼 List를 순회하는 Linked List로 동작하기 때문입니다. 튜토리얼 이벤트에서 특정 미션을 수행했는지 확인하는 메소드를 delegate로 선언하고 해당 메소드를 type으로 하는 변수 "checker"를 만들었습니다. 미션을 순서대로 진행해야 하므로 Tutorial 미션을 진행하는 기능은 Coroutine으로 구현했습니다. 미션마다 상태 변화를 확인하는 메소드를 checker에 대입하여 순서에 따라 미션이 수행될 때까지 기다리게 하였고 모든 미션을 완수하면 다음 Scene으로 넘어갈 수 있도록 설정했습니다.

처음 delegate를 사용할 당시에는 delegate가 왜 이런 방식으로 동작하는지 미처 이해하지 못한 상태로 구현했기 때문에 함수를 대체하거나 추가, 삭제하는 방식이 낯설었습니다. 게다가 이벤트가 들어가지 않은 checker를 호출했을 때에 대한 예외처리에 대한 생각조차 못 했기 때문에 구현 당시에는 많은 시간을 들여 구현했습니다. 특히 멀티스레딩 환경에서 동작하는 checker의 경우 조건문을 통해 null인지를 확인하고 delegate를 수행했으나 직전에 다른 스레드에서 등록된 메소드를 지운 경우가 발생하여 NullReferenceException 에러를 발생하는 문제도 겪어보며 동작 방식과 사용 방법에 대해서도 많이 고민할 수 있었습니다. 여러 시행착오 끝에 null 조건 연산자(?.)를 이용하여 checker가 null이 아닌 경우에만 동작하도록 예외처리를 수행하여 안정적으로 동작하도록 구현할 수 있었습니다.

덧붙여서 delegate가 제공하는 편리한 기능이 많다 보니 이에 대해 숙지할 시간이 오래 걸렸지만, 결과적으로는 원하는 이벤트가 발생하는 것에 대한 처리를 안정적으로 구현할 수 있었습니다. 공부하는 과정에서 비슷한 기능으로 Unity에서 제공하는 UnityEvent가 있음을 알게 되었습니다. UnityEvent 관련 스크립트가 컴포넌트로 추가된 객체의 Inspector에서 직접 옵저버를 등록하고 이벤트 발생 시의 처리를 설정할 수 있어 좀 더 시각적으로 옵저버 패턴이 동작하는 방식을 알 수 있었습니다. 해당 기능은 저학년 학생들도 알고 사용하면 편하리라 생각되어 지난 하계 방학 동안 진행했던 Unity(C#) 기초 학술회에서 다루기도 했습니다.

5 경험

5.1 기술보고서 작성 경험

기술보고서를 작성하면서 내용에 대한 고민을 많이 했습니다. 가능하면 어떤 디자인 패턴을 사용했는지, 그것이 무슨 기능을 하는지, 장단점은 무엇인지를 서술하고 구현한 프로젝트에서 어떤 문제가 있었는지, 그 문제를 해결하기 위해 디자인 패턴을 사용하는 과정, 그 과정에서 발생한 다른 문제는 없었는지, 적용한 결과는 어땠는지를 상세히 기술하고자 했습니다. 사진이나 그래프, 코드 등을 최대한 쓰지 않는 방향으로 작성해보았는데 어떤 내용에 대하여 논리적으로 풀어내어 설명하기가 쉽지만은 않았습니다.

5.2 심도 있는 이해를 위한 공부

특히 기술보고서를 작성하면서 제가 알고 있는 내용이 명확하지 않다는 생각이 들었습니다. 스스로 의문을 가지고 그에 대한 명료한 답을 많이 찾아볼 수 있었습니다. 그 과정에서 자주 사용해보지 않아 낯선 기능을 공부하고 이용해보았고 디자인패턴을 적용하기 전과 후를 비교 및 분석할 수 있었습니다. 해당 디자인 패턴이 얼마나 효율적인지를 확인하는 과정을 통해 개선 이전의 프로젝트는 모바일 환경에서 프로그램이 강제로 종료되거나 제대로 돌아가지 않는 문제가 발생할 수 있겠다는 생각이 들었습니다. 최적화가 얼마나 중요한지에 대해 다시 한번 많은 생각과 고민이 필요하다 생각됩니다.

얕게 알고 있던 내용을 더 상세히 찾아보며 공부한 예로 싱글톤 패턴의 경우 간단하게 구현하는 것만 알고 있었는데 멀티스레딩 환경에서의 문제나 코드의 효율성을 고려한 방법들을 통해 다양한 환경에서 문제가 발생할 수 있다는 것을 알게 됐습니다. 또한, 옵저버 패턴의 경우 동작 방식을 이해하는 과정에서 새로 알게 된 기능을 직접 사용하면서 비로소 동작에 대한 이해를 얻기도 했습니다. 구현의 비효율적인 부분이 있음에도 고치지 못했던 부분은 기술보고서 작성 중 새로운 아이디어를 얻어 기존의 코드를 다시 수정해보기도 했습니다. 리팩토링 과정을 거쳐 자잘한 버그도 수정할 수 있었고 디자인패턴을 적용하여 유의미하게 개선되는 결과를 확인했습니다.

5.3 향후 개선 방향

본 기술보고서를 준비 및 작성하는 과정에서 어떤 주제에 대하여 설명하는 연습을 할 수 있었고 더불어 어떤 부분에 대한 지식이 부족한지를 파악할 수 있어 좋았습니다. 다만 아쉬웠던 부분은 더 많은 디자인 패턴을 적용하여 개선할 수 있는 여지가 있음에도 적용해보지 못한 코드들이 있고 새롭게 알게 된 기술적인 디자인 패턴을 사용해보고 테스트하는 과정을 아직 수행해보지 못했습니다. 현재 취업을 준비하는 과정에서 작품들을 개선하는 과정을 계획

하고 있는데 개선 중 적용할 수 있는 패턴을 적용해보는 것과 동시에 테스트가 가능한 경우를 연출하여 직접 이해하는 기회로 삼을 예정입니다.

References

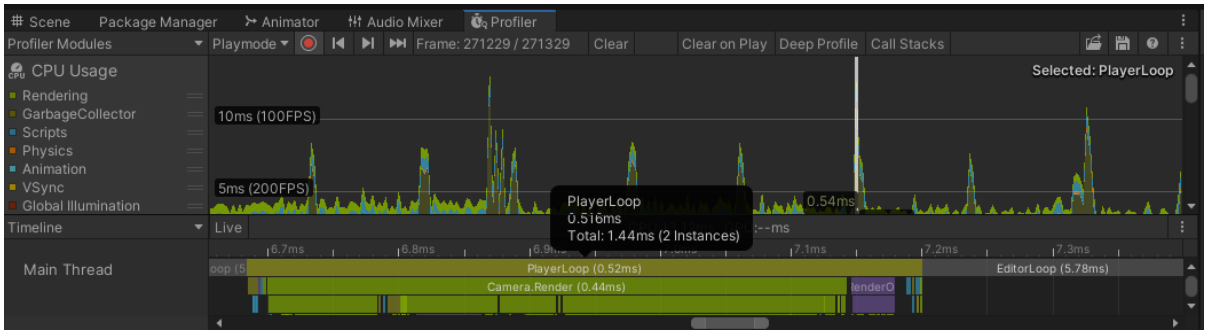
[1] 에릭 감마, 『GoF의 디자인 패턴 』, 김정아, 프로텍미디어, 2015.03.26

[2] Unity Documentation – Component
https://docs.unity3d.com/Manual/30_search.html?q=component

[3] Unity Documentation – Prefab System
https://docs.unity3d.com/kr/current/Manual/Prefabs.html

[4] Microsoft Doccs – Garbage Collection
https://docs.microsoft.com/ko-kr/dotnet/standard/garbage-collection/

[그림1] Object Pool 적용 이전 – Instantiate와 Destroy로 인한 GC Spike



[그림2] Object Pool 적용 이후 – 현저히 줄어든 GC Spike와 Frame Drop