BallonShooter Refactoring

홍익대학교 세종캠퍼스 게임소프트웨어공학
B977082백정열

I. 서론

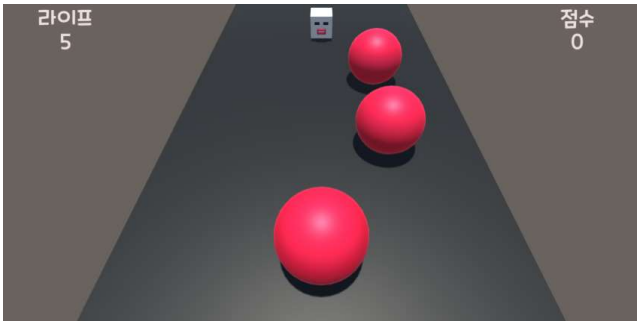
이 프로젝트는 기존 진행한 개인 프로젝트인 BallonShooter를 GPP수업에서 학습한 패턴을 적용하여 Refactoring 하는 프로젝트이다. 기존 BallonShooter의 패턴이 적용되어있지 않은 설계와 다듬어지지 않은 코드들을 StatePattern, CommandPattern, ObserverPattern 세가지 패턴을 적용하여 더 효율적이고 코드의 수정과 추가, 삭제등이 용이한 코드로 Refactoring한것에 대한 설명과 의도, 패턴을 적용함으로써 얻는 장점등에 대한 설명이다.

BallonShooter는 간단하게 방향키와 SPACE를 이용하여 플레이어 캐릭터를 조작해 플레이어 캐릭터방향으로 날아오는 Ballon을 모두 터트리면 되는 게임이다. 이렇게만 설명하면 간단하고 쉬운게임이라고 생각이 들지만, 시간의 흐름에 따라 카메라의 위치가 변하며 보이는 맵의크기와 Ballon의 속도등이 변하고 최종 스코어에서 120점 이상을 달성하기는 어렵도록 설계하였다.

Refactoring 주제를 BallonShooter로 정하게된 이유는 다음과 같다. 비록 플래시게임 수준이지만 내가 처음으로 완성한 게임이고, 빌드하여 친구들에게 플레이시켜준 경험과 재미가 있다는 말을 들었던 것과 이 게임으로 내기를 하였던 것 등에서 의미가 깊었고, BallonShooter를 발전시켜 많은 요소를 추가하여 좀 더 재밌는 게임이 되기위해서는 어떤점을 추가하고 어떤점을 개선해나가야 할까 라는 생각을 계속 해왔었기 때문이다. 새롭게 적용한 패턴을 활용해서 기능을 추가하고싶은 마음도 있었지만, 해당 과제는 Refactoring과제이기 때문에 기능을 추가하면 안된다고 생각하여 최대한 기능을 추가하지는 않는 방향으로 Refactoring 하였다.

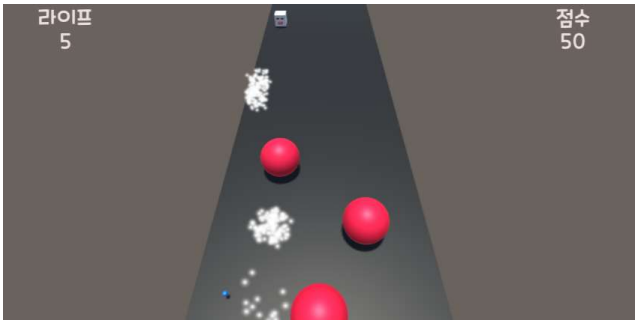
II. 본론

먼저 **BallonShooter 게임의 기본적인 설명**이다.

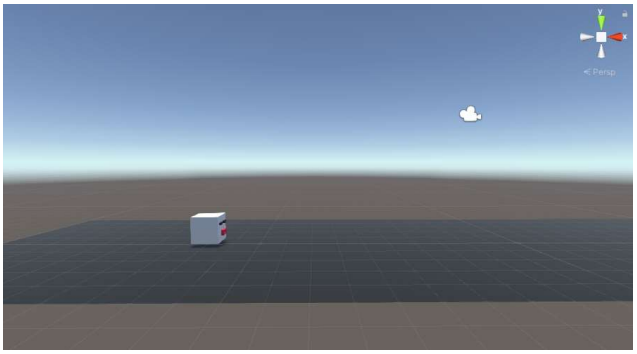


1초주기로 화면 아래에서 위쪽 방향으로 Target이 생성된다. 플레이어는 SPACE입력을 통해 Bullet을 발사하여 Target이 플레이어 뒤로 지나가기전에 모두 제거해야하며, Target이 플레이어 뒤로 넘어가게되면 라이프가 1만큼 감소하게된다.

우측 이미지는 플레이어가 어느정도 게임을 플레이하는 도중의 이미지이며, 플레이어가 50개의 Target을 제거하고 라이프는 잃지않은 상황의 이미지이다. 플레이어가 Target을 제거할때마다 점수를 1만큼 획득하게되며, Target은 사라질때 Particle효과를 남기며 오브젝트가 제거된다.



라이프가 모두 소진될 시 GAMEOVER 텍스트와 함께 이번 게임의 점수와 다시시작 버튼을 플로팅하고, 다시시작 버튼을 클릭하게되면 게임이 처음부터 다시 실행되게 된다.



게임의 기본적인 시나리오는 이렇게 구성되며 세부적인 설계를 살펴보게되면, 우측그림은 측면에서 Scene을 바라본 시점인데 게임이 시작되고 진행되어 시간이 흐름에 따라 카메라의 위치가 플레이어로부터 점점 멀어지며 보이는 맵의 크기가 넓어지게된다. 추가적으로 Target의 속도도 시간에 흐름에 따라

빨라지게되는데, 이를 설계한 의도는 게임의 시간에 흐름에 따른 난이도를 조정하기 위해서이다. 계속 동일한 맵에서 동일한 속도의 Target을 제거하게되면 플레이어의 숙련도에따라 게임이 끝나지 않을 가능성이 있기 때문이다. 그래서 게임시간기준 2분이 넘어가게되면 맵은 기존 맵보다 3배가량 더 커지게되고, Target의 속도는 모두 제거하는 것이 불가능에 가까운 속도가 된다. 게임시간기준 2분 부터는 플레이어의 판단에 따라 Target을 파괴시키는 것을 몇 개 포기함으로써 추가적인 점수를 얻을 수 있게 설계하고자하는 의도였다.

다음은 Target의 생성과 플레이어 Bullet의 생성에 대한설명이다.
Target은 맵을 가상의 5개 레인으로 생각하고 1초 주기로 랜덤하게 하나의 레인으로 보내는 형식이다. Bullet은 플레이어캐릭터의 입부분에 위치한 BulletStart 게임오브젝트 위치에서 발사되게된다. 플레이어가 화면에 Target이 보이지않는데 미리 총알을 날려 Target오브젝트가 파괴되는 것을 방지하기위해 OnBecameInvisible()함수를 사용하여 카메라상 화면에 보이지않게되면 Destroy시켜 이를 해결하였다.

다음은 Refactoring전의 기존 프로젝트를 구성하는 Class의 설명이다.

Player.cs

- 좌우 방향키 입력을 체크하여 플레이어 캐릭터의 좌우 움직임을 제어
- SPACE키 입력을 체크하여 Bullet을 인스턴스화 하고 타이머를 통해 총알발사에대한 쿨타임을 부여

GameManager.cs

- MainCamera에 부착되며 카메라의 움직임을 제어
- Target의 생성과 파괴될때의 라이프와 점수에대한 함수제공
- 게임시작과 게임오버의 UI플로팅을 제어
- 라이프와 점수의 UI를 제어

Target.cs

- Target의 움직임에 대한 제어
- Bullet과 라이프를 감소시킬 EndLine과의 충돌에대한 검사와 오브젝트의 삭제

Bullet.cs

- Bullet의 움직임과 카메라에 보이지않을때 오브젝트 삭제에 대한 제어
- Target과의 충돌에대한 검사와 오브젝트의 삭제

- ObserverPattern

아래는 Refactoring의 세가지 패턴적용중 **ObserverPattern적용의 설명**이다.

먼저 필요없는 코드와 코드의 가독성을 증대시키기위한 작업을 실행하였다.

기존 Target.cs의 speed, timer, targetStart등의 변수를 정리하고 Target의 움직임에 대한 코드를 정리하여 한줄로 축약시켰고, GameManager를 변수로 생성하여 GameManager의 함수를 Target.cs에서 직접 호출하던 방식의 구조를 삭제하였다.

GameManager.cs의 [SerializeField]를 사용하여 유니티 Hierarchy에 프리펩화된 GameObject를 마우스로 드래그하여 스크립트에서 사용하던 부분을 Resources폴더 생성 후 Resources.Load("Gameobject") as GameObject 로 가져오는 방식으로 바꾸었다. CreateTarget()에서 작성된 코드도 축약하여 구조를 바꾸었다.

먼저 ObserverPattern을 적용시켜 구현할 내용은 Target.cs 에 Observer를 등록할 수 있는 코드를 설계하고 GameManager에서 Target.cs이 삭제되는것에 대한 알림을 받아 UI에서 점수와 라이프에대한 제어를 하는 의도이다.

기존 UI에 대한 제어는 GameManager.cs에서 public으로 선언된 함수인 점수를 증가시키는 ScoreIncrease()와 라이프를 감소시키는 LifeDecrease()를 Target.cs에서 직접 호출하여 GameManager에서 관리하는 라이프와 점수를 증감시키는 형식이다. 해당 구조로 프로젝트가 확장된다면 Target이 파괴될때 알려야하는 Observer의 수가 많아질수록 여기저기의 여러 클래스의 public으로 선언된 함수를 호출하여 일일이 전달하는구조가 되는데, 이는 이상한 구조라고 표현 할 수 있을 것이다.

ObserverPattern을 구현하는 방법은 Observer를 Add, Remove하는 방식이나 Delegate를 선언하고 사용하는 방법등 여러 방법이 있을텐데 이중 선택한 방법은 Target.cs 에서 public event Action 으로 이벤트를 정의하고 해당 메세지가 필요한 GameManager.cs에서 이벤트를 연결하여 구현하는 방식을 선택하였다.

현재 구현된 코드상에서는 GameManager.cs 의 CreateTarget() 즉, Target이 생성될때 이벤트를 연결하는 방식이다. 이런 방식을 선택한 이유는 외부 클래스에서 직접 접근하는 것을 제한시키고 Action<int,Transform> 과 같은 설계로 Target이 파괴될때 플레이어가 얻을 점수와 Particle을 재생할 Transform을 받아오기위함이다.

이렇게 설계했을때의 이점은 인자로 받은 int값을 스코어로 사용하여 Target이 지금과 같은 1종류가 아닌 여러가지색의 Target이 구성되었을때를 가정하면, Target은 색에따라 다른 점수가 부여되어야 할 것이다. 이 경우의 구현이 간편해지고, 어떤색의 Target이 파괴되었는지에 대한 구분도 가능할 것이다.

현재 코드상에서의 Transform정보를 가져오는 이유는 Target이 파괴될때 Particle을 재생할 위치를 가져오는 정도지만, 이를 발전시켜 응용한다면 Target이 파괴되는 지점을 기준으로 레인에서 플레이어와 더 가까워질수록 획득하는 점수가 낮아지는 기믹이 추가되거나, Particle종류를 변경할 때 GameManager에서 관리하기가 편하다는 장점이 있을 것이라고 생각한다.

다음은 위와 같이 ObserverPattern을 적용시키게된 의도와 그 이유이다.

이 게임에서의 주요한 이벤트 발생의 주체는 Target이라고 생각한다. Target의 파괴여부에 따라 UI의 라이프와 점수가 변경되며 이는 게임의 진행과 종료의 분기가 된다고 표현할 수 있을 것이다. Target에서 이벤트를 정의하고 Target에서 생성된 알림이 필요한 곳에서 이벤트를 연결하여 이에대한 정보를 얻게되는 구조가 이상적이라고 생각하였다.

현재 GameManager.cs의 코드상에서는 연결한 이벤트가 많지 않지만 게임이 확장되어 예를들어, NPC가 풍선 500개를 파괴하는 퀘스트를 부여하였을때 이를 구현하기위해 현재 정의된 이벤트에 연결하기만 한다면 손쉽게 구현 할 수 있다.

분명히 ObserverPattern은 이벤트가 발생되는 주체인 Subject에게서 정보를 얻어야하는 Observer가 많아질수록 개별적인 알림이 가능한 구조이기 때문에 이 패턴을 적용하는 것에서 더 큰 효율을 얻으려면 프로젝트의 규모를 확장시켜 Observer의 수를 늘려야 할 것이다.

- CommandPattern

아래는 Refactoring의 세가지 패턴적용중 **CommandPattern 적용의 설명**이다.

Player.cs에서의 Player움직임과 bullet발사에 관한 행동을 CommandPattern 적용을 통해 각 행동을 클래스화 시켜 관리하였다. 먼저 현재 코드에서 필수적으로 선언되지는 않아도 된다고 생각하지만 객체화된 Command를 응용하여 리플레이 기능을 구현하거나, 하나의 입력으로 여러행동을 재생하는 등을 고려하여 프로젝트의 확장성을 증대시키기 위해 CommandManager를 설계하고, player의 움직임에 대한 클래스인 MoveCommand와 player의 bullet발사에 대한 클래스인 ShootCommand로 구성하였다.

MoveCommand에서는 Gameobject와 움직일 방향을 int값의 인자로 받아 움직일 오브젝트와 움직이는 방향을 제어하도록 설계하였다. 이때 인자로받은 int값은 1일때 오른쪽, -1일때 왼쪽으로 움직이도록 설계하였다.

ShootCommand에서는 인스턴스화시킬 GameObject와 생성될 위치의 Transform을 인자로 받아 해당 Gameobject를 Transform위치에 인스턴스화 시키도록 설계하였다.

이와 같은 구조는 현재는 구현되어있지 않은 내용이지만 예를들어, Target이 일정한 확률로 플레이어의 bullet을 다른형태 혹은 다른 특성을 가진 Bullet으로 변경하는 아이템을 드랍하여 플레이어가 이를 획득한경우나 bullet이 인스턴스화 될 Transform이 달라지는 경우를 가정하였을때 Command를 생성할 때의 GameObject인자 혹은 Transform인자 만을 변경하는 것으로 쉽게 구현할 수 있다.

현재 코드에서의 Move와 Shoot은 복잡한 로직을 가지지도않고 처리해야할 요청의 수도 적기 때문에 효율적인 Refactoring을 하였다고 말하기 어렵다. 하지만 위의 설명과 비슷한 맥락으로 프로젝트의 로직이 복잡해지고 커맨드화되어 재사용될 명령들의 수가 많아진다면 현재 Player.cs 파일처럼 클래스들이 Player.cs에 모두 선언되어있는 구조가 아니고 클래스별로 파일을 분리하게되면서 각 수정 혹은 추가가 필요한 Command마다 따로 관리할 수 있게되면 효율적인 구조가 될 것이라고 생각한다.

- StatePattern

아래는 Refactoring의 세가지 패턴적용중 **StatePattern적용의 설명**이다.

먼저 UI설계에 대한 설명을 하면, 부모 Canvas아래에 MainUI와 GameUI가 들어가있는 구조로 설계되어있다. 이는 게임상에서 플로팅되어야할 정보인 라이프와 점수를 보여줄 UI와 실제 게임상의 시간은 멈춰있는 게임시작화면과 게임종료 UI를 나누는 기준이다. MainUI에는 GameStartUI와 GameOverUI가 속해있고, GameUI에는 Life와 스코어의 Text가 속해있다.

GameUI가 화면상에 나타날때에는 Cursor.visible과 Cursor.lockState를 사용하여 커서를 감추고 커서에대한 입력을 제한시켰고, MainUI를 화면상에 나타낼때에는 Cursor.visible과 Cursor.lockState에서 감추었던 커서를 보이게하고 커서의 입력에대한 잠금을 해제하였다.

Hierarchy상의 UI설계나 제어는 크게 개선해야할 문제점이 보이지않았지만, 문제는 GameManager.cs의 코드 설계였다. 현재 몇 안되는 MainUI와 GameUI를 교체하는 코드에서도 중복되는 코드가 많은데 프로젝트가 더 커진다면 MainUI와 GameUI를 바꾸어가며 틀어야할 상황이 수도없이 많이 생겨나며 코드의 길이는 더욱 길고 복잡해질 것이다.

이는 중복되는 코드를 함수로 묶어 호출하는 방식으로도 구현이 가능하겠지만, 이 경우 StatePattern을 적용하여 State에 따라 커서를 활성화시키고 시간을 멈추고 UI의 SetActive를 조정하는 방식으로 구현하였다.

이렇게 설계하게되면 예를들어, player에게 쿨타임이 있는 스킬이 부여되거나, 다른 State가 추가되어야할 경우에 State에 따른 제어를 맡는 코드부분만을 수정하거나 State를 추가하고 StateUpdate()에 해당 State에 실행되어야할 코드를 추가하기만 하면 되는 방식이 되어 코드의 수정 및 추가가 쉬워지고 코드 가독성이 증가할 것이다.

Refactoring한 GameManager의 설계는 다음과 같다.

enum GameState의 열거형 멤버는 IsOnUI, IsOffUI가 있고, GameManager에 현재 state를 저장할 변수를 선언한뒤 이 변수를 기준으로 제어하도록 설계하였다.

기존 LifeDecrease()와 GameStart()등에서 실행되던 Cursor를 제어하는 코드와, GameUI와 MainUI의 SetActive코드도 State를 조건으로하는 Switch문에서 일괄 관리하도록

설계하였다. 해당 State에 따른 실행부분을 클래스로 만들어 제어할 수도 있지만, 해당 프로젝트에서의 적용의 경우 코드가 많지않기 때문에 Switch문에서 직접 실행되는 것이 더 올바르다고 생각하여 지금과같이 설계하였다. 하지만 State에 따른 실행되어야할 코드의 양이 많아진다면 State별 클래스를 선언하여 관리하는 것이 더 효율적일 것이다.

State가 바뀔때마다 StateUpdate()를 호출하게되며, IsOnUI State일때는 MainUI를 화면에 표시하고 GameUI를 비활성화하며, Lock시킨 Cursor를 활성화시키고 시간을 멈추게된다. IsOffUI는 IsOnUI와 반대되는 코드가 실행되는 구조이다.

현재 적용한 State패턴으로의 Refactoring역시 프로젝트의 구현기능이 많지않고 제어해야할 UI와 코드역시 많지않기 때문에 조건 로직이 복잡한 것을 해소하거나 코드의 길이를 줄이는등의 효율이나 가독성이 크게 증가하지는 못하였다.

하지만 GameManager에서 State를 싱글톤으로 관리하게한다면 튜토리얼을 구현하여 조작법에대한 튜토리얼을 진행하는 경우 혹은 아이템을 구현하여 일정시간 플레이어의 조작이나 특성이 달라지는등 특정 상황이 부여되어 이에대한 제어가 필요할 때를 대비한 확장성을 가지게 되었다고 생각한다.

하지만 개인적으로 아직 State패턴을 적용할 사례를 많이 경험하지 못하였거나 State를 나누는 기준에 대한 고민이 적은 이유에서인지 State패턴을 설계할 때 고려해야 할 점들을 구상하는것에 시간이 많이 필요하다. Refactoring한 결과가 더 복잡해지거나 State별 실행되는 코드가 명확하게 구분되어지지 않을때는 StatePattern를 굳이 적용하지 않아도 되는 상황이 나오게되는데, 이를 잘 판단하고 가장 이해하기 쉽고 유지보수나 확장성에 최적화된 코드를 설계하는 것이 좋은 개발자라고 생각하여 이에대한 고민은 앞으로도 계속 되어야할것이라고 생각한다.

III. 결론

BallonShooter에 StatePattern, CommandPattern, ObserverPattern 세가지 패턴을 적용시켜 Refactoring하는 과정에서 거의 모든 클래스의 구조와 새롭게 설계되었다. 그와 동시에 중복되는 코드가 많이 축약되었고 가독성이 증가하였으며 확장성과 코드의 수정 및 추가가 용이해졌다.

이 프로젝트에서 적용된 패턴들은 행위 패턴 즉, 객체나 클래스 사이의 알고리즘이나 책임 분배에 관련된 패턴을 위주로 적용하게되었다. CommandPattern을 통해 실행될 기능을 캡슐화하여 재사용성이 높은 클래스를 설계한 것과 Observer패턴을 통해 한 객체의 상태 변화에 따라 다른 객체의 상태도 연동되는 일 대 다 의존관계를 구성하는 것, State패턴을 통해 객체의 상태를 기준으로 행동을 제어한 것들이 해당된다.

이와 같은 패턴들을 적용하지 않고 프로젝트가 계속 진행되어 크기가 커진다면 분명히 스파게티코드가 되어 수정 혹은 추가하고자하는 코드를 찾으려면 많은 시간이 소요되고 그에 관련된 변수, 함수 등이 얽혀 작은 문제에도 많은 수고가 필요하게 되었을것이다.

그리고 디자인 패턴을 적용시키며 느낀점으로는 내가 디자인 패턴을 알고 적용시키는것도 중요하지만 많이 이용되는 디자인 패턴을 모두 이해하고 있어야 다른사람이 설계한 코드에 어떤 디자인 패턴이 적용되어있는지 파악하여 코드를 빠르게 이해할 수 있을 것이라는 것이었다.

이 프로젝트에서 적용된 패턴은 객체나 클래스사이의 책임분배에 관련된 행위패턴만을 적용하였지만 클래스나 객체를 조합하여 더 큰 구조를 만드는 구조패턴과 객체의 생성과 조합을 캡슐화하여 유연성을 제공하는 생성패턴을 더욱 완벽히 공부하고 적용시키는 연습을 통해 계속 숙지해나가야 할 것이다.

개인적으로는 아쉬움이 남는 프로젝트였다고 생각한다. Refactoring할 프로젝트를 선정하고 기존 코드의 개선해야할 부분을 수정하고 패턴을 적용시켜 새롭게 설계하면서 계속 드는생각이 있었는데, 그것은 프로젝트의 규모가 더 크고 Refactoring했을때의 결과가 더 효율적인 설계로 바뀌었다면 더 좋은 Refactoring이 되었을 것이라는 생각과 Refactoring함과 동시에 새롭게 기능을 구현하여 바로 적용시켜보는 형식이었다면 결과에서 보이는 각 Pattern을 적용시키는것의 기능과 효율성이 더 잘 나타났을것이라는 생각이었다.

기존 진행한 프로젝트를 Refactoring하며 GPP수업을 복습하는것도 의미가 있었지만 패턴을 직접 적용해보는 것이 정말 중요한 경험이였다고 생각한다. 수업을 들으며 이해하는 수준에서는 해당 패턴이 어떤 상황에 적용되며 어떻게 동작하는지에대한 이해만을 공부하게된다면, 이를 기존 프로젝트를 Refactoring하며 적용되어야할 부분을 판단하고 직접 설계하는 것은 코드를 짤때 판단하여 생각할 힘을 기르는 연습이 되었다고 생각한다.