

미사일 커맨더를 통해 배우는

UNITY C#

프로그래밍 연습

김웅남 지음



BatStudio Press

유니티 C# 스크립트 프로그래밍 연습 시리즈 1.

미사일 커맨더를 통해 배우는

유니티 C#

프로그래밍 연습

김웅남 지음

BatStudio Press

유니티 C# 스크립트 프로그래밍 연습 시리즈에 대하여

유니티 C# 스크립트 연습 시리즈는 **유니티 사용법과 C# 프로그래밍의 기본을 어느 정도 마스터하신 분들을 대상**으로 합니다. 매 시리즈에서는 간단한 게임의 로직에 해당하는 부분을 만들어 보면서, 왜 이 부분은 이렇게 코딩을 했으며 다른 방법으로 코딩할 때와 비교했을 때 나은 점은 무엇인지 등에 대해 함께 고민해 보게 됩니다. 일반적인 초보자 대상의 책이나 동영상 강좌와 달리, 이 시리즈에서는 유니티의 기본 사용법이라든지 애니메이션 세팅 등 코딩과 직접적인 관계가 없는 부분은 최대한 적게 다루고, 가급적 코딩 자체에 집중해서 진행할 것입니다. 따라서 막 입문 단계를 벗어나서 좀 더 실무적인 코딩 능력을 원하는 분들이라면 이 시리즈를 통해 좀 더 효과적으로 유니티 C# 게임 프로그래밍을 하는 방법을 배우실 수 있을 것입니다.

이 시리즈는 과거에 제가 만들었던 유니티 강좌 시리즈와 달리, 동영상이 아닌 전자책 방식으로 제작됩니다. 프로그래밍 자체에 초점을 맞추기 위해서는 코드 한 줄에 대해서도 많은 설명이 필요할 때가 있는데, 동영상 강의 방식으로는 자세히 설명하는데 한계가 있기 때문입니다. 코드 한줄을 추가할 때마다 왜 이 부분을 이렇게 작성했는지를 설명하면서 진행할 예정인데, 동영상 강좌의 경우 시간에 쫓기는 측면이 있기 때문에 좀 더 여유를 가지고 보실 수 있도록 전자책이라는 포맷을택한 것입니다.

비록 전자책 포맷이지만 동영상을 완전히 배제한 것은 아닙니다. **각각의 항목이 끝날 때마다 책으로 설명한 내용을 동영상으로 다시 녹화해서 mp4 포맷으로 함께 제공**하

고 있기 때문입니다. 따라서 책을 보시다 이해가 안되실 경우, 책에 표시된 동영상 예제 파일을 열어 보시면 됩니다. (참고로 동영상에는 음성 설명이 없으며 코딩 및 유니티 에디터 사용 장면만 군더더기 없이 편집해서 빠르게 보실 수 있게 만들었습니다. 따라서 책의 설명을 먼저 보시고 나서 리뷰라는 느낌으로 동영상을 돌려 보시면 책의 내용을 더 이해하시기 쉬울 것입니다.)

현재 시중에는 좋은 유니티 입문용 책과 강좌들이 많이 나와 있습니다. 하지만 중급 수준을 지향하는 책이나 강좌를 찾아보기는 쉽지 않습니다. 그 이유는 시장이 너무 작기 때문입니다. 초보자 대상의 책은 언제나 수요가 있지만, 중급자 이상을 지향하는 책을 출판했을 경우, 최소한의 매출을 기대하기가 어렵습니다. 따라서 이런 영역은 기존의 기업형 출판사보다는 저와 같은 1인 독립 출판을 통해서 커버할 수밖에 없을 듯 합니다. 제가 이 시리즈를 언제까지 계속 낼 수 있을지는 모르지만, 제 능력과 시간이 허용하는 한까지는 최대한 열심히 책을 써서, 유니티를 배우는 분들께 조금이나마 도움을 드릴 수 있기를 희망합니다.

유니티 C# 스크립트 프로그래밍 연습 시리즈에 대하여	3
수록된 파일 안내 및 이 책을 보시는 방법	10
동영상 예제 파일	10
C# 스크립트 파일 이름 확인	11
이 책에서 다루는 내용들	12
느슨한 커플링	12
객체 지향적 프로그래밍	13
이벤트 주도형 프로그래밍	14
디펜던시 인젝션(Dependency Injection)	15
싱글톤으로 대표되는 글로벌 참조의 최소화	16
오브젝트 풀링이 적용된 팩토리(Factory)	16
컴포지션 루트(Composition Root) 개념의 사용	17
유니티 버전	18
프로젝트 생성	19
애셋 불러오기	21
게임 뷰(Game View) 비율 설정	23
미사일 커맨더 예제 게임 플레이	24
필요한 게임 오브젝트 리스트	28
총알 발사대(BulletLauncher) 제작	29
총알 발사대 만들기	29
스프라이트 살펴 보기	30
총알 발사대 게임 오브젝트 만들기	33
마우스 입력(Input) 받아들이기	38
키보드 입력 방식 추가하기	40
또 다른 콘트롤러 추가로 지원하기	41
BulletLauncher.cs 코드 초기화	43
IGameController 인터페이스 만들기	44

MouseGameController.cs 만들기	48
FireButtonPressed() 함수 구현하기	51
KeyGameController.cs 만들기	54
총알 발사대(BulletLauncher)에서 IGameController 사용하기	58
GameManager 클래스 만들기	66
GameManager 클래스의 역할	66
BulletLauncher 프리팹으로 저장하기	68
GameManager.cs 코드 작성하기	71
씬(Scene)에 게임 매니저를 만들자	75
게임 콘트롤러를 마우스에서 키보드로 바꾸기	78
디펜던시 인젝션(의존성 주입) 개념의 이해	80
디펜던시(dependency)란 무엇인가?	80
디펜던시 인젝션(Dependency Injection)의 기본 원리	84
디펜던시 인젝션(Dependency Injection)이 꼭 필요한가?	89
이벤트(Event)를 이용한 IGameController 사용법	91
예제 코드 상태 확인하기	91
이벤트(event)를 이용한 커뮤니케이션	93
액션(Action)으로 이벤트를 구현해 보자	99
이벤트 수신 함수를 만들자	104
이벤트 송신자와 수신자의 연결	107
입력 방식을 키보드로 바꿔 보자	116
두 개의 게임 입력 방식 동시 지원하기	119
다시 마우스 입력만 사용하자	122
마우스 입력 이벤트에 클릭 지점 좌표를 함께 전달하기	123
마우스 클릭 지점의 실제 좌표를 전달하자	130
총알(Bullet)을 만들자	135
총알의 프리팹을 만들자	137
BulletLauncher를 이용해서 총알을 발사해 보자	140

총알이 총구에서 발사되도록 하자	144
총알이 클릭 지점으로 날아가게 하자	148
오브젝트 풀링이 적용된 Factory 만들기	153
재활용 오브젝트 전용 클래스인 RecycleObject 를 만들자	155
풀(pool) 생성 함수를 만들자	162
필요한 오브젝트를 요청하는 Get 함수 생성	165
오브젝트 반납 함수(Restore)를 만들자	169
완성된 Factory 사용하기	171
사용한 총알을 Factory 로 다시 회수하기	175
총알이 목표지점에 도달했는지 이벤트로 알려 주자	180
총알 발사 지연 시간(쿨타임) 설정	184
폭발 효과를 만들자	190
Explosion 의 프리팹을 만들자	196
총알이 목적지에 도달하면 Explosion 생성하기	198
폭발 효과가 일정 시간 후에 사라지게 하자	203
폭발효과를 Factory 로 회수하자	210
공통된 부분을 RecycleObject 에 정의하자	215
총알 발사대의 위치를 수동 지정하자	230
빌딩(Building)을 만들자	234
빌딩 프리팹을 만들자	238
빌딩 매니저(BuildingManager)를 만들자	240
TimeManager 를 이용한 게임 시작 딜레이 처리	255
Unbind 함수도 만들자	275
적 미사일(Missile)을 만들자	279
미사일(Missile) 프리팹을 만들자	283
미사일 매니저(MissileManager)를 만들자	285
화면 상단의 랜덤 위치에서 미사일 생성	297

미사일이 생겨나는 시점을 조절하자	303
빌딩 중 랜덤하게 하나를 골라 위치를 가져 오자	306
이제 미사일을 전진시키자	309
빌딩과의 충돌을 체크하자	311
빌딩과 충돌 순간 미사일을 회수하자	314
충돌 사실을 빌딩에 어떻게 알려 줄 것인가?	319
빌딩 파괴 이벤트를 Action으로 만들자	322
빌딩 파괴 이펙트 프리팹 만들기	327
빌딩 파괴 효과(DestroyEffect) 전용 팩토리 만들기	330
시간이 지나면 폭발 효과 사라지게 하자	338
미사일을 자동으로 생성하자	343
남아 있는 빌딩이 있는지 확인해 보자	353
미사일과 총알의 폭발효과의 충돌 처리	358
ScoreManager로 점수를 계산하자	362
간단한 UI를 만들어 점수를 표시하자	381
UIRoot.cs 스크립트를 만들자	384
UI에서 스코어(score)를 업데이트하자	392
스크린 바깥으로 나간 미사일들을 회수하자	395
게임 오버 처리하기	405
승리 조건을 체크하자	411
두 조건이 동시에 충족될 때의 승패 판정	416
게임 종료 이벤트 처리	420
게임 종료 이벤트 수신 함수들을 만들자	424
승리시의 보너스 점수 계산	432
승패 결과를 화면에 표시하기	436
오디오 매니저(AudioManager)를 만드는 방법	441
스크립터블 오브젝트를 이용하여 오디오 파일을 관리하자	443
오디오 매니저(AudioManager)를 만들자	455

오디오 매니저(AudioManager)를 이용해서 소리를 내자 마무리하며	460 464
---	-------------------

수록된 파일 안내 및 이 책을 보시는 방법

구매 후 다운 받으신 압축 파일을 풀어 보시면, 다음과 같은 파일들이 들어 있을 것입니다. 필요에 맞게 사용하시면 됩니다.

Book_MissileCommander.pdf	PDF 포맷의 전자책 파일입니다. 어도비 리더 등으로 보실 수 있습니다.
Book_MissileCommander.epub	Epub 포맷의 전자책 파일입니다. 애플의 iBooks 등으로 보실 수 있습니다.
Video Samples 폴더	동영상 예제 파일들이 들어 있습니다
MissileCommanderAssets.unitypackage	스프라이트와 오디오 에셋이 들어 있는 유니티 커스텀 패키지입니다.
MissileCommanderFinal.unitypackage	강의에서 완성한 샘플 게임 예제 패키지입니다.

한편, 이 책을 보시는 데 특별한 방법이 필요한 것은 아닙니다. 다음의 두 가지 특징만 염두에 두시면 됩니다.

동영상 예제 파일

책의 중요한 챕터마다 앞의 챕터 이후로 추가된 코드나 유니티 에디터 설정을 동영상으로 다시 한번 빠르게 보여 드리는 동영상 예제 파일의 이름이 들어 있습니다. 예를 들어 챕터 끝에 다음과 같은 표시가 있을 경우,

[동영상 예제 파일명: 097_play_sound_with_audio_manager.mp4]

“097_play_sound_with_audio_manager.mp4”라는 이름의 파일을 열어 보라는 의미입니다.

물론, 동영상 예제를 꼭 보셔야 하는 것은 아닙니다. 책으로 설명을 따라 가는데 문제가 없으시다면 굳이 동영상 예제를 열어 보실 필요가 없습니다. 책으로는 이해가 안 될 경우 참고용으로 만든 영상에 불과하기 때문입니다. (영상에는 음성 해설이 없습니다. 단순한 참고용이기 때문입니다.)

C# 스크립트 파일 이름 확인

각각의 예제 코드 상단 우측에는 해당 코드가 들어 있는 C# 파일의 이름이 표기되어 있습니다. 책을 읽는 도중 현재 어느 C# 클래스 파일에 코드를 작성하고 있는지 이해가 가지 않으실 경우, 이 파일 이름을 통해 빠르게 파악하실 수가 있습니다. 참고로 아래의 예시 코드는 GameManager.cs라는 C# 파일에 들어 있습니다.

```
GameManager.cs
void OnAllBuildingDestroyed()
{
    isAllBuildingDestroyed = true;
    GameEnded?.Invoke(false, buildingManager.BuildingCount);
    AudioManager.instance.PlaySound(SoundId.GameEnd);
}
```

이 책에서 다루는 내용들

미사일 커맨더(Missile Commander)는 우리에게 잘 알려진 고전 아케이드 게임으로서, 아주 간단한 게임 규칙을 가지고 있습니다. 이 책이 초급자 대상이었다면 아주 쉽고 간단한 방식으로 만들었겠지만, 중급자를 지향하는 분들을 대상으로 하고 있기 때문에 가급적이면 확장성과 유지 보수성을 염두에 두고 제작해 보았습니다. 예제 코드를 작성하는 동안 제가 주로 중점을 두었던 부분은 다음과 같습니다.

느슨한 커플링

게임을 개발하는 과정에서 우리는 어쩔 수 없이 원래의 기획을 변경해야 하는 경우를 마주치게 됩니다. 기획이 변경되면 이에 따라 새로운 코드를 추가하거나 기존 코드를 변경할 수밖에 없는데, 게임 개발의 속성상 이러한 상황은 피할 수가 없는 숙명과도 같습니다. 따라서 프로그래머는 자신이 지금 작성하는 코드에 대해 반드시 수정 요구가 들어올 것이라는 점을 염두에 두고, 미리 이에 대비한 구조 설계를 해 두어야 합니다. 이러한 대비책 중 하나는 느슨한 커플링(느슨한 클래스간의 결합) 구조를 만드는 것입니다. 다시 말해서 각각의 클래스들이 서로에 대해 너무 의존하지 않도록 설계해야 합니다.

클래스들이 너무 강하게 결합되어 있으면, 기획의 변경 등으로 인해 어떤 하나의 클래스를 변경했을 때 연쇄적으로 그 영향을 받는 클래스들이 생기게 됩니다. 심할 경우, 수정한 클래스와 직접 관련이 없어 보이는 클래스까지도 변경해야 하는 경우가 있습니다. 이런 일이 많아지면 코드의 유지 보수 기간과 비용이 급증하게 되어 전체적인

생산성이 크게 떨어질 수 밖에 없습니다. 여러분이 게임 프로그래밍을 할 때, 느슨한 커플링(결합)을 항상 염두에 두어야 하는 이유가 바로 여기에 있습니다. (클래스간의 상호 의존도가 높은 경우를 ‘강하게 커플링되었다’고 하고, 그 반대의 경우를 ‘느슨하게 커플링되었다’고 표현합니다.)

객체 지향적 프로그래밍

유니티의 차기 버전이 공개되면서, 유니티 C# 프로그래밍 방식이 기존의 객체 지향적 프로그래밍(Object-Oriented Programming)에서 데이터 지향적 프로그래밍(Data-Oriented Programming)으로 바뀌어 가는 추세입니다. (최근 코펜하겐에서 열린 Unite 2019 행사에서는 이 데이터 지향적 프로그래밍 방식인 DOTS에 대한 강연이 주를 이루었습니다. 따라서 유니티를 공부하시는 분들은 이제 이 방법론에 대해서도 관심을 가지셔야 합니다. DOTS은 아직 완성 단계가 아니라 시중에 관련된 책은 찾아 보기 어려우며 유니티 홈페이지에서 영문 자료로 정보를 습득하실 수 있습니다.)

그럼에도 불구하고 객체 지향적 프로그래밍은 아직까지 게임 프로그래머라면 반드시 깊이 있게 이해해야 할 분야입니다. 따라서 예제 게임인 미사일 커맨더(Missile Commander)를 만드는 과정에서 저는 최대한 객체 지향의 원칙에 충실하려고 애를 썼습니다. 간단한 코드들이지만 캡슐화와 추상화, 다형성, 그리고 상속성이라는 ‘객체 지향 프로그래밍의 원리’들을 모두 사용해 보았습니다. 이 중에서 상속은 자칫하면 강한 커플링의 원인이 되기 때문에 상속을 부득이하게 사용할 경우 1단계 이상의 상속은 사용하지 않았으며, 캡슐화와 추상화, 다형성은 느슨한 커플링을 위해 매우 중요한

원칙이므로 최대한 이에 충실한 코딩을 하기 위해 노력하였습니다. 각각의 코드를 설명하는 가운데에서 이들 개념을 직접적으로 사용하지는 않았지만, 이 책의 예제를 통해 충분한 실습을 하신 뒤 여러분이 가지고 계신 C# 입문서의 OOP(객체 지향적 프로그래밍) 항목을 다시 읽어 보시면 어떤 부분에서 객체 지향적 방법을 사용했는지 쉽게 이해하실 수 있을 것입니다.

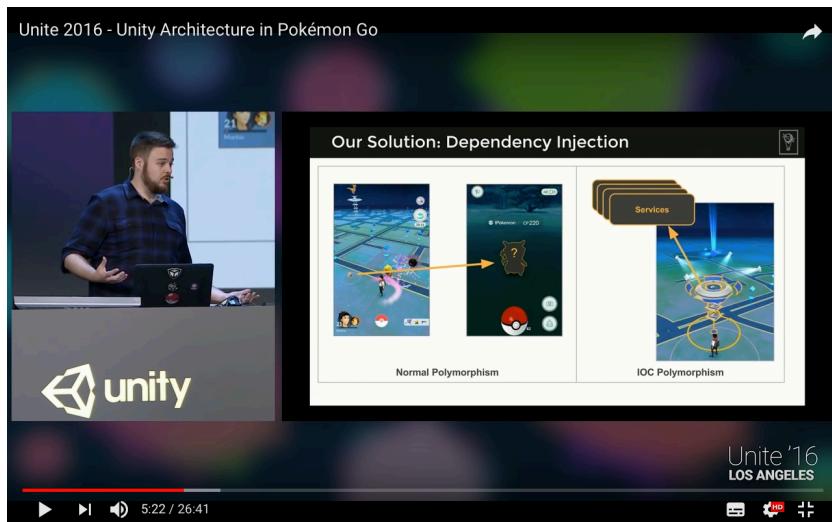
이벤트 주도형 프로그래밍

느슨한 커플링을 위해 제가 많이 사용한 방법은 이벤트(Event)를 이용하는 것입니다. 클래스의 인스턴스들(예를 들어 총알, 빌딩, 폭발 효과 등)이 서로 커뮤니케이션할 수 있게 하는 방법은 다양하지만, 확장성을 고려할 때 가장 이상적인 방법은 각각의 인스턴스들이 서로의 존재를 모르는 가운데에서도 필요한 정보를 주고 받을 수 있도록 하는 것입니다. 이벤트(Event)를 사용하면 이런 식의 게임 프로그래밍이 가능합니다.

이 책에서 제가 보여 드리는 예제를 따라 가시다 보면, 때로는 지나쳐 보일 정도로 이벤트 중심의 코딩을 하는 것을 경험하실 수가 있을 것입니다. 물론 실제 현업에서 코딩을 할 때 이렇게 모든 것을 이벤트 중심으로만 작성하지는 않지만, 이런 연습을 통해 이 방식의 장점과 단점을 파악할 기회를 가져 보시는 것은 매우 중요합니다. 특히 기획의 변경이 잦은 게임 개발 프로젝트를 할 경우, 이벤트 중심으로 코딩을 하게 되면 코드의 추가나 수정으로 인해 야기되는 ‘도미노 효과’를 큰 어려움 없이 피할 수가 있습니다.

디펜던시 인젝션(Dependency Injection)

흔히 ‘의존성 주입’이라고 번역되는 ‘디펜던시 인젝션’은 느슨한 커플링을 위해 많이 사용되는 방식입니다. 클래스 A가 어떤 일을 하기 위해 클래스 B의 인스턴스에 의존 해야 하는 경우, B를 A의 디펜던시(의존성)이라고 합니다. 그리고 디펜던시 인젝션이란 클래스 A내부에서 클래스 B의 인스턴스를 직접 만들어 참조하는 방법 대신, 클래스 A 외부에서 만들어진 인스턴스를 (외부에서 내부로) 주입받아 참조하는 방식을 가리킵니다. 이에 대해서는 제가 나중에 예제를 통해 자세히 설명 드릴 예정이므로 여기에서는 생소한 용어 때문에 너무 고민하실 필요는 없습니다. 책을 읽어 나가시는 과정에서 이 개념에 자연스럽게 친숙해지실 것입니다. (참고로 ‘포켓몬 고’ 개발자들도 이 개념에 근거하여 게임을 개발했다고 유나이트 2016에서 발표한 적이 있습니다. 관심 있는 분들은 유튜브에서 해당 영상 <https://youtu.be/8hru629dkRY> 을 찾아 보시기 바랍니다.)



싱글톤으로 대표되는 글로벌 참조의 최소화

초급자 대상으로 쓰여진 유니티 입문서나 기타 동영상 강좌들 중에는 싱글톤(Singleton) 패턴을 지나치게 선호하는 경우가 있습니다. 싱글톤을 일단 만들어 놓으면 사용하기가 매우 편합니다. 하지만 이 방식은 프로그래머들에게 금기라고 할 수 있는 글로벌 참조(언제 어디서나 바로 참조할 수 있는 클래스 인스턴스)를 기반으로 하고 있기 때문에 남용할 경우 위험합니다. 또한 이 방식은 클래스들이 강하게 결합하는 구조를 만들게 하므로 아무 생각 없이 싱글톤을 남발하게 되면 나중에 그 대가를 톡톡히 치르게 되는 날이 오게 됩니다. 따라서 이 책에서는 싱글톤 사용을 최대한 자제하였고, 맨 마지막의 오디오 매니저(AudioManager)를 만들 때만 이 방식을 제한적으로 사용하였습니다.

오브젝트 풀링이 적용된 팩토리(Factory)

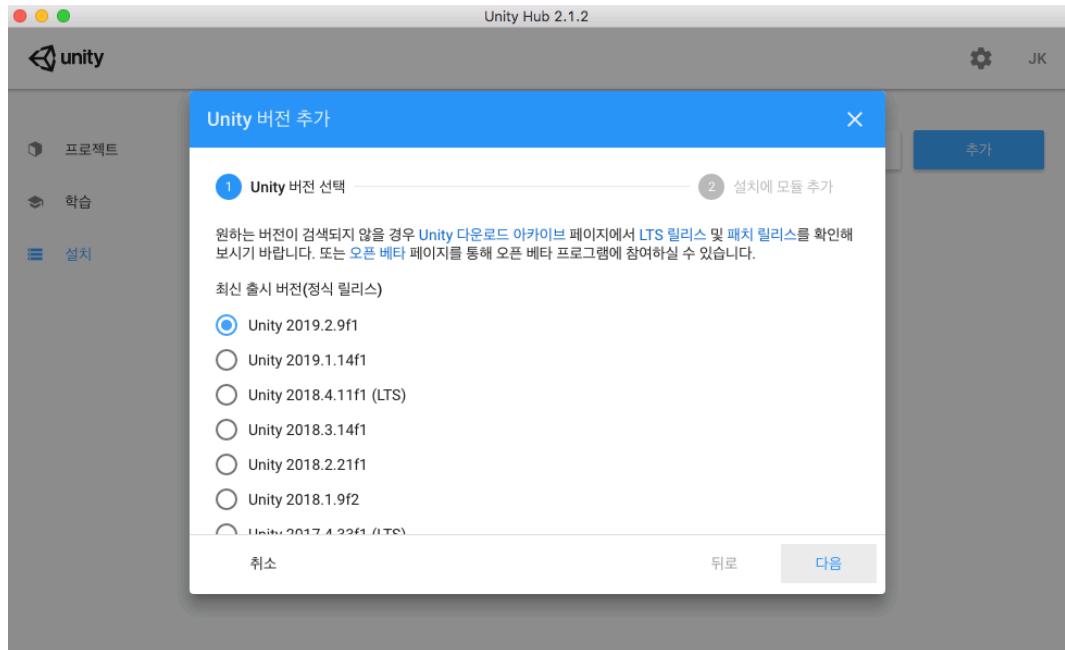
오브젝트 풀링(Object Pooling)은 클래스의 인스턴스들을 재활용해서 메모리를 절약하고, 동시에 가비지 컬렉션을 수행하는 횟수를 최소화해서 게임 도중에 퍼포먼스가 일시적으로 저하되지 않도록 하기 위한 방법입니다. 미사일 커맨더 게임 예제에서는 싱글톤 방식으로 글로벌하게 접근할 수 있는 오브젝트 풀링 대신, 디펜던시 인젝션(의존성 주입) 방식으로 사용할 수 있는 오브젝트 풀링 클래스를 구현하고 여기에 팩토리(Factory) 개념을 적용하여 범용적으로 사용할 수 있게 해 보았습니다.

컴포지션 루트(Composition Root) 개념의 사용

게임에 사용되는 각각의 클래스들(정확하게는 클래스의 인스턴스들)이 서로의 존재를 모르는 상태에서도 커뮤니케이션할 수 있게 하기 위해서는 게임의 시작 포인트(entry point)에 해당하는 클래스(여기에서는 GameManager) 한 곳에서 이들을 생성하고 서로 연동해 주는 작업을 전담하는 것이 좋습니다. 이런 방식을 컴포지션 루트(Composition Root)라고 합니다. 유니티에서는 컴포지션 루트의 역할을 하는 시작포인트가 엔진 차원에서 제공되고 있지는 않습니다. 따라서 원래는 이 방식을 ‘이론 그대로’ 사용할 수 없습니다. 하지만 그 원리를 부분적으로 나마 구현하는 방법이 없는 것은 아니며, 잘만 사용하면 상당히 편리합니다. 미사일 커맨더 게임 예제에서는 게임 매니저(GameManager) 클래스를 마치 컴포지션 루트처럼 사용하여, 중요한 클래스들이 서로의 존재를 모른 상태에서도 문제 없이 소통할 수 있게 만들어 보았습니다.

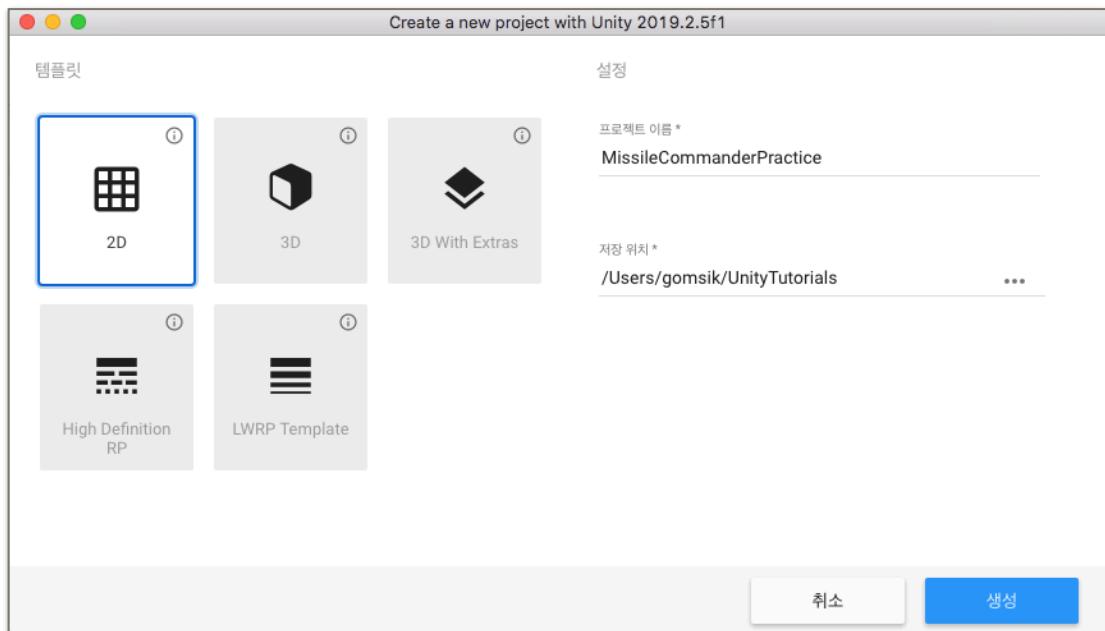
유니티 버전

이 책에서 사용한 유니티 버전은 2019.2.5f1 입니다. 하지만 미사일 커맨더(Missile Commander) 예제에서는 거의 C# 프로그래밍만 다루고 나머지 요소들은 가장 기본적인 것들만 사용하고 있기 때문에, 유니티 2019.2로 시작하는 어떤 버전을 사용하셔도 문제가 없습니다. 만약 해당 버전의 유니티가 아직 설치되어 있지 않다면, 유니티 허브에서 2019.2.x 번대의 버전을 찾아서 설치하시면 됩니다.

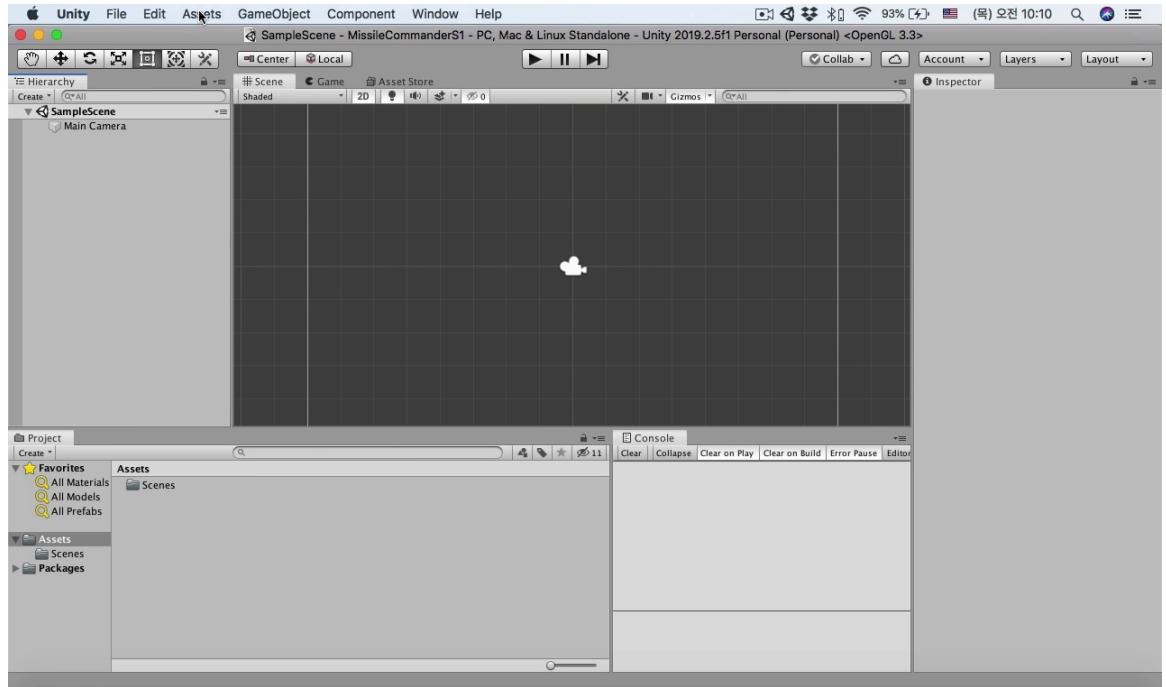


프로젝트 생성

제일 먼저 해야 할 일은 프로젝트를 생성하는 것입니다. 우선 유니티 허브에서 새로운 프로젝트를 만들고, 이름을 Missile Commander Practice 라고 정한 다음, 폴더 위치를 정하고 [생성] 버튼을 눌러서 프로젝트를 생성합니다. 이 때 주의할 것은 템플릿을 2D로 선택해야 한다는 것입니다. 우리가 만들고자 하는 예제 게임이 스프라이트 기반 2D 게임이기 때문입니다.



유니티 에디터의 레이아웃은 여러분이 편하신 대로 설정하시면 되지만, 저의 경우는 사용하고 있는 맥북의 해상도가 낮기 때문에 디폴트 상태로 정한 뒤, 콘솔창만 앞으로 꺼내도록 하겠습니다. 여러분들은 저보다 해상도가 높은 모니터를 사용하실 가능성 이 높으므로 저처럼 하지 않으셔도 됩니다.



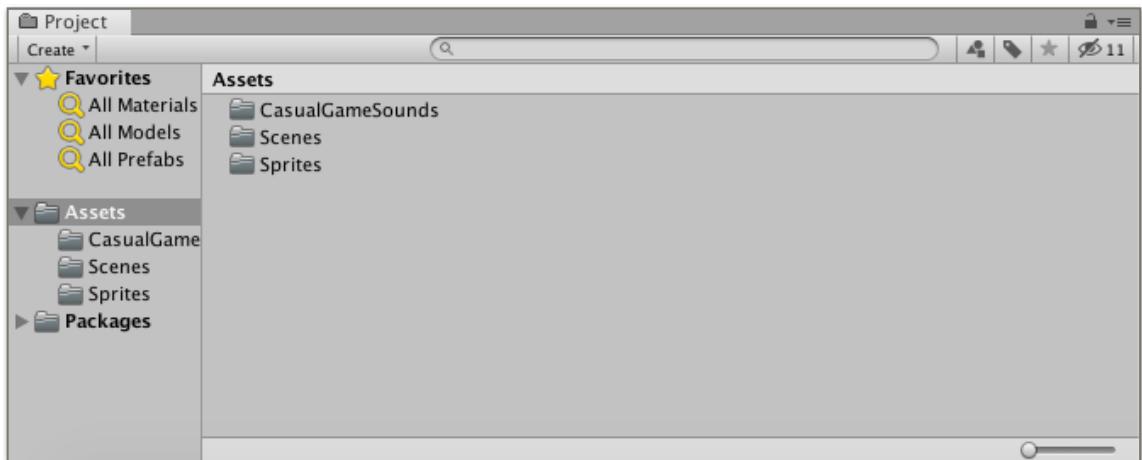
에셋 불러오기

다음으로 해야 할 것은 예제 게임에 사용할 에셋을 임포트(import)하는 것입니다. 이 책자와 함께 제공된 파일 중, 다음과 같은 이름을 가진 패키지 파일이 있을 것입니다.
이것을 유니티 프로젝트로 임포트하시면 됩니다.

파일명: MissileCommanderAssets.unitypackage



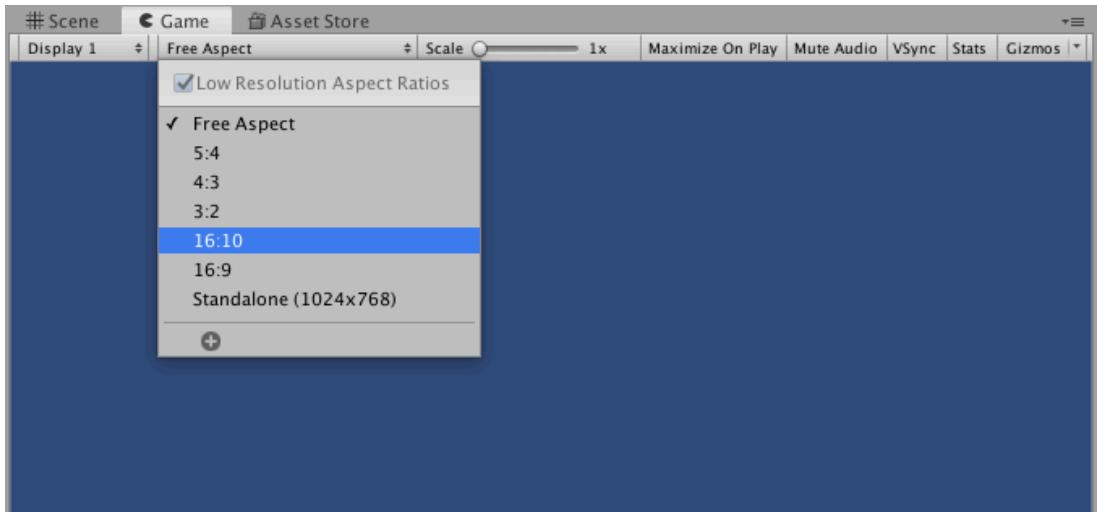
이 패키지 파일 안에는 제가 만든 아주 조잡한 스프라이트 파일들과 유니티 에셋스토어에 공개되어 있는 프리-라이센스 사운드 파일(CasualGameSounds)이 몇 개 들어 있습니다. 이것이 전부입니다. 코딩 위주로 설명할 것이기 때문에 그 흔한 애니메이션도 없습니다. 애니메이션을 만들고 설정하는 번거로운 과정 자체를 피하기 위해서 그렇게 하였습니다. 에셋에 대해서는 너무 신경쓰지말고, 코딩 자체에 집중하시면 됩니다.



[동영상 예제 파일명: 000_import_package.mp4]

게임 뷰(Game View) 비율 설정

다음으로는 게임 뷰(Game View)에서 게임 화면의 가로세로 비율을 조정해 보겠습니다. 어떤 식으로 하셔도 상관은 없지만 저는 그냥 16:10으로 지정하겠습니다.

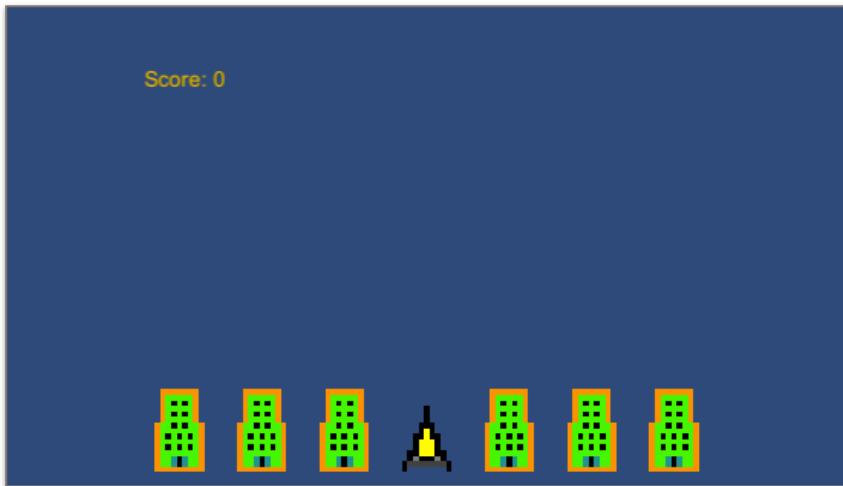


이제 본격적인 코딩 작업으로 들어가기 전의 사전 작업은 다 끝났습니다. 그럼 이제부터 게임 자체에 대해 이야기를 해 보겠습니다.

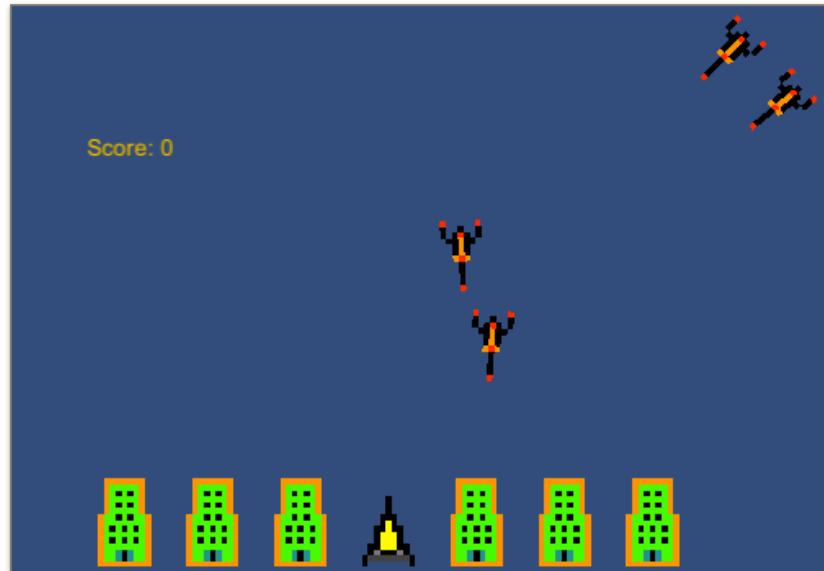
미사일 커맨더 예제 게임 플레이

지금부터 만들고자 하는 예제 게임은 아주 간단한 기능을 가지고 있습니다. 이 책에서는 본격적인 게임을 만들고자 하는 것이 아닌, 핵심 게임 플레이를 구현하기 위한 프로그래밍 방법론을 다루는 것이 목표이기 때문입니다. 여하튼 미리 만들어 놓은 게임을 플레이하면서 이 게임의 규칙을 간단히 소개해 보겠습니다.

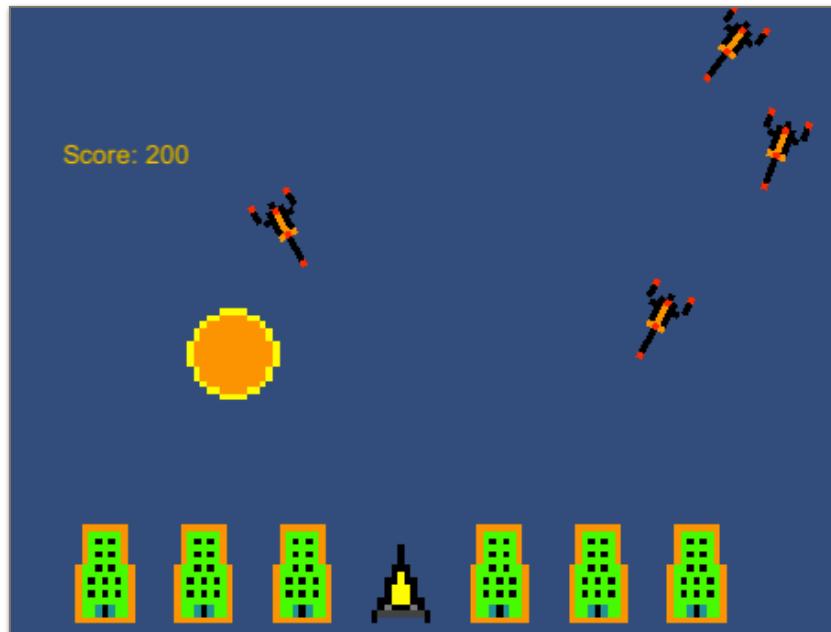
게임이 시작되고 약간의 시간(스탠바이 타임)이 지나면 화면에 미리 지정한 개수의 빌딩과, 빌딩을 지키기 위한 총알 발사대(MissileLauncher)가 나타납니다. 화면 왼쪽 상단에는 스코어 표시가 되어 있습니다.



그리고 난 뒤, 하늘에서 적의 미사일이 자동으로 생성되어 빌딩을 향해 날아 옵니다.
적의 미사일이 목표로 하는 빌딩은 파괴되지 않는 것들 중에서 랜덤으로 선택됩니다.



미사일이 날아오기 시작하면 이쪽에서도 응사를 해야 합니다. 마우스로 원하는 지점 을 클릭하면 해당 지점을 향해 총알(대공포)이 발사됩니다. 발사된 총알이 마우스 클 리к 지점에 도달하면 자동으로 폭발하며, 이 폭발 영역에 휘말린(충돌한) 적의 미사일 은 파괴됩니다. 적의 미사일을 파괴할 때마다 스코어가 올라갑니다.



빌딩이 모두 파괴되기 전에 날아 오는 적의 미사일을 모두 파괴하면 게임에서 승리하게 됩니다. 반면에 빌딩이 모두 파괴되면 게임에서 패배하게 됩니다. 그리고 그 결과가 화면에 나타납니다.



[동영상 예제 파일명: 001_gameplay_preview.mp4]

이것으로 끝입니다. 진짜 간단한 게임입니다. 사실 유니티와 C# 스크립트의 기본을 알고 계신 분들이라면 이 정도 게임을 만드는 것은 간단한 일일 것입니다. 하지만 이 책에서는 아무리 간단해 보이는 게임이라고 해도, 확장성 있고 유지 보수가 쉽도록 개발하려면 생각보다 많은 것을 고려해야 한다는 점을 보여 드리려고 합니다. 따라서 게임이 간단하다고 너무 실망하지 마시고 계속 더 읽어 보시기를 권하고 싶습니다.

필요한 게임 오브젝트 리스트

우선 미사일 커맨더 게임을 만들기 위해 필요한 게임 오브젝트(GameObject)들에는 어떤 것들이 있을지 한번 생각해 보도록 하겠습니다. 아마 다음과 같은 게임 오브젝트들이 필요할 것입니다.

- 총알 발사대 (Bullet Launcher)
- 총알(Bullet)
- 적 미사일(Enemy Missile)
- 폭발 효과(Explosion)
- 파괴 효과(Destroy Effect)
- 빌딩(Building)

그럼 이 중에서 플레이어(player)가 직접 컨트롤하는 게임 오브젝트가 무엇인지 찾아보겠습니다. 현재 나열된 게임 오브젝트 중에서 사용자의 입력을 받아 어떤 행동을 하는 것은 총알 발사대(Bullet Launcher) 하나 밖에 없습니다. 따라서 이것을 가장 먼저 구현해 보는 것으로 하겠습니다.

총알 발사대(BulletLauncher) 제작

총알 발사대 만들기

총알 발사대(BulletLauncher)를 제일 먼저 만드는 이유는, 이것이 플레이어의 입력을 받아서 행동하는 유일한 클래스이기 때문입니다. 총알 발사대는 플레이어가 마우스를 클릭할 때마다 총알을 생성합니다. 총알 발사대가 할 일은 이것이 거의 전부입니다. 생성된 총알이 무엇과 충돌해서 어떤 결과를 야기하는지에 대해서는 상관하지 않습니다. 오로지 총알의 생성과 파괴, 그리고 이와 관련한 이벤트를 발생시켜 외부의 다른 클래스와 서로 소통하는 것이 총알 발사대의 중요한 역할입니다.

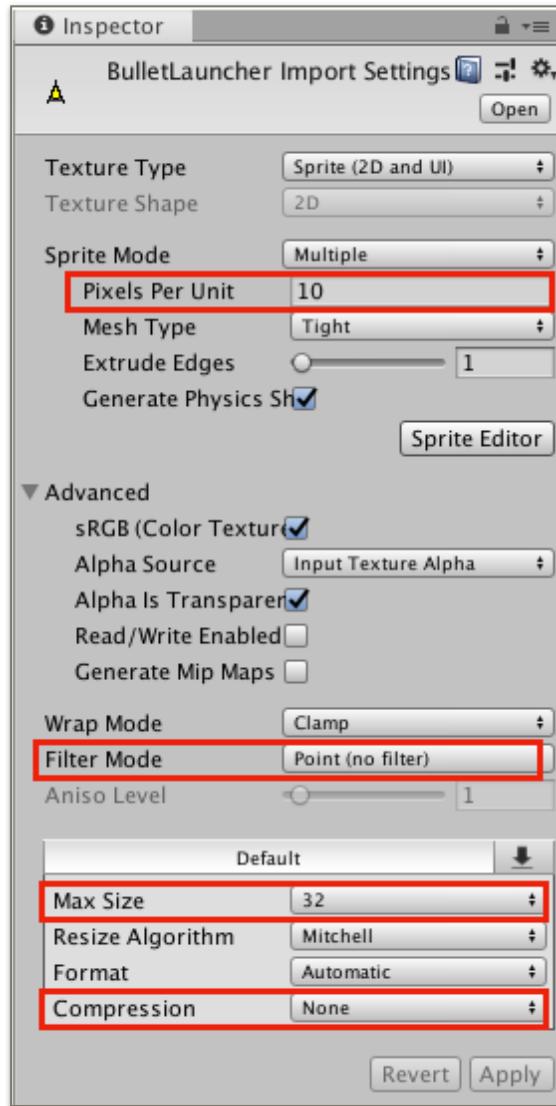
'S.O.L.I.D'라는 이름의 소프트웨어 개발 원리 중에 '단일 책임의 원리(Single Responsibility Principle)'라는 것이 있습니다. 모든 클래스 또는 메소드(함수)는 단 하나의 분명한 역할만을 책임져야 한다는 이야기입니다. 이 원리에 따라, 총알 발사대의 역할도 '총알의 생성과 관리'라는 간단 명확한 부분으로 한정시켰습니다. (클래스 하나에 수 천~ 수 만 줄의 코드가 들어 있는 것을 상상해 보시기 바랍니다. 코드 작성 후 한달이 지난 뒤에 열어 보면 분명히 자신이 짠 코드임에도 불구하고 그 내용을 다시 이해하는 데 꽤 많은 시간이 필요할 것입니다. 코딩 도중 클래스 하나의 역할이 너무 많아진다고 생각될 경우, 과감하게 클래스를 기능에 따라 분리해서 단일 책임의 원리에 충실히 지繇하고 노력하는 것이 좋습니다.)

스프라이트 살펴 보기

우선은 앞에서 임포트(import)하신 스프라이트를 이용하여 총알 발사대를 만들어 보겠습니다. Sprites라는 이름의 폴더에 들어가 보시면, 제가 그런 정말로 조잡한 스프라이트들이 보일 것입니다.



현재 이 스프라이트들은 제가 미리 만들어서 필요한 세팅을 다 해 놓은 상태입니다. 이 중에서 BulletLauncher 스프라이트를 선택한 뒤, 인스펙터(Inspector)를 한번 살펴보겠습니다.



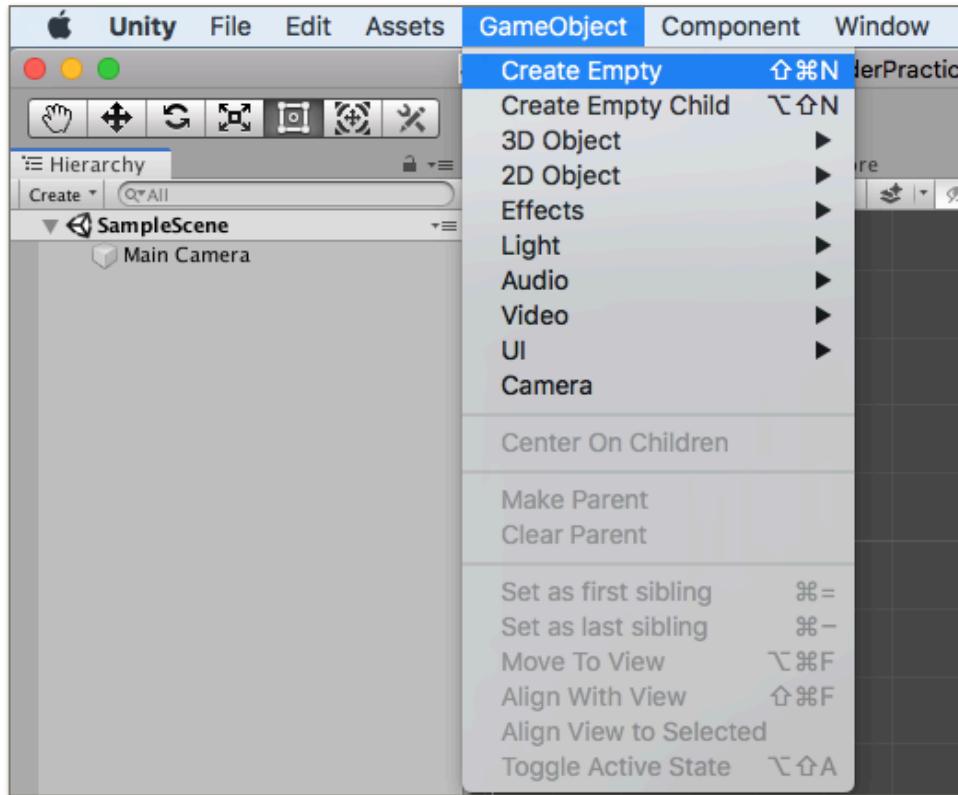
이 중에서 눈여겨 보실 부분은 Pixels Per Unit 을 10 으로 한 부분과 Filter Mode를 Point(no filter)로 한 부분, 그리고 Max Size를 32, Compression 을 None으로 한 부분입니다.

사실 이 세팅들이 큰 의미는 없고, 그냥 제가 그린 픽셀 아트의 사이즈가 워낙 작아서 이렇게 값을 조정한 것입니다. 위와 같이 하지 않으면 게임에서 스프라이트 이미지를 이 깨알같이 작게 나타나는 데다가, 이미지 자체도 뭉개져 보일 것이기 때문에 1 픽셀이 1 Unit 을 차지하도록 단위를 키워서, 스프라이트 이미지가 화면에 크게 보이도록 설정한 것입니다. 이미지 압축(Compression)을 하지 않은 이유도 그 때문입니다. 안 그래도 작은 픽셀 이미지를 압축까지 하면 뭉개져 버릴테니까요. (Max Size 를 32로 한 이유도 이미지 자체가 너무 작아서 그런 것인니까 큰 의미는 없습니다)

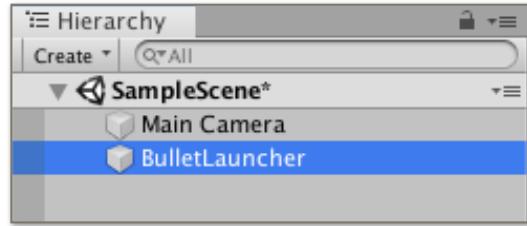
여하튼 이러한 스프라이트 세팅은 이 책자의 주제인 코딩 자체와는 직접적인 상관 관계가 없으니까, 그냥 이런 게 있다고 생각하시고 넘어가시면 됩니다.

총알 발사대 게임 오브젝트 만들기

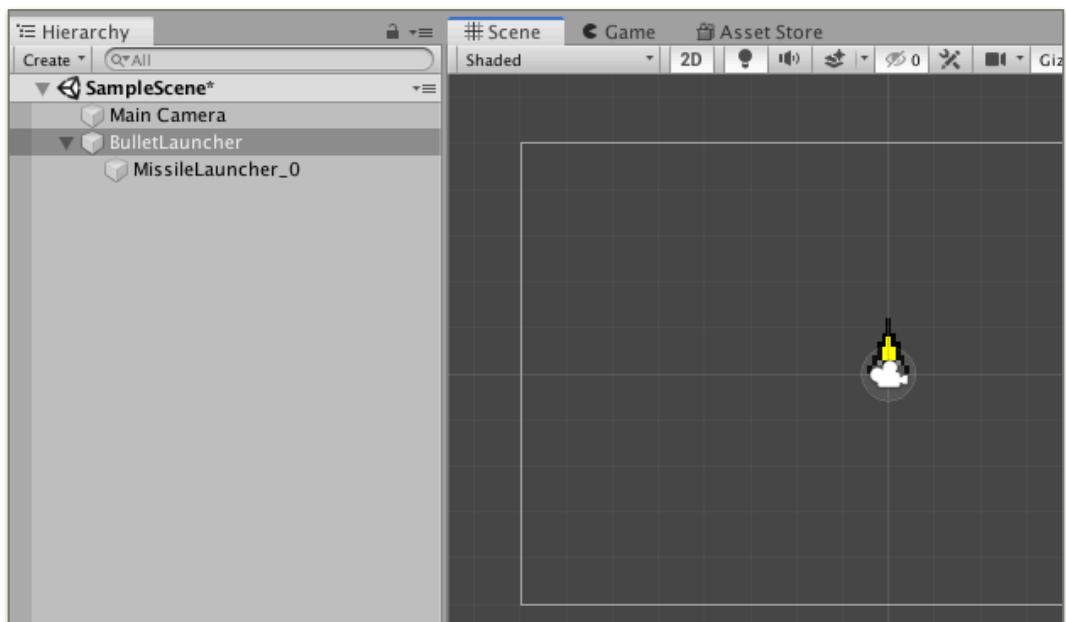
이제 총알 발사대(BulletLauncher) 게임 오브젝트를 만들겠습니다. 우선 상단의 메뉴에서 GameObject->Create Empty를 선택해서 빈 게임 오브젝트를 현재의 씬(Scene)에 생성하겠습니다.



그리고 나서 계층류(Hierarchy)에서 지금 만든 빈 게임 오브젝트의 이름을 BulletLauncher로 바꾸겠습니다.

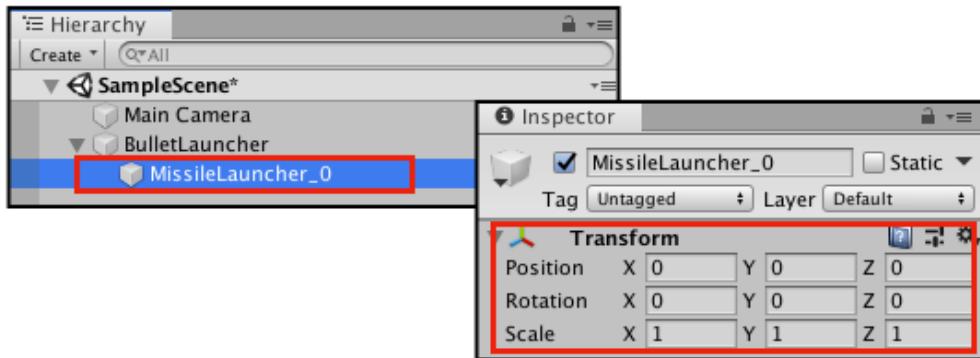


다음으로는 프로젝트뷰(Project)의 Sprites 폴더에서 아까 BulletLauncher 스프라이트를 드래그해서 BulletLauncher 게임 오브젝트의 자식 오브젝트로 만듭니다.



그러면 보시는 것처럼 씬의 한 가운데에 총알 발사대(Bullet Launcher)의 스프라이트 이미지가 놓여지게 될 것입니다.

확인을 위해, MissileLauncher_0 을 누르신 뒤, 인스펙터(Inspector)를 보시기 바랍니다. 로컬 포지션(position)과 로테이션(Rotation)이 각각 0,0,0 그리고 스케일(Scale)이 1,1,1 이 되어야 합니다. 만약 이렇게 되지 않았다면 트랜스폼 컴포넌트에 있는 Reset 를 눌러서 값을 초기화하시기 바랍니다.

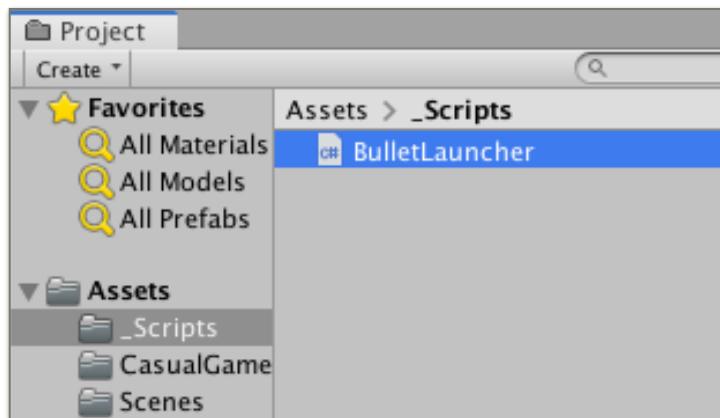


[동영상 예제 파일명: 002_bulletLauncher_creation.mp4]

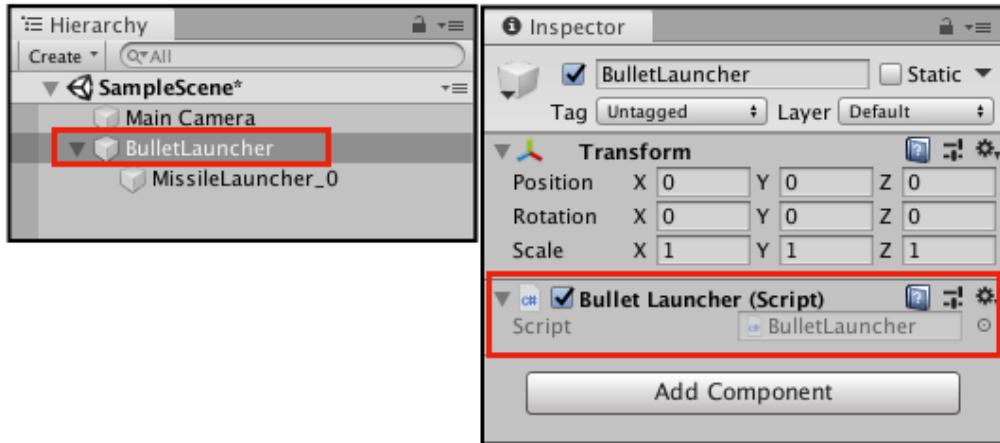
이제 모든 준비가 되었으니 코딩을 시작하겠습니다. 우선 프로젝트 뷰의 Assets 폴더 아래에 _Scripts 라는 이름의 폴더를 하나 생성하겠습니다.

폴더의 이름에 언더스코어(_)를 붙여서 _Script 라고 이름 지은 이유는 스크립트 폴더가 항상 리스트의 맨 위에 표시되도록 하기 위해서입니다. 미사일 커맨더와 같은 간단한 게임을 만들 때는 꼭 이렇게 하지 않아도 되지만, 만약에 여러분의 프로젝트 규모가 커지고, 이에 따라 폴더가 너무 많아지게 되면 스크립트 폴더 하나를 찾는데에도 오랜 시간이 걸릴 것입니다. 따라서 이와 같은 방식으로 이름을 붙이면 편리합니다. (여하튼 이건 별로 중요한 것은 아니고, 프로그래머들이 자주 사용하는 팁 같은 것이라서 알려 드렸습니다.)

이제 스크립트 폴더로 가서 새로운 C# 스크립트 파일을 하나 만들고, 이름을 BulletLauncher 라고 짓도록 하겠습니다.



그리고 난 다음에, 계층(Hierarchy)에서 BulletLauncher 게임 오브젝트를 선택한 다음에, 방금 만든 스크립트를 드래그해서 붙여 주겠습니다.



이제 스크립트가 연결되었으니, 이것을 더블클릭해서 비주얼 스튜디오(Visual Studio)로 파일을 열어 보겠습니다. 저의 경우는 맥(Mac) 버전을 사용하고 있기 때문에, 마이크로소프트 윈도우즈(Windows)를 사용하시는 분들은 화면 구성이 조금 다를 것입니다. 하지만 비주얼 스튜디오의 기본적인 기능과 코드 내용은 동일하니까 신경쓰지 않으셔도 됩니다.

[동영상 예제 파일명: 003_bulletLauncher_script_creation.mp4]

마우스 입력(Input) 받아들이기

방금 만든 BulletLauncher.cs 파일을 열어 보시면 다음과 같이 되어 있을 것입니다.

BulletLauncher.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BulletLauncher : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

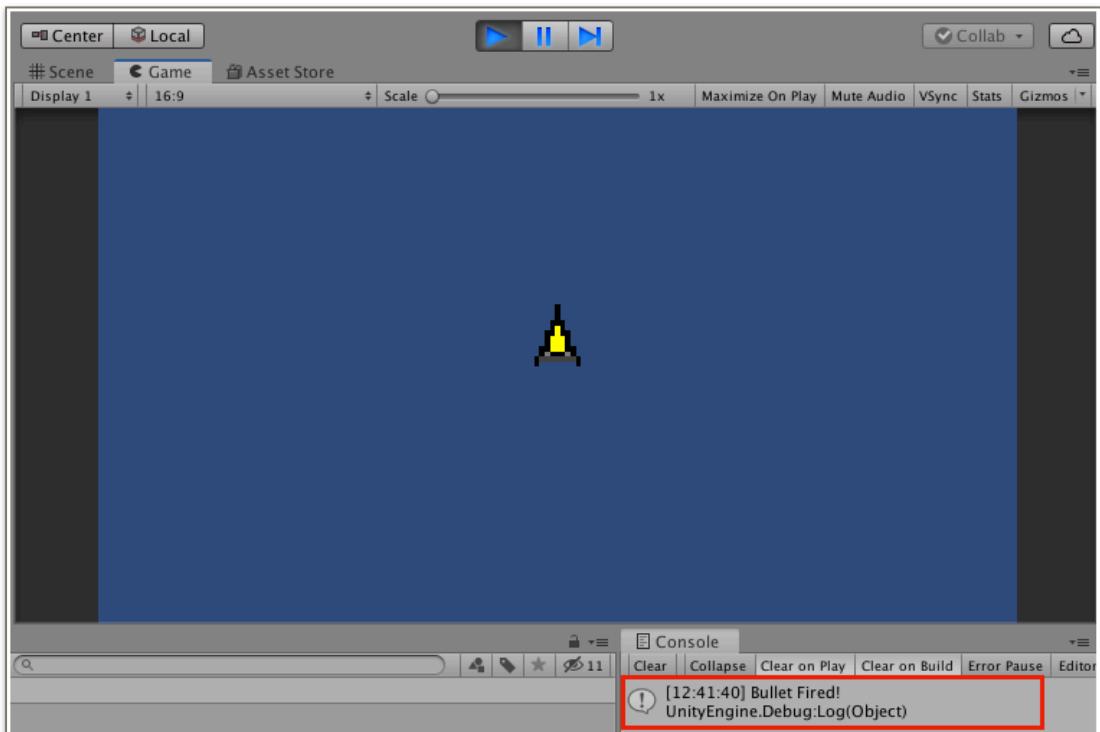
    }
}
```

제일 먼저 플레이어가 마우스 왼쪽 버튼을 클릭했을 때, 이에 반응하여 콘솔창에 간단한 텍스트를 표시하는 코드를 작성해 보겠습니다. 일단은 우리가 일반적으로 많이 사용하는 방법부터 사용해 보겠습니다. Update() 함수 안에 다음과 같이 코드를 입력합니다.

BulletLauncher.cs

```
void Update()
{
    if (Input.GetMouseButtonDown(0))
    {
        Debug.Log("Bullet Fired!");
    }
}
```

이 상태에서 스크립트 파일을 저장한 뒤에 유니티 에디터로 돌아가서 게임을 실행해 보겠습니다. 게임을 실행하고 게임 화면을 마우스로 클릭하면 다음과 같이 “Bullet Fired!”라는 메시지가 콘솔창에 뜨게 될 것입니다.



[동영상 예제 파일명: 004_Input_mouse_basic.mp4]

키보드 입력 방식 추가하기

이것이 일반적으로 마우스 입력을 게임에서 받아 들이는 코딩 방식입니다. 대부분의 초보자 대상의 유니티 입문서나 동영상 강좌에서는 이러한 방식으로 사용자 입력을 받아 들입니다. 게임의 콘트롤 방식이 하나인 경우(예를 들어 마우스 전용 게임 같은 것)에는 이런 식으로 만들어도 문제가 없습니다. 하지만 게임이 출시된 후, ‘키보드 입력을 통해서도 게임을 콘트롤 할 수 있게 하자’고 게임 기획, 운영상의 변경이 발생했을 때는 어떻게 해야 할까요? 이런 경우에는 다음과 같이 Update() 함수에 추가 코드를 넣는 수밖에 없을 것입니다.

BulletLauncher.cs

```
void Update()
{
    if (Input.GetMouseButtonDown(0) || Input.GetKeyDown(KeyCode.Space))
    {
        Debug.Log("Bullet Fired!");
    }
}
```

위의 코드를 보시면, Input.GetMouseButtonDown(0)이나
Input.GetKeyDown(KeyCode.Space)는 서로 다른 명령어를 사용하고 있지만, 실제로 이 두 가지 다른 명령어가 하는 일은 동일합니다. 즉, ’총알을 발사한다’는 똑같은 일을 할 뿐입니다.

또 다른 콘트롤러 추가로 지원하기

만약 여기에서 그치지 않고, 나중에 특정한 서드파티 회사(예를 들어 오쿨러스)의 콘트롤러도 지원해야 하는 경우가 다시 생긴다면 어떻게 될까요? 역시 사용자 입력을 처리하는 코드를 다시 찾아서 코드를 수정해 주어야 할 것입니다. 예를 들어 다음과 같은 식입니다. (참고로 아래 코드는 따라서 입력하지 마시기 바랍니다. 그냥 예를 든 것이므로 따라서 입력하시면 에러가 날 것입니다)

```
void Update()
{
    if (Input.GetMouseButtonDown(0) || Input.GetKeyDown(KeyCode.Space) ||
        OVRInput.GetDown(OVRInput.Button.One))
    {
        Debug.Log("Bullet Fired!");
    }
}
```

이런 식으로 새로운 콘트롤 방식을 추가해야 할 때마다 일일이 코드를 수정하는 것은 매우 비효율적인 일입니다. 또한 개발을 하다 보면 새로운 콘트롤 방식을 추가하는 것 뿐 아니라 빼야 할 경우도 있을 텐데 그 때마다 기존 코드를 찾아서 일일이 수정해야 한다면 정말 번거로운 일이 아닐 수 없습니다. 따라서 지금 보여 드린 방식으로 계속 작업한다는 것은 바람직하지 않습니다. 그럼 어떻게 하는 것이 좋을까요? 여기에 서 사용할 수 있는 것이 ‘인터페이스(interface)를 이용하는 방법’입니다.

C# 프로그래밍에 익숙하지 않은 분들을 위해 참고로 말씀드리면, 여기에서 말하는 인터페이스(interface)는 우리가 흔히 말하는 ‘그래픽 사용자 인터페이스(GUI)’와는 다

른 ‘프로그래밍적’인 개념입니다. 인터페이스라는 것은 클래스와 매우 비슷하게 생겼지만 클래스와 달리 실제 구현된 내용이 없습니다. 대신 클래스 작성을 위한 규약(약속)에 해당하는 항목들만 들어 있습니다. 예를 들어 “이 인터페이스를 구현한 클래스는 반드시 특정한 형식을 갖춘 함수를 꼭 만들어야 한다”는 식입니다. 이렇게 설명 드리면 잘 이해가 안되실텐데니까, 일단은 직접 만들어 보면서 차근차근 설명드리도록 하겠습니다.

BulletLauncher.cs 코드 초기화

일단 진행하기 전에 아까 예제로 만들었던 코드를 다 지우겠습니다.

BulletLauncher.cs에서 Update()에 들어 있던 코드를 다 지우고 초기 상태로 되돌립니다.

```
BulletLauncher.cs
void Update()
{
    if (Input.GetMouseButtonDown(0) || Input.GetKeyDown(KeyCode.Space) ||
        OVRInput.GetDown(OVRInput.Button.One))
    {
        Debug.Log("Bullet Fired!");
    }
}
```

그리면 BulletLauncher.cs에는 원래대로 Start()와 Update() 함수만 남아 있게 될 것입니다.

```
BulletLauncher.cs
public class BulletLauncher : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
    }
}
```

IGameController 인터페이스 만들기

우선 모든 게임 컨트롤러 관련 클래스에 공통적으로 적용될 규칙을 담을 인터페이스를 만들겠습니다. 인터페이스의 이름을 정할 때는 일반적으로 인터페이스를 의미하는 알파벳 대문자 'I'를 맨 앞에 붙이는 것이 권장됩니다. 또한 반드시 그래야 하는 것은 아니지만 이름의 마지막 부분을 어떤 속성이나 성질, 가능성을 나타내는 'able'을 붙이는 경우가 많습니다. 하지만 여기에서는 그냥 이를 무시하고, 알기 쉽게 IGameController 라고 이름을 붙이도록 하겠습니다.

먼저 유니티 에디터의 [Script] 폴더 안에 IGameController.cs라는 이름의 C# 스크립트 파일을 만들어 넣고 더블클릭해서 오픈해 보겠습니다.

IGameController.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class IGameController : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

유니티 에디터 안에서 C# 스크립트 파일을 만들었기 때문에 IGameController 클래스는 MonoBehaviour 의 파생 클래스로 자동 설정되어 있는 것을 확인할 수 있습니다. 인터페이스를 만들 때 이 상속 관계는 필요가 없으므로 다 지우겠습니다.

IGameController.cs

```
public class IGameController : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

그리고 난 뒤, class 부분을 지우고,

IGameController.cs

```
public class IGameController
{
}
```

이 부분을 아래와 같이 interface 라는 키워드로 바꿔 줍니다.

IGameController.cs

```
public interface IGameController
{
}
```

다음으로는 인터페이스에 필요한 함수 규약을 추가할 차례입니다. 여기서 말하는 ‘규약’이란, ‘이 IGameController 인터페이스에서 파생된 모든 클래스는 반드시 여기에 서 지정한 함수를 똑같은 형식으로 구현(implementation)해야 한다’는 규칙을 말합니다. 일단 코드를 작성해 놓고 다시 설명 드리겠습니다. 다음과 같이 코드를 입력합니다.

```
IController.cs  
public interface IGameController  
{  
    bool FireButtonPressed();  
}
```

인터페이스에 대한 사전 지식이 없으시다면 왜 함수에 중괄호 영역 { } 이 없나 하고 궁금하실 것입니다. 인터페이스는 단지 규칙만 정하는 것이고, 실제 함수 내용은 이 인터페이스를 구현한 클래스(class) 쪽에서 구현하도록 되어 있습니다. 따라서 위의 예시문처럼 ‘리턴값 타입’과 ‘함수 이름’만 설정하면 끝나게 됩니다. 참고로 함수 앞에 public이나 private 같은 접근 제어자(access modifier)도 붙일 필요가 없습니다. (만약 함수에 인자가 있을 경우에는 이에 대해서도 정의해 줍니다. 예를 들어 int Add(int a, int b); 와 같은 식으로 정의합니다.)

이제 인터페이스를 만들었으니 이 인터페이스를 구현한 클래스들을 만들어 보겠습니다. 계속 진행하기 전에, 지금 만든 FireButtonPressed()이라는 함수의 역할에 대해 설명 드리자면, 이 함수는 플레이어가 마우스로 화면을 클릭하거나 키보드 또는 게임 콘트롤러의 발사 버튼을 눌렀을 때 ‘true’ 값을 리턴함으로써, 이에 대해 알려 주는 역할을 하게 됩니다. 그럼 이 인터페이스에 근거하여, 먼저 마우스 입력을 이용하는 게

임 콘트롤러 클래스를 만들고 다음으로는 키보드 입력을 이용하는 클래스를 만들어 보겠습니다

[동영상 예제 파일명: 005_IGameController_interface.mp4]

MouseGameController.cs 만들기

우선 새로운 유니티 C# 스크립트 파일을 만들고 이름을 MouseGameController.cs라고 이름 붙이도록 하겠습니다. 그리고 이 파일을 비주얼 스튜디오로 열어 보시면, 다음과 같이 MonoBehaviour 클래스를 상속한 보통의 유니티 C# 클래스를 보실 수 있을 것입니다.

MouseGameController.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MouseGameController : MonoBehaviour
{
    // Use this for initialization
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

그런데 우리가 원하는 것은 MouseGameController 가 앞에서 만든 IGameController 인터페이스를 구현하도록 하는 것입니다. 따라서 MonoBehaviour 상속과 관련된 부분들을 모두 지워 줍니다

```

public class MouseGameController : MonoBehaviour
{
    // Use this for initialization
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}

```

다음으로는 이 클래스가 IGameController 인터페이스에서 파생되도록 하겠습니다.

```

public class MouseGameController : IGameController
{
}

```

그런데 이렇게 코드를 바꾸고 나면, 비주얼 스튜디오에 다음 그림과 같이

IGameController 부분에 밑줄이 생기는 것을 보실 수 있습니다.

```

public class MouseGameController : IGameController
{
}

```

IGameController 인터페이스에서 파생된 클래스는 이 인터페이스에서 규정한 FireButtonPressed()이라는 함수를 반드시 구현해야 하는데, 현재

MouseGameController.cs 에는 이러한 함수를 구현한 부분이 없기 때문에 경고가 나오는 것입니다. 그럼 지금부터 실제 함수를 구현하도록 하겠습니다.

FireButtonPressed() 함수 구현하기

IGameController 인터페이스에 정의된 FireButtonPressed() 함수를 구현하기 위해서는 일일이 이 함수를 타이핑해서 구현해도 되지만 비주얼 스튜디오 내장 기능을 이용하면 번거로운 타이핑 과정 없이도 함수 구현부를 쉽게 작성할 수 있습니다. 커서를 IGameController 에 위치시킨 다음에 맥(Mac)에서 옵션키 + 엔터키를 누르거나, 윈도우 pc 에서 ALT + 엔터키를 누르면 다음과 같은 메뉴가 뜨게 됩니다.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class MouseGameController : IGameController
6  {
7      명시적으로 인터페이스 구현
8      생성자 'MouseGameController()' 생성
9 }
```

위의 그림처럼 ‘인터페이스 구현’이 선택된 기본 상태에서 엔터키를 누르면 다음과 같이 IGameController 인터페이스에 정의된 FireButtonPressed() 함수를 구현할 수 있도록 자동으로 함수의 정의에 해당하는 코드가 만들어집니다.

```
MouseGameController.cs
public class MouseGameController : IGameController
{
    public bool FireButtonPressed()
    {
        throw new System.NotImplementedException();
    }
}
```

만약에 비주얼 스튜디오를 사용하지 않거나, 아니면 다른 어떤 이유로 위와 같이 함수가 자동으로 만들어지지 않으면 그냥 위의 함수 코드(빨간색으로 표시한 부분)를 그대로 타이핑하시면 됩니다.

그리고 위 함수를 잘 보시면 public이라는 접근 제어자(access modifier)가 자동으로 붙어 있는 것을 확인하실 수 있습니다. 분명히 IGameController 인터페이스에서는 bool FireButtonPressed() 앞에 아무런 접근 제어자를 붙이지 않았는데, 이 인터페이스를 구현한 MouseGameController 클래스에는 ‘public’이 자동으로 붙은 것입니다. 인터페이스에서 정의한 함수는 외부와 소통하기 위한 수단이므로 이를 ‘public’이 아닌 ‘private’으로 설정하게 되면 사실상 인터페이스를 사용하는 의미가 없습니다. 따라서 인터페이스를 구현한 클래스에서는 무조건 public으로 자동 설정한다고 생각하시고 진행하시면 되겠습니다.

여하튼 자동으로 작성된 코드의 FireButtonPressed() 함수 부분을 잘 보시면, 다음과 같은 코드가 들어 있는 것을 확인하실 수 있습니다.

```
throw new System.NotImplementedException();
```

이 부분은 비주얼 스튜디오에서 자동으로 작성한 코드인데, 아직 코드의 실질적인 내용이 구현되지 않았다(not implemented)는 일종의 경고를 하는 것입니다. 따라서 우리가 해야 할 일은 이 명령어를 지우고, 그 안에 실제 코드를 채워 넣는 것입니다. 다음과 같이 해당 코드를 지우도록 하겠습니다.

```
public bool FireButtonPressed()
{
    throw new System.NotImplementedException();
}
```

이제 이 안에 다음과 같은 코드를 작성해서, 마우스 왼쪽 버튼이 클릭되었는지를 리턴 값으로 전달하도록 하겠습니다.

```
public bool FireButtonPressed()
{
    return Input.GetMouseButton(0);
}
```

이렇게 해서 MouseGameController.cs 코딩은 완료되었습니다. 그럼 이것을 어떻게 사용하는지 알려드리기 전에 똑같은 IGameController 인터페이스를 구현한 또 다른 클래스인 KeyGameController.cs 를 하나 더 만든 뒤 계속 진행하도록 하겠습니다.

[동영상 예제 파일명: 006_MouseGameController_creation.mp4]

KeyGameController.cs 만들기

이번에는 MouseGameController.cs 와 마찬가지로 IGameController 인터페이스에서 파생된 또 다른 클래스인 KeyGameController.cs 를 만들어 보겠습니다. 우선 유니티에서 새로운 C# 클래스를 하나 만들고, 이름을 KeyGameController 라고 바꾸도록 하겠습니다. 파일을 열어 보시면 다음과 같이 되어 있을 것입니다.

KeyGameController.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class KeyGameController : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

이번에도 MonoBehaviour 와 관련된 항목들은 필요가 없으므로 다 지워 줍니다.

KeyGameController.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

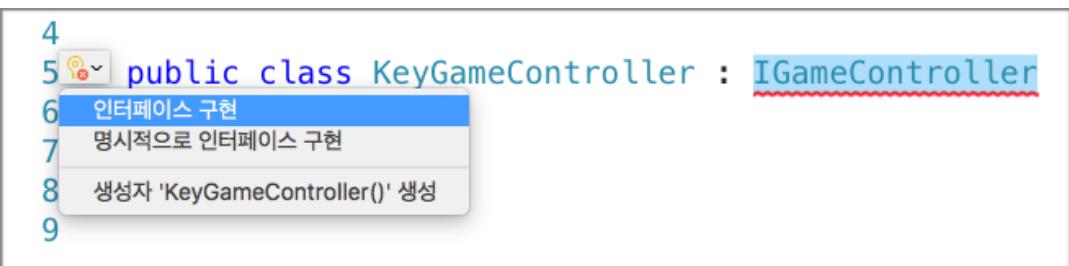
public class KeyGameController : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
}
```

```
{  
}  
  
// Update is called once per frame  
void Update()  
{  
}  
}  
}
```

그리고 나서 MonoBehaviour 가 지워진 자리에 IGameController 라고 인터페이스 이름을 입력합니다.

```
KeyGameController.cs  
public class KeyGameController : IGameController  
{  
}
```

그리고 나서 IGameController 인터페이스에 정의되어 있는 FireButtonPressed() 함수의 구현부(implementation)를 작성합니다. 앞에서 알려 드린 비주얼 스튜디오의 ‘자동으로 구현하기’ 기능을 이용하셔도 되고 수동으로 하셔도 됩니다. 저는 이번에도 자동으로 만들어 보았습니다. IGameInterface 에 마우스 커서를 갖다 대고 ‘option + 엔터키(Mac)’를 누르거나 ‘Alt + 엔터키(Windows)’를 누르시면 됩니다.



이렇게 하면 다음과 같이 FireButtonPressed() 함수의 구현부가 자동으로 만들어집니다.

```
KeyGameController.cs
public class KeyGameController : IGameController
{
    public bool FireButtonPressed()
    {
        throw new System.NotImplementedException();
    }
}
```

역시 앞에서와 마찬가지로 throw로 시작하는 자동 명령문을 삭제합니다.

```
KeyGameController.cs
public class KeyGameController : IGameController
{
    public bool FireButtonPressed()
    {
        throw new System.NotImplementedException();
    }
}
```

지금 만든 함수는 키보드의 특정한 키를 눌렀는지를 확인하는 함수입니다. 스페이스(Space) 키를 눌렀는지 여부를 리턴 값으로 전달하도록 다음과 같은 명령어를 추가해 보겠습니다.

```
KeyGameController.cs
public bool FireButtonPressed()
{
    return Input.GetKeyDown(KeyCode.Space);
}
```

이제 IGameController 인터페이스에서 파생된 두 개의 클래스, MouseGameController 와 KeyGameController 를 모두 만들었습니다. 그러면 총알

발사대인 BulletLauncher.cs 에서 이것을 사용할 수 있도록 코드를 작성해 보겠습니다.

[동영상 예제 파일명: 007_KeyGameController_creation.mp4]

총알 발사대(BulletLauncher)에서 IGameController 사용하기

이제 BulletLauncher.cs 파일을 다시 열어 보겠습니다. 현재 BulletLauncher 클래스는 다음과 같이 되어 있습니다.

```
BulletLauncher.cs
public class BulletLauncher : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

우선 Start() 함수 윗 부분에 IGameController 타입의 변수를 하나 만들겠습니다. 변수 이름은 controller 라고 하겠습니다.

```
BulletLauncher.cs
public class BulletLauncher : MonoBehaviour
{
    IGameController controller;

    // Start is called before the first frame update
    void Start()
    {

    }

    ...
}
```

이제 MouseGameController의 인스턴스를 만들고, controller 변수에 이에 대한 참조 값을 할당해야 하는데, 두 가지 방법이 있습니다. 클래스 내부에서 하는 방법과 외부에서 하는 방법입니다. 우선 클래스 내부에서 인스턴스를 만들고 변수에 할당하는 방법을 구현해 보겠습니다. 먼저 다음 예제 코드와 같이 Start() 함수에서 MouseGameController 의 인스턴스를 만든 다음, controller에 할당하겠습니다.

BulletLauncher.cs

```
public class BulletLauncher : MonoBehaviour
{
    IGameController controller;

    // Start is called before the first frame update
    void Start()
    {
        controller = new MouseGameController();
    }

    // Update is called once per frame
    void Update()
    {
    }
}
```

이런 식으로 해도 당장은 상관은 없지만, 이런 방식을 택하면 나중에 게임 콘트롤 방식을 마우스에서 키보드로 바꿔야 할 경우에는 BulletLauncher 클래스를 다시 수정해야 합니다. 예를 들어 다음과 같은 식입니다. (**다음 보라색 코드는 따라서 입력하지 마시기 바랍니다**)

BulletLauncher.cs

```
public class BulletLauncher : MonoBehaviour
{
    IGameController controller;

    // Start is called before the first frame update
```

```
void Start()
{
    controller = new MouseGameController();
    controller = new KeyGameController();
}

// Update is called once per frame
void Update()
{
}
```

이런 식으로 게임 콘트롤 방식을 변경해야 할 때마다 클래스 내부로 들어가서 기존 코드를 삭제하고 새로운 코드를 작성해야 한다면, 나중에 프로젝트 규모가 커질 경우 굉장히 번거롭고 짜증나게 될 것입니다. 따라서 이렇게 BulletLauncher 클래스 내부에서 MouseGameController 나 KeyGameController 와 같은 게임 콘트롤러 클래스의 인스턴스를 만들기 보다는 BulletLauncher 클래스 바깥에서 만들어진 인스턴스를 그냥 전달 받는 식으로 하는 것이 더 낫습니다. 그렇게 하면 나중에 게임 콘트롤 방식을 변경할 때 BulletLauncher 클래스 내부 코드는 전혀 손대지 않아도 되기 때문입니다. 그러면 어떻게 하면 되는지를 이해하기 위해 먼저 Start() 함수 안에 방금 제가 작성했던 코드를 지우고 다른 함수를 만들어 보겠습니다.

우선 기존 코드부터 지우겠습니다. 그런데 단순히 명령문만 지우는 것이 아니라 Start() 함수 자체를 모두 지우도록 하겠습니다. 당장은 Start() 함수도 필요 없기 때문입니다.

```

public class BulletLauncher : MonoBehaviour
{
    IGameController controller;

    // Start is called before the first frame update
    void Start()
    {
        controller = new MouseGameController();
    }

    // Update is called once per frame
    void Update()
    {
    }
}

```

이제 외부에서 IGameController 의 파생 클래스의 인스턴스를 주입(Injection) 받기 위한 함수, SetGameController()를 만들어 보겠습니다. 우선 함수의 형식은 다음과 같이 될 것입니다.

```

public class BulletLauncher : MonoBehaviour
{
    IGameController controller;

    public void SetGameController(IGameController controller)
    {

    }

    // Update is called once per frame
    void Update()
    {
    }
}

```

여기에서 눈여겨 보실 부분은 SetGameController() 함수의 인자로 IGameController라는 인터페이스 타입이 전달되었다는 것입니다. 이것은 IGameController 인터페이스를 구현한 어떤 클래스의 인스턴스라도 다 인자로 받아 들일 수 있다는 이야기입니다. 다시 말해, 앞에서 이 인터페이스를 구현한 MouseGameController 와 KeyGameController 클래스의 인스턴스 뿐 아니라, 미래에 추가될지도 모르는 어떤 새로운 게임 콘트롤러 클래스의 인스턴스들도 다 인자로 받아들일 수 있다는 이야기가 됩니다. (미래에 추가될 새로운 게임 콘트롤러가 IGameController 인터페이스를 구현하기만 한다면 말입니다.)

이제 SetGameController() 함수 안에 다음과 같은 코드를 작성해 넣겠습니다.

BulletLauncher.cs

```
public void SetGameController(IGameController controller)
{
    this.controller = controller;
}
```

이렇게 하면 외부에서 주입된 controller 가 내부의 private 변수인 this.controller 에 전달됩니다. 위의 예제에서는 함수의 인자로 전달된 controller 와 내부 변수 controller 의 이름이 같기 때문에 내부 변수에 this 를 붙인 것입니다.
(this.controller)

이제 Update() 함수에서 이 controller를 사용해 보겠습니다. 다음과 같이 코드를 작성합니다.

BulletLauncher.cs

```
void Update()
{
    if (controller.FireButtonPressed())
    {
        Debug.Log("Fired a bullet!");
    }
}
```

이것은 게임 컨트롤러의 FireButtonPressed() 가 true 일 경우, 콘솔창에 “Fired a bullet!” 이라는 메시지를 띄우는 것입니다. 그런데 한 가지 문제가 있습니다. Update() 의 이 코드가 실행되려면 controller 가 null 이어서는 안됩니다. 앞에서 만든 SetGameController() 를 통해서 어떤 참조값이 전달되지 않았다면 이 명령문은 ‘null 참조 에러’를 발생시킬 것입니다. 따라서 이 경우를 피해갈 수 있도록 방어용 코드를 추가하도록 하겠습니다.

BulletLauncher.cs

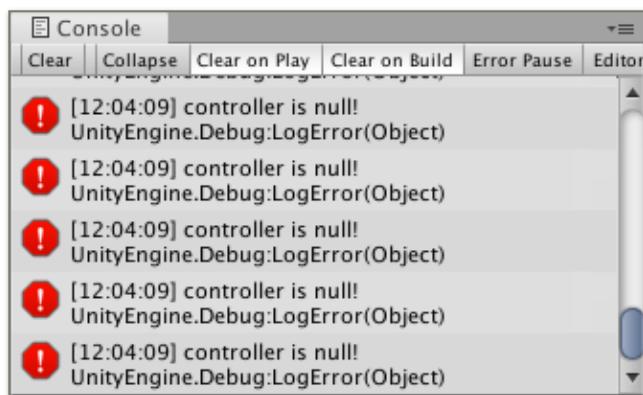
```
void Update()
{
    if (controller != null)
    {
        if (controller.FireButtonPressed())
        {
            Debug.Log("Fired a bullet!");
        }
    }
}
```

그런데 이렇게만 하고 끝나면, controller 가 null 인 경우에도 아무런 에러가 발생하지 않기 때문에 우리가 작업하는 과정에서 controller 에 문제가 있다는 것을 알아차리지

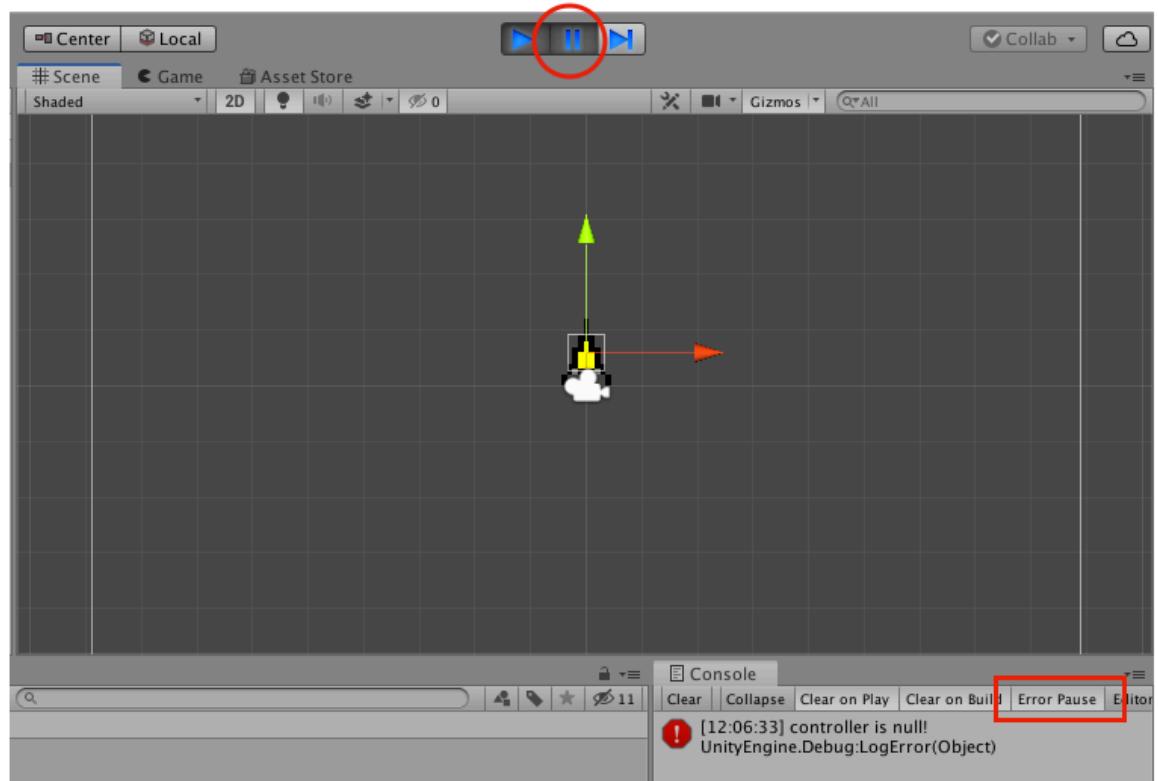
못할 가능성이 있습니다. 따라서 Debug.LogError() 를 이용하여, controller 가 null 일 경우에는 콘솔창에 경고 메시지가 표시되도록 해 보겠습니다.

```
BulletLauncher.cs  
void Update()  
{  
    if (controller != null)  
    {  
        if (controller.FireButtonPressed())  
        {  
            Debug.Log("Fired a bullet!");  
        }  
    }  
    else  
    {  
        Debug.LogError("controller is null!");  
    }  
}
```

위와 같이 하면, controller 에 아무런 참조값이 할당되지 않은 상태에서 게임을 실행할 경우 유니티 에디터의 콘솔창에 다음과 같은 에러 메시지가 뜨게 됩니다.



그리고 만약에 콘솔창 상단의 [Error Pause] 를 선택한 상태로 게임을 실행하면, 위의 여러 메시지가 표시되는 순간 유니티 에디터상에서 게임이 일시 정지되므로 문제 상황이 발생했을 때 바로 인지하고 찾아낼 수 있습니다.



[동영상 예제 파일명: 008_use_IGameController.mp4]

GameManager 클래스 만들기

이제 BulletLauncher 쪽의 코드는 모두 작성 완료되었습니다. 이제 BulletLauncher의 인스턴스를 생성하고 관리할 클래스가 필요합니다. 바로 게임 매니저(GameManager) 클래스입니다.

GameManager 클래스의 역할

지금부터 만들 게임 매니저(GameManager)는 미사일 커맨더 게임에 필요한 주요 클래스들의 인스턴스들을 생성하고 이들이 서로 커뮤니케이션을 할 수 있도록 연동해주는 역할을 하게 될 것입니다.

먼저 유니티 에디터에서 새로운 C# 스크립트 파일을 만들고, 이름을 GameManager로 바꾸겠습니다. 그리고 나서 파일을 열어 보면 다음과 같이 되어 있을 것입니다.

GameManager.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class GameManager : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
```

```
    }  
}
```

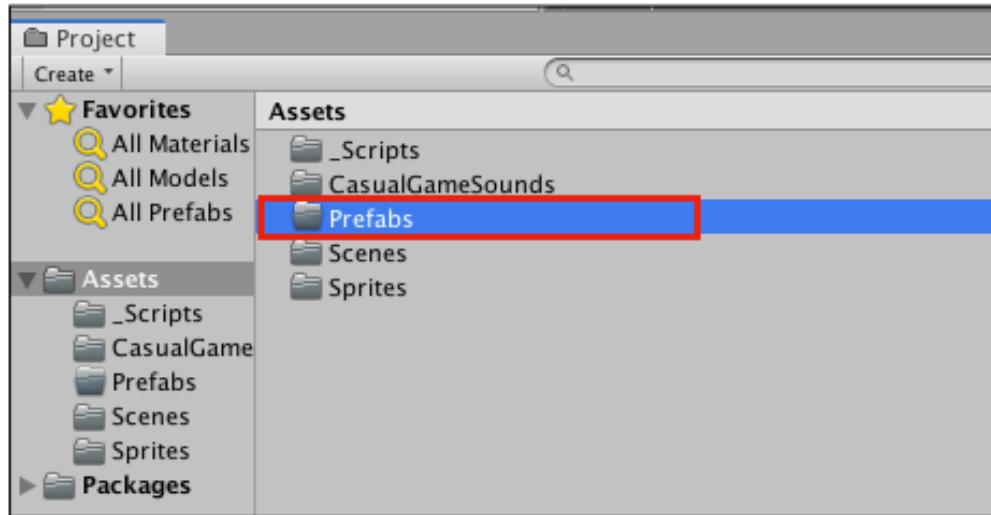
이 GameManager 클래스는 정상적인 MonoBehaviour 기반의 유니티 클래스입니다. 여기에서 지금부터 할 일은 다음과 같습니다.

1. 총알 발사대(BulletLauncher) 프리팹으로부터 인스턴스를 생성해서 적당한 위치에 놓는다.
2. MouseGameController 나 KeyGameController의 인스턴스를 생성한 다음에, 이를 BulletLauncher에 주입한다.

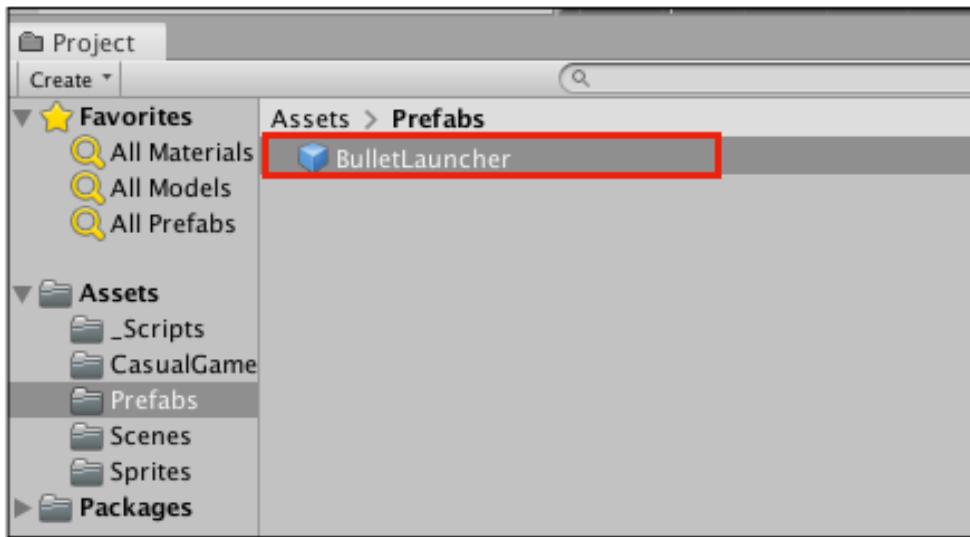
그럼 코딩을 시작하기 전에, 먼저 유니티 에디터로 가서 우리가 지금까지 만들었던 BulletLauncher 게임 오브젝트를 프리팹으로 저장하는 것부터 하겠습니다.

BulletLauncher 프리팹으로 저장하기

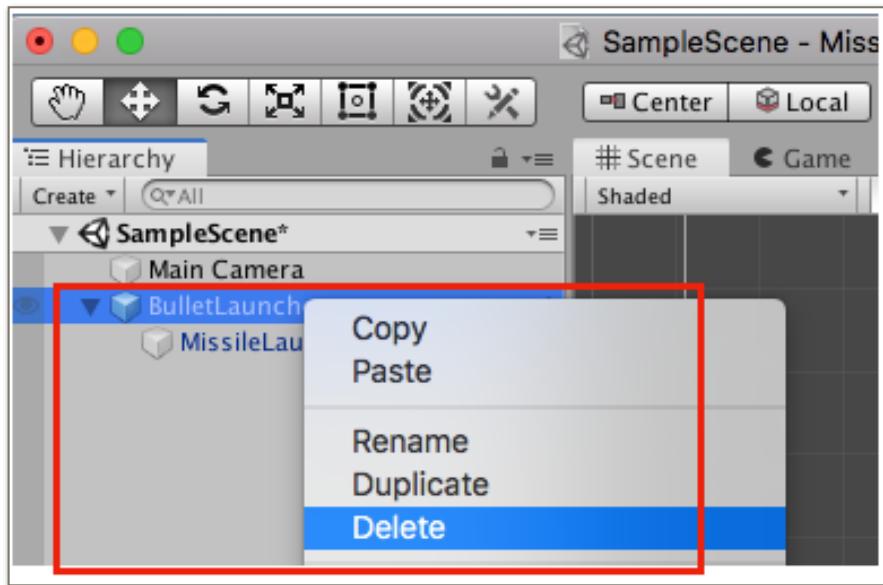
현재 유니티 에디터의 계층구조(Hierarchy)를 보시면 우리가 만들어 놓은 BulletLauncher가 있습니다. 이것을 프리팹으로 만들기 위해, 먼저 하단의 Assets 폴더 아래에 Prefabs라는 이름의 폴더를 만들겠습니다.



다음으로는 계층구조(Hierarchy)에 있는 BulletLauncher 게임 오브젝트를 Prefabs 폴더 안으로 드래그해서 갖다 놓습니다. 그러면 BulletLauncher가 프리팹으로 변환되면서 계층구조에서 이름이 파란색으로 표시될 것입니다.



이제 계층에서 BulletLauncher 게임 오브젝트를 삭제하겠습니다.



[동영상 예제 파일명: 009_bulletLauncher_prefab.mp4]

GameManager.cs 코드 작성하기

이제 GameManager.cs 의 코드를 작성하겠습니다. 먼저 방금 만들었던 총알 발사대 (BulletLauncher)의 프리팹을 참조할 변수가 필요합니다. 아래와 같이 코드를 작성합니다.

```
GameManager.cs
```

```
public class GameManager : MonoBehaviour
{
    [SerializeField]
    BulletLauncher launcherPrefab;

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

이 부분을 public 으로 선언 안하고 [SerializeField]를 이용하는 이유는, 최대한 public 변수 사용을 피하기 위해서입니다. launcherPrefab을 public 으로 선언할 경우, 나중에 누군가가 어디에서 이 변수에 접근해서 값을 바꿔 버려도 우리는 이것을 추적해내기가 쉽지 않습니다. 따라서 꼭 필요한 경우가 아니라면 최대한 public 사용을 자제하는 것이 안전한 습관입니다.

이 경우에 public 대신 [SerializeField]를 사용하면, 유니티 에디터의 인스펙터(Inspector)를 이용하여 변수에 참조 대상을 ‘드래그 앤 드롭’으로 할당하는 것이 가능한 반면, 다른 클래스에서는 이 변수에 직접적으로 접근할 수가 없으므로 안전합니다. 여러분들도 앞으로는 가급적이면 public 대신 [SerializeField]를 사용하는 습관을 들이시기 바랍니다.

다음에 할 작업은 방금 만든 launcherPrefab 으로부터 만들어진 인스턴스를 참조할 변수를 만드는 것입니다. 아래와 같이 코드를 작성해 넣겠습니다.

```
GameManager.cs
```

```
public class GameManager : MonoBehaviour
{
    [SerializeField]
    BulletLauncher launcherPrefab;
    BulletLauncher launcher;

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

}
}
```

이제 Start() 함수에서 총알 발사대의 인스턴스를 만든 뒤, 이를 launcher 에 할당하겠습니다. 여기에서는 생성된 총알 발사대의 위치(position)를 따로 설정하지 않았기 때

문에, 생성된 총알 발사대는 (0,0,0) 의 기본 위치에 생성될 것입니다. 지금은 위치가 중요한 것이 아니므로, 게임 화면에 총알 발사대가 표시되기만 하면 됩니다.

GameManager.cs

```
void Start()
{
    launcher = Instantiate(launcherPrefab);
}
```

다음으로는 launcher 에 만들어 놓은 함수인 SetGameController() 를 이용하여 MouseGameController 클래스의 인스턴스를 생성해서 주입해 보겠습니다. 함수의 인자로 new MouseGameController() 를 다음 예제처럼 전달하면 됩니다.

GameManager.cs

```
void Start()
{
    launcher = Instantiate(launcherPrefab);
    launcher.SetGameController(new MouseGameController());
}
```

여기에서 눈여겨 보실 부분은 launcher의 SetGameController() 안에 new 를 이용하여 MouseGameController 클래스의 인스턴스를 직접 만들어 넣었다는 것입니다. 원래 SetGameController() 함수는 인자로 IGameController 인터페이스 파생 클래스의 인스턴스를 전달 받도록 되어 있습니다. 예를 들어, BulletLauncher.cs 에서 이 함수는 다음과 같은 식으로 구현되어 있었습니다.

BulletLauncher.cs

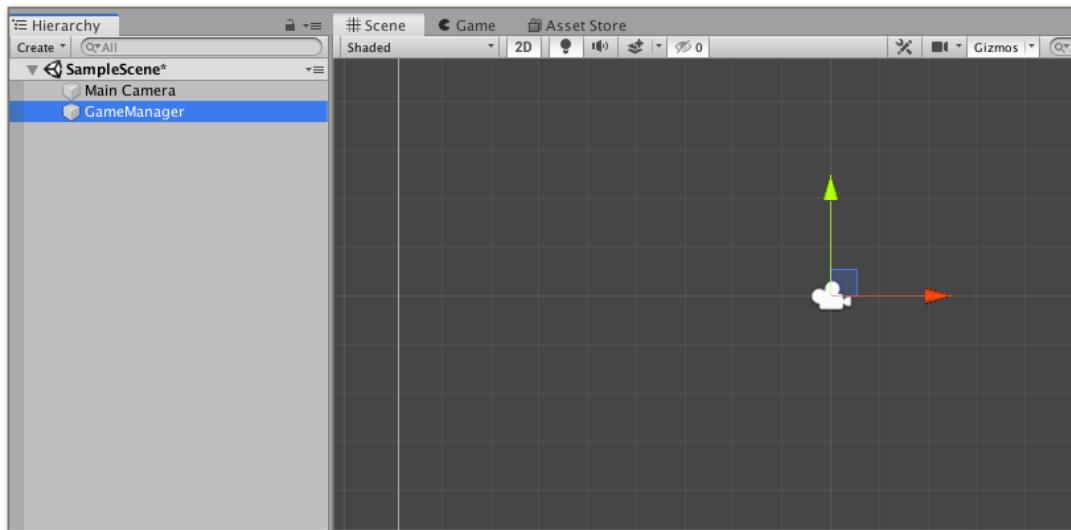
```
public void SetGameController(IGameController controller)
{
    this.controller = controller;
}
```

MouseGameController 나 KeyGameController 모두 IGameController 인터페이스로부터 파생된 클래스이므로, 우리는 SetGameController() 함수에 두 개의 클래스 중 어느 것을 넣어도 문제가 없습니다. 이 개념을 처음 보신 분은 이해가 잘 안 가실 수 있겠지만, 일단 책을 쭉 읽어 보시다가 나중에 이 부분을 몇번 다시 반복해서 읽으시면 어느 순간 이해되는 것을 느끼실 수 있을 것입니다.

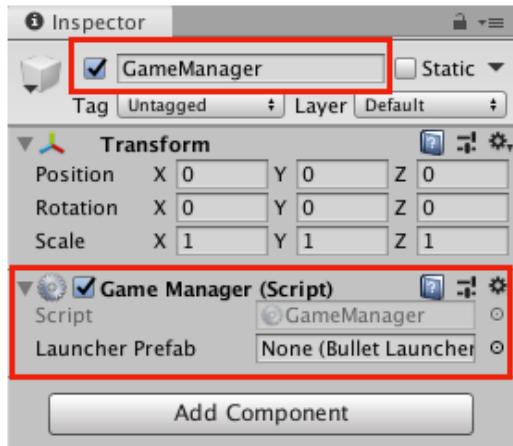
일단 GameManager.cs 의 코딩은 이것으로 끝났습니다. 스크립트를 저장하신 다음에 유니티 에디터로 가셔서 새로운 게임 오브젝트를 만들고 여기에 GameManager.cs 스크립트를 연동하도록 하겠습니다.

씬(Scene)에 게임 매니저를 만들자

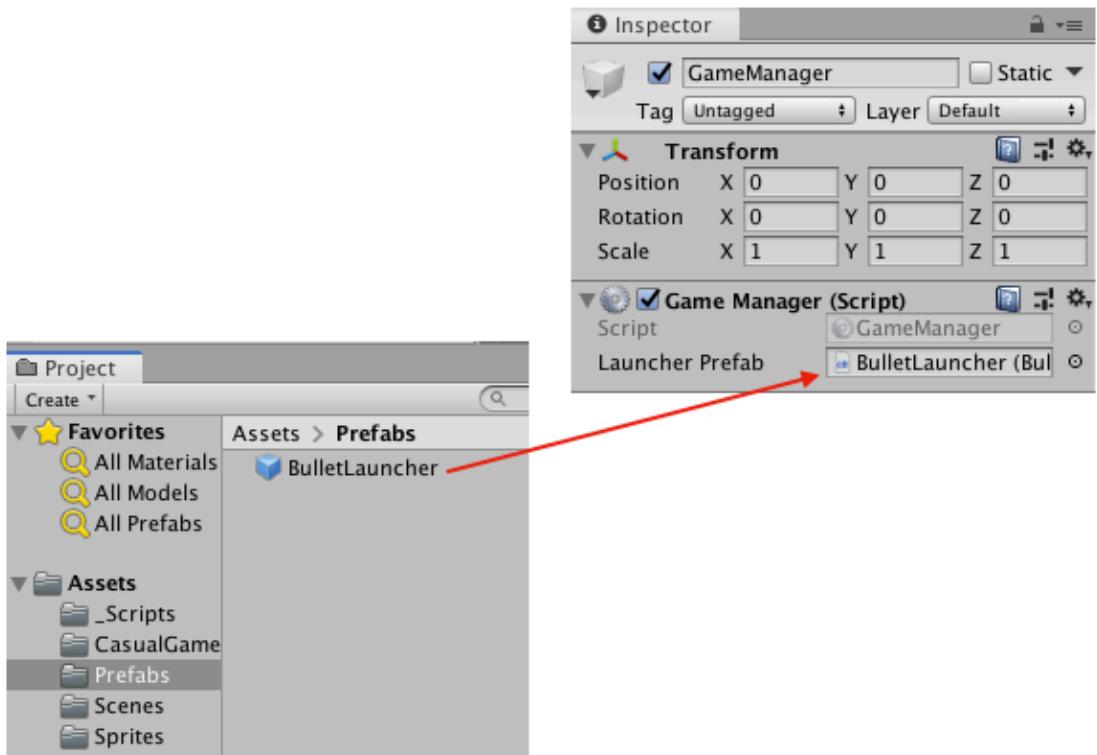
우선 유니티 에디터의 계층구조(Hierarchy)에 빈 게임 오브젝트를 하나 만들고 이름을 GameManager라고 짓도록 하겠습니다. 이 때 주의하실 점은 이 게임 오브젝트가 화면 중간에 표시되어야 한다는 점입니다. 총알 발사대(BulletLauncher)의 인스턴스가 이곳에 위치할 것이기 때문입니다. (Transform 의 position 값이 0,0,0 인지 확인하시기 바랍니다)



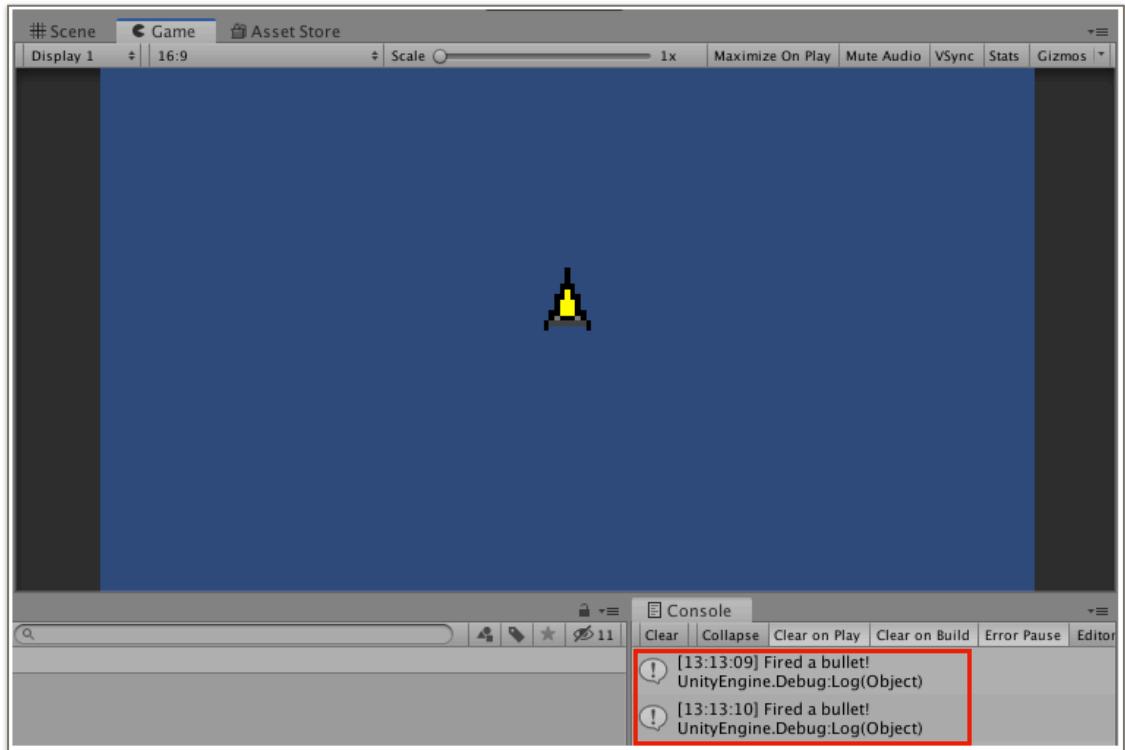
다음으로는 아까 만들었던 GameManager.cs 스크립트를 연결하는 것입니다. 아래와 같이 GameManager 게임 오브젝트에 GameManager.cs 스크립트 파일을 연결하면 됩니다.



다음으로는 [Prefabs] 폴더에 있는 BulletLauncher 프리팹을 드래그해서 GameManager 스크립트에 있는 LauncherPrefab에 연결해야 합니다. 다음과 같이 하시면 됩니다.



이제 준비가 모두 끝났습니다. 유니티 에디터에서 플레이 버튼을 눌러서 게임을 실행하신 뒤, 마우스로 화면 아무 곳이나 클릭하면, 그림과 같이 콘솔창에 “Fired a bullet!”이라는 메시지가 표시되는 것을 확인하실 수 있습니다.



[동영상 예제 파일명: 010_create_GamerManager.mp4]

게임 콘트롤러를 마우스에서 키보드로 바꾸기

이제 마우스 입력을 통한 게임 콘트롤이 잘 작동하는 것을 확인하였으니, 다음으로 해야 할 것은 게임 콘트롤 방식을 마우스가 아닌 키보드로 바꾸는 것입니다. 과거의 경우, 게임 콘트롤 방식을 바꾸기 위해서는 총알 발사대의 BulletLauncher.cs 클래스를 수정해야 했습니다. 하지만 우리가 지금 만든 방식을 사용하게 되면 BulletLauncher 클래스는 전혀 건드릴 필요가 없습니다. 이 클래스 바깥에 있는 GameManager라는 외부 클래스에서 필요한 게임 콘트롤러를 교체해 주면 되기 때문입니다.

우선 GameManager.cs 로 가서, launcher.SetGameController() 의 인자 전달 코드를 다음과 같이 삭제합니다.

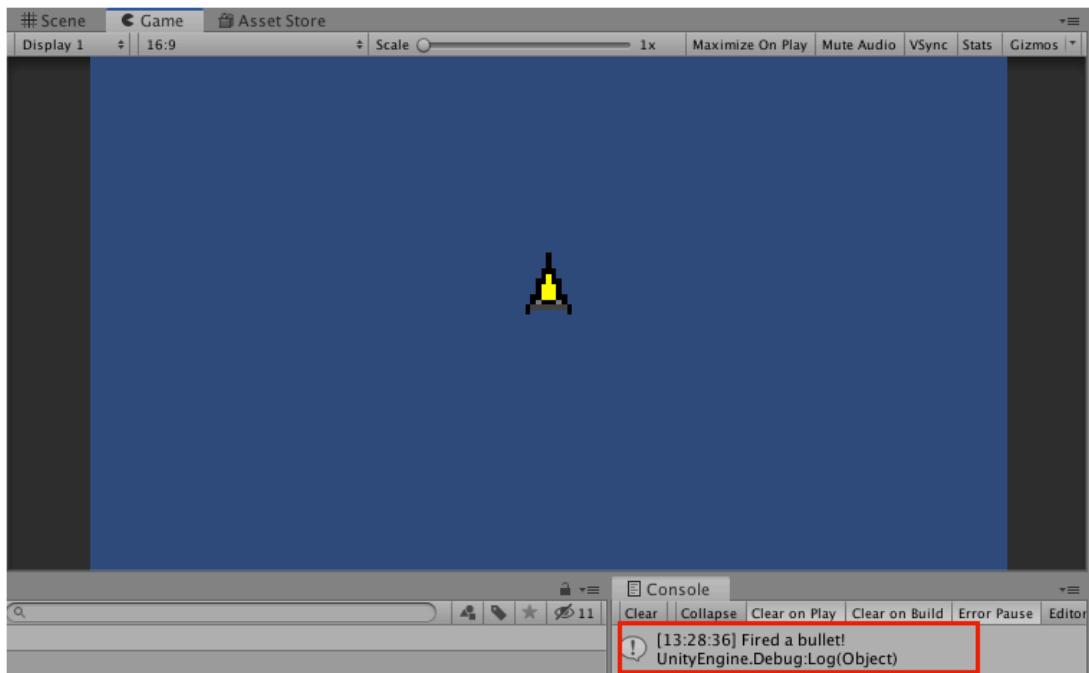
```
GameManager.cs
void Start()
{
    launcher = Instantiate(launcherPrefab);
    launcher.SetGameController(new MouseGameController());
}
```

그리고 이 부분을 아래와 같이 바꿉니다.

```
GameManager.cs
void Start()
{
    launcher = Instantiate(launcherPrefab);
    launcher.SetGameController(new KeyGameController());
}
```

보시는 것처럼, 다른 것은 그대로 두고 오직 전달되는 게임 콘트롤러만 기존의 MouseGameController 의 인스턴스에서 KeyGameController 의 인스턴스로 바꾸었

습니다. 그럼 이 상태에서 저장한 뒤, 유니티 에디터로 가서 게임을 플레이해 보겠습니다. 먼저 아까와 마찬가지로 마우스 클릭을 해 보고, 그 다음에 키보드의 스페이스바를 눌러 보겠습니다.



테스트를 해 보시면, 마우스를 클릭할 때는 아무 반응이 없다가, 키보드의 스페이스바를 누르면 그 때 콘솔창에 메시지가 뜨는 것을 확인하실 수 있습니다. 게임 컨트롤러가 기존의 MouseGameController에서 KeyGameController로 바뀌었기 때문입니다.

[동영상 예제 파일명: 011_change_game_controller_keyboard.mp4]

디펜던시 인젝션(의존성 주입) 개념의 이해

앞에서는 IGameController 인터페이스를 이용하여, 서로 다른 게임 콘트롤러 클래스의 인스턴스를 손쉽게 바꿔가며 사용할 수 있도록 코드를 작성하였습니다. 이 때 사용한 것이 디펜던시 인젝션(Dependency Injection; 의존성 주입)의 개념입니다. 우선 디펜던시(dependency)가 무엇인지 살펴 보도록 하겠습니다.

디펜던시(dependency)란 무엇인가?

여기서 두 개의 클래스 A, B가 있다고 해 보겠습니다. 만약 클래스 A가 어떤 일을 하기 위해 클래스 B의 인스턴스를 필요로 한다면, 우리는 B를 A의 디펜던시(dependency)라고 말합니다. 디펜던시라고 영어를 사용하니까 개념이 즉각적으로 와닿지 않는 분들도 계실텐데요, 디펜던시란 ‘의존’이라는 의미를 가지고 있습니다. 앞의 예에서는 클래스 A가 작동하기 위해서는 클래스 B의 인스턴스가 반드시 필요하기 때문에, ‘A가 B에 의존하고 있다’고 이야기할 수 있는 것입니다.

앞에서 우리가 작성한 BulletLauncher 클래스와 MouseGameController 또는 KeyGameController의 관계를 살펴 보겠습니다. BulletLauncher가 제대로 자기 역할을 하기 위해서는 반드시 게임 콘트롤러가 필요합니다. 게임 콘트롤러가 마우스 기반의 콘트롤러(MouseGameController)인지, 아니면 키보드 기반의 콘트롤러(KeyGameController)인지는 몰라도, 어쨌건 IGameController 인터페이스를 구현한

클래스의 인스턴스가 필요하다는 것은 확실합니다. 따라서 BulletLauncher 클래스는 IGameController의 구현 클래스에 의존하고 있다고 말할 수 있는 것입니다.

그런데 이처럼 하나의 클래스가 다른 클래스에 의존하고 있을 때, 이 의존성(dependency)을 어떻게 구현할 것인가가 문제가 됩니다. 예를 들어 유니티는 기본적으로 게임 오브젝트(Entity)에 다양한 컴포넌트(Component)들을 붙여서 사용하도록 설계되어 있습니다. 하나의 게임 오브젝트에 다양한 Component 들이 붙어 있을 때, 이들 각각의 컴포넌트들을 GetComponent<T>() 함수를 이용하여 참조하는 경우를 많이 보셨을 것입니다. 게임 오브젝트에 붙어 있는 각각의 컴포넌트들은 모두 MonoBehaviour 클래스의 인스턴스들이기 때문에 이들 인스턴스를 사용한다는 이야기는 곧 ‘이들 인스턴스들에 의존한다’는 것과 같은 의미라고 볼 수 있습니다.

예를 들어 아래의 예제에서 Arrow 가 움직이기 위해서는 리지드바디(Rigidbody) 클래스의 인스턴스가 필요합니다. 따라서 Arrow 라는 클래스는 Rigidbody 라는 클래스에 의존한다고 이야기할 수가 있을 것입니다. (다음의 코드는 따라하실 필요 없습니다. 개념 이해를 위해 그냥 보시기만 하면 됩니다.)

```
public class Arrow : MonoBehaviour
{
    Rigidbody body;

    void Awake()
    {
        body = GetComponent<Rigidbody>();
    }
}
```

```
void Shoot()
{
    body.AddForce(transform.forward * 100.0f);
}
```

위의 예제 중 Awake() 함수를 보겠습니다. 이곳에서는

GetComponent<Rigidbody>() 명령어를 이용하여 Rigidbody 클래스의 인스턴스를 참조하고 있습니다. 그리고 하단에 있는 Shoot()이라는 함수에서 이 참조 변수를 이용, 어떤 행동(화살 쏘기)을 구현하고 있습니다.

유니티에서 권장하는 디펜던시 참조 방법은 바로 이것입니다. 유니티 엔진은 컴포넌트 중심으로 설계되어 있기 때문에 위와 같은 방식을 사용해서 클래스의 디펜던시를 처리하는 것이 기본입니다. 그런데 문제가 하나 있습니다. 유니티에서 권장하는 이 방법은, 다수의 클래스 인스턴스들이 동일한 하나의 게임 오브젝트에 연결되어 있을 때는 아주 훌륭하게 작동합니다 하지만 ‘서로 다른’ 게임 오브젝트들에 연결되어 있는 클래스 인스턴스끼리 서로 참조하거나 커뮤니케이션해야 할 경우에는 그렇지 않습니다.

또한 MonoBehaviour의 파생 클래스가 아닌 보통의 C# 클래스에서 (new 키워드를 이용해서) 디펜던시가 만들어졌을 경우에도 마찬가지입니다. 이 경우에는 GetComponent<T>() 명령어를 이용한 참조가 불가능하기 때문입니다.

이런 경우, `GameObject.Find("오브젝트 이름")` 이나 `GameObject.FindWithTag("태그 이름")`, 또는 `Object.FindObjectOfType(타입)` 과 같은 명령어를 사용할 수 있습니다. 혹은 싱글톤(Singleton)과 같이 언제 어디서나 바로 접근할 수 있는 글로벌 참조 방식을 사용할 수도 있습니다. (싱글톤이 무엇이고 어떤 식으로 사용하는지에 대해서는 이 책의 맨 마지막 부분의 오디오 매니저 구현 항목에서 배우실 수 있습니다.) 이런 방법을 사용할 경우 일단은 편리하기는 하지만, 프로젝트 규모가 커지고 개발에 참여한 프로그래머의 수가 늘어날 수록 점점 복잡한 문제가 생기게 됩니다.

예를 들어, `GameObject.Find("이름")`을 너무 많이 사용하게 되면, 나중에 누군가가 실수로 게임 오브젝트의 이름을 바꿔버렸을 때 문제가 생깁니다. 또한 싱글톤(Singleton)과 같이 어떤 클래스에서든 바로 접근할 수 있는 방법을 사용할 경우, 에러가 발생했을 때 원인을 찾아내는 것이 무척 어렵습니다. 자칫하면 이 싱글톤을 참조하는 프로젝트 안의 모든 코드들을 일일이 다 추적해야 할 수도 있기 때문입니다.

이런 문제들에 대해 자세히 설명하자면 끝이 없기 때문에 일단 여기에서 더 이상 관련된 이야기를 진행하지는 않겠습니다. 다만 확실히 말씀드리고자 하는 것은 디펜던시(의존성) 문제를 잘 처리하지 못할 경우, 프로젝트 도중에 발생한 변경 사항에 대처하기가 무척 어렵다는 것입니다. 이 챕터에서 소개해 드릴 '디펜던시 인젝션(Dependency Injection)' 또한 이러한 문제를 해결하기 위해 고안된 방법 중 하나입니다.

디펜던시 인젝션(Dependency Injection)의 기본 원리

앞에서 디펜던시(Dependency)가 무엇인지에 대해 알아 보았습니다. 그럼 이번에는 다음으로는 인젝션(Injection)이라는 단어를 살펴 보겠습니다. 영어사전에서 injection 이라는 단어를 찾아 보면 ‘주입’이나 주사’라는 의미입니다. 따라서 디펜던시 인젝션(dependency injection)이란 디펜던시를 클래스 외부에서 내부로, 마치 주사를 놓듯 주입해 넣는다는 의미라고 이해하시면 됩니다.

이와 관련하여, 우리가 앞에서 만들었던 BulletLauncher.cs 클래스를 보시면, IGameController 타입의 디펜던시를 가지고 있는 것을 확인할 수 있습니다. (아래 코드 파란색 부분 참조)

```
BulletLauncher.cs
public class BulletLauncher : MonoBehaviour
{
    IGameController controller;

    public void SetGameController(IGameController controller)
    {
        this.controller = controller;
    }

    ...
}
```

여기에서 controller 를 사용하기 위해서는 IGameController 를 구현한 클래스의 인스턴스를 참조값으로 전달해야 합니다. 이 때, 위의 코드에서는 아래와 같이 SetGameController() 함수를 이용해서 외부에서 주입(injection)하는 방법을 택하고 있습니다.

```
public void SetGameController(IGameController controller)
{
    this.controller = controller;
}
```

그런데 이 방법에는 한 가지 문제가 있습니다. SetGameController() 함수를 이용해서 디펜던시를 주입하는 행위가 강제로 진행되는 것이 아니기 때문에, 프로그래머가 코딩을 하다가 이 함수 호출을 깨먹을 수 있습니다. 그 경우, controller 변수를 사용하는 순간 null 참조로 인한 에러가 발생하게 됩니다. Update() 함수에서 null 을 체크하기 위한 조건문을 만들어서, 방어적으로 코딩을 한 이유는 바로 그 때문입니다.(아래 코드의 파란색 부분 다시 참조)

```
void Update()
{
    if (controller != null)
    {
        if (controller.FireButtonPressed())
        {
            Debug.Log("Fired a bullet!");
        }
    }
    else
    {
        Debug.LogError("controller is null!");
    }
}
```

사실 디펜던시 인젝션의 정석은 SetGameController() 와 같은 별도의 함수에서 디펜던시를 주입하는 것이 아니라 클래스의 생성자(constructor)에서 강제로 주입하는 방법을 사용하는 것입니다. 생성자는 클래스의 인스턴스가 만들어지는 순간 무조건 자동 실행되기 때문에 디펜던시 주입을 빼먹을 수가 없습니다. 그 경우 바로 에러가 발

생해서 우리가 즉시 알 수 있기 때문입니다. 예를 들어 위의 BulletLauncher 클래스가 생성자를 가질 수 있다면, GameManager 클래스에서는 우리가 사용한 아래의 방법 대신

```
void Start()
{
    launcher = Instantiate(launcherPrefab);
    launcher.SetGameController(new KeyGameController());
}
```

다음과 같은 식으로 디펜던시를 주입했을 것입니다. (아래 예제는 실제로는 작동하지 않으니까 따라 하지 마시고 그냥 참고만 하시기 바랍니다)

```
void Start()
{
    launcher = new BulletLauncher(new KeyGameController());
}
```

그리고 위와 같은 코딩이 가능하려면 BulletLauncher.cs에서 다음과 같은 생성자 코드를 작성했어야 합니다. (아래 코드 역시 실제로는 불가능합니다. BulletLauncher는 유니티의 MonoBehaviour 파생 클래스이기 때문에 생성자를 만들 수 없습니다. 따라서 코딩 하지 마시고 그냥 참고로 보기만 하십시오)

```
public class BulletLauncher : MonoBehaviour
{
    IGameController controller;

    public BulletLauncher(IGameController controller)
    {
        this.controller = controller;
    }

    ...
}
```

이러한 방법을 가리켜, ‘생성자(constructor)를 통한 디펜던시 주입’이라고 합니다.

그리고 이런 방법이 안전한 이유는 생성자에서 controller 가 null 인지를 즉시 체크해서, 만약 null 이라면 바로 예외를 발생시키거나 경고를 날릴 수 있기 때문입니다. 아래 예제 코드가 그 예입니다. (역시 코드를 따라 하지 마시고 참고하시기 바랍니다. 유니티에서는 이 방법을 사용할 수 없습니다)

```
public BulletLauncher(IGameController controller)
{
    this.controller = controller;

    if (controller == null)
    {
        throw new System.Exception("controller is null!");
    }
}
```

위와 같은 식으로 코드를 작성했다면, GameManager 클래스에서 아래와 같은 방식으로 BulletLauncher의 인스턴스를 생성하려면 반드시 new MouseGameController()나 new KeyGameController() 를 생성자의 인자로 전달해야만 합니다. (만약 이 과정을 빼 먹었다면 인자의 값이 null 이 될 것이기 때문에, 생성자에서 그것을 즉시 간파해서 경고 메시지를 날릴 것입니다.)

```
void Start()
{
    launcher = new BulletLauncher(new KeyGameController());
}
```

이처럼 디펜던시를 외부에서 주입하는 경우에는 생성자(constructor)를 통해서 주입하는 것이 가장 보편적이고 안전한 방법입니다. 하지만 유니티의 MonoBehaviour 파생 클래스에서는 불행히도 생성자(constructor)를 사용할 수가 없습니다. 바로 이러한

이유 때문에 저는 아래의 원래 코드처럼 SetGameController() 함수를 이용해서 디펜던시를 주입했던 것입니다. 이러한 방법을 “함수(메소드)에 의한 디펜던시 주입”이라고 말합니다.

```
void Start()
{
    launcher = Instantiate(launcherPrefab);
    launcher.SetGameController(new KeyGameController());
}
```

(참고로 이 챕터의 설명이 너무 어려워서 이해가 안되는 분들은 너무 신경 쓰지 마시고 다음으로 넘어가시기 바랍니다. 이 챕터에서 초록색으로 설명한 코드들은 실제 이 책의 예제에서는 사용하지 않는 단순한 이론 설명일 뿐입니다. 그냥 진행하시고, 제 코드 예제나 동영상 예제를 참고해서 코딩을 여러번 해 보시다 보면 내용을 자연스럽게 이해하시게 될 것입니다. 실제 코딩을 해 보는 것보다 더 좋은 공부법은 없으니까요.)

디펜던시 인젝션(Dependency Injection)이 꼭 필요한가?

유니티로 게임을 개발할 때, 디펜던시 인젝션을 꼭 사용해야 하는 것은 아닙니다. 앞에서도 설명 드린 것처럼, 유니티는 기본적으로 컴포넌트(Component) 기반으로 설계되었기 때문에, `GetComponent<T>()` 명령어를 이용해서 디펜던시를 사용하는 것이 일반적인 방법입니다. 따라서 MonoBehaviour 클래스에서 파생된 유니티 클래스들을 사용할 경우에는 유니티에서 권장하는 방식을 사용하시면 됩니다..

하지만 앞에서 제가 예로 들었던 것처럼 MonoBehaviour 의 상속 클래스를 사용하지 않는 경우나 하나의 인터페이스로부터 파생된 다수의 클래스 인스턴스들을 손쉽게 교체해 가면서 사용하고자 할 때, 또는 가급적 느슨한 커플링을 구현하고자 하는 경우, 디펜던시 인젝션은 권장할만한 방법으로 최근 유니티 개발자 사이에서 각광을 받고 있습니다.

디펜던시 인젝션은 다양한 디자인 패턴(Design Pattern)들과 연관되어 있는 방법론으로, 깊이 들어가기로 작정하면 별도의 책이나 강의들을 찾아가면서 공부하셔야 할 것입니다. 하지만 이 책의 목적은 디펜던시 인젝션 자체에 대해 깊이 알기 위한 것이 아니므로 더 이상 이 주제에 대해 논의하지는 않겠습니다. 다만 느슨한 커플링과 좀 더 유연하고 확장성 있는 유니티 프로그래밍을 위해서는 디펜던시 인젝션의 개념과 기본적인 사용법 정도는 알고 계시는 것이 중요하다는 점을 강조드리고 싶습니다.

다음에는 디펜던시 인젝션이 아닌 이벤트(Event)를 이용하여, 어떤 식으로
IGameController 의 구현 클래스들이 총알 발사대(bulletLauncher)와 커뮤니케이션
할 수 있는지에 대해 살펴 보도록 하겠습니다. 미사일 커맨더 게임은 이 방법을 주로
사용해서 만들어 나갈 예정입니다.

이벤트(Event)를 이용한 IGameController 사용법

예제 코드 상태 확인하기

우선 이벤트 사용법에 대해 이야기를 시작하기 전에, 혹시라도 앞에서 디펜던시 인젝션을 설명하느라 제가 예시로 보여 드렸던 코드(생성자를 이용하는 방법)를 그대로 따라서 타이핑해 보신 분들이 있다면 원래 상태로 되돌려 놓으시라고 말씀 드리고 싶습니다. 이번 챕터를 시작하는 시점에서 GameManager.cs 클래스는 다음과 같이 되어 있어야 합니다.

GameManager.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class GameManager : MonoBehaviour
{
    [SerializeField]
    BulletLauncher launcherPrefab;
    BulletLauncher launcher;

    // Start is called before the first frame update
    void Start()
    {
        launcher = Instantiate(launcherPrefab);
        launcher.SetGameController(new KeyGameController());
    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

또한 BulletLauncher.cs 는 다음과 같이 되어 있어야 합니다.

BulletLauncher.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BulletLauncher : MonoBehaviour
{
    IGameController controller;

    public void SetGameController(IGameController controller)
    {
        this.controller = controller;
    }

    // Update is called once per frame
    void Update()
    {
        if (controller != null)
        {
            if (controller.FireButtonPressed())
            {
                Debug.Log("Fired a bullet!");
            }
        }
        else
        {
            Debug.LogError("controller is null!");
        }
    }
}
```

따라서 만약 이렇게 되어 있지 않다면, 위의 코드를 참고하셔서 똑같이 해 놓으신 상태로 책을 읽으시기 바랍니다. 그럼 계속 진행하도록 하겠습니다.

이벤트(event)를 이용한 커뮤니케이션

앞의 예에서는 디펜던시 인젝션을 이용, 게임 콘트롤 방식을 마우스에서 키보드로 쉽게 교체하는 것을 보여 드렸습니다. 그런데 게임 기획이 또 바뀌어서, XBOX 게임 컨트롤러 또는 오culus 리프트의 게임 컨트롤러를 사용해야 하는 경우가 발생할 경우에는 어떻게 해야 할까요?

이 때도 BulletLauncher 클래스를 전혀 손대지 않고 새로운 콘트롤 방식을 쉽게 추가할 수 있습니다. 다음과 같이 하시면 됩니다.

1. IGameController 인터페이스를 구현한 XBoxGameController 클래스나 OculusGameController 클래스를 작성한다.
2. bulletLauncher.SetGameController() 함수 안에 new 키워드를 이용하여 인스턴스를 생성, 주입한다.

이와 같이, 어떤 방식의 새로운 게임 컨트롤 방식이 요구된다고 해도 기존의 클래스에 대한 수정 없이 기능을 구현할 수 있는 것입니다.

```
launcher.SetGameController(new XBoxGameController());
```

그런데 이 외에도 다른 방법이 하나 더 있습니다. 바로 이벤트(Event)를 사용하는 것입니다. 이 방법을 이용하면 디펜던시 자체에 대해 전혀 신경 쓰지 않고 총알 발사대와 게임 콘트롤러가 서로 커뮤니케이션을 할 수 있습니다. 다시 말해서 총알 발사대와 게임 콘트롤러가 서로의 존재에 대해 전혀 알지 못하는 가운데에도 필요한 정보를 주고 받을 수 있습니다. 직접 코드를 작성하면서 설명드리도록 하겠습니다

우선 이벤트를 방식으로 코딩을 하기 전에 기존에 작성한 코드들을 지우도록 하겠습니다. 먼저 BulletLauncher.cs로 가셔서 다음과 같이 기존 코드들을 삭제합니다.

```
BulletLauncher.cs
public class BulletLauncher : MonoBehaviour
{
    IGameController controller;

    public void SetGameController(IGameController controller)
    {
        this.controller = controller;
    }

    // Update is called once per frame
    void Update()
    {
        if(controller != null)
        {
            if(controller.FireButtonPressed())
            {
                Debug.Log("Fired a bullet!");
            }
        }
        else
        {
            Debug.LogError("controller is null!");
        }
    }
}
```

이렇게 기존 코드들을 다 지우고 나면, BulletLauncher.cs 에는 다음과 같이 아무 내용이 없는 Update() 함수 하나만 남아 있게 될 것입니다.

```
BulletLauncher.cs  
public class BulletLauncher : MonoBehaviour  
{  
    // Update is called once per frame  
    void Update()  
    {  
    }  
}
```

위와 같이 코드를 정리하는 과정에서 SetGameController() 함수를 지워 버렸기 때문에, 게임 매니저(GameManager)에서 이 함수를 사용하는 부분도 필요가 없을 것입니다. GameManager.cs 로 이동해서 관련된 코드를 지우도록 하겠습니다.

```
GameManager.cs  
public class GameManager : MonoBehaviour  
{  
    [SerializeField]  
    BulletLauncher launcherPrefab;  
    BulletLauncher launcher;  
  
    // Start is called before the first frame update  
    void Start()  
    {  
        launcher = Instantiate(launcherPrefab);  
        launcher.SetGameController(new KeyGameController());  
    }  
  
    // Update is called once per frame  
    void Update()  
    {  
    }  
}
```

이렇게 기존의 코드를 삭제함으로써, 이제 BulletLauncher 와 IGameController 의 직접적인 연관 관계는 완전히 끊어지게 됩니다. BulletLauncher 입장에서는 이 세상에 IGameController 나 IGameController 파생 클래스가 존재한다는 사실 자체를 알 수가 없습니다. 그러면 이제 어떻게 BulletLauncher 가 플레이어의 입력을 알아채고 이에 맞는 행동을 하도록 만들 수 있을까요? 이벤트를 이용해서 이를 구현해 보겠습니다.

이벤트를 사용한다는 것의 의미는 마치 라디오 방송국에서 전파를 날리는 것과 마찬 가지입니다. 라디오 방송국에서 방송 신호를 내 보낼 때, 누가 그 방송을 듣는지에 대해서는 고려하지 않습니다. 그냥 신호를 내 보낼 뿐입니다. 만약 라디오 수신기를 가지고 있는 사람이 있다면 그 방송에 채널을 맞추고 있다가 방송이 들어 오면 그냥 듣기만 하면 됩니다. 송신자 입장에서는 그냥 신호를 발사 할 뿐이고, 수신자 입장에서는 그 채널을 통해 들어오는 신호가 있다면 수신해서 들을 뿐입니다. 누가 신호를 발사했는지에 대해 수신자는 상관하지 않습니다. 앞으로 우리도 이와 비슷한 원리에 따라 코딩을 하려고 합니다.

우선 플레이어의 게임 컨트롤과 관련된 이벤트란 “총알 발사 버튼이 눌려졌다”와 같은 신호입니다. 그리고 이러한 이벤트를 발생시키는 쪽은 물론 MouseGameController나 KeyGameController와 같은 게임 컨트롤러 쪽일 것입니다. 따라서 먼저 이를 두 개의 클래스로 가서 해당 이벤트를 발생시키는 코드를 작성해 보겠습니다.

우선 MouseGameController.cs 로 가 보겠습니다. 현재 MouseGameController.cs 의 코드는 다음과 같이 되어 있을 것입니다.

MouseGameController.cs

```
public class MouseGameController : IGameController
{
    public bool FireButtonPressed()
    {
        return Input.GetMouseButton(0);
    }
}
```

일단 다음과 같이 기존의 코드를 지우겠습니다.

MouseGameController.cs

```
public class MouseGameController : IGameController
{
    public bool FireButtonPressed()
    {
        return Input.GetMouseButton(0);
    }
}
```

그런데 이렇게 지우게 되면, 코드에 문제가 있다는 것을 나타내는 빨간 줄이 IGameController 아래에 표시됩니다. (비쥬얼 스튜디오를 사용하는 경우)

```
public class MouseGameController : IGameController
{
    ...
}
```

이것은 우리가 방금 지웠던 FireButtonPressed() 함수가 IGameController 인터페이스에 규정되어 있기 때문입니다. IGameController 인터페이스에서 파생된 클래스들은

무조건 FireButtonPressed() 함수를 구현(implementation)해야 하는데, 방금 우리가 그 함수를 지워 버렸기 때문에 이와 같은 경고가 표시되는 것입니다. 따라서 IGameController.cs 로 이동해서 해당되는 코드를 지우도록 하겠습니다.

```
I	GameController.cs  
using System.Collections;  
using System.Collections.Generic;  
using UnityEngine;  
  
public interface IGameController  
{  
    bool FireButtonPressed();  
}
```

이제 IGameController 인터페이스에서 더 이상 FireButtonPressed() 함수를 구현하라고 강제하지 않습니다. 따라서 다시 MouseGameController.cs 로 가 보시면 아까 있었던 빨간 줄이 사라지고 없는 것을 확인하실 수가 있을 것입니다.

```
public class MouseGameController : IGameController  
{  
    ...  
}
```

[동영상 예제 파일명: 012_delete_existing_codes.mp4]

액션(Action)으로 이벤트를 구현해 보자

다음으로는 이벤트를 하나 만들어 넣겠습니다. 이 때 사용할 수 있는 이벤트로는 Unity에서 제공하는 UnityEvent 나 C# 언어 자체(정확하게는 닷넷 프레임워크)에서 제공하는 EventHandler 가 있지만, 현실에서 많은 프로그래머들은 Action이라는, C#의 닷넷 프레임워크에서 제공하는 자체 딜리게이트(delegate)를 이벤트 핸들러 대신으로 사용하곤 합니다. 사용하기가 편하기 때문에 저도 Action을 사용하도록 하겠습니다.

그런데 Action 을 사용하기 위해서는 클래스 상단에 using System; 을 추가해야만 합니다. 따라서 다음과 같이 해당 코드를 먼저 추가하겠습니다.

```
MouseGameController.cs
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MouseGameController : IGameController
{}
```

그리고 나서 MouseGameController 클래스 내부에 다음과 같이 FireButtonPressed라는 이름의 Action 을 public 으로 선언합니다.

```
MouseGameController.cs
public class MouseGameController : IGameController
{
    public Action FireButtonPressed;
}
```

Action 은 C#의 닷넷 프레임워크에서 자체 제공하는 델리게이트(delegate)입니다.

C# 프로그래밍의 기초를 배우신 분들은 델리게이트가 무엇인지 아실 것입니다. 간단히 말씀 드리면 ‘함수에 대한 참조’라고 할 수가 있습니다. 델리게이트라는 단어의 뜻 자체는 ‘무엇을 대신한다’ 또는 ‘대리한다’는 의미를 가지고 있습니다. 따라서 우리는 FireButtonPressed 라는 델리게이트에 어떤 함수를 할당해 놓고 (함수는 여러개를 동시에 할당하는 것이 가능합니다. 이것을 ‘델리게이트 체인’이라고 부릅니다.), 특정한 상황에서 FireButtonPressed 를 호출함으로써, 여기에 연결된 다수의 다른 함수들을 동시에 호출하는 것이 가능해 집니다. 말로만 설명 드리면 이해가 잘 안되실 수 있으니까, 일단은 코딩을 계속 해 보겠습니다.

위에서 FireButtonPressed 라는 Action을 하나 만들었으므로 이제 이것을 호출하는 코드를 작성하겠습니다. 이번에는 일반적인 유니티 클래스와 마찬가지로 Update() 함수를 사용해 보려고 합니다. 그런데 이렇게 하기 위해서는 MouseGameController 클래스가 MonoBehaviour 클래스를 상속받도록 해야 합니다. 유니티의 Update() 함수는 MonoBehaviour 의 파생 클래스에서만 사용할 수가 있기 때문입니다. 따라서 다음 코드와 같이 MonoBehaviour 와의 상속 관계를 표시해 주겠습니다.

MouseGameController.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MouseGameController : MonoBehaviour, IGameController
{
    public Action FireButtonPressed;
```

위와 같이 하면, 이제 MouseGameController 는 IGameController 뿐 아니라 MonoBehaviour 의 파생 클래스이기도 한 상태가 됩니다. 어떤 사람들은 이렇게 여러 개의 클래스를 상속한다고 해서 이것을 ‘다중 상속’이라고 말하기도 하는데, 사실 MouseGameController 가 상속하는 것은 MonoBehaviour 일 뿐이고, IGameController 는 그냥 ‘이 인터페이스를 구현하겠다는 약속’에 불과하기 때문에 엄밀하게 말하면 이것을 다중 상속이라고 이야기할 수는 없습니다. IGameController 라는 인터페이스 자체에는 상속할만한 실제 코드 내용이 하나도 없기 때문입니다. 여하튼 이와 같은 형태가 다중 상속이냐 아니냐는 프로그래머들 사이의 이론적인 논쟁거리일 뿐이므로 더 이상 신경 쓰지 마시고, 그냥 진행하시기 바랍니다.

어쨌건 이제 MouseGameController 는 MonoBehaviour 의 파생 클래스이므로 다른 유니티 스크립트들과 마찬가지로 Update() 나 Start() 와 같은 함수를 사용할 수가 있습니다. 그럼 여기에 Update() 함수를 만들어 넣어 보겠습니다. (오타가 없도록 특히 대소문자를 정확하게 입력하시기 바랍니다)

```
MouseGameController.cs
public class MouseGameController : MonoBehaviour, IGameController
{
    public Action FireButtonPressed;

    void Update()
    {

    }
}
```

그리고 Update() 함수 안에 다음과 같은 코드를 추가합니다.

```
MouseGameController.cs
public class MouseGameController : MonoBehaviour, IGameController
{
    public Action FireButtonPressed;

    void Update()
    {
        if (Input.GetMouseButton(0))
        {
            FireButtonPressed();
        }
    }
}
```

위의 코드는, 마우스 왼쪽 버튼을 클릭하면 FireButtonPressed() 를 호출하라는 의미입니다 FireButtonPressed는 텔리게이트이므로, 어떤 다른 함수를 대신하고 있을 것입니다. 따라서 FireButtonPressed() 를 호출한다는 것은, 이 텔리게이트에 연결된 모든 함수들을 동시에 실행하라는 의미가 됩니다.

그런데 여기에 문제가 하나 있습니다. 만약에 FireButtonPressed 라는 텔리게이트에 아무런 함수가 연결되어 있지 않다면 이 텔리게이트를 호출한다는 것은 아무런 의미가 없습니다. 따라서 텔리게이트 함수를 호출할 때는 항상 이것이 null 인지를 체크하는 습관을 가지셔야 합니다. 다음과 같이 코드를 추가하도록 하겠습니다

```
public class MouseGameController : MonoBehaviour, IGameController
{
    public Action FireButtonPressed;

    void Update()
    {
        if (Input.GetMouseButtonUp(0))
        {
            if (FireButtonPressed != null)
            {
                FireButtonPressed();
            }
        }
    }
}
```

위와 같이 하면, 이 코드는 다음과 같은 의미를 가지게 됩니다. “만약 마우스 왼쪽 클릭이 감지되면, FireButtonPressd 에 연결된 모든 함수들이 실행되도록 한다”. 이것이 Action 을 이용한 간단한 ‘이벤트 송신’입니다.

[동영상 예제 파일명: 013_create_mouse_click_action.mp4]

이벤트 수신 함수를 만들자

이제 이벤트를 송신하는 쪽의 코드가 완성되었으므로, 다음으로는 이벤트를 수신하는 쪽의 코드를 작성해 보겠습니다. 현재 게임에서 이벤트를 수신할 쪽은 총알 발사대(BulletLauncher)일 것입니다. 따라서 BulletLauncher.cs로 가서, FireButtonPressed 이벤트를 수신할 함수를 만들어 보겠습니다. 우선 다음과 같은 코드를 BulletLauncher.cs에 작성합니다.

BulletLauncher.cs

```
public class BulletLauncher : MonoBehaviour
{
    // Update is called once per frame
    void Update()
    {

    }

    public void OnFireButtonPressed()
    {
        Debug.Log("Fired a bullet!");
    }
}
```

여기서 주의해서 보셔야 할 것은, 이벤트 수신을 위해 만든 함수가 void 타입의 함수라는 것입니다. 이것은 우리가 앞에서 MouseGameController.cs에서 선언한 액션(Action)이 다음과 같이 되어 있기 때문입니다.

```
public Action FireButtonPressed;
```

Action 을 이용해서 텔리게이트를 만들 때의 약속은 다음과 같습니다. 우선 위의 예처럼 Action 이라는 키워드를 사용했을 경우, 이 텔리게이트가 대신할 수 있는 함수는 void 타입이어야 합니다. (만약 리턴값이 있는 함수를 대신하고 싶다면 Action이 아닌 Func 라는 키워드를 사용해야 하지만, 여기에서는 더 깊이 말씀드리지 않겠습니다. 우리가 만들고 있는, 보통의 이벤트들은 리턴값이 필요하지 않기 때문입니다.)

그리고 주의해서 보셔야 할 부분이 하나 더 있습니다. 이벤트 수신을 위해 우리가 방금 만든 OnFireButtonPressed() 함수는 인자가 없는 함수입니다. Action 이라는 키워드를 이용해서 만든 텔리게이트는 인자가 없는 함수만을 대신할 수 있기 때문입니다. 그러면 만약 다음과 같이 int 타입의 인자를 받는 함수의 텔리게이트를 만들고 싶다면 어떻게 해야 할까요? (다음의 초록색으로 표시한 부분은 설명을 위해 넣은 코드이므로 따라서 입력하시지 마시기 바랍니다. 이 책을 보시는 독자 여러분은 오로지 빨간색으로 입력한 코드만을 따라서 입력하셔야 합니다)

```
public void OnFireButtonPressed(int count)
{
}
```

이 경우에는 Action이 아닌, Action<int> 라는 키워드를 사용하시면 됩니다. 예를 들어서 다음과 같은 식입니다.

```
public Action<int> FireButtonPressed;
```

같은 원리로, int 타입의 변수와 string 타입의 변수 두 개를 인자로 받는 함수의 멜리
게이트를 만들고 싶다면 Action<int, string> 키워드를 사용하시면 됩니다. 예를 들어
다음과 같은 식입니다.

```
public Action<int, string> FireButtonPressed;
```

그러면 이제 마우스 게임 콘트롤러(MouseGameController)의 송신 이벤트와 총알 발
사대(BulletLauncher)의 이벤트 수신 함수를 어떻게 연결하는지 살펴 보도록 하겠습니다.

이벤트 송신자와 수신자의 연결

이벤트를 연동하는 방법으로 생각할 수 있는 방법은 두 가지입니다. 하나는 수신자(여기에서는 BulletLauncher)의 내부에서 연결하는 것이고 또 하나는 수신자의 외부에서 연결하는 방법입니다. 저는 수신자의 외부에서 연결하는 방법을 사용하겠습니다. 이 방법의 핵심은 MouseGameController 나 BulletLauncher 가 "서로의 존재를 모르면서도" 커뮤니케이션 할 수 있도록 하는 것입니다. 그런데 이렇게 하기 위해서는 MouseGameController 와 BulletLauncher 모두의 존재를 알고 있는 어떤 중재자 역할의 클래스가 필요합니다. 그 역할을 하는 것이 바로 게임 매니저(GameManager)입니다.

우선 GameManager.cs 를 열어 보겠습니다. 현재 GameManager.cs 는 다음과 같이 되어 있을 것입니다.

```
GameManager.cs
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class GameManager : MonoBehaviour
{
    [SerializeField]
    BulletLauncher launcherPrefab;
    BulletLauncher launcher;

    // Start is called before the first frame update
    void Start()
    {
        launcher = Instantiate(launcherPrefab);
    }

    // Update is called once per frame
```

```
void Update()
{
}
}
```

위와 같이 GameManager.cs 의 Start() 함수에는 총알 발사대의 인스턴스(launcher) 가 이미 만들어져 있습니다. 그 아래에 다음 예제와 같이 MouseGameController의 인스턴스를 생성해 보겠습니다.

GameManager.cs

```
void Start()
{
    launcher = Instantiate(launcherPrefab);

    MouseGameController mouseController =
        gameObject.AddComponent<MouseGameController>();
}
```

앞의 디펜던시 인젝션(Dependency Injection) 방법을 사용할 때와 달리 new 키워드 를 이용하여 MouseGameController의 인스턴스를 생성하지 않은 이유는, 이제 MouseGameController 클래스가 MonoBehaviour 의 파생클래스가 되었기 때문입니다. MonoBehaviour 파생 클래스에서는 new 키워드를 이용해서 인스턴스를 만들 수 없습니다. 따라서 이런 경우에는 위와 같이 AddComponent 함수를 이용하셔야 합니다.

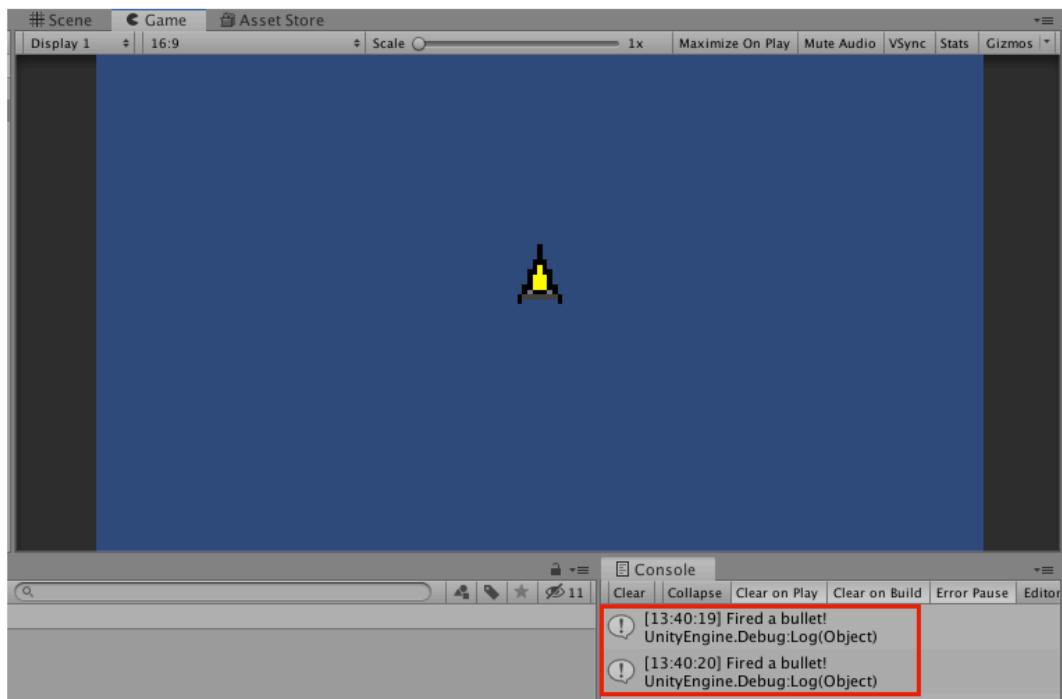
다음으로는 mouseController의 이벤트 송신자 함수와 launcher의 이벤트 수신자 함수를 헬리게이트 체인을 통해 연결하도록 하겠습니다.

```
void Start()
{
    launcher = Instantiate(launcherPrefab);

    MouseGameController mouseController =
        gameObject.AddComponent<MouseGameController>();
    mouseController.FireButtonPressed += launcher.OnFireButtonPressed;
}
```

C# 프로그래밍 입문서를 읽어 보신 분들은 ‘델리게이트 체인’이라는 개념에 대해 알고 계실 것입니다. 함수를 대신할 수 있는 델리게이트에, 여러개의 함수를 마치 사슬 체인처럼 연결해서 한번에 호출할 수 있도록 하는 것입니다. 이를 위해서 사용하는 방법이 바로 += 라는 키워드입니다. 위의 코드에서도 이 델리게이트 체인을 이용하여 mouseController의 FireButtonPressed 와 launcher의 OnFireButtonPressed 를 연결 하였습니다.

그럼 이제 다 끝났으니, 유니티 에디터로 가서 테스트를 해 보겠습니다. 게임을 실행하고 화면 이곳 저곳을 마우스로 클릭해 보면 다음과 같이 콘솔창에 메시지가 표시될 것입니다.



[동영상 예제 파일명: 014_bind_mouse_click_event.mp4]

게임 자체는 앞에서 디펜던시 인젝션(Dependency Injection)을 사용했을 때와 달라진
것이 없습니다. 하지만 지금 사용한 방식이 앞의 방식과 다른 점은, 총알 발사대
(BulletLauncher)와 마우스 게임 컨트롤러(MouseGameController) 가 서로의 존재를
전혀 모르는데도 마우스 버튼 클릭에 따른 반응이 총알 발사대(BulletLauncher) 쪽에
서 일어나고 있다는 점입니다. 이를 다시 한번 확인하기 위해서 먼저
BulletLauncher.cs 를 열어 보겠습니다.

BulletLauncher.cs

```
public class BulletLauncher : MonoBehaviour
{
    // Update is called once per frame
    void Update()
    {

    }

    public void OnFireButtonPressed()
    {
        Debug.Log("Fired a bullet!");
    }
}
```

보시는 것처럼 BulletLauncher 클래스 어느 곳에도 MouseGameController와 관련
된 항목은 존재하지 않습니다. 따라서 우리가 나중에 MouseGameController가 아니
라 KeyGameController를 사용하건, 아니면 더 나아가서 XBoxGameController나
PlayStationGameController, 또는 OculusGameController 와 같은 다른 게임 컨트롤
러를 사용하는 경우가 생기더라도 BulletLauncher는 전혀 상관하지 않습니다. 그저
외부에서 어떤 이벤트가 발생해서, OnFireButtonPressed()에 전달되기만 하면 됩니다.
외부에서 아무 이벤트가 전달되지 않아도 상관 없습니다. 그 경우에는 아무 일도
일어나지 않을 것이므로 문제될 것은 하나도 없습니다.

또한, 이벤트를 발생시키는 쪽과 수신하는 쪽은 이렇게 서로의 존재를 모르기 때문에 나중에 어느 한 쪽의 코드를 수정하는 일이 있다고 해도, 이로 인해서 다른 쪽의 코드를 수정할 필요가 없다는 점도 중요합니다. 그야 말로 ‘느슨하게 커플링된’ 코드의 전형이라고 볼 수 있을 것입니다.

그리면 참고 삼아 이벤트를 발생시키는 MouseGameController.cs 코드도 다시 한번 열어 보겠습니다.

```
MouseGameController.cs
public class MouseGameController : MonoBehaviour, IGameController
{
    public Action FireButtonPressed;

    void Update()
    {
        if (Input.GetMouseButtonUp(0))
        {
            if (FireButtonPressed != null)
            {
                FireButtonPressed();
            }
        }
    }
}
```

보시다시피 MouseGameController 쪽에도 BulletLauncher 와 관계된 것은 하나도 보이지 않습니다. 누가 이벤트를 수신하건 상관 없이, 어떤 조건(마우스 왼쪽 버튼 클릭)이 충족되면 FireButtonPressed 라는 이벤트를 그냥 날려 보낼 뿐입니다. 따라서 우리의 게임에서 나중에 누군가가 BulletLauncher 클래스를 수정하는 경우가 발생한다고 해도 전혀 문제될 것이 없습니다. MouseGameController는 BulletLauncher 클래스의 존재 자체를 모르기 때문입니다.

이제 MouseGameController.cs 작성 및 이벤트 연동이 모두 끝났으니, 이번에는 KeyGameController.cs 를 수정해 보겠습니다. 현재 KeyGameController.cs 코드는 다음과 같은 상태일 것입니다.

```
KeyGameController.cs
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class KeyGameController : IGameController
{
    public bool FireButtonPressed()
    {
        return Input.GetKeyDown(KeyCode.Space);
    }
}
```

먼저 FireButtonPress() 함수 부분을 삭제하도록 하겠습니다.

```
KeyGameController.cs
public class KeyGameController : IGameController
{
    public bool FireButtonPressed()
    {
        return Input.GetKeyDown(KeyCode.Space);
    }
}
```

다음으로는 KeyGameController 클래스를 MonoBehaviour 의 파생 클래스로 전환하겠습니다. 다음과 같이 MonoBehaviour 라는 클래스 이름을 “KeyGameController:” 다음에 추가한 뒤, 쉼표(,)를 덧붙여 주면 됩니다.

```
KeyGameController.cs
public class KeyGameController : MonoBehaviour, IGameController
{
}
```

다음으로 Action을 추가합니다. Action 을 사용하려면 먼저 ‘Using System’ 을 추가해야 합니다. 이 코드를 네임스페이스 맨 상단에 추가하도록 하겠습니다. 그리고 MouseGameController 의 경우와 마찬가지로 FireButtonPressed 액션을 추가합니다.

KeyGameController.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class KeyGameController : MonoBehaviour, IGameController
{
    public Action FireButtonPressed;
}
```

역시 Update() 유니티 함수를 추가합니다. 이 함수에서 플레이어의 키보드 입력을 받아들일 예정입니다.

KeyGameController.cs

```
public class KeyGameController : MonoBehaviour, IGameController
{
    public Action FireButtonPressed;

    void Update()
    {
    }
}
```

다음으로는 Input.GetKeyDown(KeyCode.Space) 를 이용해서 플레이어가 Space 키를 입력했는지 체크하는 조건문을 작성합니다.

```
KeyGameController.cs
void Update()
{
    if (Input.GetKeyDown(KeyCode.Space))
    {
        }
}
```

그리고 나서, 먼저 FireButtonPressed 가 null 인지를 확인합니다. 이 Action 에 연동된 이벤트 수신 함수가 하나도 없다면 이 Action 이벤트를 발동시킬 필요가 없기 때문입니다. 다음과 같이 null 체크를 한 뒤, null 이 아닌 경우에 FireButtonPressed() 를 실행해 줍니다.

```
KeyGameController.cs
void Update()
{
    if (Input.GetKeyDown(KeyCode.Space))
    {
        if (FireButtonPressed != null)
        {
            FireButtonPressed();
        }
    }
}
```

이제 KeyGameController.cs 쪽의 코드도 모두 작성했습니다. 다음으로는 게임 매니저(GameManager.cs) 로 가서 MouseGameController 대신, 지금 만든 KeyGameController를 이용하도록 코드를 수정해 보겠습니다.

입력 방식을 키보드로 바꿔 보자

먼저 비쥬얼 스튜디오로 GameManager.cs 파일을 연 다음, Start() 함수를 찾아 MouseGameController 와 관련된 부분을 찾아서 다음과 같이 주석 처리해 주겠습니다. (삭제가 아니고 주석처리입니다. 나중에 필요해서 그렇게 하는 거니까 코드를 지우지 마시기 바랍니다.)

GameManager.cs

```
void Start()
{
    launcher = Instantiate(launcherPrefab);

    //MouseGameController mouseController =
    //    gameObject.AddComponent<MouseGameController>();
    //mouseController.FireButtonPressed += launcher.OnFireButtonPressed;

}
```

그리고 나서, gameObject.AddComponent<KeyGameController>() 명령어를 이용하여 이번에는 KeyGameController 의 인스턴스를 한 개 생성한 다음에, 이를 keyController 라는 로컬 변수에 할당하도록 하겠습니다.

GameManager.cs

```
void Start()
{
    launcher = Instantiate(launcherPrefab);

    //MouseGameController mouseController =
    //    gameObject.AddComponent<MouseGameController>();
    //mouseController.FireButtonPressed += launcher.OnFireButtonPressed;

    KeyGameController keyController
        = gameObject.AddComponent<KeyGameController>();

}
```

다음으로는 앞에서와 마찬가지로 keyController의 FireButtonPress 이벤트에 launcher의 OnFireButtonPressed 를 멀리게이트 체인 방식으로 연결해 줍니다. 이렇게 하면 launcher 쪽에는 아무런 변경을 가하지 않고 기존의 마우스 입력을 키보드 입력으로 전환할 수 있습니다.

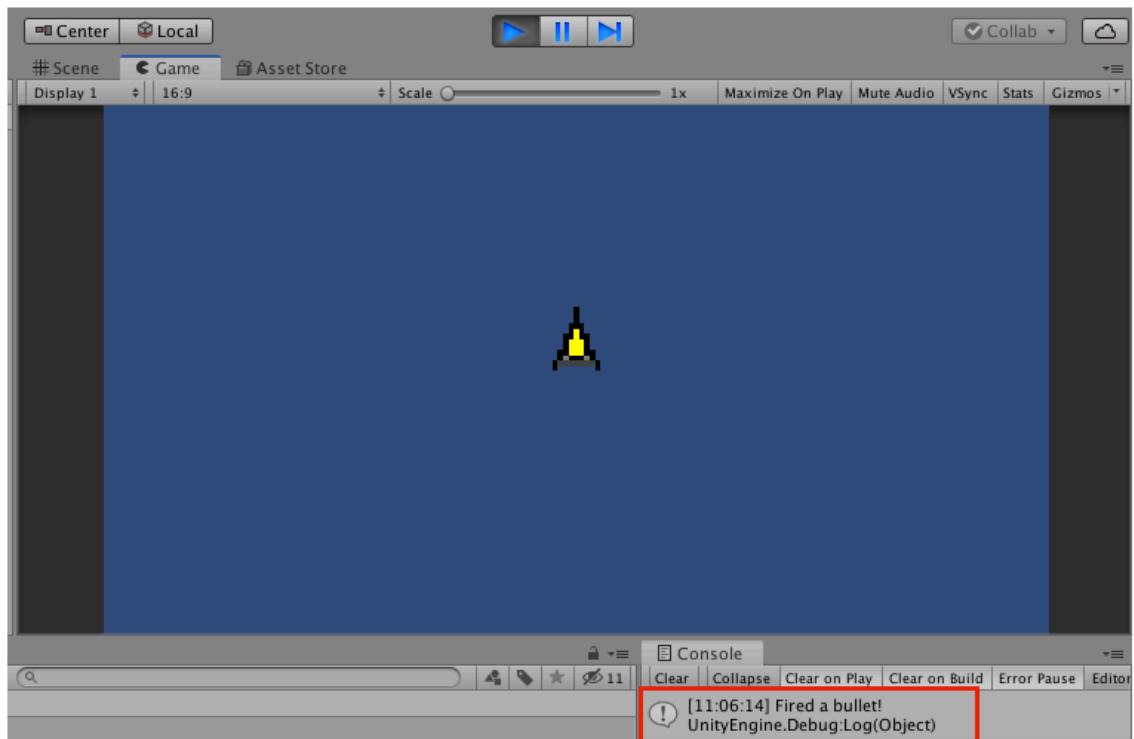
GameManager.cs

```
void Start()
{
    launcher = Instantiate(launcherPrefab);

    //MouseGameController mouseController =
    //    //gameObject.AddComponent<MouseGameController>();
    //mouseController.FireButtonPressed += launcher.OnFireButtonPressed;

    KeyGameController keyController =
        gameObject.AddComponent<KeyGameController>();
    keyController.FireButtonPressed += launcher.OnFireButtonPressed;
}
```

이제 유니티 에디터로 가서 Play 버튼을 눌어서 게임을 실행하고 테스트 해 보겠습니다. 게임이 실행된 뒤, 화면을 마우스로 클릭해 보면 이전과는 달리 아무런 반응이 없는 것을 확인할 수 있습니다. 하지만 키보드의 스페이스바를 누르면 아래 스크린샷처럼 콘솔창에 “Fired a bullet!”이라는 메시지가 뜹니다. 마우스 입력 방식에서 키보드 입력 방식으로 성공적으로 교체된 것입니다.



[동영상 예제 파일명: 015_bind_keyboard_event.mp4]

두 개의 게임 입력 방식 동시 지원하기

지금까지 launcher 쪽에는 아무런 수정을 가하지 않고 마우스 입력 방식을 키보드 입력 방식으로 바꾸는 것을 보여 드렸습니다. 이 방식이 편리하기는 하지만, 한번에 하나의 입력 방식만 지원한다는 점에서 다소 불편함이 있을 수 있습니다. 따라서 이번에는 마우스 입력 방식과 키보드 입력 방식을 동시에 지원하는 방법에 대해 살펴 보도록 하겠습니다.

우선 코딩에 들어가기 전에 기존에 사용했던 두 가지 방법을 비교해 보면, 디펜던시(dependency)를 외부에서 주입하는 방법을 사용했을 때는 게임 콘트롤 방식을 “마우스에서 키보드로” 또는 “키보드에서 마우스로” 바꾸는 것만 가능했습니다. 하지만 지금 우리가 사용했던 이벤트 방식을 이용할 경우에는 한번에 여러 개의 입력 방식을 동시에 지원할 수 있습니다. 예를 들어 마우스 입력과 키보드 입력을 동시에 지원할 수가 있는 것입니다. 여기에서 중요한 사실은, 이렇게 여러 개의 입력 방식을 동시에 지원해야 하는 경우에도 각각의 클래스 내부의 코드를 전혀 수정할 필요가 없다는 것입니다. 오직 게임 매니저(GameManager.cs)에서 텔리게이트 체인을 덧붙이거나 빼는 것만으로 새로운 사용자 입력 방식을 얼마든지 추가하거나 제거할 수가 있기 때문입니다.

그러면 현재 상태에서 다시 마우스 입력을 추가하려면 어떻게 하면 될까요? 방법은 간단합니다. 앞에서 주석 처리한 MouseGameController 관련 코드에서 주석 부분을 다시 제거하면 됩니다.

```

void Start()
{
    launcher = Instantiate(launcherPrefab);

    #MouseGameController mouseController =
        #gameObject.AddComponent<MouseGameController>();
    #mouseController.FireButtonPressed += launcher.OnFireButtonPressed;

    KeyGameController keyController =
        gameObject.AddComponent<KeyGameController>();
    keyController.FireButtonPressed += launcher.OnFireButtonPressed;
}

```

이렇게 하면 launcher의 OnFireButtonPressed 이벤트 수신 함수는 마우스 콘트롤러 (mouseController)와 키보드 콘트롤러(keyboardController) 모두에게서 FirePressed라는 이벤트를 받아 들어하게 됩니다. 따라서 두 가지 입력 방식 모두를 지원할 수 있게 되는 것입니다.

그럼 앞의 주석을 해제한 상태에서의 코드를 다시 확인해 보겠습니다. 다음과 같이 되어 있을 것입니다.

```

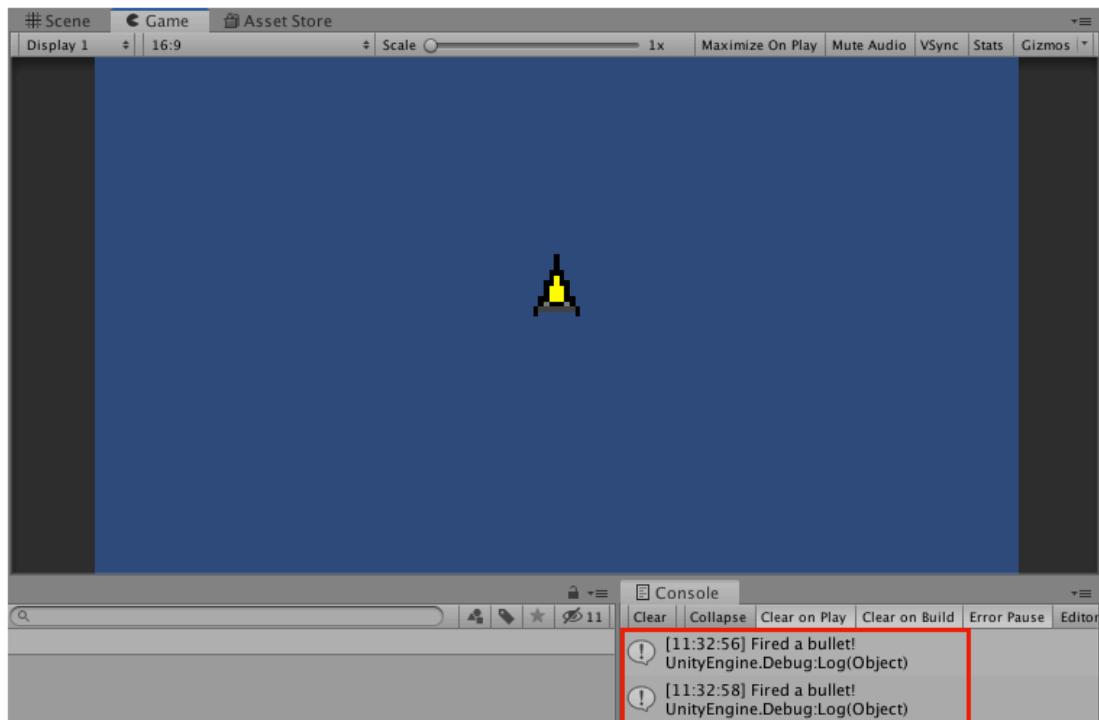
void Start()
{
    launcher = Instantiate(launcherPrefab);

    MouseGameController mouseController =
        gameObject.AddComponent<MouseGameController>();
    mouseController.FireButtonPressed += launcher.OnFireButtonPressed;

    KeyGameController keyController =
        gameObject.AddComponent<KeyGameController>();
    keyController.FireButtonPressed += launcher.OnFireButtonPressed;
}

```

그럼 이제 유니티 에디터로 이동해서 테스트를 해 보도록 하겠습니다. 플레이버튼을 눌러서 게임을 실행한 다음, 화면에 마우스 커서를 대고 클릭하거나 키보드의 스페이스 바를 입력해 보면, 두 가지 입력 방식 모두 잘 작동하는 것을 확인하실 수 있습니다.



[동영상 예제 파일명: 016_bind_multiple_events.mp4]

다시 마우스 입력만 사용하자

지금까지 두 개의 게임 컨트롤 방식을 구현해 보았습니다. 하지만 지금부터 작성할 미사일 커맨더 게임 로직에서는 마우스 입력만 사용하도록 하겠습니다. 게임 자체가 마우스 입력에 맞도록 고안되었기 때문입니다. 따라서 우선 GameManager.cs 로 가서 앞에서 작성했던 KeyGameController 관련된 부분을 모두 지우도록 하겠습니다.

GameManager.cs

```
void Start()
{
    launcher = Instantiate(launcherPrefab);

    MouseGameController mouseController =
        gameObject.AddComponent<MouseGameController>();
    mouseController.FireButtonPressed += launcher.OnFireButtonPressed;

    KeyGameController keyController =
        gameObject.AddComponent<KeyGameController>();
    keyController.FireButtonPressed += launcher.OnFireButtonPressed;
}
```

또한 KeyGameController.cs 클래스 자체도 이제부터는 사용할 일이 없으니까 지워 도 됩니다.

마우스 입력 이벤트에 클릭 지점 좌표를 함께 전달하기

MouseGameController.cs 에서는 플레이어가 마우스를 클릭하는 순간, 이를 알리기 위한 이벤트를 발생시켰습니다. 하지만 마우스를 클릭했다는 사실을 아는 것만으로는 미사일 커맨더 게임을 만들 수 없습니다. 마우스를 클릭한 위치의 좌표를 알아야 그쪽으로 총알을 발사할 수 있기 때문입니다.

따라서 이제부터는 MouseGameController 에 만들었던 Action 이벤트를 수정해서, 이벤트 발생 시 마우스 클릭 지점의 좌표를 함께 전달할 수 있도록 해 보겠습니다. 이를 위해서 먼저 MouseGameController.cs 를 비쥬얼 스튜디오로 열어 보겠습니다. 현재까지의 코드는 다음과 같이 되어 있을 것입니다.

```
MouseGameController.cs
public class MouseGameController : MonoBehaviour, IGameController
{
    public Action FireButtonPressed;

    void Update()
    {
        if (Input.GetMouseButtonDown(0))
        {
            if (FireButtonPressed != null)
            {
                FireButtonPressed();
            }
        }
    }
}
```

일단 Action 의 타입부터 변경하도록 하겠습니다. 현재 Action은 아무런 매개 변수를 전달하지 않고 있는데, FireButtonPressed 이벤트가 발생할 때 마우스 클릭 지점의 좌표를 Vector3 타입으로 전달하도록 Action 타입을 다음과 같이 변경하도록 하겠습니다.

```
MouseGameController.cs
public class MouseGameController : MonoBehaviour, IGameController
{
    public Action<Vector3> FireButtonPressed;

    void Update()
    {
        if (Input.GetMouseButtonUp(0))
        {
            if (FireButtonPressed != null)
            {
                FireButtonPressed();
            }
        }
    }
}
```

Action의 타입이 변경되었으므로, Update() 함수 안에 있는 FireButtonPressed() 에서도 Vector3 값을 매개 변수로 전달해야만 합니다. 그렇게 하지 않으면 에러가 나기 때문입니다. 아직까지 마우스 클릭 지점을 알아 내는 코드를 작성하지 않았기 때문에, 일단은 에러 메시지를 방지하기 위해 임시로 Vector3.zero 를 넣어 주기로 하겠습니다. (Vector3.zero 는 0,0,0 을 가리킵니다.)

MouseGameController.cs

```
public class MouseGameController : MonoBehaviour, IGameController
{
    public Action<Vector3> FireButtonPressed;

    void Update()
    {
        if (Input.GetMouseButtonUp(0))
        {
            if (FireButtonPressed != null)
            {
                FireButtonPressed(Vector3.zero);
            }
        }
    }
}
```

이제 FireButtonPressed 의 형식이 바뀌었으므로, 이 이벤트를 받아서 총알을 발사하는 일을 하는 BulletLauncher.cs 의 OnFireButtonPressed 함수도 수정해야 할 것입니다. 하지만 BulletLauncher.cs 로 가 보면 아무런 예러나 경고 표시도 보이지 않는 것을 확인하실 수 있습니다.

BulletLauncher.cs

```
public class BulletLauncher : MonoBehaviour
{
    // Update is called once per frame
    void Update()
    {

    }

    public void OnFireButtonPressed()
    {
        Debug.Log("Fired a bullet!");
    }
}
```

BulletLauncher.cs 에 아무런 경고 표시가 보이지 않는 이유는, BulletLauncher 와 MouseGameController 가 서로 완전히 독립되어 있기 때문입니다. BulletLauncher 에서는 MouseGameController 라는 것이 존재한다는 사실 자체를 알지 못합니다. BulletLauncher 클래스의 OnFireButtonPressed() 는 그저 같은 형식의 이벤트와 연동되어 있을 때, 해당 이벤트가 발생하면 거기에 반응할 뿐입니다. 누가 그 이벤트를 발생시키는지에 대해서는 전혀 상관하지 않습니다.

또한 이들 이벤트 발신자와 수신자를 연결하는 일도 BulletLauncher.cs 의 역할이 아닙니다. 따라서 MouseGameController 에서 어떤 코드 수정이 발생했다고 해도 그것은 BulletLauncher와는 전혀 상관 없는 일입니다. BulletLauncher 쪽에 아무런 경고 메시지가 보이지 않는 것은 바로 이 때문입니다.

그러면 MouseGameController 의 코드 변경으로 인해 영향을 받는 클래스는 존재하지 않을까요? 그렇지 않습니다. MouseGameController 의 FireButtonPressed() 이벤트와 다른 클래스의 이벤트 수신 함수를 서로 연결해 주는 역할을 하는 게임 매니저 (GameManager) 가 영향을 받게 됩니다. MouseGameController 쪽의 변경으로 인해, 이제 게임 매니저는 서로 다른 규약을 가진 이벤트 수신자와 송신자를 연결하게 되었고, 그 결과로 인해 발생하는 여러의 책임은 게임 매니저가 지게 됩니다. 그럼 GameManager.cs 로 가 보겠습니다.

GameManager.cs로 가 보시면, 다음과 같이 mouseController의 FireButtonPressed 와 bulletLauncher의 OnFireButtonPressed를 연동해 주는 부분에 경고 표시가 나타

나 있는 것을 확인하실 수 있습니다. 현재 FireButtonPressed 는 Action<Vector3> 형식으로 변경되었습니다. 따라서 이 이벤트를 수신하는 OnFireButtonPressed() 함수도 Vector3 형식의 매개 변수를 받도록 형식이 정의되어 있어야 하는데, 그렇지 않기 때문에 경고 표시가 뜨는 것입니다.

```
void Start()
{
    launcher = Instantiate(launcherPrefab);

    MouseGameController mouseController =
        gameObject.AddComponent<MouseGameController>();
    mouseController.FireButtonPressed += launcher.OnFireButtonPressed;
}
```

다시 말해서 GameManager.cs에서 경고 표시가 뜨는 이유는 형식이 다른 이벤트 송신자와 수신자를 뚫으려고 했기 때문입니다. 따라서 이 경고 표시를 없애기 위해서는 BulletLauncher.cs의 OnFireButtonPressed() 함수가 Vector3 형식의 매개 변수를 받도록 하면 됩니다. 그럼 BulletLauncher.cs로 가서 다음과 같이 해당 함수에 매개 변수를 추가하도록 하겠습니다.

```
BulletLauncher.cs

public class BulletLauncher : MonoBehaviour
{
    // Update is called once per frame
    void Update()
    {

    }

    public void OnFireButtonPressed(Vector3 position)
    {
        Debug.Log("Fired a bullet!");
    }
}
```

그리고 나서 테스트를 위해 기존에 작성해 놓았던 Debug.Log() 명령문을 약간 수정해서, 인자로 전달된 Vector3 타입의 값(position)을 콘솔창에 표시하도록 하겠습니다.

BulletLauncher.cs

```
public class BulletLauncher : MonoBehaviour
{
    // Update is called once per frame
    void Update()
    {

    }

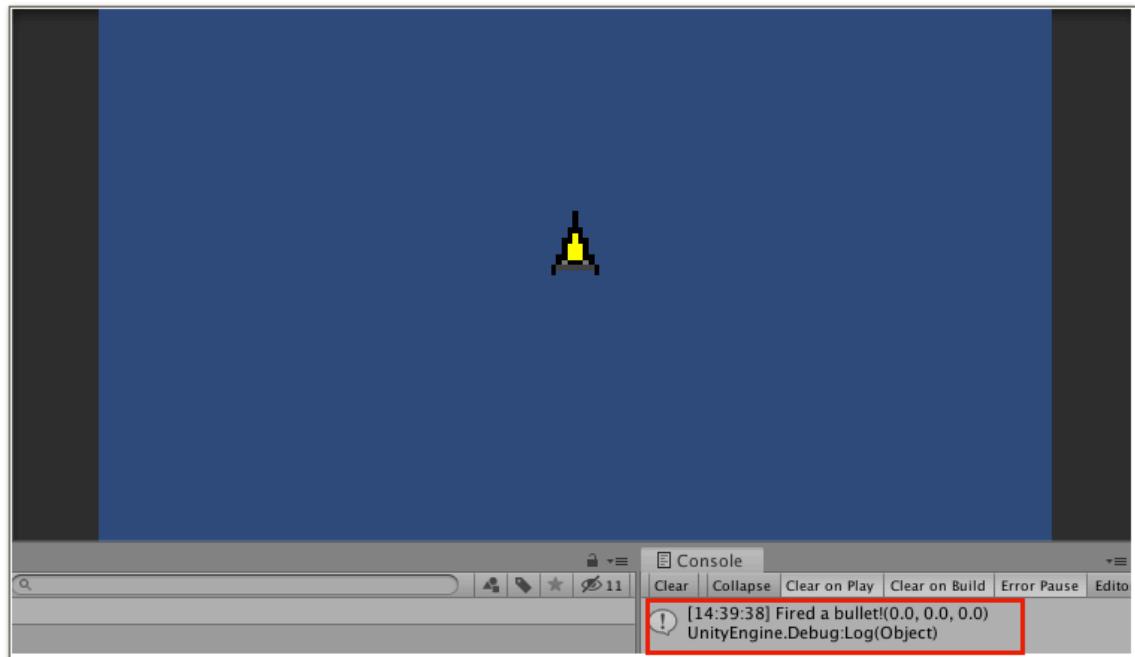
    public void OnFireButtonPressed(Vector3 position)
    {
        Debug.Log("Fired a bullet!" + position);
    }
}
```

이제 이벤트 송신자 쪽과 수신자 쪽의 형식을 일치시켰습니다. 따라서 이제 GameManager.cs로 다시 가 보면, 아까의 경고 표시가 사라져 있는 것을 확인하실 수 있습니다.

```
void Start()
{
    launcher = Instantiate(launcherPrefab);

    MouseGameController mouseController =
        gameObject.AddComponent<MouseGameController>();
    mouseController.FireButtonPressed += launcher.OnFireButtonPressed;
}
```

그럼 다시 유니티 에디터로 가서 테스트를 해 보겠습니다. 플레이 버튼을 눌러서 게임을 실행한 다음, 화면 아무 곳에나 마우스 커서를 대고 클릭해 봅니다. 그러면 다음 그림처럼 콘솔창에 임시로 전달한 위치값 (0,0,0) 이 로그 메시지와 함께 표시되는 것을 확인하실 수 있습니다.



[동영상 예제 파일명: 017_event_with_mouse_click_position.mp4]

마우스 클릭 지점의 실제 좌표를 전달하자

이제 MouseGameController.cs 로 가서 임시 벡터 값이 아닌 실제 값을 전달하도록 코드를 수정해 보도록 하겠습니다. 우선 Update() 안에 있는 코드 중, FireButtonPressed() 함수를 찾아서, 매개 변수로 전달된 Vector3.zero 부분을 삭제하도록 하겠습니다.

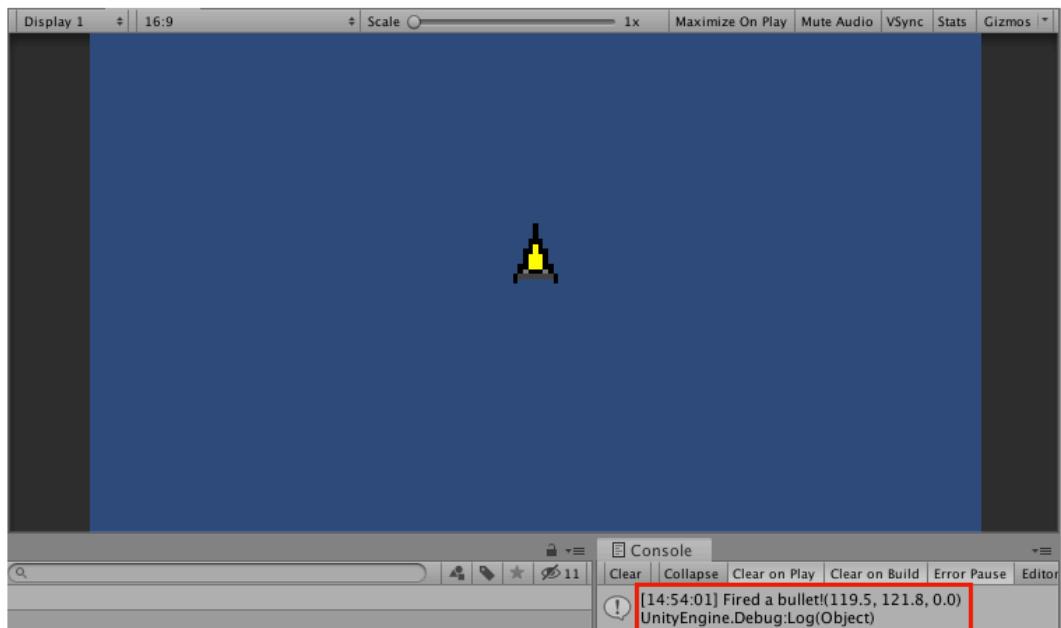
```
MouseGameController.cs
public class MouseGameController : MonoBehaviour, IGameController
{
    public Action<Vector3> FireButtonPressed;

    void Update()
    {
        if (Input.GetMouseButtonUp(0))
        {
            if (FireButtonPressed != null)
            {
                FireButtonPressed(Vector3.zero);
            }
        }
    }
}
```

우리가 전달받고자 하는 것은 마우스 클릭 지점의 좌표입니다. 마우스 클릭 지점이라고 할 때 제일 먼저 떠올릴 수 있는 것은 Input.mousePosition 입니다. 하지만 이것은 우리가 원하는 좌표가 아닙니다. 왜냐하면 Input.mousePosition은 게임의 월드 좌표가 아니라 화면상의 스크린 좌표 값을 알려 주기 때문입니다. 이를 확인하기 위해 FireButtonPressed() 의 인자로 Input.mousePosition 을 넣어 보도록 하겠습니다.

```
void Update()
{
    if (Input.GetMouseButtonUp(0))
    {
        if (FireButtonPressed != null)
        {
            FireButtonPressed(Input.mousePosition);
        }
    }
}
```

이 상태에서 유니티 에디터에 가서 테스트를 해 보겠습니다. 플레이 버튼을 눌러서 게임을 실행한 다음 화면을 클릭한 다음, 콘솔창의 메시지를 보시면 게임 월드상의 실제 좌표가 아닌, 화면 해상도를 기준으로 한 스크린 좌표가 표시되는 것을 확인하실 수 있습니다.



[동영상 예제 파일명: 018_display_mouse_position.mp4]

이 좌표가 무엇을 의미하는지 확인하고 싶으시다면 화면의 좌측 하단과 우측 상단 맨 끝을 차례로 클릭해 보시면 이해하실 수 있을 것입니다. 현재 화면상의 총알 발사대의 위치는 월드 좌표상으로 (0,0,0)입니다. 따라서 게임의 월드 좌표 기준으로는 화면 정 중앙의 좌표가 (0,0,0)이 되어야 합니다. 하지만 Input.mousePosition 을 통해 얻은 좌표값은 화면의 좌측 하단이 (0,0,0)으로 표시됩니다. 이것은 Input.mousePosition 으로 얻을 수 있는 좌표값이 월드 좌표가 아니라 스크린 좌표이기 때문입니다.

따라서 이 Input.mousePosition 으로 얻은 좌표 값으로는 게임을 만들 수가 없습니다. 스크린 좌표를 게임 월드상의 좌표로 변환하는 과정이 필요합니다. 이러한 작업을 담당할 함수를 MouseGameController.cs 에 추가해 보도록 하겠습니다.

우선 MouseGameController.cs 로 가셔서 다음과 같이 Vector3 값을 리턴하는 GetCurrentClickPoint() 라는 이름의 함수를 하나 만들도록 하겠습니다.

```
MouseGameController.cs
public class MouseGameController : MonoBehaviour, IGameController
{
    public Action<Vector3> FireButtonPressed;

    void Update()
    {
        ...

        Vector3 GetCurrentClickPoint(Vector3 mousePosition)
        {
            ...
        }
    }
}
```

이 함수에서 하는 일은 함수의 인자로 전달받은 Input.mousePosition 값을 월드 좌표
로 변환하여 반환하는 것입니다. 이를 위해 다음과 같이
Camera.main.ScreenToWorldPoint() 함수를 이용하여 스크린 좌표를 월드 좌표로
변환한 다음에 이를 Vector3 타입의 로컬 변수인 point에 담겠습니다.

MouseGameController.cs

```
Vector3 GetCurrentClickPoint(Vector3.mousePosition)
{
    Vector3 point = Camera.main.ScreenToWorldPoint(mousePosition);
}
```

다음으로는 이렇게 얻은 point의 z 값을 0으로 변환한 다음에 point를 리턴하도록 하겠습니다. 이렇게 하는 이유는 우리가 만드는 게임이 2D 게임이기 때문입니다. 스크린 좌표를 월드 좌표로 변환하는 과정에서 혹시라도 z값이 0이 아닌 값이 나오게 될 경우를 방지하기 위해 안전 차원에서 z 값에 0을 입력한 뒤 반환한 것입니다.

MouseGameController.cs

```
Vector3 GetCurrentClickPoint(Vector3.mousePosition)
{
    Vector3 point = Camera.main.ScreenToWorldPoint(mousePosition);
    point.z = 0f;
    return point;
}
```

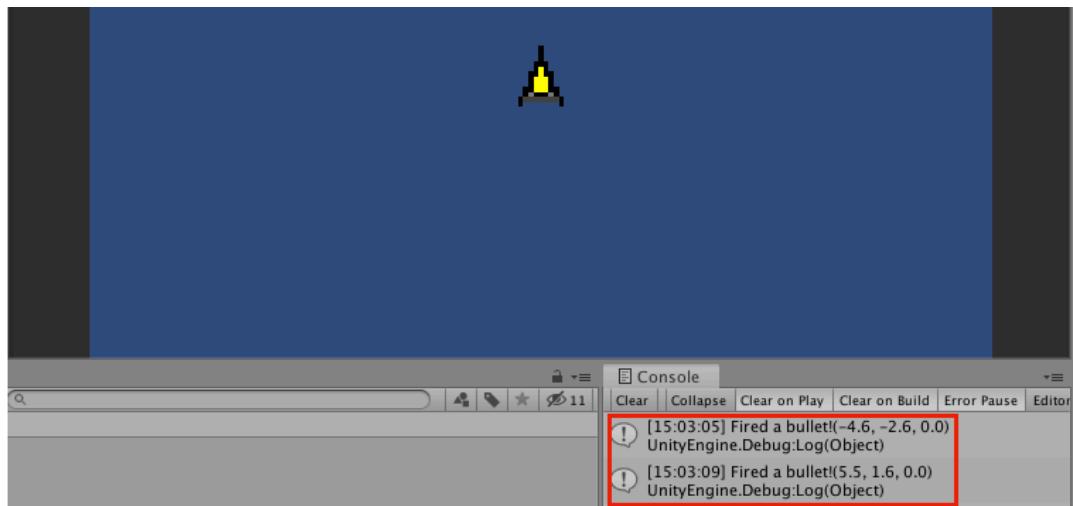
이제 Update() 함수로 가서 지금 만든 GetCurrentClickPoint() 함수를 이용하여
mousePosition을 월드 좌표로 변환한 값을 구한 뒤, 이를 FireButtonPressed()를 통해
전달하도록 하겠습니다.

```

void Update()
{
    if (Input.GetMouseButtonUp(0))
    {
        if (FireButtonPressed != null)
        {
            FireButtonPressed(GetCurrentClickPoint(Input.mousePosition));
        }
    }
}

```

이제 유니티 에디터에서 테스트해 보겠습니다. 플레이 버튼을 눌러서 게임을 실행한 다음, 화면 아무 곳이나 클릭해 보시면 이제 월드 좌표 기준으로 좌표값이 콘솔창에 표시되는 것을 확인하실 수 있을 것입니다. 정확한 테스트를 위해 화면 정 가운데의 총알 발사대가 있는 부분을 마우스로 클릭해 보시면 (0,0,0)에 가까운 좌표값이 표시되는 것을 알 수 있습니다.



[동영상 예제 파일명: 019_mouse_position_to_world_position.mp4]

총알(Bullet)을 만들자

이제 정확한 마우스 클릭 지점을 알아냈으니, 마우스로 화면을 클릭하는 순간 총알을 만들어서 그쪽으로 발사하는 코드를 작성해 보겠습니다.

우선 Bullet.cs라는 유니티 C# 스크립트 파일을 만들겠습니다. 이 파일을 비쥬얼 스튜디오로 열어 보시면 다음과 같이 되어 있을 것입니다.

```
Bullet.cs
public class Bullet : MonoBehaviour
{
    // Use this for initialization
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
    }
}
```

제일 먼저 해야 할 일은 총알의 이동 속도를 지정하기 위한 변수 moveSpeed를 선언하는 것입니다. 이 책에서 우리가 만들 모든 변수들은 외부에서 절대로 값을 변경할 수 없도록 public이 아닌 private으로 선언할 것입니다. (이 때, 변수 이름 앞에 private라는 꼭 키워드를 붙일 필요는 없습니다. 그 경우 자동으로 private 변수가 됩니다.)

하지만 초기값을 유니티 에디터의 인스펙터(Inspector)에서 변경할 수는 있어야 하므로 다음과 같이 [SerializeField] 어트리뷰트를 변수 선언문 상단에 붙여서 유니티 에디터에 노출되도록 하겠습니다. 이 때 moveSpeed 의 초기값은 5f 로 지정하겠습니다.

Bullet.cs

```
public class Bullet : MonoBehaviour
{
    [SerializeField]
    float moveSpeed = 5f;

    // Use this for initialization
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

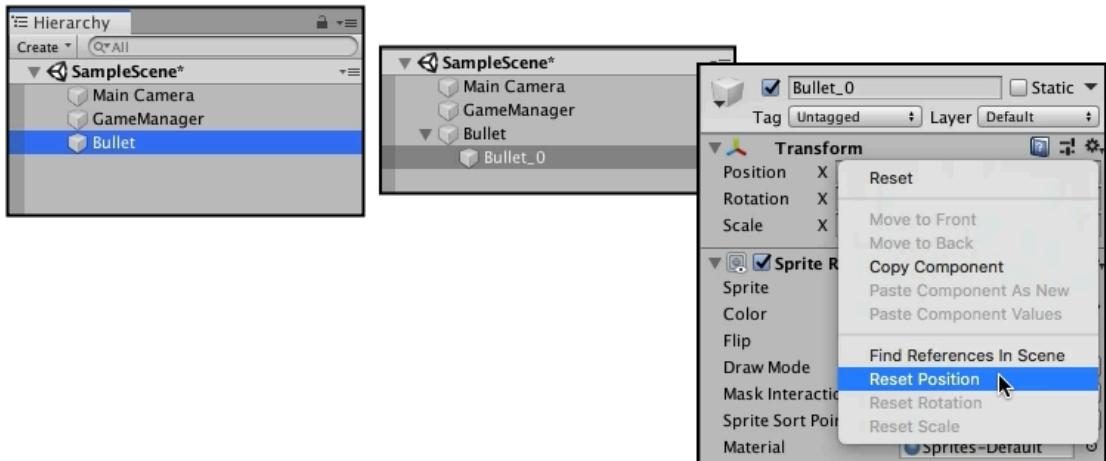
이제 Update() 함수에서 총알이 생성되면 전방을 향해서 일정 속도로 이동하는 로직을 생성하겠습니다. 2D 좌표를 기준으로 했을 때, 이 총알의 전방은 곧 위쪽(transform.up)을 의미하므로 다음과 같이 로컬 기준으로 위쪽 방향을 향해 일정 속도로 이동하는 로직을 작성하도록 하겠습니다.

Bullet.cs

```
void Update()
{
    transform.position += transform.up * moveSpeed * Time.deltaTime;
}
```

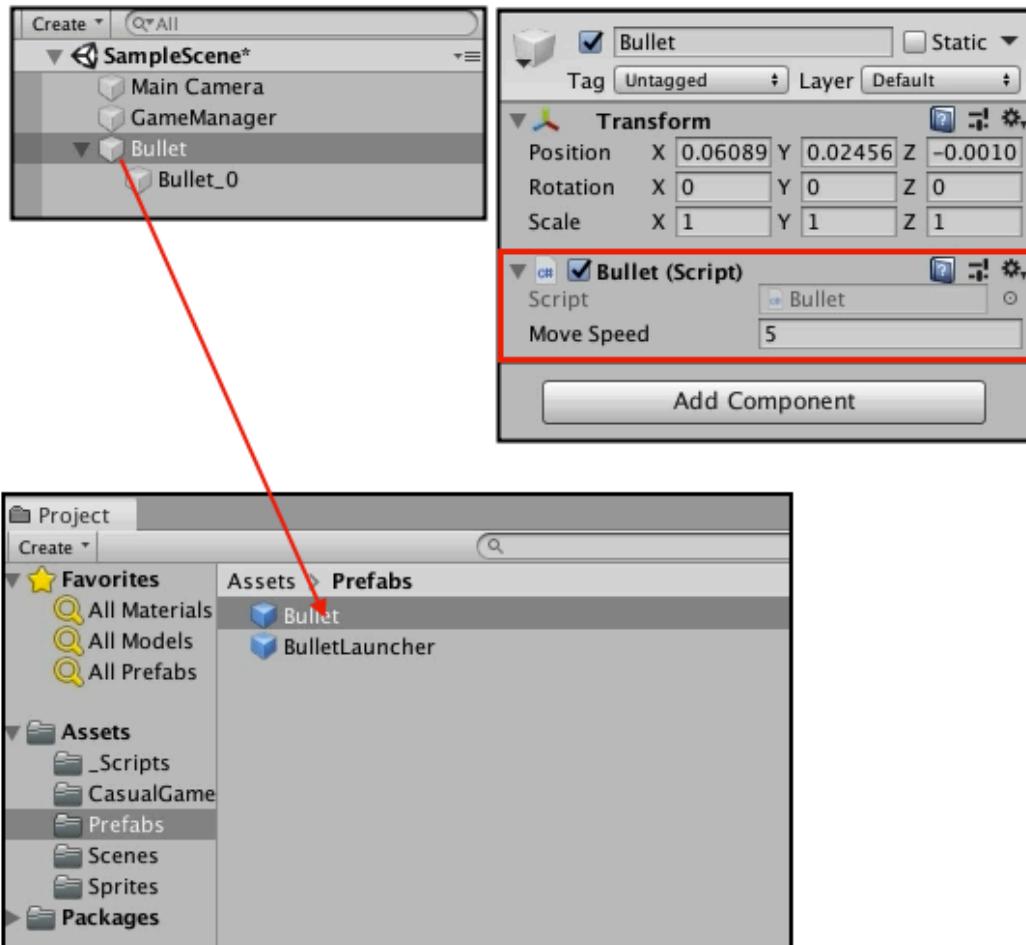
총알의 프리팹을 만들자

이제 스크립트 작성이 끝났으니, 실제 게임에서 사용할 총알 프리팹을 만들어 보겠습니다. 먼저 유니티 에디터로 가서 씬(Scene)에 빈 게임 오브젝트를 만들고 이름을 Bullet으로 변경하겠습니다. 그리고 나서 하단의 Sprites 폴더에서 총알 스프라이트를 찾은 뒤, 이를 드래그해서 방금 만든 Bullet 게임 오브젝트의 자식 오브젝트로 만들어 넣도록 하겠습니다. 이 과정에서 스프라이트의 로컬 좌표가 0,0,0 이 되어야 하므로, 트랜스폼에서 position 값이 0,0,0 인지 먼저 확인한 뒤, 만약 좌표값이 다를 경우에는 Reset Position 을 이용하여 0,0,0 으로 맞춰 주도록 하겠습니다.



그 다음으로는 Bullet 게임 오브젝트를 다시 클릭한 뒤에 방금 만든 Bullet.cs 스크립트를 붙여 주겠습니다.

그리고 나서 Bullet 게임 오브젝트를 하단의 Prefabs 폴더에 드래그해서 프리팹을 완성하도록 하겠습니다.



이제 씬에 남은 총알은 삭제예정이지만, 일단은 먼저 플레이버튼을 눌러서 총알이 전방으로 잘 이동하는지 확인해 보도록 하겠습니다. 아래 그림에 보시는 것처럼 총알이 전방으로 잘 이동하는 것을 보신 다음, 씬(Scene)에서 Bullet 게임 오브젝트를 삭제하시면 됩니다.



[동영상 예제 파일명: 020_create_bullet.mp4]

BulletLauncher를 이용해서 총알을 발사해 보자

이제 총알 발사대(BulletLauncher)를 이용해서 총알을 발사하도록 하겠습니다. 총알은 기존에 만든 게임 매니저(GameManager)에서 생성해도 되지만, GameManager는 게임 구조상 최상위에 있는 일종의 진입 포인트(entry point) 역할을 담당하도록 하고, 총알과 같은 개별 게임 오브젝트들은 일종의 중간 관리자 역할을 하는 별도의 클래스에서 처리하도록 하겠습니다.

예를 들어 총알과 폭발 이펙트의 생성과 관리는 총알 발사대(BulletLauncher) 클래스가 맡고, 적 미사일의 생성과 관리는 앞으로 만들 미사일 매니저(MissileManager) 클래스가 맡는 식입니다. 그리고 가장 최상위에 있는 GameManager 클래스가 이들 중간 관리자 역할의 클래스들(총알 발사대와 미사일 매니저)의 인스턴스를 만들고, 서로의 이벤트를 연결하거나 해제해 주는 총괄 관리자 역할을 하도록 하겠습니다. 현재로서는 이런 설명만으로 이해하시기 어려울 테니까, 일단은 코드 작성 과정을 쭉 따라오시기 바랍니다. 이 과정을 몇 번 따라서 반복적으로 연습해 보시면 왜 이렇게 하는지 자연스럽게 이해하시게 될 것입니다.

우선 총알 프리팹으로부터 총알을 만들어 보는 로직을 작성하겠습니다. 나중에는 Factory 라는 것을 이용해서 총알과 같은 게임 오브젝트들의 인스턴스들을 만들고 관리할 예정이지만, 일단 지금 당장은 일반적인 방법으로 총알의 인스턴스를 만들겠습니다.

먼저 기존에 작성한 스크립트 파일 중에서 BulletLauncher.cs 를 찾아서 연 다음, Bullet 프리팹을 담을 변수를 선언한 뒤, [SerializeField] 어트리뷰트를 덧붙여서 유니티 에디터에서 접근할 수 있도록 하겠습니다. 그리고 그 아래에, 프리팹으로부터 생성된 총알을 담을 변수도 하나 더 선언해 줍니다.

BulletLauncher.cs

```
public class BulletLauncher : MonoBehaviour
{
    [SerializeField]
    Bullet bulletPrefab;
    Bullet bullet;

    // Update is called once per frame
    void Update()
    {

    }

    public void OnFireButtonPressed(Vector3 position)
    {
        Debug.Log("Fired a bullet!" + position);
    }
}
```

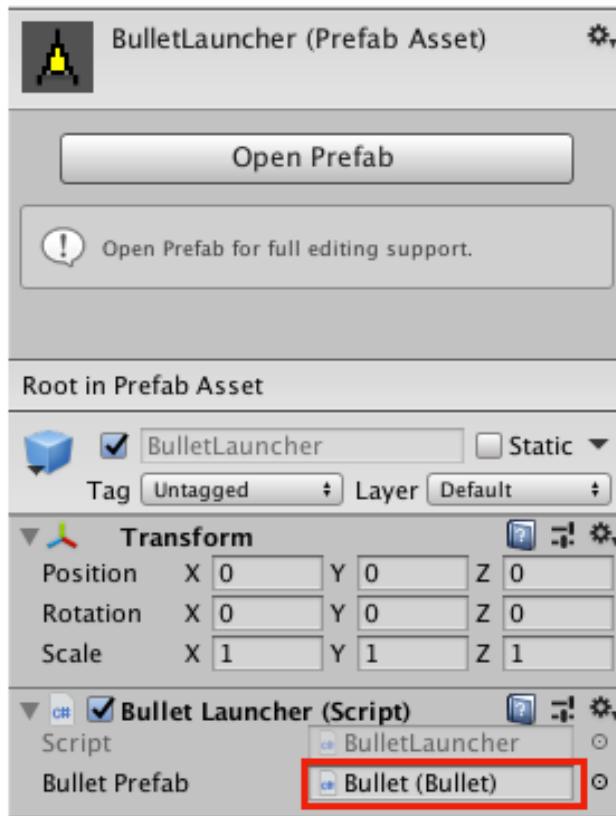
다음으로 해야 할 것은 플레이어가 마우스를 클릭하면 총알을 생성하는 것입니다. 이를 위해 OnFireButtonPressed() 함수 안에 다음과 같이 총알의 인스턴스를 생성하고, 마우스 클릭 지점에 총알을 위치시키는 코드를 작성해 넣도록 하겠습니다.

BulletLauncher.cs

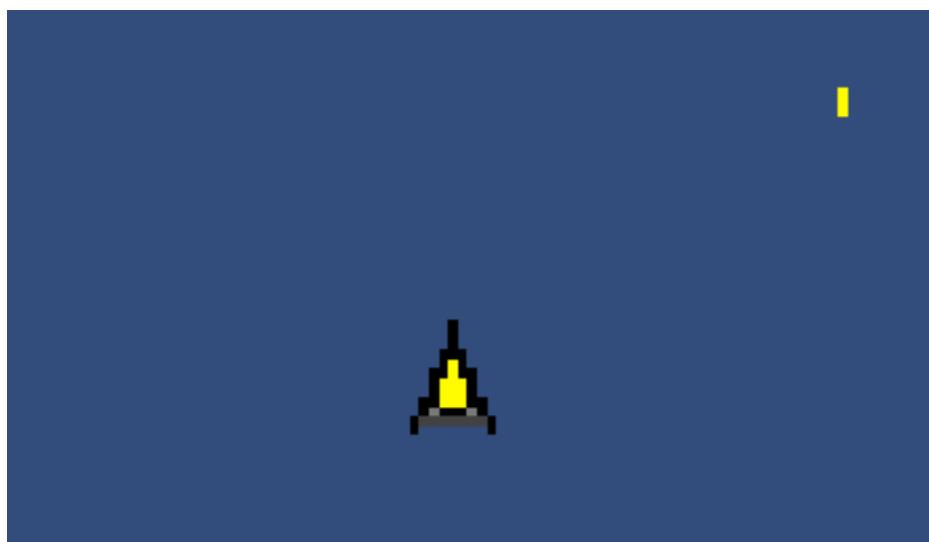
```
public void OnFireButtonPressed(Vector3 position)
{
    Debug.Log("Fired a bullet!" + position);

    bullet = Instantiate(bulletPrefab);
    bullet.transform.position = position;
}
```

이제 유니티 에디터로 가서, BulletLauncher 게임 오브젝트를 선택합니다. 다음으로 하단의 Prefabs 폴더에서 Bullet 프리팹을 선택하고, 이것을 드래그해서 인스펙터에 노출된 bulletPrefab 에 연결해 주겠습니다.



이제 준비가 끝났으니 플레이 버튼을 눌러 게임을 실행해 보겠습니다. 이 상태에서 마우스로 화면을 클릭하면, 총알이 생성되어 위로 날아가는 것을 확인할 수 있습니다. 문제는 총알이 총알 발사대(BulletLauncher)에서 만들어지는 것이 아니라 마우스 클릭지점에 만들어진다는 점입니다. 그럼 이 부분을 수정해서 총알이 총알 발사대에서 만들어지도록 해 보겠습니다.



[동영상 예제 파일명: 021_instantiate_bullet_by_launcher.mp4]

총알이 총구에서 발사되도록 하자

이제 마우스 클릭시, 총알 발사대(BulletLauncher)의 특정 위치(총구)에서 총알이 생성된 뒤, 이 총알이 마우스 클릭 지점을 겨냥해서 날아가도록 해 보겠습니다.
이를 위해서 먼저 해야 할 일은 총구의 위치를 담을 변수를 선언하는 것입니다.
firePosition이라는 Transform 타입의 변수를 선언한 뒤, 이를 인스펙터에 노출시키기 위해 [SerializeField] 어트리뷰트를 붙이겠습니다.

BulletLauncher.cs

```
public class BulletLauncher : MonoBehaviour
{
    [SerializeField]
    Bullet bulletPrefab;
    Bullet bullet;

    [SerializeField]
    Transform firePosition;

    // Update is called once per frame
    void Update()
    {

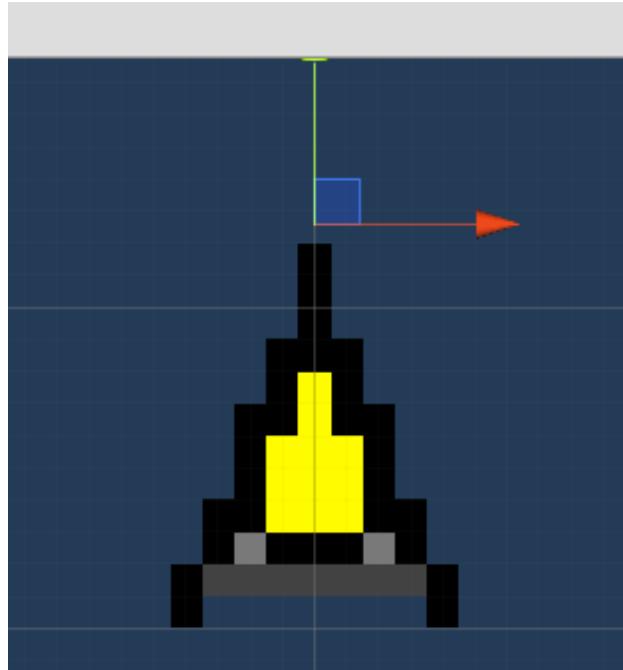
    }

    public void OnFireButtonPressed(Vector3 position)
    {
        Debug.Log("Fired a bullet!" + position);

        bullet = Instantiate(bulletPrefab);
        bullet.transform.position = position;
    }
}
```

이제 유니티 에디터로 가서, 하단 [Prefabs] 폴더에서 BulletLauncher 프리팹을 선택하겠습니다.

그리고 인스펙터에서 프리팹 편집 모드로 들어가서, BulletLauncher 아래에 빈 게임 오브젝트(Empty GameObject)를 자식 오브젝트로 생성하여 총알이 생성될 위치를 만들고 이름을 FirePosition으로 변경한 뒤, 이를 인스펙터에 노출된 변수 firePosition에 드래그해서 연결하도록 하겠습니다. 그리고 나서 프리팹 편집 모드를 빠져 나오겠습니다. (이 챕터 맨 마지막에 제시된 동영상 예제를 참고하시기 바랍니다)



이제 코드로 돌아가겠습니다.

BulletLauncher.cs 를 열어서 OnFireButtonPressed() 함수로 가신 다음에, bullet.transform.position 에 기존의 전달받은 position 값이 아니라 firePosition 의 position 값을 지정해 주면 되겠습니다. 이를 위해서 먼저 기존 코드에서 position 부분을 지워 줍니다.

BulletLauncher.cs

```
public void OnFireButtonPressed(Vector3 position)
{
    Debug.Log("Fired a bullet!" + position);

    bullet = Instantiate(bulletPrefab);
    bullet.transform.position = position;
}
```

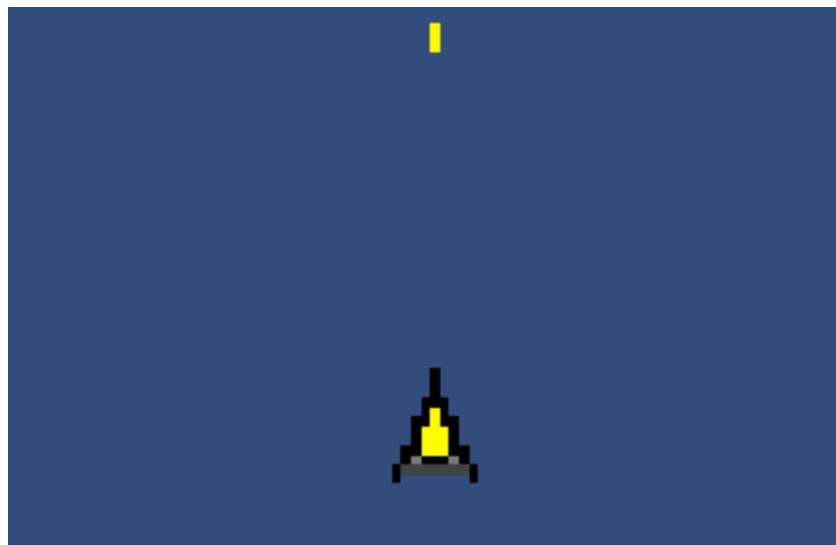
그리고 나서 다음과 같이 firePosition 의 position 값으로 해당 부분을 교체해 주면, 이제부터 마우스를 클릭할 때마다 총알이 총알 발사대(BulletLauncher)의 firePosition 위치에서 생성되어 날아가게 됩니다.

BulletLauncher.cs

```
public void OnFireButtonPressed(Vector3 position)
{
    Debug.Log("Fired a bullet!" + position);

    Bullet bullet = Instantiate(bulletPrefab);
    bullet.transform.position = firePosition.position;
}
```

코딩이 끝났으니 이제 유니티 에디터로 가서 테스트를 해 보겠습니다. 플레이 버튼을 누르신 뒤, 화면의 아무 곳이나 마우스 커서를 대고 클릭하시면, 보시는 것처럼 총알이 총구 부분에서 생성되어 위로 날아가는 것을 확인하실 수 있습니다.



[동영상 예제 파일명: 022_shoot_bullet_from_launcher.mp4]

총알이 클릭 지점으로 날아가게 하자

이제까지의 작업으로 총알이 총구에서 생성되는 것까지는 잘 되지만, 아직까지는 그 냥 위를 향해 날아갈 뿐입니다. 따라서 이번에는 발사된 총알이 마우스 클릭 지점을 향해 날아가게 해 보겠습니다. 이를 위해서 해야 할 작업은 간단합니다. 총알은 무조건 전방으로 날아가게 만들어져 있기 때문에, 총알의 위쪽 방향(머리 부분)이 마우스 클릭 지점을 가리키도록 총알을 회전시키기만 하면 됩니다. 그럼 이를 위한 함수를 하나 만들어 보겠습니다.

먼저 Bullet.cs 에 다음과 같이 Activate() 함수를 만들도록 하겠습니다. 이 함수는 두 개의 인자를 받습니다. 첫번 째 인자는 총알이 생성되는 지점으로, 이름을 startPosition 이라고 하겠습니다. 두번 째 인자는 총알이 도달해야 할 목적지, 다시 말해서 마우스 커서를 클릭한 지점의 좌표값으로, 이름을 targetPosition 이라고 하겠습니다.

```

public class Bullet : MonoBehaviour
{
    [SerializeField]
    float moveSpeed = 5f;

    // Use this for initialization
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        transform.position += transform.up * moveSpeed * Time.deltaTime;
    }

    public void Activate(Vector3 startPosition, Vector3 targetPosition)
    {

    }
}

```

이 함수에서 제일 먼저 해야 할 일은, startPosition 값을 전달 받아서 총알의 위치를 이곳으로 지정하는 것입니다.

```

public void Activate(Vector3 startPosition, Vector3 targetPosition)
{
    transform.position = startPosition;
}

```

다음으로 해야 할 일은 마우스 클릭 지점의 좌표(targetPosition)를 향해 총알이 회전하도록 하는 것입니다. 이를 위해 Quaternion.LookRotation() 함수를 사용할 예정인데, 이 함수를 사용하기 위해서는 먼저 총알의 현재 위치에서 마우스 클릭 지점을 향하는 방향 벡터를 구해야 합니다.

보통 방향 벡터는 목표 지점의 좌표에서 출발 지점의 좌표를 뺀 뒤, 이 벡터의 사이즈를 1로 만들기 위한 정규화(Normalize)를 진행함으로써 얻게 됩니다. 따라서 다음과 같이 코드를 작성하겠습니다. (이 부분은 간단한 벡터 수학인데, 좀 더 깊이 들어가시려면 수학을 따로 공부하셔야 합니다. 하지만 간단한 게임 제작에 필요한 수학 공식은 그냥 이런 식으로 필요할 때마다 외워서 사용하시면 됩니다.)

Bullet.cs

```
public void Activate(Vector3 startPosition, Vector3 targetPosition)
{
    transform.position = startPosition;
    Vector3 dir = (targetPosition - startPosition).normalized;
}
```

다음에 해야 할 일은 Quaternion.LookRotation() 함수를 이용해서 총알이 마우스 클릭 지점을 향해 회전하기 위한 회전값(rotation)을 구하면 됩니다. 이 회전값(rotation)은 다음 코드와 같이 Quaternion.LookRotation(transform.forward, dir) 을 이용하여 구할 수 있습니다.

참고로 Quaternion.LookRotation() 첫번 째 인자로 forward 벡터 값을, 두번 째 인자로 up 벡터 값을 받아서 rotation 값을 구해 리턴하는 함수입니다. 현재 우리가 구한 dir 은 총알의 위쪽(up) 부분이 가리킬 방향이므로 두번 째 인자로 전달한 것입니다.

이제 구해진 rotation 값을 총알의 transform.rotation 에 바로 대입하면, 총알의 위쪽 부분(전방)이 클릭한 지점을 가리키도록 회전하게 됩니다.

```
public void Activate(Vector3 startPosition, Vector3 targetPosition)
{
    transform.position = startPosition;
    Vector3 dir = (targetPosition - startPosition).normalized;
    transform.rotation = Quaternion.LookRotation(transform.forward, dir);
}
```

이제 BulletLauncher.cs 로 이동한 뒤, OnFireButtonPressed() 함수로 가 보겠습니다.

이곳을 보시면 총알을 생성한 뒤 포지션을 firePosition.position 으로 지정해 주는 부분이 있는데, 이 코드를 삭제하겠습니다.

```
public void OnFireButtonPressed(Vector3 position)
{
    Debug.Log("Fired a bullet!" + position);

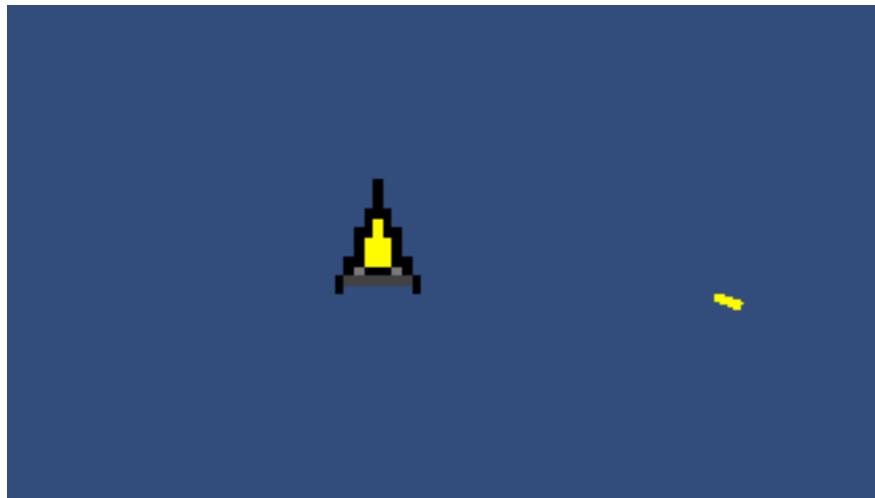
    Bullet bullet = Instantiate(bulletPrefab);
    bullet.transform.position = firePosition.position;
}
```

그리고 나서 다음과 같이, 방금 만든 Activate() 함수를 사용하는 방식으로 새로운 코드를 추가하도록 하겠습니다.

```
public void OnFireButtonPressed(Vector3 position)
{
    Debug.Log("Fired a bullet!" + position);

    Bullet bullet = Instantiate(bulletPrefab);
    bullet.Activate(firePosition.position, position);
}
```

이제 모든 게 잘 되었는지 유니티 에디터로 가서 테스트해 보겠습니다. 플레이 버튼을 눌러서 게임을 실행한 다음에 마우스로 화면을 클릭하면 총알이 만들어져서 해당 위치로 방향을 돌려 날아가는 것을 확인하실 수가 있습니다.



[동영상 예제 파일명: 023_make_bullet_move_to_click_position.mp4]

오브젝트 풀링이 적용된 Factory 만들기

앞에서 총알을 생성할 때는 `Instantiate()` 명령어를 사용하여 프리팹으로부터 총알 게임 오브젝트를 생성하는 방식을 사용했습니다. 이런 방식은 당장은 편리하지만 효율적인 방법은 아닙니다. 특히 총알과 같이 생성과 파괴를 자주해야 하는 게임 오브젝트를 다룰 때는, 매번 총알이 필요할 때마다 `Instantiate()` 명령어로 총알을 만들어내는 방식 보다는 총알 생성을 전담하는 `Factory` 와 같은 클래스를 만들어, 이 클래스가 필요한 게임 오브젝트의 생성과 제거를 전담하도록 하는 것이 좋습니다.

또한 생성된 총알을 한번 사용하고 없애 버리는 것이 아니라, 필요할 때마다 재활용할 수 있도록 오브젝트 풀링(Object Pooling) 기법을 이용하는 것도 중요합니다. 오브젝트 풀링을 사용하면 전체적으로 메모리를 절약할 수 있을 뿐 아니라 가비지 컬렉터(Garbage Collector)가 불필요하게 자주 작동하는 것을 피할 수가 있어서 전체적인 성능에 도움이 되기 때문입니다.

유니티 입문서들 중에는 오브젝트 풀링을 싱글톤(Singleton) 방식으로 구현해서 언제 어디서나 쉽게 접근할 수 있는 방식으로 만드는 경우가 많습니다. 하지만 이 책에서는 싱글톤(Singleton)과 같은 글로벌 참조를 최대한 피할 예정이므로 이 방법을 사용하지 않을 것입니다. 대신, 오브젝트 풀링에 기반을 둔 하나의 클래스(`Factory.cs`)를 만든 뒤, 필요할 때마다 이 클래스의 인스턴스를 만들어 사용하는 방법을 사용하도록 하겠습니다. 실제 코딩 과정을 보지 않고, 지금의 설명만으로는 제가 어떻게 하려고 하는지 파악하시기 힘들테니까, 이해가 안된다고 고민하지 마시고 일단은 제가 제시

하는 방법을 쪽 따라 오시기 바랍니다. 그 과정에서 저의 의도나 사용하는 방법에 대해서도 자연스럽게 이해하시게 될 것입니다.

재활용 오브젝트 전용 클래스인 RecycleObject 를 만들자

저의 의도는 하나의 Factory 클래스를 만든 뒤, 여기에서 생성된 다양한 개별 인스턴스들을 이용하여 총알, 총알 폭발 효과, 빌딩 파괴 효과 등을 각각 관리하는 것입니다. 이렇게 하기 위해서는 총알과 같은 모든 재사용 가능한 클래스들이 하나의 클래스에서 파생되도록 하는 것이 편리합니다. 따라서 RecycleObject 라는 이름의 MonoBehaviour 의 파생 클래스를 만든 뒤, 이것을 앞으로 만들 모든 “재활용 가능한 클래스”들의 부모 클래스로 지정하도록 하겠습니다.

이를 위해 유니티 C# 스크립트 파일을 하나 만들고, 이름을 RecycleObject.cs 으로 지정하도록 하겠습니다. 비쥬얼 스튜디오를 이용하여 RecycleObject.cs 를 열어 보시면 다음과 같이 되어 있을 것입니다.

RecycleObject.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class RecycleObject : MonoBehaviour
{
    // Use this for initialization
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
    }
}
```

이제 Bullet.cs 로 가셔서, 상속 관계를 다음과 같이 기존의 MonoBehaviour 에서 RecycleObject 로 바꾸도록 하겠습니다. 먼저 MonoBehaviour 를 지워 줍니다.

```
Bullet.cs  
public class Bullet : MonoBehaviour  
{  
    [SerializeField]  
    float moveSpeed = 5f;  
  
    ...  
}
```

그리고 나서 RecycleObject 를 부모 클래스로 바꿔 줍니다.

```
Bullet.cs  
public class Bullet : RecycleObject  
{  
    [SerializeField]  
    float moveSpeed = 5f;  
}
```

[동영상 예제 파일명: 024_create_recycleobject_class.mp4]

다음으로는 앞으로 우리가 재활용 가능한 게임 오브젝트들을 만들 때 사용할 Factory라는 클래스를 만들어 보겠습니다. 먼저 Factory.cs라는 이름의 유니티 C# 클래스 파일을 하나 만들고 비쥬얼 스튜디오로 열어 봅니다.

```
Factory.cs
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Factory : MonoBehaviour
{
    // Use this for initialization
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

그런데 Factory 클래스는 MonoBehaviour 와 상관 없는 순수한 C# 클래스로 만들 예정입니다. 따라서 다음과 같이 MonoBehaviour 관련된 부분들을 모두 삭제하겠습니다.

```
Factory.cs
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Factory : MonoBehaviour
{
    // Use this for initialization
    void Start()
    {
    }
}
```

```
}

// Update is called once per frame
void Update()
{
}

}
```

다음으로는 재사용 가능한 오브젝트들을 관리할 List 를 하나 만들겠습니다. 다음과 같이 pool 이라는 이름의 List<RecycleObject> 를 하나 생성합니다.

```
Factory.cs

public class Factory
{
    List<RecycleObject> pool = new List<RecycleObject>();
}
```

그 다음에는 이 오브젝트 풀의 기본 사이즈를 지정하기 위한 int 타입의 변수 defaultPoolSize 를 선언하겠습니다. 예를 들어 defaultPoolSize가 5이면, 재사용을 위한 새로운 오브젝트를 생성할 때 5개 단위로 만들게 될 것입니다.

```
Factory.cs

public class Factory
{
    List<RecycleObject> pool = new List<RecycleObject>();
    int defaultPoolSize;
}
```

다음으로 해야 할 일은 재사용 가능한 오브젝트의 프리팹을 담을 변수 prefab 을 선언하는 것입니다. 이 prefab 은 일반적인 MonoBehaviour 클래스에서와 달리 인스펙터를 이용해서 직접적으로 참조를 전달하지 않을 것입니다. 대신에 디펜던시 인젝션 (Dependency Injection) 을 사용하여 Factory 클래스의 인스턴스를 만들 때 생성자를

통해 외부에서 주입할 예정입니다. 꼭 이런 방법으로 할 필요는 없지만, 저는 이 방법을 통해 ‘생성자를 통한 디펜던시 인젝션(의존성 주입)’이 어떤 것인지를 보여 드리려고 합니다. 일단은 다음과 같이 prefab이라는 이름의 변수를 선언하겠습니다.

```
Factory.cs  
public class Factory  
{  
    List<RecycleObject> pool = new List<RecycleObject>();  
    int defaultPoolSize;  
    RecycleObject prefab;  
}
```

이제 생성자를 만들어야 합니다. 다음과 같이 생성자를 만들고, 이 생성자에서 2개의 인자를 받도록 하겠습니다. 첫번 째는 RecycleObject 타입의 prefab 으로, 이것을 이용해서 우리는 재활용 가능한 게임 오브젝트들을 만들어낼 것입니다. 두번 째 인자는 int 타입의 defaultPoolSize 로서, 기본 값을 5로 지정하겠습니다. 이후의 코드를 통해, 여기에 지정된 개수만큼의 게임 오브젝트를 한번에 만들어내게 될 것입니다.

```
Factory.cs  
public class Factory  
{  
    List<RecycleObject> pool = new List<RecycleObject>();  
    int defaultPoolSize;  
    RecycleObject prefab;  
  
    public Factory(RecycleObject prefab, int defaultPoolSize = 5)  
    {  
    }  
}
```

다음으로는 생성자를 통해 주입된 prefab 과 defaultPoolSize를 Factory 클래스의 내부 변수에 각각 저장하는 코드를 작성하겠습니다. 이런 방식이 바로 “생성자를 이용한

디펜던시 인젝션(의존성 주입)"의 기본 패턴입니다. 이 패턴을 꼭 기억해 두시기 바랍니다.

Factory.cs

```
public Factory(RecycleObject prefab, int defaultPoolSize = 5)
{
    this.prefab = prefab;
    this.defaultPoolSize = defaultPoolSize;
}
```

다음으로는 Debug.Assert()를 이용하여, 만약에 prefab 이 null 인지를 확인하겠습니다. 생성자는 클래스의 인스턴스를 만드는 과정에서 100% 자동으로 실행됩니다. 따라서 생성자를 통해 prefab 을 주입받기로 했는데, 이 값이 제대로 들어오지 않고 null 값이 들어온다면 이후의 모든 로직에 문제가 생기게 됩니다. 따라서 null 값이 들어왔는지 여부를 즉각적으로 감지하고, 그 경우 경고나 예외를 발생시키는 작업이 필요합니다. 일반적으로는 이 상황에서 예외(Exception)를 발생시켜서 프로그램 실행을 중단시키지만, 지금은 개발상의 편의를 위해 그냥 콘솔창에 경고를 띠우고 게임을 일시정지시키는 작업만 하겠습니다.

이를 위해서는 유니티의 Debug.Assert() 함수를 사용하면 됩니다. 함수의 첫번 째 인자로 내가 기대하는 조건(this.prefab != null)과, 그 조건이 거짓인 경우 표시할 경고 메시지("prefab is null!") 을 넣어 주면 됩니다. 이 경우, Factory 의 인스턴스가 생성되는 순간 자동으로 prefab 이 null 인지 여부를 체크해서, null 일 경우 유니티의 콘솔창에 경고 메시지를 출력시킵니다. (유니티 콘솔창에서 [Error Pause] 버튼을 활성화 한 경우에는 경고와 함께 게임이 일시 정지합니다.)

```
public Factory(RecycleObject prefab, int defaultPoolSize = 5)
{
    this.prefab = prefab;
    this.defaultPoolSize = defaultPoolSize;

    Debug.Assert(this.prefab != null, "Prefab is null!");
}
```

[동영상 예제 파일명: 025_factory_class_creation_1.mp4]

풀(pool) 생성 함수를 만들자

다음으로 해야 할 일은 생성자를 통해 외부에서 주입받은 prefab 과 defaultPoolSize 를 토대로 오브젝트 풀을 만들기 위한 함수, CreatePool() 를 만드는 것입니다. 다음과 같이 void 타입으로 CreatePool() 함수를 만들겠습니다.

```
Factory.cs
public Factory(RecycleObject prefab, int defaultPoolSize = 5)
{
    this.prefab = prefab;
    this.defaultPoolSize = defaultPoolSize;

    Debug.Assert(this.prefab != null, "Prefab is null!");
}

void CreatePool()
{
}

}
```

이제 CreatePool() 함수 안에, for 루프를 하나 작성해 넣도록 하겠습니다.

```
Factory.cs
void CreatePool()
{
    for (int i = 0; i < defaultPoolSize; i++)
    {

    }
}
```

그리고 나서, for 루프 안에 RecycleObject 타입의 로컬 변수인 obj 를 선언한 다음, 프리팹의 인스턴스를 하나 생성해서 여기에 할당합니다. 이 때, as RecycleObject 를 붙여 안전하게 캐스팅하는 것이 중요합니다.

```

void CreatePool()
{
    for (int i = 0; i < defaultPoolSize; i++)
    {
        RecycleObject obj = GameObject.Instantiate(prefab) as RecycleObject;
    }
}

```

다음으로는 SetActive(false) 명령어를 이용하여, 이렇게 생성된 게임 오브젝트들을 비활성시켜야 합니다. 만들어진 게임 오브젝트를 당장 사용할 것이 아니므로, 일단은 비활성화시켜 두는 것입니다. 이렇게 비활성화한 게임 오브젝트들은 나중에 사용할 때가 되면, 그 때 다시 활성화시킬 것입니다.

```

void CreatePool()
{
    for (int i = 0; i < defaultPoolSize; i++)
    {
        RecycleObject obj = GameObject.Instantiate(prefab) as RecycleObject;
        obj.gameObject.SetActive(false);
    }
}

```

이제 만들어진 게임 오브젝트를 리스트인 pool에 추가합니다. 이런 식으로 루프를 한번 돌고 나면, defaultPoolSize에 정의된 숫자만큼의 게임 오브젝트가 만들어져 pool에 보관될 것입니다.

```
void CreatePool()
{
    for (int i = 0; i < defaultPoolSize; i++)
    {
        RecycleObject obj = GameObject.Instantiate(prefab) as RecycleObject;
        obj.gameObject.SetActive(false);
        pool.Add(obj);
    }
}
```

[동영상 예제 파일명: 026_factory_class_creation_2.mp4]

필요한 오브젝트를 요청하는 Get 함수 생성

이제 이렇게 만들어진 오브젝트를 필요할 때마다 활성화시켜서 가져오는 함수 Get()을 만들어 보겠습니다. 먼저 RecycleObject 를 리턴하는 public 타입의 함수인 Get()을 다음 예제와 같이 생성합니다.

```
Factory.cs
public class Factory
{
    ...
    void CreatePool()
    {
        for (int i = 0; i < defaultPoolSize; i++)
        {
            RecycleObject obj = GameObject.Instantiate(prefab) as RecycleObject;
            obj.gameObject.SetActive(false);
            pool.Add(obj);
        }
    }

    public RecycleObject Get()
    {
    }
}
```

이렇게 만든 Get() 함수를 통해서 필요한 게임 오브젝트를 요청하면, Factory는 해당 게임 오브젝트가 현재 미리 만들어져 있는지를 확인한 뒤, 만들어 놓은 게 있으면 그 것을 찾아서 리턴하고 그렇지 않은 경우에는 CreatePool() 함수를 먼저 실행한 다음에 요청받은 게임 오브젝트를 리턴하게 될 것입니다.

따라서 Get() 함수가 호출되었을 때 가장 먼저 해야 할 것은 pool에 재활용할 게임 오브젝트가 남아 있느냐를 체크하는 것입니다. 이것은 pool.Count를 통해서 확인할 수 있습니다. pool.Count가 0이면, 다시 말해서 pool에 아무 것도 없다면, CreatePool()을 호출해서 필요한 게임 오브젝트들을 새로 만드는 것입니다.

Factory.cs

```
public RecycleObject Get()
{
    if (pool.Count == 0)
    {
        CreatePool();
    }
}
```

다음으로는 pool 리스트 안에 들어 있는 재활용 가능한 게임 오브젝트 중에서 맨 뒤에 있는 오브젝트를 꺼내어 리턴합니다. 앞에서부터 꺼내지 않고 뒤에서부터 꺼내는 이유는, 앞에서 꺼낼 경우 리스트의 요소들을 재배열하는 내부 과정이 필요하고, 이로 인해 퍼포먼스에 아주 작은 영향을 주기 때문입니다. 사실 이 영향이라는 것이 현재는 감지하기 어려울 정도로 미미하겠지만 나중에 프로젝트 규모가 커질 경우를 생각하면, 가능하면 하나라도 최적화시켜 주는 것이 좋습니다. (이 부분에 대해 자세히 이해하시려면 자료구조에 대한 공부를 하셔야 하는데, 여기에서는 그냥 이런 게 있구나 하고 넘어가시기 바랍니다. 단순히 뒤에서부터 꺼내는 것이 좀 더 성능 향상에 도움이 된다는 정도로만 생각하시기 바랍니다.)

이를 위해서 먼저 pool의 마지막 인덱스 값을 구해 온 뒤에 ($pool.Count - 1$) 마지막 인덱스 값입니다. 인덱스는 0부터 시작하기 때문입니다.) 이 인덱스가 가리키는 게임 오브젝트를 pool에서 꺼내 오도록 하겠습니다.

```

public RecycleObject Get()
{
    if (pool.Count == 0)
    {
        CreatePool();
    }

    int lastIndex = pool.Count - 1;
    RecycleObject obj = pool[lastIndex];
}

```

다음으로는 다시 lastIndex 를 이용하여 pool 리스트에서 해당 오브젝트를 제거해 주겠습니다. 이런 식으로 하면, 사용한 오브젝트는 pool 에서 제거되므로, pool 안에는 항상 ‘아직 사용하지 않은’ 오브젝트들만 남아 있게 됩니다. (참고로, 나중에 사용한 오브젝트를 반납하게 되면 그 때 다시 이를 재활용할 수 있게 pool 에 다시 넣어 줄 예정입니다. 일단은 계속 진행하시기 바랍니다.)

```

public RecycleObject Get()
{
    if (pool.Count == 0)
    {
        CreatePool();
    }

    int lastIndex = pool.Count - 1;
    RecycleObject obj = pool[lastIndex];
    pool.RemoveAt(lastIndex);
}

```

pool 에 들어 있는 오브젝트들은 모두 비활성화되어 있는 상태입니다. 따라서 이를 사용하기 위해서는 먼저 활성화를 해 주어야 합니다. 리턴하기 전에 아래 코드와 같이

SetActive() 함수를 이용, 먼저 게임 오브젝트를 활성화시켜 줍니다. 그리고 나서 리턴하면 됩니다.

```
Factory.cs
public RecycleObject Get()
{
    if (pool.Count == 0)
    {
        CreatePool();
    }

    int lastIndex = pool.Count - 1;
    RecycleObject obj = pool[lastIndex];
    pool.RemoveAt(lastIndex);
    obj.gameObject.SetActive(true);
    return obj;
}
```

[동영상 예제 파일명: 027_factory_class_creation_3.mp4]

오브젝트 반납 함수(Restore)를 만들자

다음으로는 한번 사용한 게임 오브젝트를 반환해서 다음에 다시 사용할 수 있게 준비하는 함수를 만들도록 하겠습니다. 다음과 같이 Factory.cs 에 Restore() 라는 이름의 함수를 하나 선언해 줍니다. Restore()은 한번 사용했던 게임 오브젝트를 되돌려 받기 위한 함수이므로, RecycleObject 타입의 인자를 하나 받도록 정의하겠습니다.

```
Factory.cs
public class Factory
{
    ...
    public RecycleObject Get()
    {
        if (pool.Count == 0)
        {
            CreatePool();
        }

        int lastIndex = pool.Count - 1;
        RecycleObject obj = pool[lastIndex];
        pool.RemoveAt(lastIndex);
        obj.gameObject.SetActive(true);
        return obj;
    }

    public void Restore(RecycleObject obj)
    {
    }
}
```

이 함수 안에서도 Debug.Assert() 명령어를 이용하여, 인자로 전달된 obj 가 null 인지 아닌지를 먼저 체크하겠습니다. 만약 null 일 경우에 다음과 같이 경고 메시지가 뜨도록 하겠습니다.

```
public void Restore(RecycleObject obj)
{
    Debug.Assert(obj != null, "Null object to be returned!");
}
```

다음으로는 이 게임 오브젝트를 비활성화 한 다음에, Add() 명령어를 이용하여 pool에 다시 추가하면 됩니다. 그러면 다시 재사용 가능한 상태가 됩니다.

```
public void Restore(RecycleObject obj)
{
    Debug.Assert(obj != null, "Null object to be returned!");
    obj.gameObject.SetActive(false);
    pool.Add(obj);
}
```

[동영상 예제 파일명: 028_factory_class_creation_4.mp4]

완성된 Factory 사용하기

지금까지 만든 Factory는 범용적으로 사용할 수 있게 고안되었습니다. Factory 클래스 자체는 하나이지만 이것의 인스턴스들은 프리팹을 자유롭게 바꿔 가면서 다양한 게임 오브젝트를 만들고, 회수할 수가 있습니다. 우선 총알(Bullet) 생성 단계에 이것을 적용해 보겠습니다.

우선 BulletLauncher.cs 를 열어서, Factory 클래스의 인스턴스를 담을 변수 bulletFactory 를 선언합니다.

```
BulletLauncher.cs
public class BulletLauncher : MonoBehaviour
{
    [SerializeField]
    Bullet bulletPrefab;
    Bullet bullet;

    [SerializeField]
    Transform firePosition;

    Factory bulletFactory;

    // Update is called once per frame
    void Update()
    {
        ...
    }
}
```

다음으로는 Start() 함수를 생성합니다.

BulletLauncher.cs

```
public class BulletLauncher : MonoBehaviour
{
    [SerializeField]
    Bullet bulletPrefab;
    Bullet bullet;

    [SerializeField]
    Transform firePosition;

    Factory bulletFactory;

    void Start()
    {
        ...
    }

    // Update is called once per frame
    void Update()
    {
        ...
    }
}
```

그리고 나서 이 Start() 함수 안에서 new 키워드를 이용하여 Factory의 인스턴스를 생성해 줍니다. Factory 클래스는 생성자를 통해 프리팹을 주입할 수 있도록 만들어져 있으므로, 보시는 것처럼 bulletPrefab 을 인자로 넣어 주도록 하겠습니다.

BulletLauncher.cs

```
void Start()
{
    bulletFactory = new Factory(bulletPrefab);
}
```

이제 총알을 생성하는 곳으로 가서, 기존의 Instantiate() 명령문이 아니라 방금 만든 bulletFactory를 이용하여 총알이 생성되도록 하겠습니다. 먼저 OnFireButtonPressed() 함수를 찾으신 다음에, Instantiate 부분을 삭제하겠습니다.

BulletLauncher.cs

```
public void OnFireButtonPressed(Vector3 position)
{
    Debug.Log("Fired a bullet!" + position);

    Bullet bullet = Instantiate(bulletPrefab);
    bullet.Fire(firePosition.position, position);
}
```

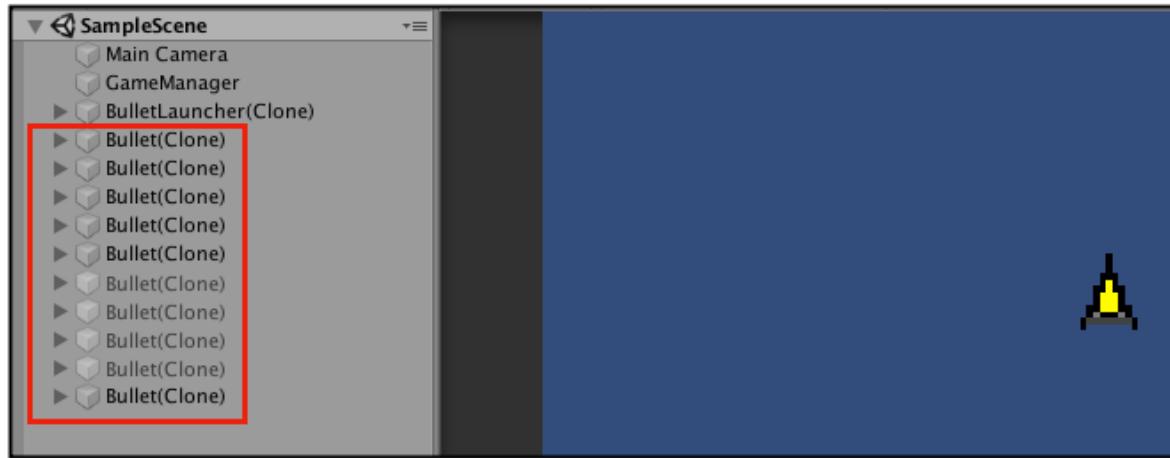
그리고 나서 지워진 코드 대신, 다음과 같이 bulletFactory 를 이용해서 총알을 만들어 사용하는 식으로 코딩하시면 됩니다.

BulletLauncher.cs

```
public void OnFireButtonPressed(Vector3 position)
{
    Debug.Log("Fired a bullet!" + position);

    Bullet bullet = bulletFactory.Get() as Bullet;
    bullet.Fire(firePosition.position, position);
}
```

이제 유니티로 가서 게임을 테스트해 보겠습니다. 플레이 버튼을 눌러서 마우스 클릭을 해 보시면, 다음의 그림과 같이 총알이 Factory 내부에서 구현한 오브젝트 풀링(Object Pooling)에 근거해서 만들어지는 것 확인하실 수 있습니다.



[동영상 예제 파일명: 029_use_factory.mp4]

사용한 총알을 Factory 로 다시 회수하기

앞에서의 테스트를 통해 총알이 오브젝트 풀링 기반으로 만들어지는 것을 확인할 수 있었습니다. 하지만 아직까지 문제가 남아 있는데, 그것은 사용된 총알이 회수되지 않고 있다는 것입니다. 이는 만들어진 총알이 파괴되는 로직을 아직 만들지 않았기 때문입니다.

따라서 이번에는 총알이 목적지(마우스로 클릭한 타겟 지점)에 도달하면 자동으로 파괴되도록 해 보겠습니다. 이를 위해서는 총알이 목적지(타겟 지점)의 위치를 항상 알고 있어야 합니다. 왜냐하면 총알의 현재 위치에서 목적지까지의 거리를 체크해야 하기 때문입니다. 따라서 이를 위한 변수를 준비해야 합니다.

먼저 Bullet.cs 스크립트 파일로 가서, 다음과 같이 targetPosition 이라는 이름의 Vector3 타입의 변수를 선언하겠습니다.

```
Bullet.cs
public class Bullet : RecycleObject
{
    [SerializeField]
    float moveSpeed = 5f;

    Vector3 targetPosition;

    // Use this for initialization
    void Start()
    {
    }

    ...
}
```

다음으로는 Activate() 함수로 가서, 외부에서 인자로 전달된 targetPosition 을 방금 만든 this.targetPosition 변수에 저장하겠습니다.

```
Bullet.cs
public void Activate(Vector3 startPosition, Vector3 targetPosition)
{
    transform.position = startPosition;
    this.targetPosition = targetPosition;
    Vector3 dir = (targetPosition - startPosition).normalized;
    transform.rotation = Quaternion.LookRotation(transform.forward, dir);
}
```

그 다음에는 총알이 목적지에 도달했는지를 체크하기 위한 함수가 필요합니다. 다음과 같이 참,거짓 값을 반환하는 bool 타입의 함수인 IsArrivedToTarget() 을 작성하겠습니다.

```
Bullet.cs
public class Bullet : RecycleObject
{
    ...
    public void Activate(Vector3 startPosition, Vector3 targetPosition)
    {
        transform.position = startPosition;
        this.targetPosition = targetPosition;
        Vector3 dir = (targetPosition - startPosition).normalized;
        transform.rotation = Quaternion.LookRotation(transform.forward, dir);
    }

    bool IsArrivedToTarget()
    {
        ...
    }
}
```

IsArrivedToTarget() 함수가 하는 일은 한 가지입니다. 현재 총알이 마우스로 클릭한 목표지점에 도달했는지를 체크해서 만약 그렇다면 참(true) 값을, 그렇지 않다면 거짓(false)을 반환하는 것입니다. 아래 코드와 같이 Vector3.Distance() 를 이용하여 총알의 현재 위치와 타겟 포지션의 위치 사이의 거리를 구한 뒤, 이 값이 0.1보다 작으면 참, 그렇지 않으면 거짓을 반환하도록 하겠습니다.

Bullet.cs

```
bool IsArrivedToTarget()
{
    float distance = Vector3.Distance(transform.position, targetPosition);
    return distance < 0.1f;
}
```

다음으로는 총알이 목적지를 향해 움직이기 위한 조건을 지정해 주겠습니다. 다음과 같이 총알이 움직일 수 있는지 여부를 판단하기 위한 bool 타입 변수, isActivated 를 선언하고 기본 값을 false 로 해 주겠습니다. (사실 false 로 지정하지 않아도 기본값은 자동으로 false가 됩니다. 하지만 코드 가독성을 위해 이렇게 명시적으로 false 라고 선언해 주었습니다.)

Bullet.cs

```
public class Bullet : RecycleObject
{
    [SerializeField]
    float moveSpeed = 5f;

    Vector3 targetPosition;
    bool isActivated = false;

    ...
}
```

다음으로는 Activate() 함수로 가서 isActivated를 true로 바꿔 주는 코드를 작성합니다.

Bullet.cs

```
public void Activate(Vector3 startPosition, Vector3 targetPosition)
{
    transform.position = startPosition;
    this.targetPosition = targetPosition;
    Vector3 dir = (targetPosition - startPosition).normalized;
    transform.rotation = Quaternion.LookRotation(transform.forward, dir);
    isActivated = true;
}
```

그리고 Update() 함수로 가서, 맨 앞에 isActivated 가 true 일 경우에만 총알이 움직일 수 있게, 조건 체크를 하는 코드를 삽입하겠습니다. 아래와 같이 하면 isActivated 값이 false 인 경우에는 그 다음에 오는 코드들이 실행되지 않습니다.

Bullet.cs

```
void Update()
{
    if (!isActivated)
        return;

    transform.position += transform.up * moveSpeed * Time.deltaTime;
}
```

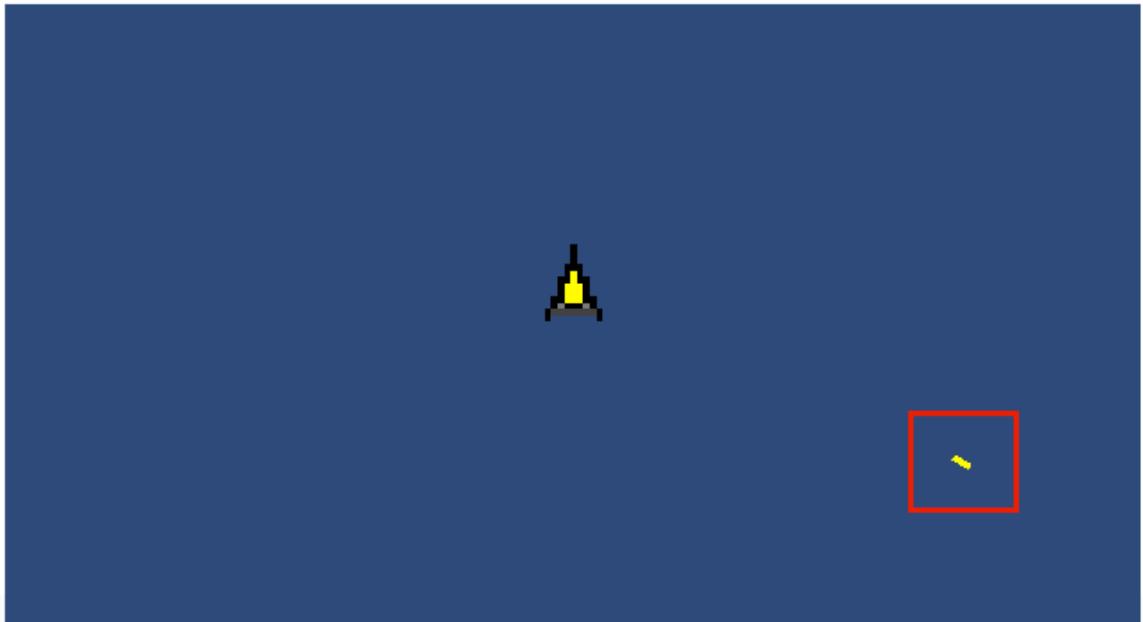
다음으로 목표 지점에 도달했으면 총알의 이동을 정지시키는 코드를 작성하겠습니다. 보시는 것처럼 IsArrivedToTarget() 함수의 값이 true인 경우, 다시 말해서 총알이 목적지에 도달한 경우에는 isActivated 값을 false 로 만들어 총알이 더 이상 이동하지 않게 만듭니다.

```
void Update()
{
    if (!activated)
        return;

    transform.position += transform.up * moveSpeed * Time.deltaTime;

    if (IsArrivedToTarget())
    {
        activated = false;
    }
}
```

이제 유니티 에디터로 가서 게임을 테스트해 보겠습니다. 플레이 버튼을 눌러서 게임을 실행한 뒤, 마우스로 화면 아무 곳이나 마우스를 클릭하면 총알이 클릭 지점 가까이까지 가서 멈추는 것을 확인하실 수 있을 것입니다.



[동영상 예제 파일명: 030_restore_used_bullet_1.mp4]

총알이 목표지점에 도달했는지 이벤트로 알려 주자

이제 총알이 목표 지점에 도달할 경우, 이를 알려 주는 Action 이벤트를 만들도록 하겠습니다. Action을 사용하기 위해서는 다음 예시와 같이 “using System;” 을 네임스페이스에 추가해야 합니다. 그리고 나서 Destroyed 라는 이름의 Action을 생성하되, Bullet 을 인자로 전달할 수 있게 Action<Bullet>으로 타입을 지정합니다. 이는 나중에 이 Destroyed 이벤트를 수신하는 쪽에서, 이벤트 발신자가 누구인지를 알 수 있게 하기 위해서입니다.

Bullet.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Bullet : RecycleObject
{
    [SerializeField]
    float moveSpeed = 5f;

    Vector3 targetPosition;
    bool isActivated = false;

    public Action<Bullet> Destroyed;

    // Use this for initialization
    void Start()
    {
    }

    ...
}
```

이제 Update() 함수로 다시 가서 IsArrivedToTarget() 을 체크하는 조건문으로 이동한 뒤, 방금 만든 Destroyed 이벤트를 발생시키도록 코드를 추가하겠습니다. 먼저 Destroyed 라는 Action 에 연결된 이벤트 수신자 함수들이 있는지 체크한 뒤, 만약 연결된 함수가 있을 경우에 Destroyed 이벤트를 발생시키면서 자기 자신(this)을 인자로 함께 전달합니다. 코드는 다음과 같습니다.

Bullet.cs

```
void Update()
{
    if (!isActivated)
        return;

    transform.position += transform.up * moveSpeed * Time.deltaTime;

    if (IsArrivedToTarget())
    {
        isActivated = false;

        if (Destroyed != null)
        {
            Destroyed(this);
        }
    }
}
```

이제 총알이 파괴되는 순간을 알리는 이벤트를 만들었으므로, 다음으로는 BulletLauncher.cs 로 가서 이 이벤트에 대한 수신 이벤트를 만들도록 하겠습니다. 아래 코드와 같이 OnBulletDestroyed() 라는 함수를 만들되, 이벤트 송신자와 마찬가지로 Bullet 타입의 인자를 받아 들이도록 하겠습니다.

```

public class BulletLauncher : MonoBehaviour
{
    ...
    public void OnFireButtonPressed(Vector3 position)
    {
        ...
    }

    void OnBulletDestroyed(Bullet usedBullet)
    {
        ...
    }
}

```

이제 OnFireButtonPressed()에서 이 이벤트 수신 함수를 bullet.Destroyed 와 연동(바인딩)합니다.

```

public void OnFireButtonPressed(Vector3 position)
{
    Debug.Log("Fired a bullet!" + position);

    Bullet bullet = bulletFactory.Get() as Bullet;
    bullet.Activate(firePosition.position, position);
    bullet.Destroyed += OnBulletDestroyed;
}

```

그리고 나서 OnBulletDestroyed() 이벤트 수신자 함수로 가서, 아까 연결한 이벤트를 해제합니다. 사용한 총알을 회수하기 전에 기존에 연동(바인딩)한 이벤트를 해제하지 않으면 나중에 이 총알이 재활용될 때마다 동일한 이벤트가 계속 덧붙여져서 문제가 발생하게 됩니다. 연결(바인딩)한 이벤트는 반드시 연결 해제(언바인딩)해야 한다는 점을 명심하시기 바랍니다.

BulletLauncher.cs

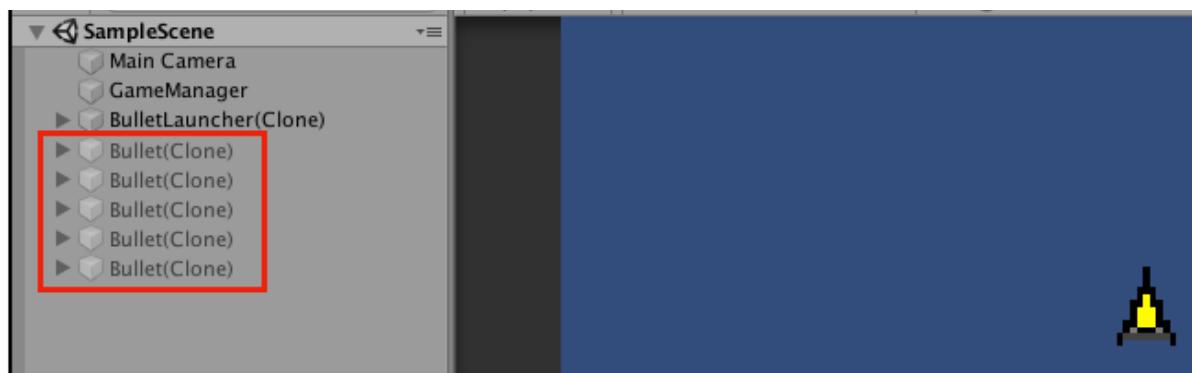
```
void OnBulletDestroyed(Bullet usedBullet)
{
    usedBullet.Destroyed -= OnBulletDestroyed;
}
```

이제 총알을 bulletFactory로 회수하여 다음에 다시 사용할 수 있도록 하겠습니다. 다음의 코드처럼 bulletFactory.Restore() 함수를 이용하여 파괴된 총알을 팩토리로 회수합니다.

BulletLauncher.cs

```
void OnBulletDestroyed(Bullet usedBullet)
{
    usedBullet.Destroyed -= OnBulletDestroyed;
    bulletFactory.Restore(usedBullet);
}
```

이제 유니티 에디터로 가서 테스트 하겠습니다. 플레이 버튼을 누르고 화면을 마우스로 클릭하면 총알이 발사되고, 목적지에 도달한 총알은 자동으로 사라지면서 사용한 총알이 비활성(회수)되는 것을 하이어라기 뷰에서 확인하실 수 있을 것입니다.



[동영상 예제 파일명: 031_restore_used_bullet_2.mp4]

총알 발사 지연 시간(쿨타임) 설정

현재 총알을 발사하고, 목적지에 도달한 총알을 회수하는 데는 문제가 없습니다. 하지만 다음 총알 발사 때까지의 쿨타임이 적용되지 않았기 때문에 총알을 원하는대로 얼마든지 발사하는 것이 가능합니다. 이렇게 되면 날아 오는 미사일을 쉽게 막아낼 수가 있어서 게임이 너무 쉬워집니다. 이를 방지하기 위해, 총알을 한번 발사하고 다음 총알을 발사할 때까지 지연시간(쿨타임)을 계산해서 적용하도록 하겠습니다.

우선 BulletLauncher.cs 로 가서, 아래와 같이 지연시간 적용에 필요한 변수들을 추가하겠습니다. 첫번 째 변수인 fireDelay 는 총알 발사 후 다음 발사까지의 지연 시간을 얼마나 줄 것인지를 지정하는 부분인데, 바로 위에 [SerializeField]라는 어트리뷰트를 적용하여 유니티 에디터의 인스펙터에서 값을 수정할 수 있게 하였습니다. 그 아래의 elapsedTime 은 실제 시간의 흐름을 계산하는 데 사용하는 변수입니다. 그리고 마지막으로 선언한 canShoot 은 bool 타입의 변수로서, 이 값이 참이면 총알을 발사할 수 있고 만약 거짓이면 총알을 발사할 수 없습니다.

```

public class BulletLauncher : MonoBehaviour
{
    [SerializeField]
    Bullet bulletPrefab;
    Bullet bullet;

    [SerializeField]
    Transform firePosition;

    [SerializeField]
    float fireDelay = 0.5f;
    float elapsedFireTime;
    bool canShoot = true;

    Factory bulletFactory;

    void Start()
    {
        bulletFactory = new Factory(bulletPrefab);
    }

    ...
}

```

다음 단계로 진행하기 전에, 먼저 OnFireButtonPressed() 함수 안에 작성해 놓았던 디버그 메시지부터 삭제하겠습니다. 이제 더 필요가 없기 때문입니다.

```

public void OnFireButtonPressed(Vector3 position)
{
    Debug.Log("Fired a bullet!" + position);

    Bullet bullet = bulletFactory.Get() as Bullet;
    bullet.Activate(firePosition.position, position);
    bullet.Destroyed += OnBulletDestroyed;
}

```

다음으로는 canShoot의 값이 true 일 경우에만 OnFireButtonPressed() 함수 내 코드가 실행되도록 조건 체크를 하는 구문을 추가하겠습니다. 아래와 같이 코드를 입력하면, canShoot의 값이 false 일 경우, 그 다음에 있는 코드들을 모두 무시하게 됩니다.

BulletLauncher.cs

```
public void OnFireButtonPressed(Vector3 position)
{
    if (!canShoot)
        return;

    Bullet bullet = bulletFactory.Get() as Bullet;
    bullet.Activate(firePosition.position, position);
    bullet.Destroyed += OnBulletDestroyed;
}
```

이제 총알 발사 후 canShoot를 false로 바꿔 주는 코드를 삽입하겠습니다. 이 코드를 통해 이전에 가능했던 ‘지연 시간 없는 연속 발사’를 방지할 수 있습니다.

BulletLauncher.cs

```
public void OnFireButtonPressed(Vector3 position)
{
    if (!canShoot)
        return;

    Bullet bullet = bulletFactory.Get() as Bullet;
    bullet.Activate(firePosition.position, position);
    bullet.Destroyed += OnBulletDestroyed;

    canShoot = false;
}
```

앞의 OnFireButtonPressed()에서 canShoot이 false로 변했으므로, 이제 일정 시간이 지나면 canShoot를 다시 true로 바꿔주는 코드를 작성해야 합니다. 이를 위해 Update() 함수로 가서, canShoot이 false인지 여부를 체크하는 조건문을 하나 만들겠습니다.

```
void Update()
{
    if (!canShoot)
    {

    }
}
```

그리고 나서 elapsedFireTime += Time.deltaTime 을 통해 경과 시간을 계산하겠습니다.

```
void Update()
{
    if (!canShoot)
    {
        elapsedFireTime += Time.deltaTime;
    }
}
```

다음으로는 경과된 시간(elapsedFireTime)이 fireDelay 에서 지정된 시간을 초과하는지 체크하는 조건문을 작성하겠습니다.

```
void Update()
{
    if (!canShoot)
    {
        elapsedFireTime += Time.deltaTime;
        if (elapsedFireTime >= fireDelay)
        {
            }

    }
}
```

그리고 경과한 시간이 fireDelay 의 값보다 크면 canShoot의 값을 다시 true 로 변경합니다. 이제 canShoot 이 true가 되었으므로 다시 총알을 발사할 수가 있습니다.

BulletLauncher.cs

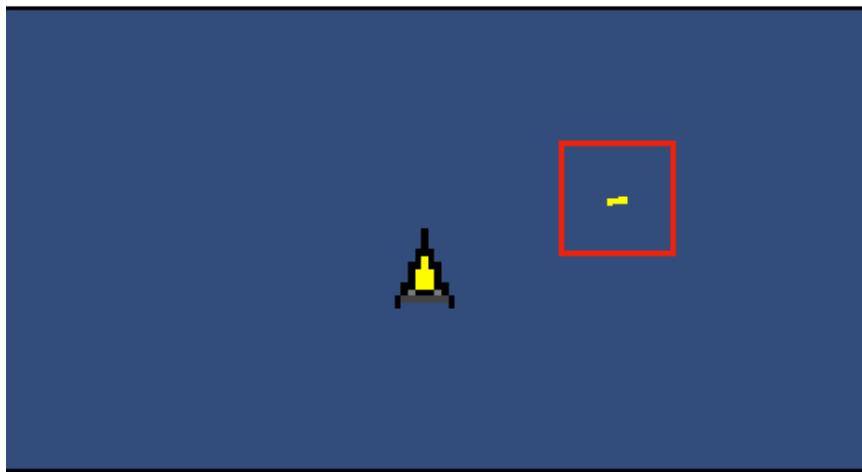
```
void Update()
{
    if (!canShoot)
    {
        elapsedFireTime += Time.deltaTime;
        if (elapsedFireTime >= fireDelay)
        {
            canShoot = true;
        }
    }
}
```

마지막으로 elapsedFireTime 을 0으로 초기화해서 다음 지연 시간 계산에 대비할 수 있게 하면 이 부분의 코딩이 완료됩니다.

BulletLauncher.cs

```
void Update()
{
    if (!canShoot)
    {
        elapsedFireTime += Time.deltaTime;
        if (elapsedFireTime >= fireDelay)
        {
            canShoot = true;
            elapsedFireTime = 0f;
        }
    }
}
```

이제 유니티 에디터로 가서 테스트를 해 보겠습니다. 그러면 다음과 같이, 아무리 마우스 버튼을 빠르게 클릭한다고 해도 총알을 무한정 발사할 수가 없고, 일정 시간이 지난 후에야 다음 총알이 발사되는 것을 확인하실 수 있습니다.



[동영상 예제 파일명: 032_delay_for_bullet_shooting.mp4]

폭발 효과를 만들자

발사한 총알이 목적지에 도달하면 사라지는 기능을 구현했지만, 현재는 그냥 사라지고 말 뿐이라서 너무 멋밋합니다. 총알이 사라지는 순간 이곳에 폭발 효과가 발생하도록 해 보겠습니다.

이를 위해 우선 Explosion.cs라는 이름의 유니티 C# 클래스를 하나 만들겠습니다. 폭발 효과 역시 Factory를 이용해서 재활용할 수 있게 만들 예정이므로, 총알(Bullet)과 마찬가지로 RecycleObject의 파생 클래스가 되도록 해야 합니다. 따라서 다음과 같이 MonoBehaviour라고 쓰여진 부분을 지우겠습니다.

Explosion.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Explosion : MonoBehaviour
{
    // Use this for initialization
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

그리고 나서, 지워진 MonoBehaviour 대신 RecycleObject를 입력해 줍니다.

```
public class Explosion : RecycleObject
{
    // Use this for initialization
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
    }
}
```

다음으로 해야 할 일은, [RequireComponent] 를 이용해서, Explosion.cs 스크립트 파일이 게임 오브젝트에 부착되는 순간 자동으로 CircleCollider2D 와 Rigidbody2D 컴포넌트의 인스턴스가 생성되도록 하는 것입니다.

물론 이렇게 하지 않고 코드 상에서

gameObject.AddComponent<CircleCollider2D>() 와 같은 명령을 이용하여 동적으로 컴포넌트를 생성해도 되지만, 이 경우에는 유니티 에디터의 인스펙터를 이용하여 충돌체크 영역의 크기를 수동으로 조절해 줄 계획이므로 [RequireComponent] 를 이용하는 방법을 택하였습니다.

```
[RequireComponent(typeof(CircleCollider2D), typeof(Rigidbody2D))]  
public class Explosion : RecycleObject  
{  
    // Use this for initialization  
    void Start()  
    {  
    }  
  
    // Update is called once per frame  
    void Update()  
    {  
    }  
}
```

다음으로는 이렇게 자동으로 붙인 컴포넌트들을 할당할 변수들을 선언해야 합니다.

다음과 같이 CircleCollider2D 와 Rigidbody2D 타입의 변수들을 각각 선언합니다.

```
[RequireComponent(typeof(CircleCollider2D), typeof(Rigidbody2D))]  
public class Explosion : RecycleObject  
{  
    CircleCollider2D circle;  
    Rigidbody2D body;  
  
    // Use this for initialization  
    void Start()  
    {  
    }  
  
    // Update is called once per frame  
    void Update()  
    {  
    }  
}
```

다음으로는 필요 없는 유니티 내장 함수를 삭제하겠습니다. 저는 Start() 대신 Awake() 를 사용할 예정이라 다음과 같이 Start() 함수 부분을 삭제하도록 하겠습니다.

```
Explosion.cs
public class Explosion : RecycleObject
{
    CircleCollider2D circle;
    Rigidbody2D body;

    // Use this for initialization
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
    }
}
```

그리고 Awake() 함수를 만들겠습니다.

```
Explosion.cs
public class Explosion : RecycleObject
{
    CircleCollider2D circle;
    Rigidbody2D body;

    void Awake()
    {

    }

    // Update is called once per frame
    void Update()
    {
    }
}
```

참고로, 비쥬얼 스튜디오를 이용해서 Awake() 함수를 만들면 void 앞에 자동으로 private 이라는 키워드가 붙을 것입니다.(동영상 예제 참고) private 이 붙거나 안 붙거나 동일하므로 신경 안 쓰셔도 됩니다.

다음으로는 Awake() 함수에서 GetComponent<T>() 명령어를 이용하여 CircleCollider2D와 Rigidbody를 각각 불러와 앞에서 만들어 놓았던 변수들에 할당하겠습니다.

Explosion.cs

```
public class Explosion : RecycleObject
{
    CircleCollider2D box;
    Rigidbody2D body;

    void Awake()
    {
        circle = GetComponent<CircleCollider2D>();
        body = GetComponent<Rigidbody2D>();
    }

    // Update is called once per frame
    void Update()
    {
    }
}
```

다음으로 해야 할 일은 이를 컴포넌트들의 설정을 변경하는 것입니다. 충돌체의 isTrigger 값을 true로, 리지드바디의 타입은 Kinematic 으로 설정하여 물리 시뮬레이션을 하지 않고 트리거 충돌 감지만하도록 하도록 하면 됩니다.

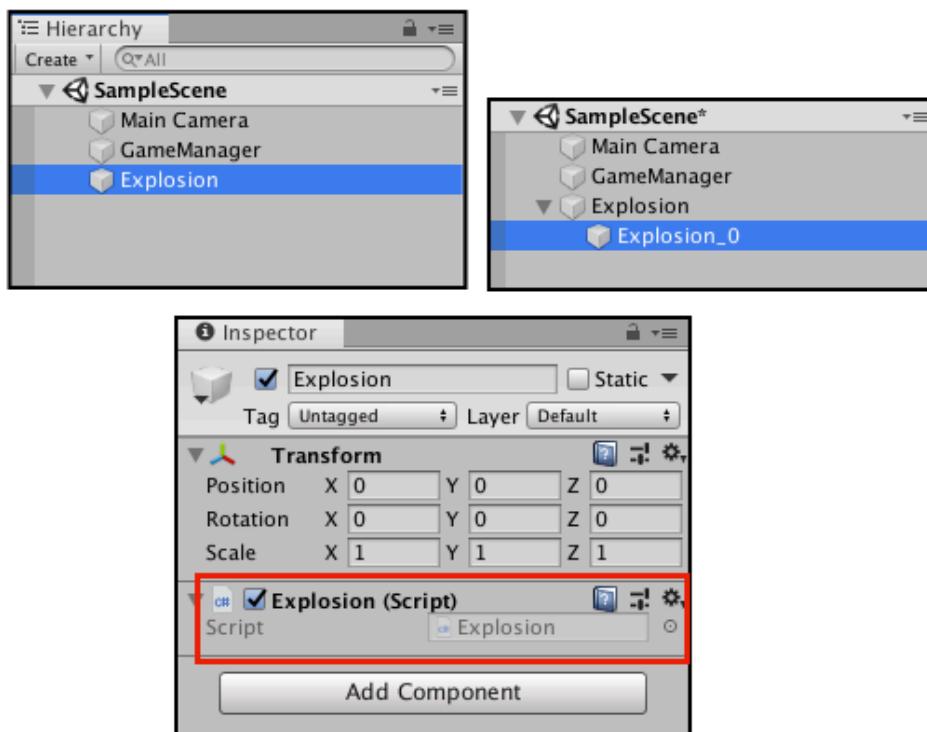
```
void Awake()
{
    circle = GetComponent<CircleCollider2D>();
    body = GetComponent<Rigidbody2D>();

    circle.isTrigger = true;
    body.bodyType = RigidbodyType2D.Kinematic;
}
```

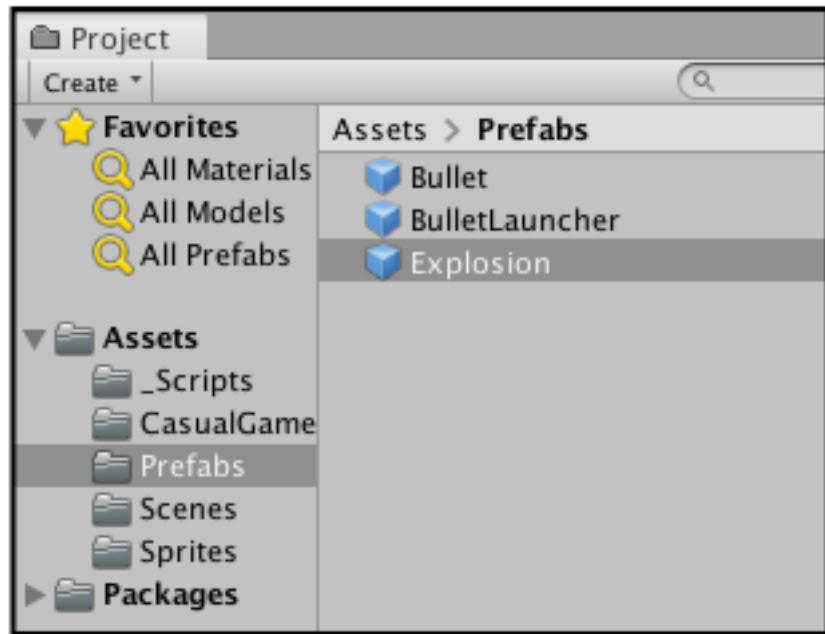
[동영상 예제 파일명: 033_explosion_class_creation.mp4]

Explosion 의 프리팹을 만들자

이제 실제 게임에서 사용할 폭발 효과(Explosion) 프리팹을 만들어 보겠습니다. 먼저 유니티로 가져서 씬에 빈 게임 오브젝트(Empty GameObject)를 하나 만들도록 하겠습니다. 그리고 나서 이름을 Explosion 으로 바꾸고, 하단의 스프라이트 폴더에서 Explosion 스프라이트를 드래그해서 방금 만든 게임 오브젝트의 자식 오브젝트로 넣어 줍니다. 그리고 나서 이 스프라이트 자식 오브젝트의 로컬 포지션을 (0,0,0) 으로 맞추어 주도록 하겠습니다. 그리고 이 상태에서 Explosion 을 다시 선택하고 방금 만든 Explosion.cs 스크립트를 붙여 주면 프리팹을 만들 준비가 모두 끝나게 됩니다.



이제 이렇게 만들어진 Explosion 게임 오브젝트를 마우스로 드래그해서 하단 프로젝트 뷰의 [Prefabs] 폴더에 갖다 넣겠습니다. 프리팹이 만들어졌으니, 씬(Scene)에 만들어 놓았던 Explosion 게임 오브젝트는 이제 삭제하겠습니다.



[동영상 예제 파일명: 034_create_explosion_prefab.mp4]

총알이 목적지에 도달하면 Explosion 생성하기

이제 다시 코딩으로 돌아가겠습니다. 총알이 사라지기 전 마지막으로 위치한 곳에 폭발 효과를 만들어 낼 예정인데, 총알의 마지막 위치를 알고 있는 것은 총알 발사대(BulletLauncher)입니다. 따라서 폭발 효과를 만들어내는 코드 역시 BulletLauncher.cs에 작성하도록 하겠습니다.

폭발 효과(Explosion)를 만들고 사용하는 데에도 역시 Factory를 이용할 예정입니다. Factory를 만들기 위해서는 폭발 효과 프리팹이 준비되어야 하므로, 다음과 같이 프리팹을 저장할 변수를 하나 선언한 뒤, [SerializeField]를 붙여서 인스펙터에 이 변수가 노출되도록 하겠습니다.

BulletLauncher.cs

```
public class BulletLauncher : MonoBehaviour
{
    [SerializeField]
    Bullet bulletPrefab;
    Bullet bullet;

    [SerializeField]
    Explosion explosionPrefab;

    [SerializeField]
    Transform firePosition;

    [SerializeField]
    float fireDelay = 0.5f;
    float elapsedFireTime;
    bool canShoot = true;

    Factory bulletFactory;

    ...
}
```

또한, 폭발 효과를 생성하고 관리할 explosionFactory 라는 Factory 타입의 변수도 다음과 같이 선언하겠습니다.

```
BulletLauncher.cs
public class BulletLauncher : MonoBehaviour
{
    [SerializeField]
    Bullet bulletPrefab;
    Bullet bullet;

    [SerializeField]
    Explosion explosionPrefab;

    [SerializeField]
    Transform firePosition;

    [SerializeField]
    float fireDelay = 0.5f;
    float elapsedFireTime;
    bool canShoot = true;

    Factory bulletFactory;
    Factory explosionFactory;

    ...
}
```

다음으로 해야 할 일은 Start()에서 폭발 효과 Factory를 생성하는 것입니다. 아래 코드와 같이 new 키워드를 이용해서 Factory의 인스턴스를 생성하면서 explosionPrefab 을 인자로 전달해 줍니다.

```
BulletLauncher.cs
void Start()
{
    bulletFactory = new Factory(bulletPrefab);
    explosionFactory = new Factory(explosionPrefab);
}
```

이제 총알이 목적지에 도달해서 사라지는 순간, 이 위치에 폭발 효과(Explosion)를 생성하는 코드를 작성해야 합니다. 이를 위해서, OnBulletDestroyed 이벤트 수신 함수로 갑니다. 여기에는 총알이 파괴되어 회수되는 코드가 들어 있는데, 총알이 회수되기 전에 총알의 마지막 위치를 Vector3 타입의 로컬 변수에 저장하겠습니다.

BulletLauncher.cs

```
void OnBulletDestroyed(Bullet usedBullet)
{
    Vector3 lastBulletPosition = usedBullet.transform.position;
    usedBullet.Exploded -= OnExploded;
    bulletFactory.Restore(usedBullet);

}
```

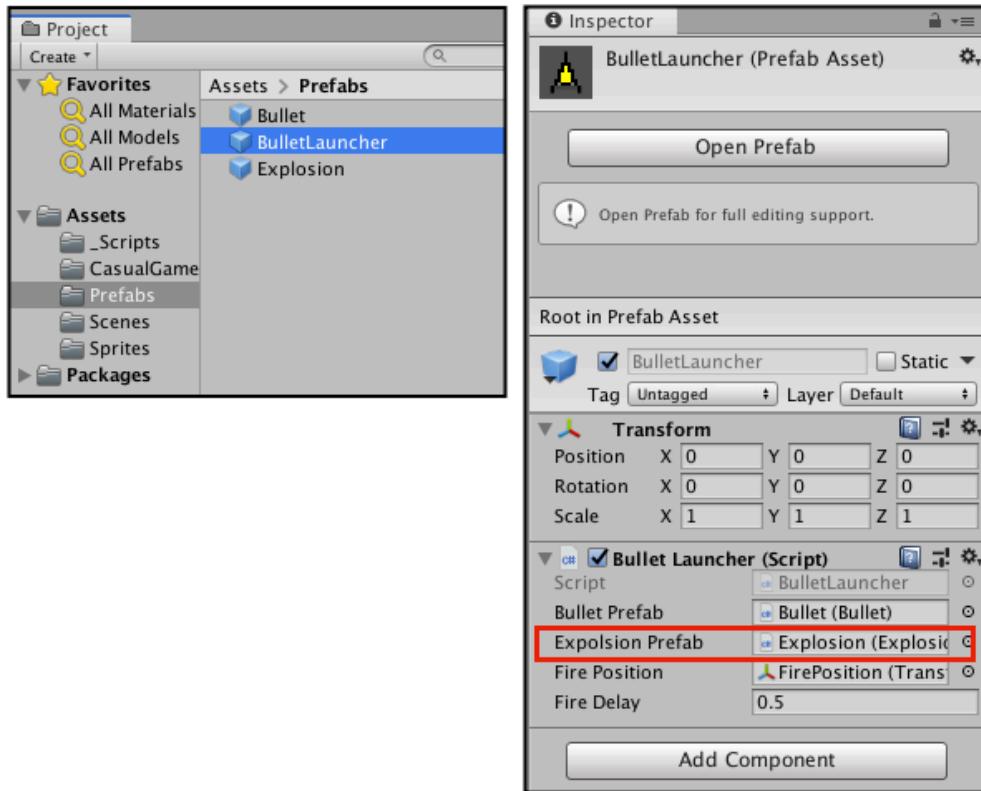
다음으로는 explosionFactory.Get() 함수를 이용하여 팩토리에서 폭발 효과를 하나 가져 온 뒤, 앞에서 저장한 ‘총알의 마지막 위치(lastBulletPosition)’에 위치시키도록 하겠습니다. 참고로 explosionFactory.Get() 함수가 리턴하는 것은 Explosion 이 아니라 RecycleObject 이므로, as Explosion 으로 안전하게 캐스팅해 주어야 합니다.

BulletLauncher.cs

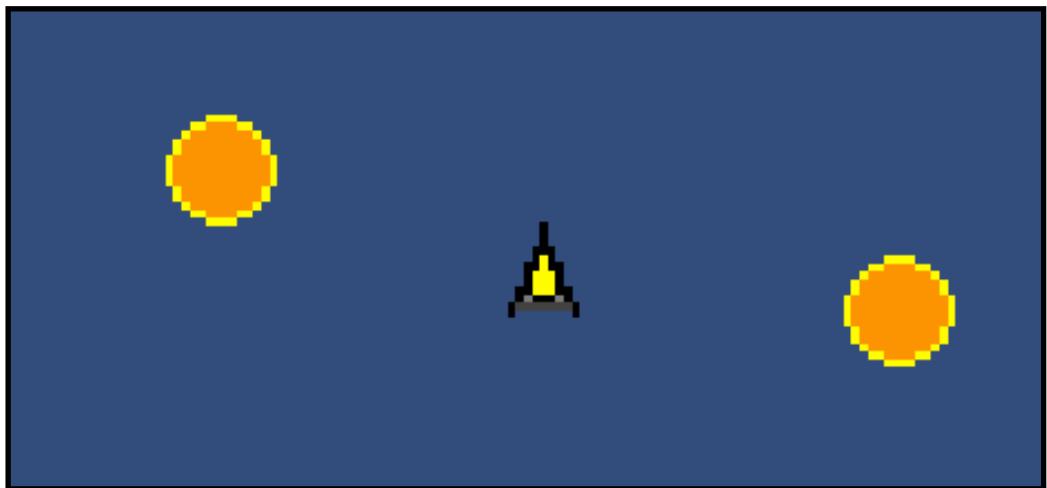
```
void OnBulletDestroyed(Bullet usedBullet)
{
    Vector3 lastBulletPosition = usedBullet.transform.position;
    usedBullet.Exploded -= OnExploded;
    bulletFactory.Restore(usedBullet);

    Explosion explosion = explosionFactory.Get() as Explosion;
    explosion.transform.position = lastBulletPosition;
}
```

이제 유니티 에디터로 가서 테스트해 보겠습니다. 먼저 [prefabs] 폴더로 가서 BulletLauncher 프리팹을 찾아 선택합니다. 그리고 인스펙터에서 BulletLauncher 스크립트를 찾은 뒤, Explosion Prefab 변수에 아까 만든 Explosion 프리팹을 드래그해서 연결해 줍니다.



다음으로 플레이 버튼을 눌러서 테스트해 보겠습니다. 게임을 실행하고 화면을 마우스로 클릭하면 총알이 발사되고, 총알이 사라지는 순간 그 자리에 폭발효과가 나타나는 것을 확인하실 수 있습니다. 하지만 한번 생겨난 폭발 효과가 사라지지 않고 그대로 남아 있습니다. 사용된 폭발 효과를 회수하는 로직을 아직 작성하지 않았기 때문입니다.



[동영상 예제 파일명: 035_create_explosion_on_target_position.mp4]

폭발 효과가 일정 시간 후에 사라지게 하자

이제 Explosion.cs 로 가서, 폭발 효과가 일정 시간이 지난 후 자동으로 사라지게 하는 로직을 만들겠습니다. 먼저 다음과 같이 필요한 변수들을 선언하겠습니다.

timeToRemove	폭발 효과가 얼마의 시간이 지난 후 자동으로 사라지는지를 결정합니다. 유니티 에디터의 인스펙터에서 값을 조정 할 수 있도록 [SerializeField] 어트리뷰트를 붙여 줍니다.
elapsedTime	경과된 시간을 계산하는 데 사용되는 변수입니다.
isActivated	이 값이 활성화되었을 때만 폭발 효과가 사라지기 위한 시간을 계산합니다.

Explosion.cs

```
public class Explosion : RecycleObject
{
    CircleCollider2D circle;
    Rigidbody2D body;

    [SerializeField]
    float timeToRemove = 1f;
    float elapsedTime = 0f;
    bool isActivated = false;

    void Awake()
    {
        circle = GetComponent<CircleCollider2D>();
        body = GetComponent<Rigidbody2D>();

        circle.isTrigger = true;
        body.bodyType = RigidbodyType2D.Kinematic;
    }

    ...
}
```

다음으로는 Update() 함수로 가서 총알이 일정 시간 후에 사라지는 로직을 작성하도록 하겠습니다. isActivated 의 값이 true 인 경우에만 로직을 작동시킬 것이므로 아래와 같은 조건문을 하나 생성합니다.

Explosion.cs

```
void Update()
{
    if (isActivated)
    {
        }
}
```

다음으로는 elapsedTime 을 이용하여 경과된 시간을 계산한 다음에, 이것을 timeToRemove에서 지정된 시간과 비교하겠습니다. elapsedTime 의 값이 timeToRemove 보다 크다면, 시간이 충분히 경과한 것으로 폭발 효과를 제거할 수 있습니다.

Explosion.cs

```
void Update()
{
    if (isActivated)
    {
        elapsedTime += Time.deltaTime;
        if (elapsedTime >= timeToRemove)
        {
            }
    }
}
```

이제 다음 번의 시간 경과 계산을 위해 elapsedTime 을 0으로 초기화해 주겠습니다.

Explosion.cs

```
void Update()
{
    if (isActivated)
    {
        elapsedTime += Time.deltaTime;
        if (elapsedTime >= timeToRemove)
        {
            elapsedTime = 0f;
        }
    }
}
```

다음으로는 Update() 함수 바깥에 DestroySelf() 라는 이름의 함수를 만들고, 이를 Update() 함수의 조건문 안에서 호출해 주겠습니다.

Explosion.cs

```
public class Explosion : RecycleObject
{
    ...
    // Update is called once per frame
    void Update()
    {
        if (isActivated)
        {
            elapsedTime += Time.deltaTime;
            if (elapsedTime >= timeToRemove)
            {
                elapsedTime = 0f;
                DestroySelf();
            }
        }
    }

    void DestroySelf()
    {
    }
}
```

이제 DestroySelf() 함수의 내용을 채우겠습니다. 제일 먼저 해야 할 일은 isActivated를 false로 바꾸는 것입니다. 폭발효과를 회수하기 전에 이 값을 false로 바꿔 주어야 나중에 이것을 재활용할 수 있습니다.

Explosion.cs

```
void DestroySelf()
{
    isActivated = false;
}
```

다음으로는 Debug.Log() 함수를 이용해서 이 게임 오브젝트(폭발 효과)가 파괴되었다는 메시지를 콘솔창에 띄우도록 하겠습니다.

Explosion.cs

```
void DestroySelf()
{
    isActivated = false;
    Debug.Log(gameObject.name + "is destroyed!");
}
```

[동영상 예제 파일명: 036_destroy_explosion_1.mp4]

앞에서 `isActivated` 를 `false`로 만들었기 때문에, 어디에서인가 `isActivated` 를 `true`로 다시 바꿔 주는 곳이 있어야 합니다. 이 역할을 하는 함수가 바로 `Activate()` 입니다. `Activate()` 함수는 앞의 총알(Bullet)에서 사용한 `Activate()` 함수와 거의 같은 역할이지만, 총알과 달리 목표 지점을 설정해 줄 필요가 없습니다. 총알은 목표지점을 향해 날아가야 하지만, 폭발 효과는 그냥 그 자리에 생겨나기만 하면 되기 때문입니다. 따라서 함수의 인자로는 `Vector3` 타입 한 개만 받도록 합니다.

Explosion.cs

```
public class Explosion : RecycleObject
{
    ...
    // Update is called once per frame
    void Update()
    {
        if (isActivated)
        {
            elapsedTime += Time.deltaTime;
            if (elapsedTime >= timeToRemove)
            {
                elapsedTime = 0f;
                DestroySelf();
            }
        }
    }

    public void Activate(Vector3 position)
    {
        ...
    }

    void DestroySelf()
    {
        isActivated = false;
        Debug.Log(gameObject.name + "is destroyed!");
    }
}
```

Activate() 함수에서 제일 먼저 해야 할 일은 isActivated 값을 true로 전환하는 것입니다. 이렇게 되면 폭발 효과가 재활용될 경우, Update() 함수에서 지속 시간을 다시 계산할 수 있게 됩니다.

Explosion.cs

```
public void Activate(Vector3 position)
{
    isActivated = true;
}
```

다음으로는 폭발 효과의 위치를 지정합니다. 함수의 인자로 전달받은 position 값을 이용하면 됩니다.

```
public void Activate(Vector3 position)
{
    isActivated = true;
    transform.position = position;
}
```

이제 폭발 효과(Explosion.cs) 쪽에서의 작업은 다 끝났습니다. 이제 총알 발사대(BulletLauncher.cs)의 OnBulletDestroyed 이벤트 수신 함수로 가서 방금 만든 Activate() 함수를 사용하도록 하겠습니다. 먼저 기존의 코드에서 explosion.transform.position 을 수동으로 지정해 주는 부분을 지워 줍니다.

BulletLauncher.cs

```
void OnBulletDestroyed(Bullet usedBullet)
{
    Vector3 lastBulletPosition = usedBullet.transform.position;
    usedBullet.Destroyed -= OnBulletDestroyed;
    bulletFactory.Restore(usedBullet);

    Explosion explosion = explosionFactory.Get() as Explosion;
    explosion.transform.position = lastBulletPosition;
}
```

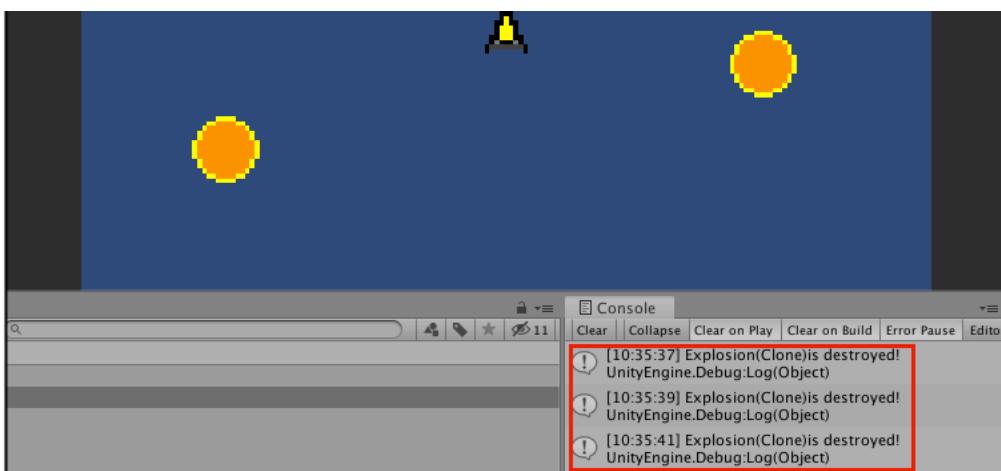
다음으로는 explosion의 Activate() 함수를 이용하여 폭발 효과의 위치를 지정할 수 있도록 다음과 같이 코드를 작성해 줍니다.

BulletLauncher.cs

```
void OnBulletDestroyed(Bullet usedBullet)
{
    Vector3 lastBulletPosition = usedBullet.transform.position;
    usedBullet.Destroyed -= OnBulletDestroyed;
    bulletFactory.Restore(usedBullet);

    Explosion explosion = explosionFactory.Get() as Explosion;
    explosion.Activate(lastBulletPosition);
}
```

이제 유니티 에디터로 가서 테스트 해 보겠습니다. 플레이 버튼을 눌러서 게임을 실행하면 폭발 효과가 만들어지고 일정 시간이 지날 때마다 콘솔창에 “Explosion(Clone) is destroyed!”라는 메시지가 뜨는 것을 확인하실 수 있습니다. 아직 폭발 효과를 회수하는 코드를 작성하지 않았기 때문에 폭발 효과가 화면에 그대로 남아 있기는 하지만, 로직 자체는 잘 작동하고 있다는 것을 알 수 있습니다.



[동영상 예제 파일명: 037_destroy_explosion_2.mp4]

폭발효과를 Factory 로 회수하자

폭발 효과를 다 사용하고 나면, 재활용을 위해 explosionFactory로 반납해야 합니다. 이를 위해서는 폭발 효과가 화면에서 없어져야 하는 타이밍을 알아야 하는데, 이를 알려 주는 새로운 Action 이벤트를 Explosion.cs 에 만들어 보겠습니다.

Action을 사용하기 위해서는 using System; 이 필요합니다. 이를 네임스페이스 맨 위에 작성해 줍니다. 그리고 나서 아래와 같이 Destroyed 라는 이름의 Action을 하나 선언합니다. Explosion 자신을 인자로 전달할 수 있도록 Action<Explosion> 타입으로 선언합니다.

Explosion.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Explosion : RecycleObject
{
    CircleCollider2D circle;
    Rigidbody2D body;

    [SerializeField]
    float timeToRemove = 1f;
    float elapsedTime = 0f;
    bool isActivated = false;

    public Action<Explosion> Destroyed;

    void Awake()
    {
        ...
    }
    ...
}
```

이제 DestorySelf() 함수에서 지금 만든 Destoryed 이벤트를 호출해야 합니다. 먼저 기존에 만들어 놓았던 Debug.Log() 사용 부분을 제거하겠습니다. 더 이상 임시 코드는 필요 없기 때문입니다.

Explosion.cs

```
void DestroySelf()
{
    isActivated = false;
    Debug.Log(gameObject.name + "is destroyed!");
}
```

다음으로는 Destroyed 에 등록된 이벤트 수신 함수들이 있는지 체크하여, 그 경우에만 Destroyed 이벤트를 발생시키도록 하는 코드를 다음과 같이 작성하겠습니다.

Explosion.cs

```
void DestroySelf()
{
    isActivated = false;
    if (Destroyed != null)
    {
        Destroyed(this);
    }
}
```

그런데 이 코드는 더 짧게 축약하는 것이 가능합니다. 더 빠른 코딩을 위해서는 코드를 축약할 수 있으면 하시는 것이 좋습니다. 앞에서 작성한 코드를 다시 삭제합니다.

Explosion.cs

```
void DestroySelf()
{
    isActivated = false;
    if (Destroyed != null)
    {
        Destroyed(this);
    }
}
```

그리고 다음과 같이 축약된 코드를 작성합니다. 코드는 축약되어 있지만, 위에서 삭제한 코드와 아래의 새로 작성한 코드는 완전히 동일한 의미입니다. (앞으로는 축약형 코드를 이용해서 코딩할 것입니다.)

Explosion.cs

```
void DestroySelf()
{
    isActivated = false;
    Destroyed?.Invoke(this);
}
```

이제 Explosion 쪽에서 이벤트를 발생시키는 부분을 다 작성했으니, 다음으로는 BulletLauncher.cs로 가서 방금 만들었던 Destroyed 이벤트를 수신하기 위한 함수를 만들겠습니다. 아래와 같이 OnBulletDestroyed 함수 바깥에 OnExplosionDestroyed라는 이벤트 수신 함수를 작성합니다. 이 때, 송신자와 같은 형식으로 만들어야 하므로, Explosion 타입의 인자를 받도록 선언합니다.

BulletLauncher.cs

```
public class BulletLauncher : MonoBehaviour
{
    ...
    void OnBulletDestroyed(Bullet usedBullet)
    {
        ...
    }

    void OnExplosionDestroyed(Explosion usedExplosion)
    {
    }
}
```

이제 OnBulletDestroyed() 로 가서, explosion의 Destroyed 이벤트에 방금 만든 이벤트 수신 함수(OnExplosionDestroyed)를 연동(바인딩)해 줍니다.

BulletLauncher.cs

```
void OnBulletDestroyed(Bullet usedBullet)
{
    Vector3 lastBulletPosition = usedBullet.transform.position;
    usedBullet.Destroyed -= OnBulletDestroyed;
    bulletFactory.Restore(usedBullet);

    Explosion explosion = explosionFactory.Get() as Explosion;
    explosion.Activate(lastBulletPosition);
    explosion.Destroyed += OnExplosionDestroyed;
}
```

앞에서 강조했듯이 연결한 이벤트는 반드시 끊어주어야 합니다(언바인딩). 따라서 다음과 같이 OnExplosionDestroyed() 함수에서 이벤트를 언바운딩하는 구문을 작성합니다. 폭발 효과를 다 사용해서 팩토리로 회수하기 전에 연결된 이벤트를 끊어 주는 것입니다.

BulletLauncher.cs

```
void OnExplosionDestroyed(Explosion usedExplosion)
{
    usedExplosion.Destroyed -= OnExplosionDestroyed;
}
```

이제 마지막으로 explosionFactory를 이용하여 다 사용한 폭발 효과를 다음과 같이 회수해 주면 모든 것이 끝납니다.

BulletLauncher.cs

```
void OnExplosionDestroyed(Explosion usedExplosion)
{
    usedExplosion.Destroyed -= OnExplosionDestroyed;
    explosionFactory.Restore(usedExplosion);
}
```

그럼 유니티로 가서 테스트해 보겠습니다. 플레이 버튼을 눌러서 게임을 실행한 뒤 마우스로 화면을 클릭해서 총알을 발사하면, 총알이 폭발한 뒤 일정 시간이 지난 후 폭발 효과가 없어지는 것을 확인하실 수가 있을 것입니다.



[동영상 예제 파일명: 038_restore_explosion_to_factory.mp4]

공통된 부분을 RecycleObject 에 정의하자

지금까지 총알(Bullet)과 폭발 효과(Explosion)이라는 두 개의 RecycleObject 파생 클래스들을 재활용하는 것을 구현해 보았습니다. 그런데 잘 살펴 보면, 이 두 개의 RecycleObject 파생 클래스들은 공통적인 부분을 가지고 있습니다. 특히 재활용과 관련된 기능은 거의 같다고 할 수 있을 정도입니다. 따라서 이 공통적인 부분들을 각각의 클래스에서 별도로 구현하기보다는 이들의 부모 클래스인 RecycleObject 에서 구현한 뒤 상속을 통해 사용하는 것이 더 효율적일 것입니다. 그럼 어떤 함수와 변수들을 공통적으로 사용할 수 있을 지 한번 검토해 보겠습니다.

isActivated	변수	활성화되었는지 여부를 체크
Activate()	함수	인자를 하나 받는 Activate() 와 두 개 받는 Activate() 가 있지만, 오버로드(overload) 개념을 이용하면 하나의 함수처럼 공통적으로 사용 가능
Destroyed	Action	대상이 파괴될 때 발생하는 Action. 모든 재사용 가능한 오브젝트들은 이 Action 이벤트를 공통적으로 사용할 수 있음

위의 표에서 설명한 것처럼, 현재 총알과 폭발 효과는 공통된 부분이 있습니다. 그럼 이들을 부모 클래스인 RecycleObject 로 옮기도록 하겠습니다.

제일 먼저 해야 할 일은 RecycleObject.cs 로 가서 사용하지 않을 함수들을 삭제하는 것입니다. Start() 와 Update()를 모두 지워 줍니다.

RecycleObject.cs

```
public class RecycleObject : MonoBehaviour
{
    // Use this for initialization
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

다음으로는 공통된 변수인 `isActivated` 를 선언하는 것입니다. `isActivated` 는 `Bullet.cs` 와 `Explosion.cs` 에서 원래 `private`로 선언된 변수이지만, 여기에서는 파생 클래스들에서 접근할 수 있어야 하므로 `protected` 로 선언해 주어야 합니다.

RecycleObject.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class RecycleObject : MonoBehaviour
{
    protected bool isActivated = false;
}
```

이제 부모 클래스인 `RecycleObject.cs` 에서 `isActivated` 가 선언되었으므로, 자식 클래스들에게는 이 변수가 필요 없습니다. 먼저 `Bullet.cs` 로 가서 `isActivated` 선언 부분을 다음과 같이 삭제합니다.

```

public class Bullet : RecycleObject
{
    [SerializeField]
    float moveSpeed = 5f;

    Vector3 targetPosition;
    bool isActivated = false;

    public Action<Bullet> Destroyed;

    ...
}

```

다음으로는 Explosion.cs 에 있는 isActivated 선언 부분도 삭제합니다.

```

public class Explosion : RecycleObject
{
    CircleCollider2D circle;
    Rigidbody2D body;

    [SerializeField]
    float timeToRemove = 1f;
    float elapsedTime = 0f;
    bool isActivated = false;

    public Action<Explosion> Destroyed;

    ...
}

```

다음으로 해야 할 것은 부모 클래스인 RecycleObject.cs 로 다시 가서 두 개의 Activate() 함수를 선언하는 것입니다. 총알이나 폭발 효과 모두 Activate() 함수를 가지고 있지만, 총알(Bullet.cs)에서 사용한 Activate() 함수는 출발 위치와 목표 지점의 위치라는 2개의 변수를 인자로 갖는 함수이고, 폭발 효과(Explosion.cs)에서 사용한 Activate() 함수는 생성 위치라는 하나의 변수만을 인자로 갖고 있습니다. 따라서 두개

의 함수는 서로 다르다고 할 수 있습니다. 하지만 우리는 앞에서 두 함수 모두에게 Activate()라는 같은 이름을 정해 주었기 때문에 C#의 오버로드(overload) 기능을 이용할 수가 있습니다. (오버로드 개념을 모르시는 분은 C# 프로그래밍 입문서를 찾아보시기 바랍니다.)

우선, 다음과 같이 이름은 같지만 인자의 개수가 다른 두 개의 Activate() 함수를 RecycleObject.cs에 만들어 주겠습니다. 이 때 중요한 것은 각각의 함수에 virtual이라는 키워드를 붙여 주는 것입니다. 왜냐하면 나중에 파생 클래스에서 이들 함수의 내용을 자신들이 편한대로 다시 정의해서 써야 하는 경우(override)가 있을 수 있기 때문입니다. 이 책에서는 자식 클래스에서 이 Activate() 함수들을 재정의해서 쓰는 부분은 없지만, 만약의 경우를 생각해서 virtual 키워드를 붙여 주었습니다.

RecycleObject.cs

```
public class RecycleObject : MonoBehaviour
{
    protected bool isActive = false;

    public virtual void Activate(Vector3 position)
    {
    }

    public virtual void Activate(Vector3 startPosition, Vector3 targetPosition)
    {
    }
}
```

[동영상 예제 파일명: 039_move_common_codes_to_recycleobject.mp4]

이제 다음에 해야 할 일은, 각각의 Activate() 함수에 코드를 삽입하는 것입니다. 첫 번째 Activate() 함수는 인자를 하나만 받는 함수입니다. 따라서 여기에는 Explosion.cs에서 작성한 것과 똑같은 내용의 코드를 작성하면 됩니다. 짧은 코드이므로 그냥 타이핑하셔도 되고, 아니면 Explosion.cs로 가서 기존 코드를 복사해서 붙이셔도 됩니다. 여하튼 RecycleObject.cs의 첫 번째 Activate() 함수에 다음의 코드를 추가합니다.

RecycleObject.cs

```
public class RecycleObject : MonoBehaviour
{
    protected bool isActivated = false;

    public Action<RecycleObject> Destroyed;

    public virtual void Activate(Vector3 position)
    {
        isActivated = true;
        transform.position = position;
    }

    public virtual void Activate(Vector3 startPosition, Vector3 targetPosition)
    {
    }
}
```

이제 Explosion.cs로 가서 다음과 같이 Activate() 함수를 삭제합니다. 동일한 내용의 함수가 부모 클래스인 RecycleObject.cs에 만들어졌으므로 파생 클래스에서 따로 이 함수를 구현할 필요가 없기 때문입니다.

```
public class Explosion : RecycleObject
{
    ...
    // Update is called once per frame
    void Update()
    {
        if (isActivated)
        {
            elapsedTime += Time.deltaTime;
            if (elapsedTime >= timeToRemove)
            {
                elapsedTime = 0f;
                DestroySelf();
            }
        }
    }

    public void Activate(Vector3 position)
    {
        isActivated = true;
        transform.position = position;
    }

    void DestroySelf()
    {
        isActivated = false;
        Destroyed?.Invoke(this);
    }
}
```

다음으로는 인자를 두 개 받는 Activate() 함수를 만들 차례입니다. 다시 RecycleObject.cs 로 가서 이번에는 두번 째 Activate() 함수의 내용을 Bullet.cs 에 있는 내용과 일치하도록 코드를 작성합니다. (Bullet.cs 의 Activate() 함수에 있는 코드를 카피해서 이곳(RecycleObject.cs)에 복사해 넣어도 됩니다.)

RecycleObject.cs

```
public virtual void Activate(Vector3 startPosition, Vector3 targetPosition)
{
    transform.position = startPosition;
    this.targetPosition = targetPosition;
    Vector3 dir = (targetPosition - startPosition).normalized;
    transform.rotation = Quaternion.LookRotation(transform.forward, dir);
    isActivated = true;
}
```

그런데 이렇게 할 경우, 에러가 발생하게 됩니다. 왜냐하면 this.targetPosition에 해당하는 변수가 RecycleObject.cs에는 없기 때문입니다. 따라서 targetPosition이라는 이름의 변수를 아래와 같이 RecycleObject.cs에 선언해 줍니다. 역시 파생 클래스에서 접근할 수 있어야 하므로 protected로 선언해 주어야 합니다.

RecycleObject.cs

```
public class RecycleObject : MonoBehaviour
{
    protected bool isActivated = false;

    public Action<RecycleObject> Destroyed;
    protected Vector3 targetPosition;

    public virtual void Activate(Vector3 position)
    {
        isActivated = true;
        transform.position = position;
    }

    ...
}
```

이제 부모 클래스 쪽에 targetPosition 변수가 선언되었기 때문에, Bullet.cs에는 이 변수가 필요 없습니다. Bullet.cs에 있는 targetPosition 변수를 삭제하겠습니다.

```

public class Bullet : RecycleObject
{
    [SerializeField]
    float moveSpeed = 5f;

    Vector3 targetPosition;

    // Use this for initialization
    void Start()
    {
    }

    ...
}

```

또한 Bullet.cs 에 있는 Activate() 함수도 다음과 같이 삭제합니다. 똑같은 함수가 부모 클래스인 RecycleObject.cs 에 만들어졌기 때문입니다.

```

public class Bullet : RecycleObject
{
    ...

    public void Activate(Vector3 startPosition, Vector3 targetPosition)
    {
        transform.position = startPosition;
        this.targetPosition = targetPosition;
        Vector3 dir = (targetPosition - startPosition).normalized;
        transform.rotation = Quaternion.LookRotation(transform.forward, dir);
        isActivated = true;
    }

    bool IsArrivedToTarget()
    {
        float distance = Vector3.Distance(transform.position, targetPosition);
        return distance < 0.1f;
    }
}

```

[동영상 예제 파일명: 040_move_Bullet_and_Explosion_codes.mp4]

지금까지 Activate() 함수와 관련 변수를 자식 클래스에서 부모 클래스로 옮기는 작업을 모두 마무리했습니다. 다음에 할 일은 Destroyed 라는 이름의 Action 을 부모 클래스로 옮기는 것입니다.

RecycleObject.cs 로 가서 다음과 같이 Destroyed 라는 이름의 Action 을 추가합니다. 이 때 Bullet이나 Explosion 뿐 아니라 모든 RecycleObject 의 파생 클래스들을 인자로 전달해야 하므로 Action<RecycleObject> 로 형식을 지정합니다. 이렇게 하면 앞으로 RecycleObject 로부터 파생된 어떤 클래스의 인스턴스들이라고 해도 Destroyed 에 담아서 인자로 전달할 수가 있게 됩니다. (맨 위에 using System; 을 추가하는 것을 잊지 마시기 바랍니다.)

RecycleObject.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class RecycleObject : MonoBehaviour
{
    protected bool isActive = false;

    public Action<RecycleObject> Destroyed;

    public virtual void Activate(Vector3 position)
    {
    }

    public virtual void Activate(Vector3 position, Vector3 targetPosition)
    {
    }
}
```

역시 부모 클래스 쪽에서 Destroyed 액션(Action)을 선언 했으므로 파생 클래스 쪽에서 다시 선언할 필요가 없습니다. 먼저 Bullet.cs 로 가서 아래와 같이 Action 선언부를 지워 줍니다.

Bullet.cs

```
public class Bullet : RecycleObject
{
    [SerializeField]
    float moveSpeed = 5f;

    public Action<Bullet> Destroyed;

    // Use this for initialization
    void Start()
    {
        }

    ...
}
```

또한 다른 파생 클래스인 Explosion.cs 에 있는 Action 선언부도 다음과 같이 지워줍니다.

Explosion.cs

```
public class Explosion : RecycleObject
{
    CircleCollider2D circle;
    Rigidbody2D body;

    [SerializeField]
    float timeToRemove = 1f;
    float elapsedTime = 0f;

    public Action<Explosion> Destroyed;

    ...
}
```

그런데, 이렇게 하면 비쥬얼 스튜디오에서 BulletLauncher.cs 쪽에 경고 표시를 하는 것을 보실 수 있습니다. 이것은 이벤트 송신자인 Destroyed 와 수신자 함수인 OnBulletDestroyed, OnExplosionDestroyed 의 인자 타입이 달라졌기 때문입니다. 따라서 수신자 함수 쪽의 인자 타입을 RecycleObject로 변경해 주어야 합니다. BulletLauncher.cs 로 가서 다음과 같이 Bullet 과 Explosion 을 각각 지워 줍니다.

BulletLauncher.cs

```
public class BulletLauncher : MonoBehaviour
{
    ...
    void OnBulletDestroyed(Bullet usedBullet)
    {
        Vector3 lastBulletPosition = usedBullet.transform.position;
        usedBullet.Destroyed -= OnBulletDestroyed;
        bulletFactory.Restore(usedBullet);

        Explosion explosion = explosionFactory.Get() as Explosion;
        explosion.Activate(lastBulletPosition);
        explosion.Destroyed += OnExplosionDestroyed;
    }

    void OnExplosionDestroyed(Explosion usedExplosion)
    {
        usedExplosion.Destroyed -= OnExplosionDestroyed;
        explosionFactory.Restore(usedExplosion);
    }
}
```

그리고 인자 타입을 각각 RecycleObject 로 아래와 같이 바꿔 줍니다.

BulletLauncher.cs

```
public class BulletLauncher : MonoBehaviour
{
    ...
    void OnBulletDestroyed(RecycleObject usedBullet)
    {
        Vector3 lastBulletPosition = usedBullet.transform.position;
        usedBullet.Destroyed -= OnBulletDestroyed;
        bulletFactory.Restore(usedBullet);

        Explosion explosion = explosionFactory.Get() as Explosion;
        explosion.Activate(lastBulletPosition);
        explosion.Destroyed += OnExplosionDestroyed;
    }

    void OnExplosionDestroyed(RecycleObject usedExplosion)
    {
        usedExplosion.Destroyed -= OnExplosionDestroyed;
        explosionFactory.Restore(usedExplosion);
    }
}
```

이렇게 하면, 이제는 bulletFactory.Get() 이나 explosionFactory.Get() 을 이용해서 총알(Bullet)이나 폭발 효과(Explosion) 을 꺼내 을 때 캐스팅(Casting)을 할 필요가 없어집니다. 따라서 Bullet이나 as Explosion 부분을 지우고 이를 받는 로컬 변수의 타입도 RecycleObject 로 바꿔야 합니다.

먼저 OnFireButtonPressed() 찾아서 다음과 같이 as Bullet을 삭제합니다.

BulletLauncher.cs

```
public void OnFireButtonPressed(Vector3 position)
{
    if (!canShoot)
        return;

    bullet = bulletFactory.Get() as Bullet;
    bullet.Activate(firePosition.position, position);
    bullet.Destroyed += OnBulletDestroyed;

    canShoot = false;
}
```

그리고 나서 bullet 의 데이터 타입을 RecycleObject 로 변경합니다. 이렇게 하면 bulletFactory.Get() 을 통해 전달 받은 대상의 데이터 타입(RecycleObject)과 로컬 변수인 bullet 의 데이터 타입이 같아지기 때문에 굳이 “as Bullet” 과 같은 캐스팅 절차가 필요 없게 됩니다.

BulletLauncher.cs

```
public void OnFireButtonPressed(Vector3 position)
{
    if (!canShoot)
        return;

    RecycleObject bullet = bulletFactory.Get();
    bullet.Activate(firePosition.position, position);
    bullet.Destroyed += OnBulletDestroyed;

    canShoot = false;
}
```

그런데 이렇게 하면 문제가 생깁니다. bullet 변수가 중복 선언되었기 때문입니다. 따라서 클래스 맨 위쪽으로 올라가서 동일한 이름의 변수 선언 부분을 찾아서 지워 줍니다. 그러면 비쥬얼 스튜디오의 경고 메시지가 사라질 것입니다.

BulletLauncher.cs

```
public class BulletLauncher : MonoBehaviour
{
    [SerializeField]
    Bullet bulletPrefab;
    Bullet bullet;

    [SerializeField]
    Explosion explosionPrefab;

    [SerializeField]
    Transform firePosition;

    ...
}
```

다음으로 OnBulletDestroyed()로 가서, 이쪽도 비슷한 방식으로 수정하겠습니다. 먼저 다음과 같이 Explosion 관련 부분들을 삭제합니다.

BulletLauncher.cs

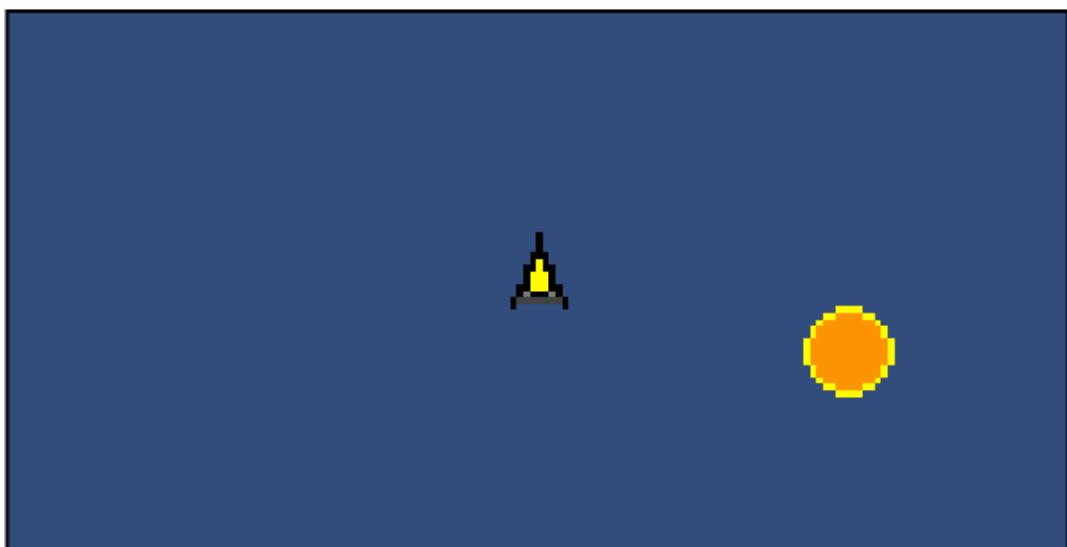
```
void OnBulletDestroyed(RecycleObject usedBullet)
{
    Vector3 lastBulletPosition = usedBullet.transform.position;
    usedBullet.Destroyed -= OnBulletDestroyed;
    bulletFactory.Restore(usedBullet);

    Explosion explosion = explosionFactory.Get() as Explosion;
    explosion.Activate(lastBulletPosition);
    explosion.Destroyed += OnExplosionDestroyed;
}
```

그리고 로컬 변수인 explosion 의 데이터 타입 역시 RecycleObject 로 바꿔 줍니다.
원리는 위의 총알의 경우와 같습니다.

```
BulletLauncher.cs  
void OnBulletDestroyed(RecycleObject usedBullet)  
{  
    Vector3 lastBulletPosition = usedBullet.transform.position;  
    usedBullet.Destroyed -= OnBulletDestroyed;  
    bulletFactory.Restore(usedBullet);  
  
    RecycleObject explosion = explosionFactory.Get();  
    explosion.Activate(lastBulletPosition);  
    explosion.Destroyed += OnExplosionDestroyed;  
}
```

이제 유니티 에디터로 가서 테스트를 해 보겠습니다. 그러면 보시는 것처럼 아무 오류 없이 전과 똑같이 게임이 동작하는 것을 확인하실 수 있습니다.



[동영상 예제 파일명: 041_add_action_on_recycleobject.mp4]

총알 발사대의 위치를 수동 지정하자

현재 총알 발사대(BulletLauncher)는 게임 화면의 정 가운데인 월드 좌표(0,0,0)에 생성됩니다. 따라서 위치를 옮기도록 하겠습니다. 작업을 쉽게 하기 위해서 앞으로는 씬(Scene)에 빈 게임 오브젝트(Empty GameObject)를 하나 만들고 이 위치에 총알 발사대를 생성할 예정입니다.

현재 총알 발사대(BulletLauncher)를 생성해서 게임 월드상에 배치하는 역할은 GameManager.cs 가 담당하고 있습니다. 따라서 먼저 GameManager.cs 로 가서, 다음과 같이 총알 발사대의 생성 위치를 담을 Transform 타입의 변수, launcherLocator 를 선언하겠습니다. 그리고 [SerializeField] 어트리뷰트를 덧붙여서 이 변수가 유니티 에디터의 인스펙터에 노출되도록 하겠습니다.

```
GameManager.cs
```

```
public class GameManager : MonoBehaviour
{
    [SerializeField]
    BulletLauncher launcherPrefab;
    BulletLauncher launcher;

    [SerializeField]
    Transform launcherLocator;

    // Start is called before the first frame update
    void Start()
    {
        ...
    }
    ...
}
```

다음으로는 Start() 함수로 가서, launcher의 position 과 launcherLocator의 position 을 일치시키는 코드를 작성하겠습니다.

```
GameManager.cs
```

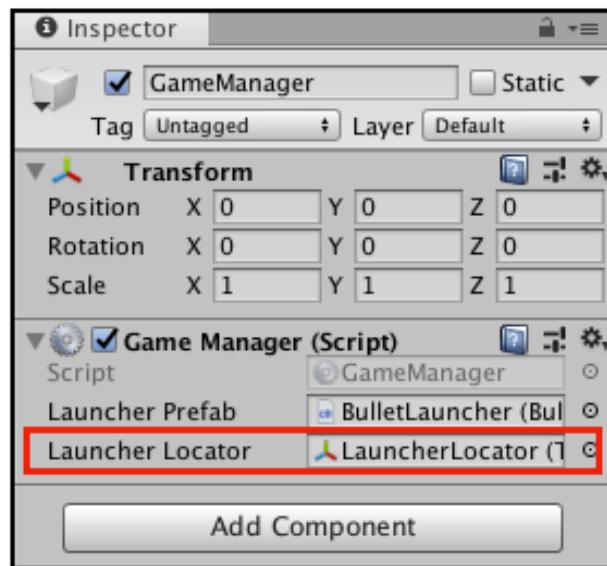
```
void Start()
{
    launcher = Instantiate(launcherPrefab);
    launcher.transform.position = launcherLocator.position;

    MouseGameController mouseController =
        gameObject.AddComponent<MouseGameController>();
    mouseController.FireButtonPressed += launcher.OnFireButtonPressed;
}
```

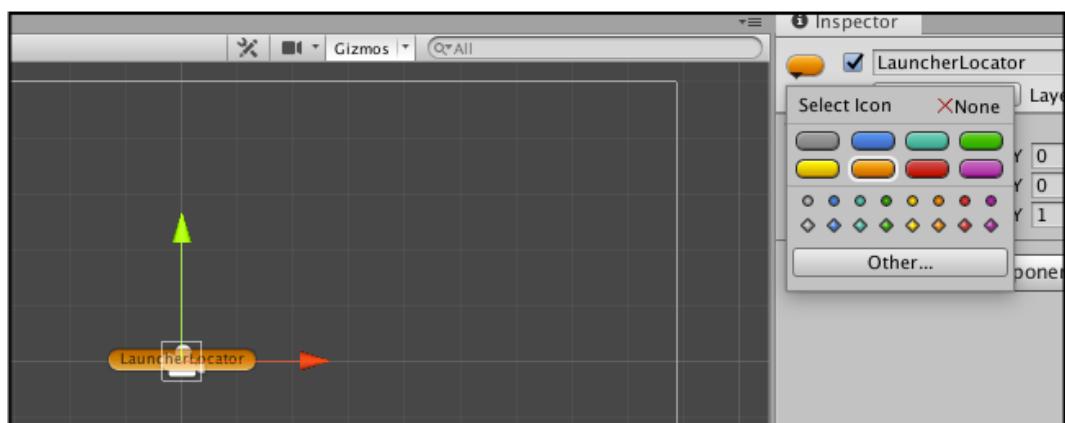
이제 코딩이 끝났으니, 유니티 에디터로 가서 나머지 작업을 수행하겠습니다. 먼저 하이어라키 뷰에서 GameManager 게임 오브젝트를 선택하고, 자식 오브젝트를 하나 만들어 붙이도록 하겠습니다. 그리고 새로 만든 자식 오브젝트의 이름을 LauncherLocator라고 변경합니다.



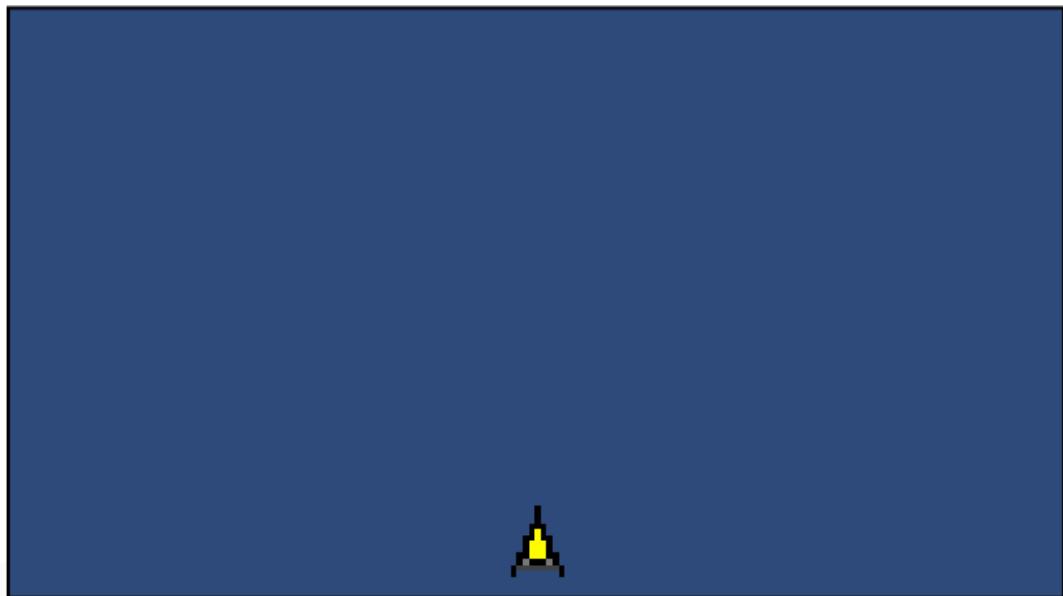
그리고 나서 이 자식 오브젝트(launcherLocator)를 드래그해서 GameManager에 노출된 'launcherLocator' 변수에 넣어 줍니다.



다음으로는 씬 뷰에서 이 launcherLocator의 위치를 쉽게 파악할 수 있도록 인스펙터에서 라벨을 붙이겠습니다. 그리고 나서 마우스로 드래그해서 launcherLocator의 위치를 씬 뷰의 하단 쪽으로 옮겨 줍니다.



이제 플레이 버튼을 눌러서 게임을 실행해 보겠습니다. 그러면 보시는 것처럼 총알 발사대 위치가 달라져 있는 것을 확인하실 수 있습니다.



[동영상 예제 파일명: 042_manually_set_the_launcher_location.mp4]

빌딩(Building)을 만들자

다음으로는 빌딩(Building)을 만들어 보겠습니다. 빌딩은 적의 미사일이 공격하는 대상으로, 플레이어는 이 빌딩이 적의 미사일에 부딪쳐서 파괴되지 않도록 방어해야 합니다.

우선 빌딩의 모든 기능을 담당할 새로운 유니티 C# 스크립트 파일인 Building.cs 를 만들겠습니다. 총알(Bullet)이나 폭발 효과(Explosion)와 달리, 빌딩은 한번 파괴되면 그것으로 끝입니다. 따라서 다시 재활용할 필요가 없습니다. 따라서 빌딩은 RecycleObject 의 파생클래스로 만들지 않을 것입니다. Building.cs 를 처음 만들면 다음과 같이 되어 있을 것입니다.

```
Building.cs
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Building : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

}
}
```

빌딩은 적 미사일과 충돌하면 파괴됩니다. 따라서 충돌을 감지할 수 있도록 충돌체를 필요로 합니다. 빌딩은 네모난 모양이므로 아래와 같이 [RequireComponent] 를 이용하여 BoxCollider2D 컴포넌트를 강제로 붙여 주기로 하겠습니다.

Building.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[RequireComponent(typeof(BoxCollider2D))]
public class Building : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

다음으로는 BoxCollider2D 컴포넌트를 저장해서 관리할 변수를 선언해야 합니다. 아래와 같이 box 라는 간단한 이름으로 선언해 주겠습니다.

Building.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[RequireComponent(typeof(BoxCollider2D))]
public class Building : MonoBehaviour
{
    BoxCollider2D box;

    // Start is called before the first frame update
    void Start()
```

```
{  
}  
  
// Update is called once per frame  
void Update()  
{  
}  
}  
}
```

다음으로는 유니티 내장 함수인 Awake() 를 작성하겠습니다. 다음과 같이 새로운 함수를 하나 만들어 줍니다.

Building.cs

```
using System.Collections;  
using System.Collections.Generic;  
using UnityEngine;  
  
[RequireComponent(typeof(BoxCollider2D))]  
public class Building : MonoBehaviour  
{  
    BoxCollider2D box;  
  
    void Awake()  
    {  
    }  
  
    // Start is called before the first frame update  
    void Start()  
    {  
    }  
  
    ...  
}
```

이제 Awake() 함수에서 GetComponent<BoxCollider2D>() 를 이용하여, 빌딩 게임 오브젝트에 붙어 있는 BoxCollider2D 컴포넌트를 box에 할당한 다음, isTrigger 를 true로 해 줍니다. 단순한 트리거 충돌만을 감지하기 위해서입니다.

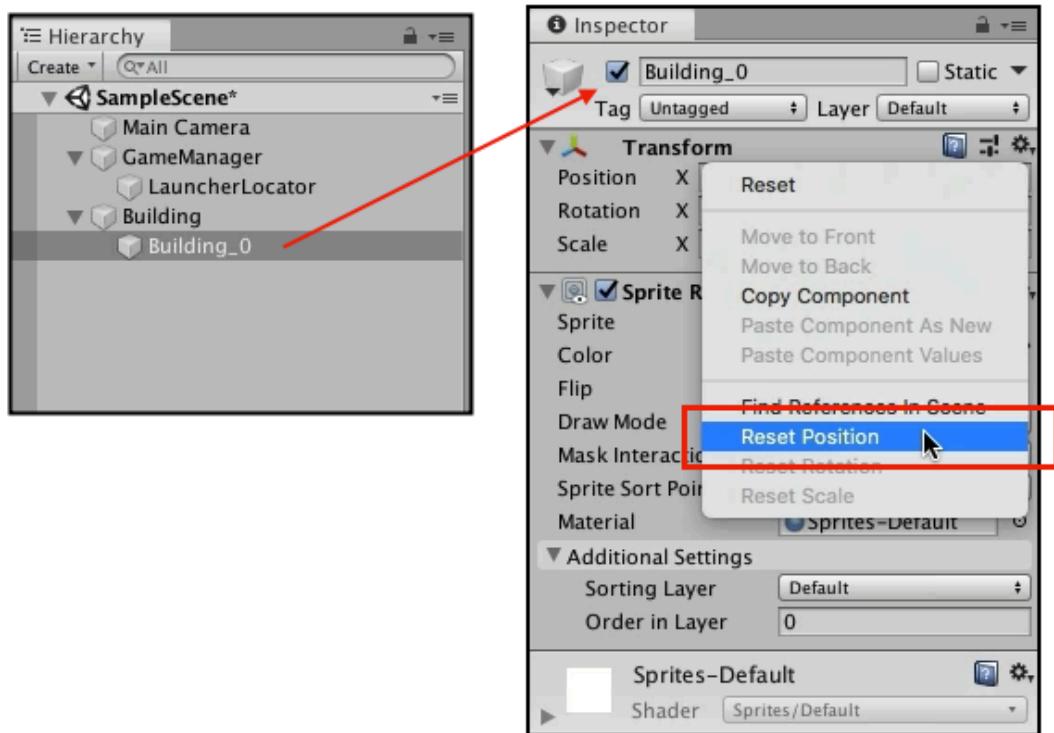
```
Building.cs  
void Awake()  
{  
    box = GetComponent<BoxCollider2D>();  
    box.isTrigger = true;  
}
```

빌딩에 리지드바디(Rigidbody2D)를 붙이지 않고 충돌체만 붙인 이유는 간단합니다. 아직 만들지는 않았지만, 나중에 빌딩과 충돌하게 될 적의 미사일이 리지드바디를 가지고 있을 것이기 때문입니다. 충돌 감지를 위해서는 두 개의 충돌 오브젝트 중 어느 한쪽에만 리지드바디가 붙어 있으면 됩니다.

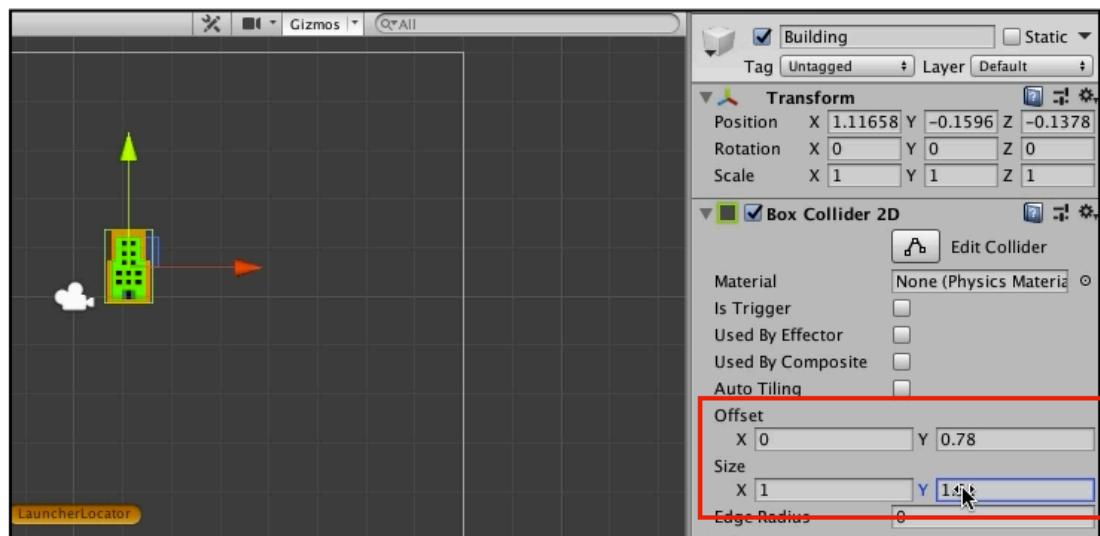
[동영상 예제 파일명: 043_building_class_creation.mp4]

빌딩 프리팹을 만들자

이제 실제 게임에서 사용할 빌딩 프리팹을 만들어 보겠습니다. 우선 유니티 에디터로 가서 씬(Scene)에 빈 게임 오브젝트(Empty GameObject)를 하나 만들고 이름을 Building 이라고 바꿔 주도록 하겠습니다. 그리고 나서 하단 프로젝트 뷰의 [Sprites] 폴더에서 빌딩 스프라이트를 찾아 드래그해서, 방금 만든 Building 게임 오브젝트의 자식 오브젝트로 붙여 놓도록 하겠습니다. 이 때 자식 오브젝트의 로컬 위치가 (0,0,0)인지 확인하신 뒤, 그렇지 않다면 위치를 맞춰 주셔야 합니다.



그 다음, 씬(Scene)이나 하이어라키(Hierarchy)에서 빌딩 게임 오브젝트를 선택하고 방금 만든 Building.cs 스크립트 파일을 붙이겠습니다. 그러면 Box2DCollider 컴포넌트가 자동으로 붙게 되는데, 충돌체의 크기가 빌딩 스프라이트의 크기와 비슷해지도록 사이즈를 조절해 주시기 바랍니다. 그리고 나서 만들어진 게임 오브젝트를 하단의 [Prefabs] 폴더로 드래그해서 프리팹으로 만든 다음, 씬에서 빌딩(Building) 게임 오브젝트를 제거해 주시면 됩니다.



[동영상 예제 파일명: 044_create_building_prefab.mp4]

빌딩 매니저(BuildingManager)를 만들자

앞에서도 이야기한 것처럼, 빌딩은 재활용할 것이 아니므로 Factory를 사용하지는 않을 것입니다. 다만 게임 매니저(GameManager.cs)와 별도로 빌딩 매니저(BuildingManager.cs)를 만들어서 빌딩과 관련된 로직을 이것이 전담하도록 하겠습니다. 게임 매니저가 있는데도 빌딩 매니저를 따로 만드는 이유는, 만약 게임 매니저에게 빌딩을 만들고 관리하는 역할까지 맡기게 되면, 게임 매니저(GameManager) 클래스가 하는 일이 너무 많아지기 때문입니다. 일반적으로 한 개의 클래스는 분명한 한 가지 역할만을 전담하게 하는 게 바람직합니다. (이를 단일 책임의 원칙이라고도 합니다) 따라서 빌딩과 관련한 것은 빌딩 매니저가 전담하고, 게임 매니저(GameManager.cs)는 다른 하위 매니저들의 생성과 커뮤니케이션만 담당하게 하는 것이 저의 목표입니다.

먼저 새로운 유니티 스크립트를 하나 만든 뒤, 이름을 BuildingManager.cs 라고 바꿔주겠습니다. 이 코드는 다음과 같이 되어 있을 것입니다.

```
BuildingManager.cs
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BuildingManager : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {
    }
```

```
// Update is called once per frame
void Update()
{
}
}
```

그런데 BuildingManager.cs 는 유니티의 MonoBehaviour 의 파생 클래스가 아닌, 보통의 클래스로 만들도록 하겠습니다. 앞에서도 비슷한 경우가 있었지만, new 키워드를 이용해서 인스턴스를 생성하고, 생성자에서 필요한 디펜던시(의존성)를 주입하는 방식을 사용해 보려는 것이 그 의도입니다. 따라서 다음과 같이 MonoBehaviour 와 관련된 것들을 모두 삭제하겠습니다.

BuildingManager.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BuildingManager : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

다음으로는 이 클래스의 생성자를 작성하겠습니다.

BuildingManager.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BuildingManager
{
    public BuildingManager()
    {

    }
}
```

그리고 나서 빌딩을 만들어서 게임 월드에 배치하기 위해 필요한 두 개의 변수를 선언하겠습니다. 첫 번째는 빌딩의 프리팹을 할당할 변수입니다. 이 변수를 private로 선언한 이유는 생성자를 통해 외부에서 인스턴스를 주입받아서 사용할 것이기 때문입니다. 또한, 만들어진 빌딩들을 모아서 관리할 Transform 배열도 추가로 선언해 줍니다.

BuildingManager.cs

```
public class BuildingManager
{
    Building prefab;
    Transform[] buildingLocators;

    public BuildingManager()
    {

    }
}
```

다음으로는 생성자에 다음과 같이 두 개의 인자를 추가합니다.

```
BuildingManager.cs
public class BuildingManager
{
    Building prefab;
    Transform[] buildingLocators;

    public BuildingManager(Building prefab, Transform[] buildingLocators)
    {
        }
}
```

이제 생성자에서 전달 받은 인자를 클래스 내부 변수에 할당합니다. 앞에서도 몇 번 사용해 본 방식입니다.

```
BuildingManager.cs
public class BuildingManager
{
    Building prefab;
    Transform[] buildingLocators;

    public BuildingManager(Building prefab, Transform[] buildingLocators)
    {
        this.prefab = prefab;
        this.buildingLocators = buildingLocators;
    }
}
```

다음으로는, 외부에서 주입된 변수의 참조값이 null인 경우 유니티 콘솔창에 경고가 뜨도록 Debug.Assert() 함수 두개를 만듭니다. 이렇게 하면 BuildingManager가 만들 어질 때 필요한 참조값이 인자로 전달되지 않으면 바로 알 수 있습니다.

```
public class BuildingManager
{
    Building prefab;
    Transform[] buildingLocators;

    public BuildingManager(Building prefab, Transform[] buildingLocators)
    {
        this.prefab = prefab;
        this.buildingLocators = buildingLocators;

        Debug.Assert(this.prefab != null, "null building prefab!");
        Debug.Assert(this.buildingLocators != null, "null buildingLocators!");
    }
}
```

[동영상 예제 파일명: 045_building_manager_class_creation.mp4]

이제 BuildingManager.cs 의 기본적인 초기화 관련 코딩이 끝났으니, 실제로 이 클래스의 인스턴스를 생성하는 부분을 다루어 보겠습니다. 아까 제가 빌딩 매니저(BuildingManager)와 같은 중간 관리자 역할을 하는 클래스들은 최상위 관리자인 게임 매니저(GameManager)에서 관리할 것이라고 했는데, 이를 위해 GameManager.cs 로 가서 필요한 작업을 진행하도록 하겠습니다.

우선 GameManager.cs 로 이동한 뒤, 다음과 같이 빌딩 프리팹을 저장할 변수를 먼저 선언하겠습니다. 이 변수는 유니티 에디터에서 만들었던 빌딩 프리팹을 담을 변수입니다. 이를 위해서는 변수를 유니티 에디터의 인스펙터에 노출시켜야 하므로 [SerializeField] 어트리뷰트를 덧붙여 주겠습니다.

```
GameManager.cs
```

```
public class GameManager : MonoBehaviour
{
    [SerializeField]
    BulletLauncher launcherPrefab;
    BulletLauncher launcher;

    [SerializeField]
    Transform launcherLocator;

    [SerializeField]
    Building buildingPrefab;

    // Start is called before the first frame update
    void Start()
    {
        launcher = Instantiate(launcherPrefab);
        launcher.transform.position = launcherLocator.position;

        MouseGameController mouseController =
            gameObject.AddComponent<MouseGameController>();
        mouseController.FireButtonPressed += launcher.OnFireButtonPressed;
    }
}
```

```
    ...  
}
```

다음으로는 생성될 빌딩들의 위치들을 지정하기 위한 트랜스폼 배열을 선언하도록 하겠습니다. 역시 [SerializeField]를 붙여서 유니티 에디터의 인스펙터에서 볼 수 있게 하겠습니다.

GameManager.cs

```
public class GameManager : MonoBehaviour  
{  
    [SerializeField]  
    BulletLauncher launcherPrefab;  
    BulletLauncher launcher;  
  
    [SerializeField]  
    Transform launcherLocator;  
  
    [SerializeField]  
    Building buildingPrefab;  
  
    [SerializeField]  
    Transform[] buildingLocators;  
  
    // Start is called before the first frame update  
    void Start()  
    {  
        ...  
    }  
  
    ...  
}
```

다음으로는 빌딩 매니저(BuildingManager.cs)의 인스턴스를 담을 변수를 선언해 주겠습니다. 이름은 buildingManager라고 하겠습니다.

```

public class GameManager : MonoBehaviour
{
    [SerializeField]
    BulletLauncher launcherPrefab;
    BulletLauncher launcher;

    [SerializeField]
    Transform launcherLocator;

    [SerializeField]
    Building buildingPrefab;

    [SerializeField]
    Transform[] buildingLocators;

    BuildingManager buildingManager;

    ...
}

```

다음에는 Start()에서 빌딩 매니저 인스턴스를 생성하겠습니다. 이 때 생성자를 통해 buildingPrefab과 buildingLocators를 인자로 전달해 줍니다.

```

void Start()
{
    launcher = Instantiate(launcherPrefab);
    launcher.transform.position = launcherLocator.position;

    MouseGameController mouseController =
        gameObject.AddComponent<MouseGameController>();
    mouseController.FireButtonPressed += launcher.OnFireButtonPressed;

    buildingManager = new BuildingManager(buildingPrefab, buildingLocators);
}

```

[동영상 예제 파일명: 046_create_instance_of_building_manager.mp4]

이제 GameManager.cs 쪽에서 할 일은 일단 끝났습니다. 이제 다시 BuildingManager.cs 로 가서 빌딩을 생성하는 로직을 작성하겠습니다. 이를 위해 제일 먼저 해야 할 일은 새로 만들 빌딩들을 모두 저장해서 관리할 리스트를 생성하는 것입니다. 다음 코드처럼 buildings 라는 이름의 리스트를 하나 만들어 줍니다.

BuildingManager.cs

```
public class BuildingManager
{
    Building prefab;
    Transform[] buildingLocators;

    List<Building> buildings = new List<Building>();

    public BuildingManager(Building prefab, Transform[] buildingLocators)
    {
        this.prefab = prefab;
        this.buildingLocators = buildingLocators;

        Debug.Assert(this.prefab != null, "null building prefab!");
        Debug.Assert(this.buildingLocators != null, "null buildingLocators!");
    }
}
```

그 다음으로 해야 할 일은 실제로 빌딩을 생성하기 위한 함수를 만드는 것입니다. 아래처럼 CreateBuildings() 라는 함수를 하나 생성합니다.

BuildingManager.cs

```
public class BuildingManager
{
    Building prefab;
    Transform[] buildingLocators;

    List<Building> buildings = new List<Building>();

    public BuildingManager(Building prefab, Transform[] buildingLocators)
    {
        this.prefab = prefab;
        this.buildingLocators = buildingLocators;
```

```
        Debug.Assert(this.prefab != null, "null building prefab!");
        Debug.Assert(this.buildingLocators != null, "null buildingLocators!");
    }

    void CreateBuildings()
    {
    }

}
```

이 함수에서 하는 일은 빌딩 프리팹으로부터 빌딩의 인스턴스들을 만드는 것입니다. 이 게임에서 빌딩은 게임이 시작될 때 단 한 번만 만들고 끝낼 것입니다. 따라서 빌딩을 만들기 전에 buildings 리스트를 확인해서, 기존에 이미 만들어진 게 있는지를 먼저 확인하는 과정을 거치도록 하겠습니다. 예를 들어 게임 도중 CreateBuildings() 함수가 어디에서인가 호출되었다고 해도, 이미 만들어진 빌딩이 있을 경우에는 다시 만들지 않을 것입니다.

기존에 이미 만들어진 빌딩이 있는지를 확인하는 방법은 간단합니다.

buildings.Count 를 이용하여, 현재 리스트에 빌딩이 존재하는지 여부를 체크하기만 하면 됩니다. 다음과 같이 조건 판단을 하여, 만들어진 빌딩이 있으면 경고 메시지를 콘솔창에 표시하고 여기에서 함수를 종료합니다.

BuildingManager.cs

```
void CreateBuildings()
{
    if (buildings.Count > 0)
    {
        Debug.LogWarning("Buildings have been already created!");
        return;
    }
}
```

다음에 해야 할 일은 실제로 빌딩을 생성하는 것입니다. 이 때 빌딩을 몇 개 만들어야 하느냐가 문제가 될 수 있는데, 여기에서는 그냥 buildingLocators 배열의 개수만큼 빌딩을 생성하겠습니다. 다음 코드와 같이, 먼저 for 루프를 하나 만들겠습니다.

BuildingManager.cs

```
void CreateBuildings()
{
    if (buildings.Count > 0)
    {
        Debug.LogWarning("Buildings have been already created!");
        return;
    }

    for (int i = 0; i < buildingLocators.Length; i++)
    {
    }
}
```

이제 for 루프 안에서 빌딩을 만들어 리스트에 추가하는 작업을 진행하겠습니다. 앞에서도 말씀 드린 것처럼 이 게임에서 빌딩(Building)은 재활용 가능한 게임 오브젝트가 아닙니다. 따라서 팩토리(Factory)를 사용하지 않습니다. 그냥 유니티의 Instantiate() 명령을 이용해서 빌딩의 인스턴스를 생성하겠습니다.

다만 여기에서 주의할 점이 있는데, BuildingManager.cs 클래스 자체는 유니티의 MonoBehaviour 파생 클래스가 아니므로 Instantiate() 함수를 사용하면 비쥬얼 스튜디오에서 경고 메시지를 보여 줄 것입니다. 따라서 다음과 같이 GameObject.Instantiate()라고 적어 주셔야 합니다.

```

void CreateBuildings()
{
    if (buildings.Count > 0)
    {
        Debug.LogWarning("Buildings have been already created!");
        return;
    }

    for (int i = 0; i < buildingLocators.Length; i++)
    {
        Building building = GameObject.Instantiate(prefab);
    }
}

```

그 다음에 생성된 개별 빌딩의 위치를 buildingLocators 배열에 들어 있는 트랜스폼의 포지션 값에 맞춰 줍니다. 그러면 생성된 빌딩들이 우리가 나중에 유니티 에디터에서 수동으로 지정할 위치에 놓여질 것입니다.

```

void CreateBuildings()
{
    if (buildings.Count > 0)
    {
        Debug.LogWarning("Buildings have been already created!");
        return;
    }

    for (int i = 0; i < buildingLocators.Length; i++)
    {
        Building building = GameObject.Instantiate(prefab);
        building.transform.position = buildingLocators[i].position;
    }
}

```

이제 마지막으로 할 일은 생성된 빌딩을 buildings.Add() 를 이용하여 리스트에 삽입하는 것입니다. 이렇게 해야 나중에 이 빌딩들의 상태를 추적해서 관리하는 것이 용이

합니다. 남아 있는 빌딩의 개수가 몇 개인지를 체크해서 게임의 승패를 결정한다거나, 아니면 게임 승리시 남아 있는 빌딩의 개수에 따른 보너스 점수를 계산한다거나 하는 경우 이 리스트가 활용될 것입니다.

BuildingManager.cs

```
void CreateBuildings()
{
    if (buildings.Count > 0)
    {
        Debug.LogWarning("Buildings have been already created!");
        return;
    }

    for (int i = 0; i < buildingLocators.Length; i++)
    {
        Building building = GameObject.Instantiate(prefab);
        building.transform.position = buildingLocators[i].position;
        buildings.Add(building);
    }
}
```

이제 CreateBuildings() 함수가 완성되었습니다. 테스트를 위해 임시로 이 함수를 생성자에서 호출해 보도록 하겠습니다.

BuildingManager.cs

```
public BuildingManager(Building prefab, Transform[] buildingLocators)
{
    this.prefab = prefab;
    this.buildingLocators = buildingLocators;

    Debug.Assert(this.prefab != null, "null building prefab!");
    Debug.Assert(this.buildingLocators != null, "null buildingLocators!");

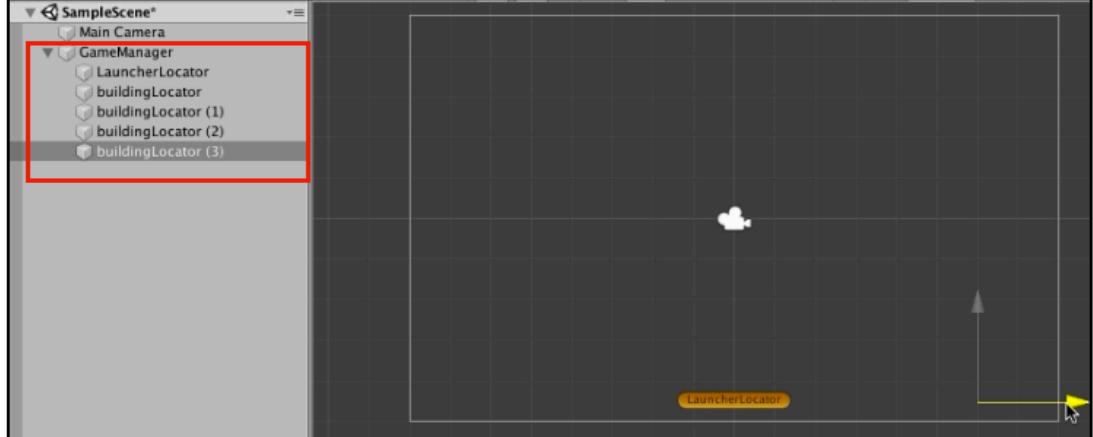
    CreateBuildings();
}
```

[동영상 예제 파일명: 047_create_buildings_by_manager.mp4]

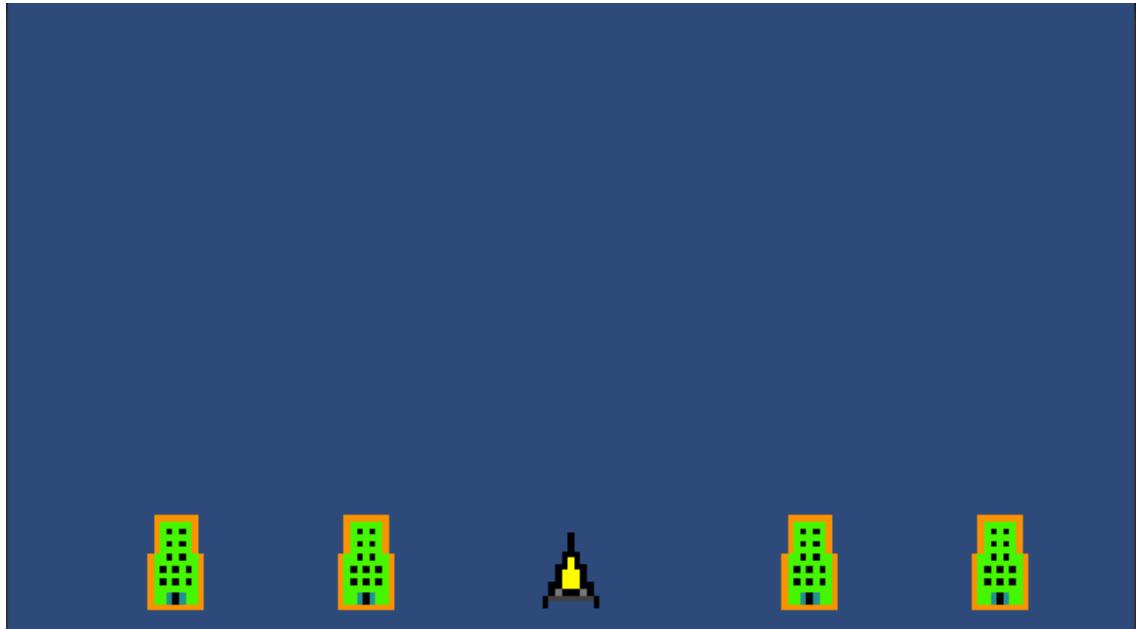
이제 기본적인 코딩은 끝났습니다. 나머지 작업은 유니티 에디터에서 해야 합니다. 이를 위해 유니티 에디터로 이동하겠습니다.

먼저 GameManager의 자식 오브젝트를 하나 만든 뒤, 이름을 buildingLocator 라고 하겠습니다. 그리고 이것을 3개 추가로 복제한 다음, 마우스를 이용해서 적당한 위치에 배치합니다. 이 오브젝트들이 위치한 곳에 빌딩이 생성될 예정이므로, 이 오브젝트들의 트랜스폼 정보를 GameManager 의 인스펙터에 노출된 Building Locators 배열의 요소로 할당해야 합니다.

이를 위해 가장 쉬운 방법은 4개의 자식 오브젝트들을 동시에 선택한 뒤, 인스펙터의 배열 이름 위에 드래그해서 갖다 놓는 것입니다. 그러면 4개의 자식 오브젝트들을 쉽게 배열에 할당할 수가 있습니다. 그리고 프로젝트 뷰에 있는 [Prefabs] 폴더에서 빌딩 프리팹을 찾아서 ‘Building Prefab’ 변수에 연결해 주면 모든 준비가 끝나게 됩니다.
(이 항목은 동영상 예제를 보시면 쉽게 이해하실 수 있을 것입니다.)



이제 모든 준비가 끝났습니다. 플레이 버튼을 눌러서 실행해 보시면 빌딩이 지정된 위치에 잘 생겨나는 것을 확인하실 수 있을 것입니다.



[동영상 예제 파일명: 048_create_buildings_in_the_game.mp4]

TimeManager 를 이용한 게임 시작 딜레이 처리

다음은 게임 시간을 관리하는 역할을 하는 타임 매니저(TimeManager) 클래스를 만 들어 보겠습니다. 진짜 게임이라면 타임 매니저에서 게임의 일시 정지, 불릿타임 모드 등 다양한 시간 관련 처리를 담당하게 하겠지만, 지금 이 책에서 다루는 것은 간단한 샘플 게임이므로 단순히 게임의 시작 타이밍을 관리하는 역할만 부여하도록 하겠습니다.

우리가 지금 계획하는 것은 게임을 시작하자마자 바로 빌딩을 생성하는 것이 아니라, 약간의 시간이 흐르기를 기다린 다음 빌딩이 생성되게 하는 것입니다. 타임 매니저가 하는 역할은 이를 위한 시간 관리 및 관련 이벤트를 발생시키는 것입니다.

이를 위해 우선 유니티 C# 스크립트를 하나 만들고, 이름을 TimeManager.cs 로 변경하겠습니다. 그리고 다음과 같이 Start() 와 Update() 함수들을 제거하겠습니다. TimeManager 에서는 Update()가 아닌 코루틴을 사용할 생각이기 때문입니다.

```
TimeManager.cs  
using System.Collections;  
using System.Collections.Generic;  
using UnityEngine;  
  
public class TimeManager : MonoBehaviour  
{  
    //Start is called before the first frame update  
    void Start()  
    {  
    }  
}
```

```
// Update is called once per frame
void Update()
{
}
}
```

다음으로는 게임 시작 명령을 내리기 위한 StartGame() 함수를 작성하겠습니다. 함수는 public 으로 선언하고, 인자로는 게임이 시작되기 전까지의 대기 시간을 전달해 줍니다.

TimeManager.cs

```
public class TimeManager : MonoBehaviour
{
    public void StartGame(float timeToStart = 3f)
    {
    }
}
```

그런데 현재 StartGame() 함수는 public 이라서, 외부에서 아무나 실행할 수 있습니다. 이 함수는 게임이 처음 시작할 때 GameManager.cs 에서 단 한번만 실행시킬 예정이지만, 나중에 프로그램 팀원 중 누가 실수로 이 함수를 다시 실행시키는 사고(?) 를 칠지 모릅니다. 이렇게 되면 진행 중인 게임이 중간에 다시 시작되는 등의 예기치 못한 사태가 발생할 수 있으므로 이를 막기 위해 StartGame() 함수가 게임 플레이 도중에는 오직 단 한번만 실행되도록 안전장치를 마련할 필요가 있습니다. (함수를 public 으로 선언하지 않았다면 이런 위험도 없을 것입니다. 가급적 불필요한 public 함수를 만들지 않으려고 노력해야 하는 이유가 여기에 있습니다. 하지만 public 을 완전히 배제하고 게임을 만들 수는 없으므로, 어쩔 수 없이 어떤 함수를 public 으로 선

언할 경우에는 이 함수를 잘못 호출하는 경우에 대한 대비책을 세워 두는 습관을 가지는 것이 좋습니다.)

이를 위해 게임이 시작되었는지 여부를 체크하기 위한 bool 타입의 변수를 하나 선언해 주겠습니다.

TimeManager.cs

```
public class TimeManager : MonoBehaviour
{
    bool isGameStarted = false;

    public void StartGame(float timeToStart = 3f)
    {
    }
}
```

그리고 나서 StartGame() 함수 안에 다음과 같이, isStarted 가 true 인 경우 return 명령어로 함수 실행을 바로 종료시키도록 하겠습니다.

TimeManager.cs

```
public class TimeManager : MonoBehaviour
{
    bool isGameStarted = false;

    public void StartGame(float timeToStart = 3f)
    {
        if (isGameStarted)
            return;
    }
}
```

그리고 나서 그 아래에서 isGameStarted 를 true로 바꿔서, 일단 StartGame() 함수가 한번이라도 실행되면 다음에는 실행되지 않도록 안전 장치를 만들어 줍니다.

```
public class TimeManager : MonoBehaviour
{
    bool isGameStarted = false;

    public void StartGame(float timeToStart = 3f)
    {
        if (isGameStarted)
            return;

        isGameStarted = true;
    }
}
```

이제 간단한 코루틴 함수를 하나 작성해서, 정해진 시간이 지나면 게임이 시작되었다는 이벤트를 발생시키도록 하겠습니다. 아래와 같이 DelayedGameStart() 라는 이름의 코루틴 함수를 하나 만듭니다.

```
public class TimeManager : MonoBehaviour
{
    bool isGameStarted = false;

    public void StartGame(float timeToStart = 3f)
    {
        if (isGameStarted)
            return;

        isGameStarted = true;
    }

    IEnumerator DelayedGameStart(float delayTime)
    {
    }
}
```

그리고 나서 인자로 전달 받은 지연 시간(delayTime) 만큼을 기다리도록 yield return 구문을 작성합니다.

TimeManager.cs

```
IEnumerator DelayedGameStart(float delayTime)
{
    yield return new WaitForSeconds(delayTime);
}
```

다음으로는 게임 시작을 알리는 이벤트를 하나 만들겠습니다. Action 을 이용해서 생성할 것이기 때문에 아래와 같이 네임스페이스에 using System; 을 추가해 주어야 합니다. 그리고 나서 GameStarted 라는 이름의 Action을 하나 선언합니다.

TimeManager.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class TimeManager : MonoBehaviour
{
    bool isGameStarted = false;
    public Action GameStarted;

    public void StartGame(float timeToStart = 3f)
    {
        if (isGameStarted)
            return;

        isGameStarted = true;
    }

    IEnumerator DelayedGameStart(float delayTime)
    {
        yield return new WaitForSeconds(delayTime);
    }
}
```

그리고 나서 DelayedGameStart() 코루틴 함수로 가서, yield return 구문 다음에 GameStarted 이벤트를 발생시키는 명령문을 작성합니다. 이번에도 측약형으로 명령어를 작성하겠습니다.

TimeManager.cs

```
IEnumerator DelayedGameStart(float delayTime)
{
    yield return new WaitForSeconds(delayTime);

    GameStarted?.Invoke();
}
```

이제 마지막으로 StartGame() 함수에서 방금 완성한 코루틴 함수를 실행하면 TimeManager.cs에서 할 일은 다 끝나게 됩니다.

TimeManager.cs

```
public void StartGame(float timeToStart = 3f)
{
    if (isGameStarted)
        return;

    isGameStarted = true;

    StartCoroutine(DelayedGameStart(timeToStart));
}
```

[동영상 예제 파일명: 049_time_manager_class_creation.mp4]

다음으로는 게임 매니저(GameManager.cs)로 이동해서, 방금 만든 TimeManager의 인스턴스를 생성하고, GameStarted 이벤트를 관련된 이벤트 수신 함수들과 연동(바인딩)해 주겠습니다. 이를 위해 제일 먼저 해야 할 것은 GameManager.cs에 다음과 같이 TimeManager 의 인스턴스를 담을 변수를 생성하는 것입니다.

GameManager.cs

```
public class GameManager : MonoBehaviour
{
    [SerializeField]
    BulletLauncher launcherPrefab;
    BulletLauncher launcher;

    [SerializeField]
    Transform launcherLocator;

    [SerializeField]
    Building buildingPrefab;

    [SerializeField]
    Transform[] buildingLocators;

    BuildingManager buildingManager;
    TimeManager timeManager;

    // Start is called before the first frame update
    void Start()
    {
        ...
    }

    ...
}
```

이제 TimeManager 클래스의 인스턴스를 생성해야 합니다. TimeManager는 MonoBehaviour 파생 클래스이므로, new 가 아닌 gameobject.AddComponent<T>() 를 사용하여 인스턴스를 생성하겠습니다.

```

void Start()
{
    launcher = Instantiate(launcherPrefab);
    launcher.transform.position = launcherLocator.position;

    MouseGameController mouseController =
        gameObject.AddComponent<MouseGameController>();
    mouseController.FireButtonPressed += launcher.OnFireButtonPressed;

    buildingManager = new BuildingManager(buildingPrefab, buildingLocators);
    timeManager = gameObject.AddComponent<TimeManager>();
}

```

어떤 클래스는 new 키워드를 이용해서 생성하고 어떤 클래스는

gameObject.AddComponent<T>() 를 이용하여 생성하는 이유는, 가급적이면 다양한 방식의 클래스 인스턴스 생성 및 디펜던시 주입 방법을 보여드리기 위해서입니다. 클래스에서 유니티의 내장 함수를 사용하고자 한다면 MonoBehaviour 파생 클래스로 만들 수밖에 없고, 이 경우에는 gameObject.AddComponent<T>() 를 이용하여 인스턴스를 만들어야 할 것입니다. 반면에 생성자를 사용해서 좀 더 일반적인 C# 프로그래밍을 하고 싶을 경우에는 보통의 C# 클래스를 만들고 new 키워드를 이용해서 인스턴스를 만드는 것이 좋을 것입니다. 어느 쪽이 더 좋다고 단언할 수는 없습니다. 여러분이 편하신대로 선택하시면 됩니다.

이제 타임 매니저로부터 게임이 시작되었다는 이벤트를 수신할 함수들을 준비해야 합니다. 먼저 빌딩 매니저에 필요한 코드를 작성하겠습니다.

BuildingManager.cs 로 이동한 다음에, 생성자를 찾아 맨 아래에 우리가 테스트를 위해 작성했던 CreateBuilding() 함수의 호출 부분을 삭제합니다.

```
BuildingManager.cs
public BuildingManager(Building prefab, Transform[] buildingLocators)
{
    this.prefab = prefab;
    this.buildingLocators = buildingLocators;

    Debug.Assert(this.prefab != null, "null building prefab!");
    Debug.Assert(this.buildingLocators != null, "null buildingLocators!");

    CreateBuildings();
}
```

다음으로는 타임 매니저의 GameStarted 이벤트를 수신하기 위한 함수인 OnGameStarted()라는 함수를 만들겠습니다. 그리고 나서, CreateBuilding() 함수를 이곳에서 실행하겠습니다. 이렇게 하고 나면, 빌딩 매니저(BuildingManager)는 앞으로 게임이 시작되었다는 이벤트를 수신한 경우에만 빌딩들을 생성하게 될 것입니다.

```

public class BuildingManager
{
    ...

    void CreateBuildings()
    {
        if (buildings.Count > 0)
        {
            Debug.LogWarning("Buildings have been already created!");
            return;
        }

        for (int i = 0; i < buildingLocators.Length; i++)
        {
            Building building = GameObject.Instantiate(prefab);
            building.transform.position = buildingLocators[i].position;
            buildings.Add(building);
        }
    }

    public void OnGameStarted()
    {
        CreateBuildings();
    }
}

```

이제 게임 매니저로 가서 방금 만든 함수를 타임 매니저의 게임 시작 이벤트와 연동해 주도록 하겠습니다. 이를 위해 GameManager.cs 로 가서 Start() 함수로 이동합니다. 이곳에서 타임 매니저의 GameStarted와 빌딩 매니저의 OnGameStarted 를 다음과 같이 바인딩합니다.

```

void Start()
{
    launcher = Instantiate(launcherPrefab);
    launcher.transform.position = launcherLocator.position;

    MouseGameController mouseController =
        gameObject.AddComponent<MouseGameController>();
}

```

```
mouseController.FireButtonPressed += launcher.OnFireButtonPressed;

buildingManager = new BuildingManager(buildingPrefab, buildingLocators);
timeManager = gameObject.AddComponent<TimeManager>();
timeManager.GameStarted += buildingManager.OnGameStarted;
}
```

다음으로는 게임 시작 명령을 내리도록 하겠습니다. 게임 시작을 위해서는 다음 예제와 같이 timeManager.StartGame() 을 실행하면 됩니다. 그러면 일정 시간이 지나기 를 기다린 뒤 timeManager의 GameStarted 이벤트가 발생할 것이고, 이와 바인딩된 다른 수신자 함수들이 이 이벤트에 반응해서 각자가 해야 할 작업들을 시작하게 될 것입니다.

```
GameManager.cs
```

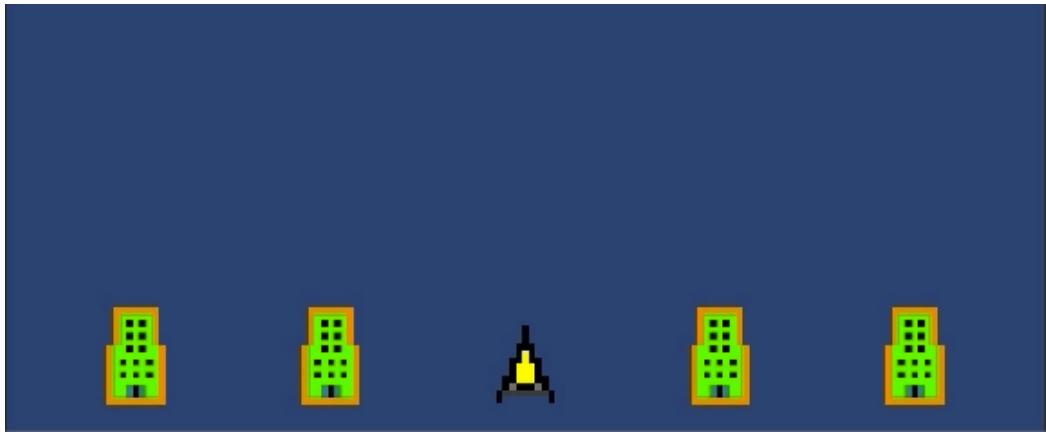
```
void Start()
{
    launcher = Instantiate(launcherPrefab);
    launcher.transform.position = launcherLocator.position;

    MouseGameController mouseController =
        gameObject.AddComponent<MouseGameController>();
    mouseController.FireButtonPressed += launcher.OnFireButtonPressed;

    buildingManager = new BuildingManager(buildingPrefab, buildingLocators);
    timeManager = gameObject.AddComponent<TimeManager>();
    timeManager.GameStarted += buildingManager.OnGameStarted;

    timeManager.StartGame();
}
```

이제 유니티 에디터로 가서 테스트해 보도록 하겠습니다. 플레이 버튼을 누르면 3초를 기다린 뒤에야 빌딩이 생겨나는 것을 확인하실 수 있을 것입니다.



만약 테스트 과정에서 기본값으로 설정된 3초의 지연 시간이 너무 길다고 생각될 경우, 다음과 같이 timeManager.StartGame() 함수에 1f라는 값을 넣어 주면 됩니다.

GameManager.cs

```
void Start()
{
    launcher = Instantiate(launcherPrefab);
    launcher.transform.position = launcherLocator.position;

    MouseGameController mouseController =
        gameObject.AddComponent<MouseGameController>();
    mouseController.FireButtonPressed += launcher.OnFireButtonPressed;

    buildingManager = new BuildingManager(buildingPrefab, buildingLocators);
    timeManager = gameObject.AddComponent<TimeManager>();
    timeManager.GameStarted += buildingManager.OnGameStarted;

    timeManager.StartGame(1f);
}
```

[동영상 예제 파일명: 050_bind_time_manager_building_manager.mp4]

그리고 앞으로는 이렇게 매니저간에 이벤트를 바인딩할 일이 점점 많아질 것입니다.

Start() 함수에서 이 모든 바인딩 작업을 하게 되면 함수가 너무 지저분해지므로, BindEvents()라는 이름의 바운딩 전용 함수를 만들고, 이벤트 바운딩과 관련된 코드들을 모두 이곳으로 이동하겠습니다. 우선 다음과 같이 BindEvents()라는 이름의 함수를 하나 만들어 줍니다.

GameManager.cs

```
void Start()
{
    launcher = Instantiate(launcherPrefab);
    launcher.transform.position = launcherLocator.position;

    MouseGameController mouseController =
        gameObject.AddComponent<MouseGameController>();
    mouseController.FireButtonPressed += launcher.OnFireButtonPressed;

    buildingManager = new BuildingManager(buildingPrefab, buildingLocators);
    timeManager = gameObject.AddComponent<TimeManager>();
    timeManager.GameStarted += buildingManager.OnGameStarted;

    timeManager.StartGame(1f);
}

void BindEvents()
{
```

그런데, 이렇게 하기 위해서는 일부 변수를 Start() 함수의 로컬 변수가 아닌, GameManager.cs의 private 변수로 바꿔 주어야 합니다. 그래야 Start() 함수 바깥에 있는 다른 함수에서도 해당 변수에 접근할 수 있기 때문입니다. 따라서 Start()에 있던 mouseController 로컬 변수를 다음과 같이 바깥으로 끄내어 private 변수로 선언하고, Start() 함수 안에서는 지역 변수 타입 선언 부분을 지우도록 하겠습니다.

```
public class GameManager : MonoBehaviour
{
    [SerializeField]
    BulletLauncher launcherPrefab;
    BulletLauncher launcher;

    [SerializeField]
    Transform launcherLocator;

    [SerializeField]
    Building buildingPrefab;

    [SerializeField]
    Transform[] buildingLocators;

    MouseGameController mouseController;
    BuildingManager buildingManager;
    TimeManager timeManager;

    // Start is called before the first frame update
    void Start()
    {
        launcher = Instantiate(launcherPrefab);
        launcher.transform.position = launcherLocator.position;

        MouseGameController mouseController =
            gameObject.AddComponent<MouseGameController>();
        mouseController.FireButtonPressed += launcher.OnFireButtonPressed;

        buildingManager = new BuildingManager(buildingPrefab, buildingLocators);
        timeManager = gameObject.AddComponent<TimeManager>();
        timeManager.GameStarted += buildingManager.OnGameStarted;

        timeManager.StartGame(1f);
    }

    ...
}
```

이제 준비가 끝났으니, Start() 함수 안에 있는 이벤트 바인딩과 관련된 모든 코드들을 새로 만든 BindEvents() 함수 안으로 이동시키겠습니다.

GameManager.cs

```
void Start()
{
    launcher = Instantiate(launcherPrefab);
    launcher.transform.position = launcherLocator.position;

    mouseController = gameObject.AddComponent<MouseGameController>();
mouseController.FireButtonPressed += launcher.OnFireButtonPressed;

    buildingManager = new BuildingManager(buildingPrefab, buildingLocators);
    timeManager = gameObject.AddComponent<TimeManager>();
timeManager.GameStarted += buildingManager.OnGameStarted;

    timeManager.StartGame(1f);
}

void BindEvents()
{
    mouseController.FireButtonPressed += launcher.OnFireButtonPressed;
    timeManager.GameStarted += buildingManager.OnGameStarted;
}
```

그리고 나서 다음과 같이 Start() 함수에서 BindEvents() 를 호출하겠습니다.

GameManager.cs

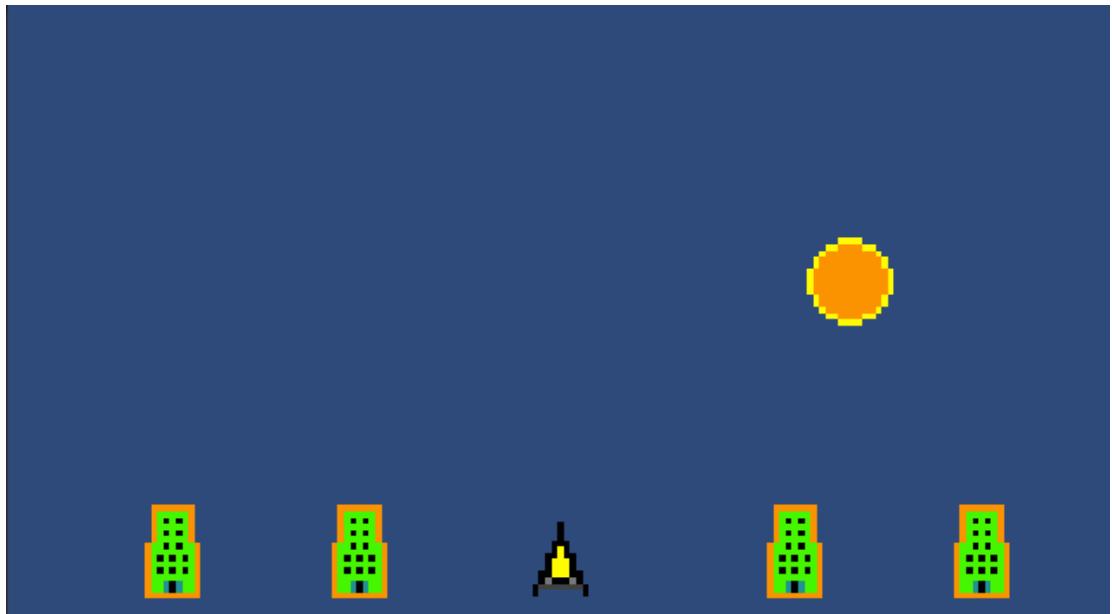
```
void Start()
{
    launcher = Instantiate(launcherPrefab);
    launcher.transform.position = launcherLocator.position;

    mouseController = gameObject.AddComponent<MouseGameController>();
    buildingManager = new BuildingManager(buildingPrefab, buildingLocators);
    timeManager = gameObject.AddComponent<TimeManager>();

    BindEvents();

    timeManager.StartGame(1f);
}
```

그럼 문제 없이 잘 작동하는지 유니티 에디터로 가서 테스트해 보겠습니다. 플레이 버튼을 실행하고 1초간 기다리면 그 때 빌딩이 생겨나는 것을 확인하실 수 있을 것입니다.



[동영상 예제 파일명: 051_bindevents_function_creation.mp4]

빌딩이 게임 시작 시간까지 기다렸다가 생성되게 하는 데에는 성공했지만, 한 가지 문제가 있습니다. 게임이 시작되고 빌딩이 아직 생겨나지 않은 상태에서도 총알이 발사되기 때문입니다. 게임이 아직 시작되지 않았다면 총알도 발사할 수 없어야 정상입니다. 따라서 타임 매니저의 GameStarted 이벤트를 수신한 경우에만 총알을 발사할 수 있도록, 총알 발사대 쪽에 새로운 코드를 작성하겠습니다.

이를 위해 먼저 총알 발사를 관리하는 총알 발사대(BulletLauncher.cs)로 가 보겠습니다. 비쥬얼 스튜디오로 BulletLauncher.cs를 열고 다음과 같이 게임이 시작되었는지를 체크하기 위한 bool형 변수를 하나 선언해 주겠습니다.

BulletLauncher.cs

```
public class BulletLauncher : MonoBehaviour
{
    [SerializeField]
    Bullet bulletPrefab;

    [SerializeField]
    Explosion explosionPrefab;

    [SerializeField]
    Transform firePosition;

    [SerializeField]
    float fireDelay = 0.5f;
    float elapsedFireTime;
    bool canShoot = true;

    bool isGameStarted = false;

    Factory bulletFactory;
    Factory explosionFactory;

    ...
}
```

그리고 나서 Update() 함수로 가서, 아직 게임이 시작되지 않았을 경우에는 더 이상의 코드를 실행시키지 않도록 다음과 같이 조건문을 추가하겠습니다.

BulletLauncher.cs

```
void Update()
{
    if (!isGameStarted)
        return;

    if (!canShoot)
    {
        elapsedFireTime += Time.deltaTime;
        if (elapsedFireTime >= fireDelay)
        {
            canShoot = true;
            elapsedFireTime = 0f;
        }
    }
}
```

또한 OnFireButtonPressed() 함수에도 똑같은 코드를 추가합니다. 그러면 이제 isGameStarted 가 false 인 동안에는 절대로 총알을 발사할 수 없습니다.

BulletLauncher.cs

```
public void OnFireButtonPressed(Vector3 position)
{
    if (!isGameStarted)
        return;

    if (!canShoot)
        return;

    RecycleObject bullet = bulletFactory.Get();
    bullet.Activate(firePosition.position, position);
    bullet.Destroyed += OnBulletDestroyed;

    canShoot = false;
}
```

다음으로 해야 할 일은 어디에서인가 isGameStarted 값을 true로 바꾸는 것입니다. 이를 위해서 OnGameStarte()라는 이름의 함수를 만들고 아래와 같이 isGameStarted 를 true로 바꾸는 코드를 삽입합니다.

BulletLauncher.cs

```
public class BulletLauncher : MonoBehaviour
{
    ...
    void OnExplosionDestroyed(RecycleObject usedExplosion)
    {
        usedExplosion.Destroyed -= OnExplosionDestroyed;
        explosionFactory.Restore(usedExplosion);
    }

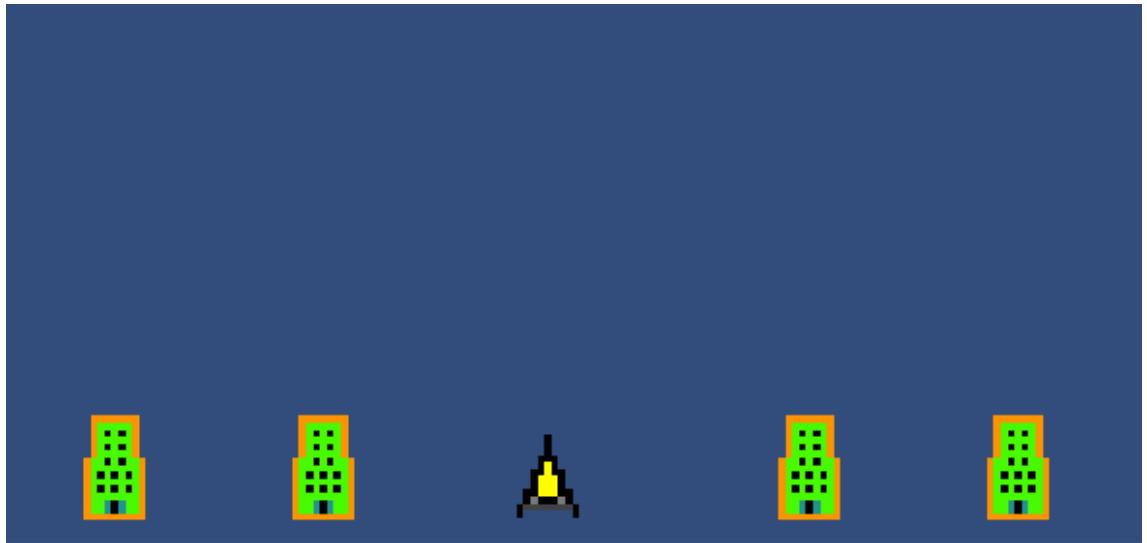
    public void OnGameStarted()
    {
        isGameStarted = true;
    }
}
```

이제 게임 매니저(GameManager.cs)로 이동해서 위에서 만든 함수를 타임 매니저의 게임 시작 이벤트와 연동하겠습니다. 다음과 같이 BindEvents() 함수에 관련 코드를 추가하면 됩니다.

GameManager.cs

```
void BindEvents()
{
    mouseController.FireButtonPressed += launcher.OnFireButtonPressed;
    timeManager.GameStarted += buildingManager.OnGameStarted;
    timeManager.GameStarted += launcher.OnGameStarted;
}
```

이제 유니티로 가서 게임을 실행해 보겠습니다. 플레이 버튼을 눌러서 테스트해 보시면, 게임이 시작되어 빌딩이 생기기 전에는 마우스를 아무리 클릭해도 총알이 발사되지 않는 것을 확인하실 수 있을 것입니다.



[동영상 예제 파일명: 052_shoot_bullet_when_game_started.mp4]

Unbind 함수도 만들자

이벤트를 바인딩했으면 잊기 전에 언바인딩을 해야 합니다. 이것은 이벤트 기반으로 게임을 만들 경우 반드시 명심해야 할 부분입니다. 언바인딩되지 않은 이벤트는 어디엔가 숨어 있다가 생각지도 않은 순간에 게임에 영향을 미칠 수 있기 때문입니다. 유니티 C# 프로그래밍에서 일반적으로 언바인딩을 수행하는 곳은 OnDisable()이나 OnDestroy() 함수입니다. 만약 MonoBehaviour 기반의 클래스가 아닌, 여러분이 직접 만든 커스텀 클래스를 사용하고 있다면 클래스의 인스턴스가 제거되는 순간에 언바인딩을 해 주는 것이 일반적입니다. 이 경우 소멸자(Destructor)에서 언바인딩을 하거나, 아니면 여러분이 직접 만든 특정한 함수 안에서 언바인딩을 꼭 해 주시기 바랍니다.

일단 이 게임에서는 GameManager.cs 의 OnDestroy() 유니티 내장 함수를 이용하여 언바인딩을 하겠습니다. 지금 만들고 있는 게임에서는 게임 매니저가 파괴될 일이 없습니다. 따라서 GameManager 클래스에서의 언바인딩이 꼭 필요한 것은 아니지만, 미래에 어떤 식으로 게임이 발전할 지 모르기 때문에 안전을 위해 미리 언바인딩 처리를 해 두는 것이 좋습니다. 우선 다음 예제와 같이 UnBindEvents() 라는 함수를 하나 만들도록 하겠습니다.

```

public class GameManager : MonoBehaviour
{
    ...
    void BindEvents()
    {
        mouseController.FireButtonPressed += launcher.OnFireButtonPressed;
        timeManager.GameStarted += buildingManager.OnGameStarted;
        timeManager.GameStarted += launcher.OnGameStarted;
    }

    void UnBindEvents()
    {

    }

    // Update is called once per frame
    void Update()
    {
    }
}

```

다음으로는 BindEvents() 함수에 있는 코드들을 UnBindEvents() 함수에 모두 카피하겠습니다. 바인딩의 플러스(+) 기호를 언바인딩에서는 마이너스(-) 기호로만 바꾸면 되므로, 같은 명령어를 복사한 뒤 += 를 -= 로만 바꿀 예정입니다.

```

void UnBindEvents()
{
    mouseController.FireButtonPressed -= launcher.OnFireButtonPressed;
    timeManager.GameStarted -= buildingManager.OnGameStarted;
    timeManager.GameStarted -= launcher.OnGameStarted;
}

```

코드를 모두 복사했으면 다음과 같이 += 를 -= 로 바꿔 주면 됩니다.

GameManager.cs

```
void UnBindEvents()
{
    mouseController.FireButtonPressed -= launcher.OnFireButtonPressed;
    timeManager.GameStarted -= buildingManager.OnGameStarted;
    timeManager.GameStarted -= launcher.OnGameStarted;
}
```

다음으로는 유니티 내장 함수인 OnDestroy() 를 다음과 같이 생성합니다.

GameManager.cs

```
public class GameManager : MonoBehaviour
{
    ...
    void UnBindEvents()
    {
        mouseController.FireButtonPressed -= launcher.OnFireButtonPressed;
        timeManager.GameStarted -= buildingManager.OnGameStarted;
        timeManager.GameStarted -= launcher.OnGameStarted;
    }

    void OnDestroy()
    {

    }

    // Update is called once per frame
    void Update()
    {
    }
}
```

그리고 나서 UnBindEvents() 를 여기에서 실행해 주면 됩니다. 이렇게 하면
GameManager 클래스의 인스턴스가 파괴되는 순간 UnBindEvents() 함수가 실행되
면서 바인딩되었던 이벤트들이 모두 언바인딩될 것입니다.

GameManager.cs

```
void OnDestroy()  
{  
    UnBindEvents();  
}
```

[동영상 예제 파일명: 053_unbindevents_function_creation.mp4]

적 미사일(Missile)을 만들자

다음으로는 적의 미사일 공격을 구현할 차례입니다. 이를 위해서 제일 먼저 해야 할 일은 미사일(Missile)을 만드는 것입니다. 미사일 역시 Factory 를 이용하여 재사용 가능한 게임 오브젝트로 만들 예정입니다. 따라서 총알(Bullet)이나 폭발 효과(Explosion)와 마찬가지로 RecycleObject 의 파생 클래스로 구현하겠습니다. 이를 위해 다음과 같이 Missile.cs 라는 이름의 유니티 C# 스크립트 파일을 하나 만든 뒤, MonoBehaviour 를 삭제해 줍니다.

```
Missile.cs
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Missile : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

그리고 나서 지워진 MonoBehaviour 대신에 RecycleObject 를 입력해 줍니다. 그러면 미사일(Missile.cs) 역시 RecycleObject의 파생 클래스가 되어 팩토리(Factory)를 이용하여 손쉽게 생성하고 재활용할 수 있습니다.

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Missile : RecycleObject
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}

```

한편, 미사일 역시 충돌 감지가 필요합니다. 따라서 리지드바디와 충돌체가 반드시 요구됩니다. 다음과 같이 RequireComponent를 이용하여 BoxCollider2D와 Rigidbody2D 가 자동으로 연결되도록 해 주겠습니다.

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[RequireComponent(typeof(BoxCollider2D), typeof(Rigidbody2D))]
public class Missile : RecycleObject
{
    // Start is called before the first frame update
    void Start()
    {

    }

    ...
}

```

그 다음에는 이들을 담을 변수들이 필요합니다. 아래의 예제와 같이 BoxCollider2D 와 Rigidbody2D 타입의 변수들인 box 와 body를 각각 선언해 줍니다.

```
Missile.cs
public class Missile : RecycleObject
{
    BoxCollider2D box;
    Rigidbody2D body;

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
    }
}
```

이제 Awake() 함수를 만들고 GetComponent<T>() 를 이용하여 BoxCollider2D와 Rigidbody2D 컴포넌트를 불러와 적절한 변수들에 할당해 주겠습니다.

```
Missile.cs
public class Missile : RecycleObject
{
    BoxCollider2D box;
    Rigidbody2D body;

    void Awake()
    {
        box = GetComponent<BoxCollider2D>();
        body = GetComponent<Rigidbody2D>();
    }

    ...
}
```

그 다음에 해야 할 일은 컴포넌트 속성을 변경하는 것입니다. 다음과 같이 리지드 바디의 타입은 Kinematic 으로 지정하고 충돌체의 속성은 isTrigger 로 해서, 트리거를 이용한 충돌 감지를 할 수 있도록 설정합니다.

Missile.cs

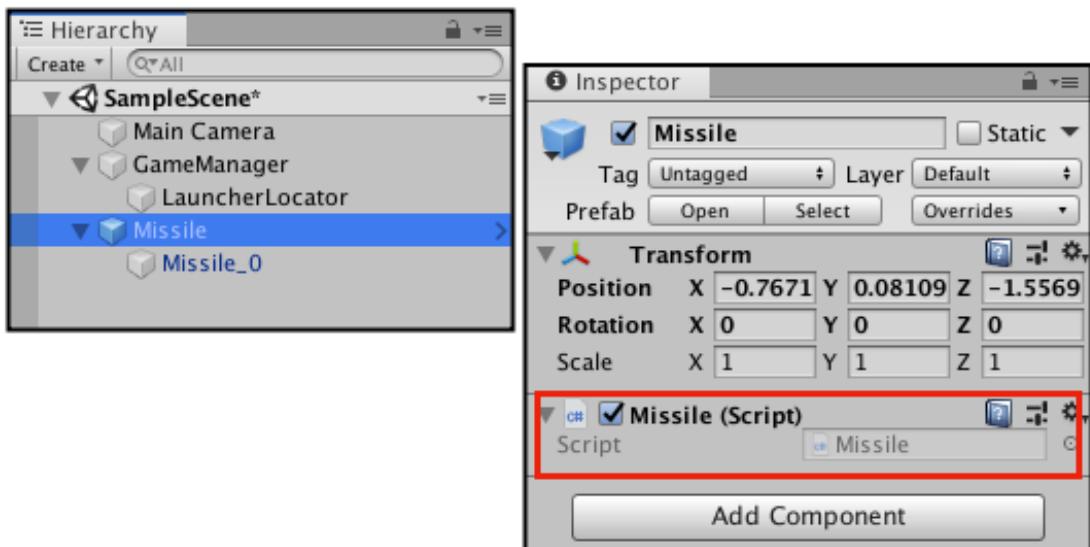
```
void Awake()
{
    box = GetComponent<BoxCollider2D>();
    body = GetComponent<Rigidbody2D>();

    body.bodyType = RigidbodyType2D.Kinematic;
    box.isTrigger = true;
}
```

[동영상 예제 파일명: 054_missile_class_creation.mp4]

미사일(Missile) 프리팹을 만들자

유니티 에디터로 가서 씬에 빈 게임 오브젝트를 만들고 이름을 Missile 이라고 변경합니다. 그리고 나서 프로젝트 뷰의 [Sprites] 폴더에서 미사일 스프라이트 찾아서 드래그한 뒤, 방금 만든 Missile 게임 오브젝트의 자식 오브젝트로 만들어 줍니다. 이 때 자식 오브젝트의 위치를 확인해서 로컬 포지션이 (0,0,0)인지 확인하고 그렇지 않다면 (0,0,0)으로 맞춰 줍니다. 그리고 나서 다시 Missile 게임 오브젝트를 선택한 뒤, 방금 만든 Missile.cs 스크립트를 연결합니다. 이렇게 미사일이 완성되면 이를 드래그해서 프로젝트 뷰에 있는 [Prefabs] 폴더로 드래그해서 갖다 놓으면 됩니다.



이제 미사일의 프리팹이 준비되었으니, 씬(Scene)에서 방금 프리팹을 만들기 위해 사용했던 Missile이라는 이름의 게임 오브젝트는 더 이상 필요 없습니다. 따라서 씬(Scene)에서 이것을 지우도록 하겠습니다.

[동영상 예제 파일명: 055_missile_prefab_creation.mp4]

미사일 매니저(MissileManager)를 만들자

다음으로는 MissileManager.cs 클래스를 만들도록 하겠습니다. 앞에서 총알과 총알의 폭발 효과는 총알 발사대(BulletLauncher)가 관리했습니다. 또한 빌딩의 경우엔 빌딩 매니저(BuildingManager)가 관리했습니다. 이와 마찬가지로 미사일과 관계된 것들만 관리할 클래스가 필요한데, 이것이 바로 지금부터 만들 미사일 매니저(MissileManager)입니다.

그럼 우선 MissileManager.cs 클래스를 다음과 같이 생성하도록 하겠습니다. 참고로, MissileManager는 MonoBehaviour 의 파생 클래스 그대로 사용하겠습니다. 그 이유는 나중에 이곳에서 코루틴(Coroutine) 함수를 사용할 예정이기 때문입니다.

MissileManager.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MissileManager : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

}
}
```

우선 사용하지 않을 함수들을 삭제하고 시작하겠습니다.

MissileManager.cs

```
public class MissileManager : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

다음으로는 디펜던시를 외부로부터 주입받기 위한 함수를 하나 만들겠습니다. 이 미사일 매니저(MissileManager.cs)는 MonoBehaviour의 파생 클래스이기 때문에 일반 클래스와 달리 생성자를 사용할 수가 없습니다. 따라서 여기에서는 생성자 대신 Initialize()라고 하는 public 함수를 만들어서, 생성자가 아닌 ‘함수’를 통한 디펜던시 인젝션(의존성 주입)이 가능하도록 해 보겠습니다.

MissileManager.cs

```
public class MissileManager : MonoBehaviour
{
    public void Initialize()
    {

    }
}
```

이제 미사일 생성을 전담할 팩토리 변수(missileFactory)와, 빌딩 매니저를 저장할 변수(buildingManager)를 각각 다음과 같이 선언하겠습니다.

```
public class MissileManager : MonoBehaviour
{
    Factory missileFactory;
    BuildingManager buildingManager;

    public void Initialize()
    {
    }
}
```

다음으로는 Initialize()가 Factory 타입의 인자와 BuildingManager 타입의 인자를 각각 받아 들일 수 있게 하겠습니다. 이는 외부에서 미사일 팩토리와 빌딩 매니저를 주입 받아서 마치 생성자처럼 사용하기 위해서입니다. (물론 생성자와 달리 이 함수를 단 한번만 강제로 실행시킬 수 없습니다. 따라서 생성자를 완전히 대신할 수는 없습니다.)

```
public class MissileManager : MonoBehaviour
{
    Factory missileFactory;
    BuildingManager buildingManager;

    public void Initialize(Factory missileFactory, BuildingManager buildingManager)
    {
    }
}
```

이제 생성자를 통한 디펜던시 인젝션을 구현할 때와 마찬가지로, 미사일 매니저의 내부 변수에 외부에서 주입한 인스턴스들을 각각 할당하는 코드를 Initialize() 함수 안에 작성하도록 하겠습니다.

```
public class MissileManager : MonoBehaviour
{
    Factory missileFactory;
    BuildingManager buildingManager;

    public void Initialize(Factory missileFactory, BuildingManager buildingManager)
    {
        this.missileFactory = missileFactory;
        this.buildingManager = buildingManager;
    }
}
```

그리고 미사일 매니저나 빌딩 매니저가 null 일 경우, 경고 메시지가 뜨도록 하는 코드를 이곳에도 작성하도록 하겠습니다.

```
public void Initialize(Factory missileFactory, BuildingManager buildingManager)
{
    this.missileFactory = missileFactory;
    this.buildingManager = buildingManager;

    Debug.Assert(this.missileFactory != null, "missile factory is null!");
    Debug.Assert(this.buildingManager != null, "building manager is null!");
}
```

일반 클래스에서 생성자는 한번만 실행되고 더 이상 다시 실행할 수 없지만,

Initialize()는 public 으로 선언된 보통의 함수이기 때문에 클래스 외부에서 몇 번이고 실행할 수가 있습니다. 따라서 안전 장치 없이 그냥 놔두면 나중에 허가받지 않은 누군가가 이 함수를 마음대로 실행해서 팩토리(missileFactory)와 빌딩 매니저(buildingManager)를 초기에 주입 받은 것과 다른 것으로 교체할 위험이 있습니다. 실수로 그러든 일부러 그러든 이런 상황이 발생하는 것을 우리는 원하지 않기 때문에, 이것을 방지하기 위한 코드를 작성해야 합니다. 다시 말해서 Initialize() 함수 안의 코

드가 게임이 실행되는 동안 오직 단 한번만 실행될 수 있도록 안전 장치를 마련해야 하는 것입니다. 이를 위해 다음과 같이 initialized라는 bool 타입의 변수를 선언하고, 기본 값을 false로 지정해 줍니다.

MissileManager.cs

```
public class MissileManager : MonoBehaviour
{
    Factory missileFactory;
    BuildingManager buildingManager;

    bool initialized = false;

    ...
}
```

다음으로는 이 initialized 변수의 값이 false인 경우에만 Initialize() 함수 안에 있는 코드들이 실행되도록 해 줍니다. 아래와 같이 코드를 작성하시면, Initialize() 함수 안의 코드는 단 한번만 실행되므로, 이제 MissileManager 클래스 바깥에서 이 함수를 다시 실행해서 팩토리(missileFactory)와 빌딩 매니저(buildingManager)를 중간에 다른 것으로 교체할 수 없습니다.

MissileManager.cs

```
public void Initialize(Factory missileFactory, BuildingManager buildingManager)
{
    if (initialized)
        return;

    this.missileFactory = missileFactory;
    this.buildingManager = buildingManager;

    initialized = true;
}
```

[동영상 예제 파일명: 056_missile_manager_class_creation.mp4]

다음으로는 게임 매니저로 이동해서 미사일 매니저의 인스턴스를 만들고, Initialize() 함수를 통해 미사일 전용 팩토리(Factory)와 빌딩 매니저(BuildingManager)의 인스턴스를 주입하는 코드를 작성하도록 하겠습니다.

우선 GameManager.cs 에 다음과 같이 두 개의 변수를 선언해 줍니다. 이 중에서 특히 missilePrefab 변수의 경우에는 [SerializeField] 어트리뷰트를 덧붙여서 유니티 에디터의 인스펙터에 노출되도록 해 주겠습니다.

GameManager.cs

```
public class GameManager : MonoBehaviour
{
    [SerializeField]
    BulletLauncher launcherPrefab;
    BulletLauncher launcher;

    [SerializeField]
    Transform launcherLocator;

    [SerializeField]
    Building buildingPrefab;

    [SerializeField]
    Transform[] buildingLocators;

    [SerializeField]
    Missile missilePrefab;

    MouseGameController mouseController;
    BuildingManager buildingManager;
    TimeManager timeManager;
    MissileManager missileManager;

    // Start is called before the first frame update
    void Start()
    {
        ...
    }
}
```

다음으로 Start() 함수로 가서 MissileManager의 인스턴스를 생성한 뒤, 앞에서 만든 missileManager 변수에 할당해 주겠습니다. 앞에서도 말씀 드렸지만, MissileManager는 MonoBehaviour의 파생 클래스이기 때문에 다음과 같이 gameObject.AddComponent<T>() 명령어를 이용하여 인스턴스를 만들어 주어야 합니다.

```
GameManager.cs
void Start()
{
    launcher = Instantiate(launcherPrefab);
    launcher.transform.position = launcherLocator.position;

    mouseController = gameObject.AddComponent<MouseGameController>();
    buildingManager = new BuildingManager(buildingPrefab, buildingLocators);
    timeManager = gameObject.AddComponent<TimeManager>();
    missileManager = gameObject.AddComponent<MissileManager>();

    BindEvents();

    timeManager.StartGame(1f);
}
```

다음으로는 missileManager.Initialize() 함수를 이용하여 새로운 팩토리(Factory)와 빌딩 매니저의 디펜던시를 주입해 줍니다. Factory의 경우 missilePrefab 을 이용하여 새로 만들어 주입하고, buildingManager는 위에서 이미 만들어 놓은 것을 그냥 참조로 전달해 주기만 하면 됩니다.

```
void Start()
{
    launcher = Instantiate(launcherPrefab);
    launcher.transform.position = launcherLocator.position;

    mouseController = gameObject.AddComponent<MouseGameController>();
    buildingManager = new BuildingManager(buildingPrefab, buildingLocators);
    timeManager = gameObject.AddComponent<TimeManager>();
    missileManager = gameObject.AddComponent<MissileManager>();
    missileManager.Initialize(new Factory(missilePrefab), buildingManager);

    BindEvents();

    timeManager.StartGame(1f);
}
```

[동영상 예제 파일명: 057_create_missile_manager_instance.mp4]

이제 GameManager.cs 에서 할 작업은 끝났으므로, 다시 MissileManager.cs 로 가서 미사일을 만들어내는 로직을 작성하겠습니다. MissileManager.cs 에 다음과 같이 SpawnMissile() 이라는 함수를 생성합니다.

MissileManager.cs

```
public class MissileManager : MonoBehaviour
{
    Factory missileFactory;
    BuildingManager buildingManager;

    public void Initialize(Factory missileFactory, BuildingManager buildingManager)
    {
        this.missileFactory = missileFactory;
        this.buildingManager = buildingManager;

        Debug.Assert(this.missileFactory != null, "missile factory is null!");
        Debug.Assert(this.buildingManager != null, "building manager is null!");
    }

    void SpawnMissile()
    {
    }
}
```

SpawnMissile()에서 제일 먼저 해야 할 일은 전달 받은 미사일 팩토리(missileFactroy)의 Get() 함수를 이용하여 미사일의 인스턴스를 가져 오는 것입니다. 그리고 나서 Activate() 함수를 이용하여 미사일이 생성될 위치를 지정해 주겠습니다. 일단은 미사일이 화면의 정가운데에서 나타나도록 Vector3.zero를 임시 위치 값으로 전달합니다.

MissileManager.cs

```
void SpawnMissile()
{
    RecycleObject missile = missileFactory.Get();
    missile.Activate(Vector3.zero);
}
```

그런데 만약 missileFactory 나 buildingManager가 null 이라면 SpawnMissile() 함수가 제대로 작동하지 않을 것입니다. GameManager.cs 에서 미사일 매니저(MissileManager)의 인스턴스를 생성한 다음에 반드시 Initialize() 를 호출해 주어야 하는데, 그것을 빼먹었다면 이런 상황이 발생할 수 있습니다. 따라서 이러한 경우가 발생할 때, 즉시 알 수 있도록 Debug.Assert() 명령어를 이용해서 경고할 수 있게 하겠습니다.

MissileManager.cs

```
void SpawnMissile()
{
    Debug.Assert(this.missileFactory != null, "missile factory is null!");
    Debug.Assert(this.buildingManager != null, "building manager is null!");

    RecycleObject missile = missileFactory.Get();
    missile.Activate(Vector3.zero);
}
```

이제 SpawnMissile() 함수가 완성되었으니 테스트를 해 보겠습니다. Initialize() 함수로 가서 함수를 호출해 주겠습니다.

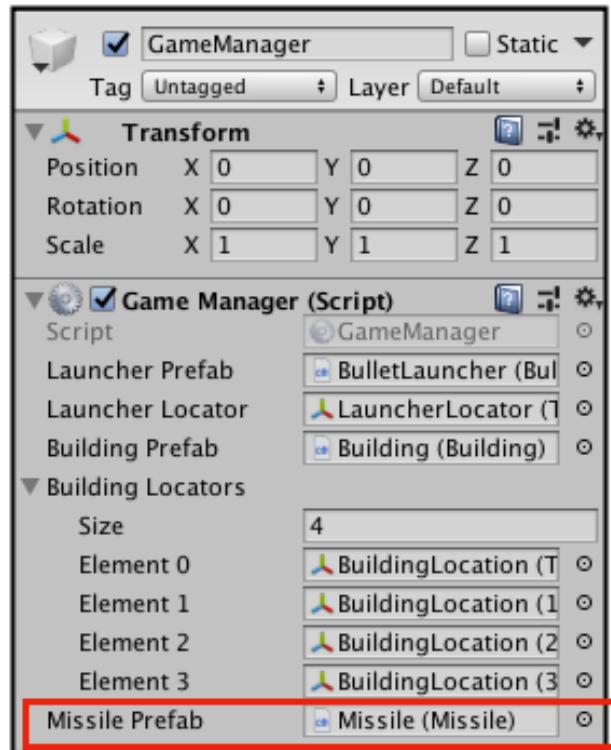
MissileManager.cs

```
public void Initialize(Factory missileFactory, BuildingManager buildingManager)
{
    if (isInitialized)
        return;

    this.missileFactory = missileFactory;
    this.buildingManager = buildingManager;

    isInitialized = true;
    SpawnMissile();
}
```

이제 유니티 에디터로 가서, GameManager 에 미사일 프리팹을 다음과 같이 연결해 준 뒤 플레이 버튼을 눌러서 미사일이 잘 생성되는지 테스트해 보기로 하겠습니다.



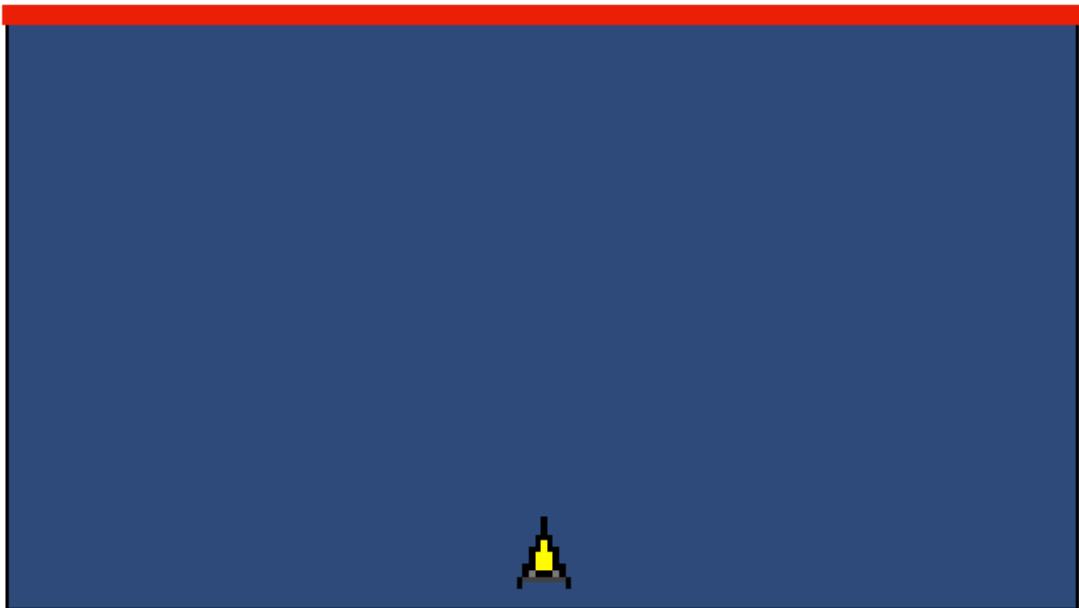
그러면 적의 미사일이 (0,0,0) 위치에 잘 생성되는 것을 확인하실 수가 있을 것입니다.



[동영상 예제 파일명: 058_create_missile_in_the_game.mp4]

화면 상단의 랜덤 위치에서 미사일 생성

지금은 임시로 게임 화면 정 가운데에서 미사일이 생겨나게 했지만, 실제 게임에서는 화면 맨 위의 랜덤 위치에서 미사일을 생성할 예정입니다. 이를 위해서는 게임 화면 맨 윗 부분을 기준으로 랜덤한 위치를 구하는 함수가 필요합니다. 예를 들어 다음 그림의 붉은 색 영역 중 아무 곳이나 랜덤 위치에 생성된 미사일을 배치하려고 하는 것입니다.



이 목적을 위해, MissileManager.cs 에 다음과 같이 GetMissileSpawnPosition() 이라는 이름의 함수를 작성하도록 하겠습니다. 이 함수는 게임 화면 상단의 랜덤 위치를 찾아서 Vector3 값으로 리턴하는 일을 하게 됩니다.

MissileManager.cs

```
public class MissileManager
{
    ...
    void SpawnMissile()
    {
        Debug.Assert(this.missileFactory != null, "missile factory is null!");
        Debug.Assert(this.buildingManager != null, "building manager is null!");

        RecycleObject missile = missileFactory.Get();
        missile.Activate(Vector3.zero);
    }

    Vector3 GetMissileSpawnPosition()
    {
    }
}
```

우선 함수 안에 Vector3 타입의 로컬 변수를 하나 만들고, 초기 값으로 Vector3.zero 를 할당하도록 하겠습니다.

MissileManager.cs

```
Vector3 GetMissileSpawnPosition()
{
    Vector3 spawnPosition = Vector3.zero;
}
```

다음으로는 화면의 왼쪽 상단이 (0,1) 이고 오른쪽 상단이 (1,1) 인 뷰포트(Viewport) 기준의 좌표를 이용해서 랜덤 위치를 구할 예정입니다. 따라서 spawnPosition.x 는 0

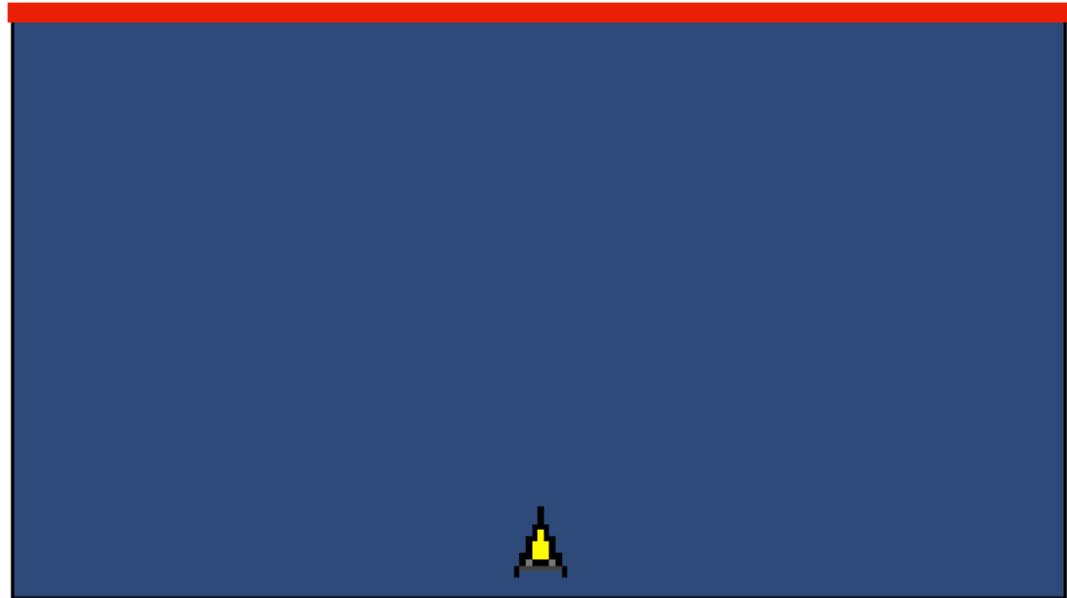
과 1 사이의 랜덤값으로 지정하고, spawnPosition.y 는 1로 지정해 주도록 하겠습니다.

```
MissileManager.cs
Vector3 GetMissileSpawnPosition()
{
    Vector3 spawnPosition = Vector3.zero;
    spawnPosition.x = Random.Range(0f, 1f);
    spawnPosition.y = 1f;
}
```

이렇게 하는 이유는 뷰포트 기준으로는 좌표가 다음과 같이 되기 때문입니다. 따라서 위와 같이 코딩을 하면 spawnPosition 은 (0,1) ~ (1,1) 사이의 랜덤값을 갖게 되어 게임 화면의 맨 위쪽 어딘가를 가리키게 됩니다.

(0,1)

(1,1)



(0,0)

(1,0)

하지만 뷰포트 좌표를 게임 월드에서 바로 사용할 수는 없습니다. 미사일을 게임 월드 안의 원하는 위치에 생성하려면, 이제 이렇게 구한 뷰포트 좌표를 월드 좌표로 변환해야 합니다. 이를 위해 다음과 같이 Camera.main.ViewportToWorldPoint() 를 이용하도록 하겠습니다.

MissileManager.cs

```
Vector3 GetMissileSpawnPosition()
{
    Vector3 spawnPosition = Vector3.zero;
    spawnPosition.x = Random.Range(0f, 1f);
    spawnPosition.y = 1f;

    spawnPosition = Camera.main.ViewportToWorldPoint(spawnPosition);
}
```

이제 미사일이 생성될 화면 상단의 랜덤 위치를 구했으니 이 값을 리턴하면 됩니다. 이 때 spawnPosition의 z 값을 0으로 만든 다음에 리턴해 주는 이유는 혹시라도 뷰포트에서 월드 좌표로 변환하는 과정에서 z 값이 0이 아닌 값이 나올 경우를 대비해서입니다. 지금 우리가 만드는 미사일 커맨더(Missile Commander) 게임은 2D 게임이므로 안전을 위해 이렇게 한 다음에 값을 리턴해 주는 것이 좋습니다.

```
Vector3 GetMissileSpawnPosition()
{
    Vector3 spawnPosition = Vector3.zero;
    spawnPosition.x = Random.Range(0f, 1f);
    spawnPosition.y = 1f;

    spawnPosition = Camera.main.ViewportToWorldPoint(spawnPosition);
    spawnPosition.z = 0f;
    return spawnPosition;
}
```

이제 SpawnMissile() 로 가서, 기존에 임시로 지정한 미사일 생성 위치인 Vector3.zero 를 지워 줍니다.

MissileManager.cs

```
void SpawnMissile()
{
    Debug.Assert(this.missileFactory != null, "missile factory is null!");
    Debug.Assert(this.buildingManager != null, "building manager is null!");

    RecycleObject missile = missileFactory.Get();
    missile.Activate(Vector3.zero);
}
```

그리고 나서 방금 만든 GetMissileSpawnPosition() 함수를 이용하여 미사일의 랜덤 위치를 구해 온 뒤 이 값을 Activate() 의 인자로 넣어 주면 됩니다.

MissileManager.cs

```
void SpawnMissile()
{
    Debug.Assert(this.missileFactory != null, "missile factory is null!");
    Debug.Assert(this.buildingManager != null, "building manager is null!");

    RecycleObject missile = missileFactory.Get();
    missile.Activate(GetMissileSpawnPosition());
}
```

이제 유니티로 가서 테스트해 보겠습니다. 그러면 보시는 것처럼 게임을 새로 실행할 때마다 화면 상단의 랜덤 위치에 미사일이 생겨나는 것을 확인하실 수 있을 것입니다.



[동영상 예제 파일명: 059_create_missile_at_random_position.mp4]

미사일이 생겨나는 시점을 조절하자

미사일의 타겟은 빌딩입니다. 따라서 빌딩의 위치를 알아야만 그쪽으로 향해 방향을 돌려 날아갈 수 있습니다. 빌딩의 위치는 빌딩매니저(BuildingManager)에게 요청하면 알아낼 수 있습니다. 하지만 빌딩은 게임이 시작된 직후에, 다시 말해서 타임 매니저(TimeManager)에서 GameStarted 이벤트를 발생시킨 이후에 만들어지므로, 미사일 생성 시점도 그 때가 되어야만 합니다. 따라서 미사일 매니저(MissileManager)에도 타임 매니저(TimeManager)의 GameStarted 이벤트에 연결할 이벤트 수신 함수를 만들어야 합니다.

이를 위해 먼저 MissileManager.cs 로 가서 다음과 같이 OnGameStarted() 함수를 public 으로 하나 생성하겠습니다.

MissileManager.cs

```
public class MissileManager
{
    ...
    public void OnGameStarted()
    {
        ...
    }
}
```

그리고 임시 테스트를 위해 생성자에서 호출했던 `SpawnMissile()`을 `OnGameStarted()`로 옮겨 줍니다.

```
MissileManager.cs
public class MissileManager
{
    Factory missileFactory;
    BuildingManager buildingManager;

    public void Initialize(Factory missileFactory, BuildingManager buildingManager)
    {
        if (isInitialized)
            return;

        this.missileFactory = missileFactory;
        this.buildingManager = buildingManager;

        isInitialized = true;

        SpawnMissile();
    }

    public void OnGameStarted()
    {
        SpawnMissile();
    }

    ...
}
```

이제 `GameStarted` 이벤트를 수신할 함수를 완성했으므로, 다음으로는 `GameManager.cs`로 이동해서 이벤트 송신자와 수신자를 바인딩시켜 주면 됩니다. `BindEvents()` 와 `UnBindEvents()` 함수에서 다음과 같이 `timeManager.GameStarted` 와 `missileManager.OnGameStarted`를 바인딩/언바인딩 해 줍니다.

```
void BindEvents()
{
    mouseController.FireButtonPressed += launcher.OnFireButtonPressed;
    timeManager.GameStarted += buildingManager.OnGameStarted;
    timeManager.GameStarted += launcher.OnGameStarted;
    timeManager.GameStarted += missileManager.OnGameStarted;
}

void UnBindEvents()
{
    mouseController.FireButtonPressed -= launcher.OnFireButtonPressed;
    timeManager.GameStarted -= buildingManager.OnGameStarted;
    timeManager.GameStarted -= launcher.OnGameStarted;
    timeManager.GameStarted -= missileManager.OnGameStarted;
}
```

이제 유니티에 가서 게임을 실행해 보겠습니다. 그러면 미사일이 처음부터 만들어지 는 것이 아니라 빌딩이 생성되는 순간까지 기다렸다가 생성되는 것을 확인하실 수가 있습니다.



[동영상 예제 파일명: 060_delayed_missile_creation_by_time_manager.mp4]

빌딩 중 랜덤하게 하나를 골라 위치를 가져 오자

적 미사일이 생성되기는 했지만, 타겟이 있어야만 공격할 수 있을 것입니다. 적 미사일이 노릴 수 있는 타겟이라고 하면 아직까지 파괴되지 않고 남아 있는 빌딩일 것입니다. 따라서 이들 빌딩 중에서 무작위로 하나를 골라서 그 위치를 가져오는 함수를 작성하기로 하겠습니다.

이를 위해 먼저 BuildingManager.cs 파일을 열고, 다음과 같이

GetRandomBuildingPosition()이라는 public 함수를 하나 만들겠습니다. 빌딩의 위치를 가져올 것이므로 리턴 값은 Vector3 타입이 될 것입니다.

```
BuildingManager.cs
public class BuildingManager
{
    ...
    public void OnGameStarted()
    {
        CreateBuildings();
    }

    public Vector3 GetRandomBuildingPosition()
    {
    }
}
```

우선, 파괴되지 않고 남아 있는 빌딩이 하나도 없으면 경고 메시지가 뜨도록 Debug.Assert() 명령어부터 추가하도록 하겠습니다.

```
public Vector3 GetRandomBuildingPosition()
{
    Debug.Assert(buildings.Count > 0, "no element in buildings!");
}
```

그 다음에는 빌딩 리스트(buildings) 안에서 랜덤으로 빌딩 하나를 선택한 뒤, 이것의
포지션 값을 리턴하도록 다음과 같이 코드를 작성하겠습니다.

```
public Vector3 GetRandomBuildingPosition()
{
    Debug.Assert(buildings.Count > 0, "no element in buildings!");

    Building building = buildings[Random.Range(0, buildings.Count)];
    return building.transform.position;
}
```

이제 MissileManager.cs로 이동해서, 미사일을 생성할 때 방금 만들었던 빌딩 매니저
의 함수를 사용해 보기로 하겠습니다. MissileManager.cs의 SpawnMissile() 함수로
가보시면, missile.Activate()이라는 함수 사용 부분이 있을 것입니다. Activate() 함
수는 타겟(목표 지점)이 없는 경우에는 인자를 한 개만 전달하면 되고, 타겟이 있는 경
우에는 인자를 두개 전달하도록 부모 클래스인 RecycleObject.cs에서 설정해 놓았습
니다. 따라서 생성된 미사일이 타겟을 가질 수 있도록, missile.Activate() 함수의 두번
째 인자로 빌딩 매니저(buildingManager)의 GetRandomBuildingPosition()를 실행
해서 얻은 값을 다음과 같이 전달해 줍니다.

```
void SpawnMissile()
{
    Debug.Assert(this.missileFactory != null, "missile factory is null!");
    Debug.Assert(this.buildingManager != null, "building manager is null!");

    RecycleObject missile = missileFactory.Get();
    missile.Activate(GetMissileSpawnPosition(),
                     buildingManager.GetRandomBuildingPosition());
}
```

이제 유니티로 가서 테스트해 보도록 하겠습니다. 게임을 실행하고 확인해 보면, 미사일이 만들어지면서 게임 화면 하단에 있는 4개의 빌딩 중 하나를 겨냥하고 있는 것을 보실 수 있을 것입니다.



[동영상 예제 파일명: 061_make_missile_rotate_to_building.mp4]

이제 미사일을 전진시키자

다음으로는 생성된 미사일이 전진하도록 만들겠습니다. 기본 로직은 앞에서 총알 (Bullet)을 이동시키는 방법과 똑같은 간단한 방식입니다. 먼저 Missile.cs 를 비주얼 스튜디오로 연 다음, 이동 속도를 지정하기 위한 변수 moveSpeed 를 선언하고 초기값 을 3으로 지정하겠습니다.

Missile.cs

```
public class Missile : RecycleObject
{
    BoxCollider2D box;
    Rigidbody2D body;

    [SerializeField]
    float moveSpeed = 3f;

    void Awake()
    {
        ...
    }

    ...
}
```

그 다음에는 Update() 에서 isActiveate 값이 false 이면 return 명령어로 더 이상의 코드가 실행되지 않도록 하겠습니다.

Missile.cs

```
void Update()
{
    if (!isActivated)
        return;
}
```

이제 미사일의 머리 방향(위쪽)을 향해 지정된 스피드(moveSpeed)로 이동하도록 코드를 작성하겠습니다.

Missile.cs

```
void Update()
{
    if (!isActivated)
        return;

    transform.position += transform.up * moveSpeed * Time.deltaTime;
}
```

이제 유니티로 가서 테스트해 보겠습니다. 그러면 생성된 미사일이 정확하게 목표 지점인 빌딩을 향해 날아가는 것을 확인하실 수 있을 것입니다.



[동영상 예제 파일명: 062_make_missile_move.mp4]

빌딩과의 충돌을 체크하자

이제 미사일과 빌딩의 충돌 순간을 체크하는 로직을 만들도록 하겠습니다. 여기에서 는 간단하게 Missile.cs 에 유니티 내장 함수인 OnTriggerEnter2D() 를 작성하여, 이 것으로 충돌 여부를 체크하도록 하겠습니다. 우선 Missile.cs 에 다음과 같이 OnTriggerEnter2D() 함수를 만들어 줍니다.

```
Missile.cs
public class Missile : RecycleObject
{
    ...
    // Update is called once per frame
    void Update()
    {
        if (!isActivated)
            return;

        transform.position += transform.up * moveSpeed * Time.deltaTime;
    }

    void OnTriggerEnter2D(Collider2D collision)
    {
    }
}
```

이제 이 트리거 충돌 이벤트가 발생할 경우, 충돌 대상이 무엇인지 알아내야 합니다. 많이 사용하는 방법으로 게임 오브젝트의 이름이나 태그(tag)를 이용하여 비교하는 방법이 있지만, 여기에서는 그냥 해당 게임 오브젝트에 Building 컴포넌트가 붙어 있는지를 확인하여 충돌 대상이 빌딩인지 아닌지를 판단하도록 하겠습니다.

우선 아래와 같이 충돌 대상이 Building이라는 컴포넌트를 가지고 있는지 확인해 봅니다. 다음과 같은 조건 판정을 이용해서 만약 Building이라는 컴포넌트를 가지고 있다면 충돌한 대상이 빌딩이라고 간주합니다.

Missile.cs

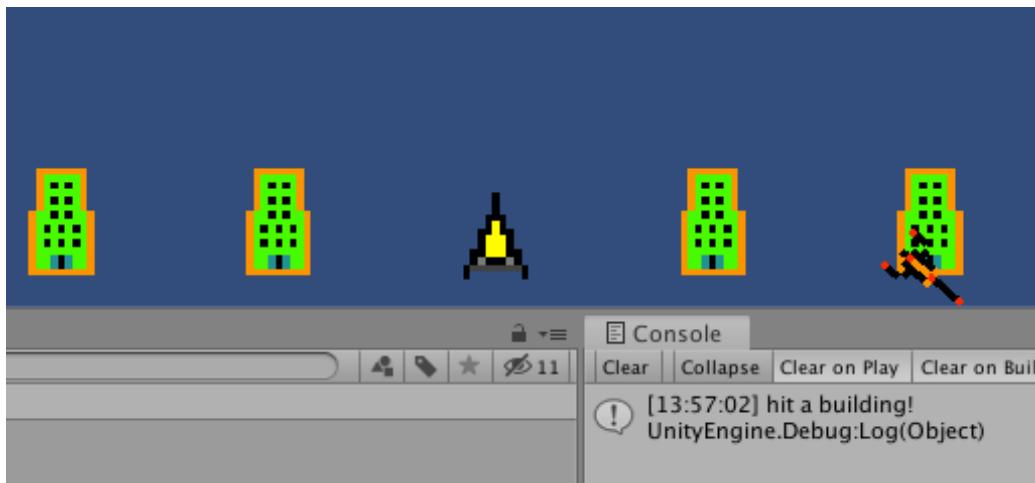
```
void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.GetComponent<Building>() != null)
    {
        }
}
```

이 경우 다음과 같이 “빌딩에 충돌했다”는 의미의 로그 메시지를 남겨 보겠습니다.

Missile.cs

```
void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.GetComponent<Building>() != null)
    {
        Debug.Log("hit a building!");
    }
}
```

이제 유니티 에디터로 가서 게임을 실행해 보겠습니다. 그러면 미사일이 빌딩과 부딪치는 순간 콘솔창에 “hit a building!”이라는 메시지가 표시되는 것을 확인하실 수 있습니다.



[동영상 예제 파일명: 063_detect_missile_collision.mp4]

빌딩과 충돌 순간 미사일을 회수하자

이제 미사일(Missile)이 빌딩에 충돌해서 파괴되었다는 것을 알려 주기 위한 Action 이벤트를 준비하겠습니다. 현재 미사일(Missile)의 부모 클래스인 RecycleObject 에 Destroyed라는 액션이 이미 정의되어 있으니, 별도의 Action 이벤트를 만들 필요 없이 이것을 사용하겠습니다.

먼저 Missile.cs로 가서 다음과 같이 DestroySelf()라는 함수를 생성합니다.

```
Missile.cs
public class Missile : RecycleObject
{
    ...
    void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.GetComponent<Building>() != null)
        {
            Debug.Log("hit a building!");
        }
    }

    void DestroySelf()
    {
    }
}
```

그리고 나서 OnTriggerEnter2D()에서 미사일이 빌딩과 충돌하면 이 함수가 실행되도록 호출해 주겠습니다. 이 때 아까 테스트를 위해 작성했던 Debug.Log() 명령문을 제거합니다.

```

void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.GetComponent<Building>() != null)
    {
        Debug.Log("hit a building!");
        DestroySelf();
    }
}

```

이제 DestroySelf() 함수에 코드를 작성하겠습니다. 여기에서는 미사일이 파괴된 상태가 될 것이므로 isActivated 의 값을 false 로 바꾸고, Destroyed 이벤트를 다음과 같이 축약형으로 발생시킵니다. (앞에서 설명했던 것처럼 Destroyed 액션 이벤트는 부모 클래스인 RecycleObject 에 정의되어 있는 것을 그냥 사용한 것입니다.)

```

void DestroySelf()
{
    isActivated = false;
    Destroyed?.Invoke(this);
}

```

이제 미사일이 파괴되었다는 것을 알리는 이벤트를 만들었으니, 다음으로는 미사일 매니저로 가서 이 이벤트를 수신할 함수를 작성하고 바인딩시키도록 하겠습니다. 먼저 MissileManager.cs 에 다음과 같이 OnMissileDestroyed() 라는 함수를 만들겠습니다. (참고로 이 함수는 미사일 매니저 내부에서 직접 바인딩할 것이라 public 으로 선언하지 않았습니다.)

```

public class MissileManager : MonoBehaviour
{
    ...
    void SpawnMissile()
    {
        RecycleObject missile = missileFactory.Get();
        missile.Activate(GetMissileSpawnPosition(),
                         buildingManager.GetRandomBuildingPosition());
    }

    void OnMissileDestroyed(RecycleObject missile)
    {
    }

    ...
}

```

다음으로는 SpawnMissile() 함수로 가서 missile의 Destroyed 이벤트와, 방금 만든 MissileManager 의 OnMissileDestroyed 이벤트 수신 함수를 바인딩해 주겠습니다. 가독성을 위해 this 라는 키워드를 붙여서 이 이벤트 수신 함수가 미사일 매니저 자신의 함수임을 분명히 드러내 줍니다. (this를 붙이지 않아도 무방하지만 나중에 다른 프로그래머가 이 코드를 보았을 때 좀 더 이해하기 쉽도록 한 것입니다.)

```

void SpawnMissile()
{
    ...
    RecycleObject missile = missileFactory.Get();
    missile.Activate(GetMissileSpawnPosition(),
                     buildingManager.GetRandomBuildingPosition());
    missile.Destroyed += this.OnMissileDestroyed;
}

```

다음으로는 OnMissileDestroyed() 함수로 가서 미사일이 파괴되었을 때의 로직을 작성하겠습니다. 일단 제일 먼저 해야 할 일은 파괴된 미사일로부터 기존에 바인딩된 이벤트를 언바인딩하는 것입니다.

MissileManager.cs

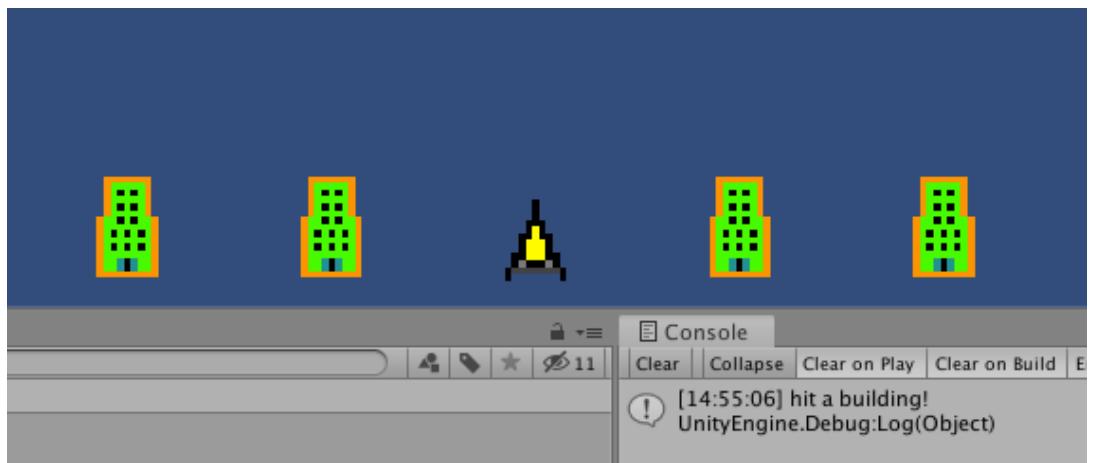
```
void OnMissileDestroyed(RecycleObject missile)
{
    missile.Destroyed -= this.OnMissileDestroyed;
}
```

그리고 파괴된 미사일을 팩토리(missileFactory)의 Restore() 함수를 이용해서 반환, 재활용할 수 있게 해 줍니다.

MissileManager.cs

```
void OnMissileDestroyed(RecycleObject missile)
{
    missile.Destroyed -= this.OnMissileDestroyed;
    missileFactory.Restore(missile);
}
```

이제 유니티로 가서 테스트해 보겠습니다. 게임을 실행하고 조금 기다리면, 미사일이 빌딩에 부딪치는 순간 사라지는 것을 확인하실 수 있을 것입니다.



[동영상 예제 파일명: 064_make_missile_disappear_on_collision.mp4]

충돌 사실을 빌딩에 어떻게 알려 줄 것인가?

다음으로는 미사일이 빌딩과 충돌했다는 사실을 빌딩에게 알려 주어야 합니다. 이를 위해 생각할 수 있는 방법이 두 개 있는데, 첫번 째 방법은 Building 에 public 함수를 하나 만들고 이를 미사일(Missile.cs) 쪽에서 직접 호출하는 방법입니다. 예를 들어서 빌딩에 building.Collide() 와 같은 public 함수가 있다면 그냥 미사일(Missile.cs) 쪽에서 빌딩에 있는 이 함수를 실행하는 것입니다. 이 방법이 사실은 사용하기 편하므로, 많은 프로그래머들이 이런 방법을 사용합니다. 하지만 이 방법은 클래스 외부에서 클래스 내부에 있는 함수에 직접 접근하는 방법이기 때문에 커플링(coupling)의 정도가 다소 높다는 단점이 있습니다.

미사일과 빌딩이 충돌했다는 사실을 빌딩에게 알려 주기 위한 두번 째 방법은, 빌딩 쪽에서도 OnTriggerEnter2D()라는 충돌 이벤트 함수를 이용해서 자신이 미사일과 충돌했다는 사실을 스스로 인지하도록 하는 것입니다. 이 방법은 첫번 째 방법보다 약간 코딩의 양이 늘어나기는 하지만 외부 요소와의 커플링(coupling)이 없기 때문에 좀 더 안전한 방법이라고 할 수 있겠습니다. 저는 두번 째 방법을 선택해서 코딩을 하도록 하겠습니다.

우선 Building.cs 로 이동해서 다음과 같이 유니티 내장 이벤트 수신 함수인 OnTriggerEnter2D() 를 작성해 줍니다. OnTriggerEnter2D() 함수는 트리거 충돌이 일어나는 순간 유니티 엔진이 알아서 호출하는 함수이므로, 그냥 사용하면 됩니다. 이

함수가 실행될 지 여부는 유니티가 알아서 판단하는 것이므로 우리는 그냥 이 안에 코드를 작성해 놓고, 때가 되면 알아서 실행되기를 기다리기만 하면 됩니다.

```
Building.cs  
[RequireComponent(typeof(BoxCollider2D))]  
public class Building : MonoBehaviour  
{  
    ...  
  
    // Update is called once per frame  
    void Update()  
    {  
    }  
  
    void OnTriggerEnter2D(Collider2D collision)  
    {  
    }  
}
```

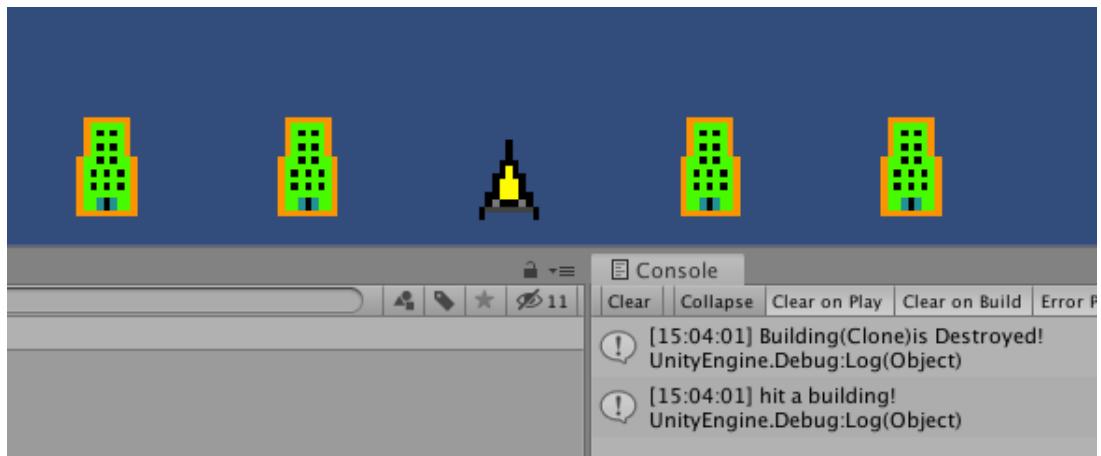
충돌한 대상을 파악하는 방법으로는 앞의 미사일에서 사용한 방법을 똑같이 사용하겠습니다. 다음과 같이 해서, 충돌 대상에게 Missile 컴포넌트가 있는지 확인하겠습니다. 만약 이 컴포넌트가 null 이 아니라면 미사일과 부딪친 것입니다.

```
Building.cs  
void OnTriggerEnter2D(Collider2D collision)  
{  
    if(collision.GetComponent<Missile>() != null)  
    {  
    }  
}
```

이제 조건문 안에 Debug.Log() 명령어를 이용하여 미사일과 충돌했다는 사실을 콘솔창에 표시해 보도록 하겠습니다.

```
void OnTriggerEnter2D(Collider2D collision)
{
    if(collision.GetComponent<Missile>() != null)
    {
        Debug.Log(gameObject.name + "is destroyed!");
    }
}
```

이제 유니티로 가서 게임을 플레이해 보겠습니다. 그러면 미사일이 빌딩과 부딪치는 순간 콘솔창에 빌딩이 파괴되었다는 메시지가 뜨는 것을 확인하실 수 있습니다.



[동영상 예제 파일명: 065_let_building_know_the_collision.mp4]

빌딩 파괴 이벤트를 Action으로 만들자

이제 빌딩이 파괴되었다는 것을 알려 주기 위한 이벤트가 필요합니다. 이를 위해 다음과 같이 Building.cs 에 Action 을 하나 만들겠습니다. 앞에서와 마찬가지로 Action을 사용하기 위해서는 using System; 을 네임스페이스 맨 위에 붙여 주어야 합니다.

만약 빌딩이 총알이나 미사일 같이 RecycleObject 의 파생 클래스였다면 따로 Destroyed 라는 액션을 만들지 않아도 됩니다. 이 액션이 부모 클래스인 RecycleObject 에 이미 정의 되어 있기 때문입니다. 하지만 빌딩은 RecycleObject 가 아닌 보통의 MonoBehavior 기반의 클래스입니다. 다음과 같이 액션을 따로 만들어 주어야 하는 것은 바로 그 때문입니다.

```
Building.cs
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[RequireComponent(typeof(BoxCollider2D))]
public class Building : MonoBehaviour
{
    BoxCollider2D box;

    public Action<Building> Destroyed;

    void Awake()
    {
        box = GetComponent<BoxCollider2D>();
        box.isTrigger = true;
    }

    ...
}
```

다음으로는 OnTriggerEnter2D() 함수에서 아까 작성한 로그 명령문을 삭제하겠습니다. 이제 더 이상 필요 없기 때문입니다. 대신 방금 만든 이벤트를 발생시키도록 다음과 같이 코드를 작성하겠습니다.

```
Building.cs
void OnTriggerEnter2D(Collider2D collision)
{
    if(collision.GetComponent<Missile>() != null)
    {
        Debug.Log(gameObject.name + "is Destroyed!");
        Destroyed?.Invoke(this);
    }
}
```

이제 빌딩이 파괴되면 그것을 알리는 이벤트가 발생합니다. 이 이벤트를 수신할 대상과, 이벤트 수신용 함수가 필요합니다. 현재 빌딩에서 발생하는 이벤트를 수신할 수 있는 것은 빌딩 매니저입니다. 따라서 BuildingManager.cs로 이동해서, 빌딩의 Destroyed 이벤트를 수신하기 위한 함수를 다음과 같이 작성하겠습니다.

```
BuildingManager.cs
public class BuildingManager
{
    ...
    void CreateBuildings()
    {
        ...
    }

    void OnBuildingDestroyed(Building building)
    {
        ...
    }
}
```

이제 CreateBuildings() 로 가서 빌딩과 빌딩 매니저의 이벤트 함수들을 다음과 같이 바인딩해 주겠습니다.

```
BuildingManager.cs
void CreateBuildings()
{
    if (buildings.Count > 0)
    {
        Debug.LogWarning("Buildings have been already created!");
        return;
    }

    for (int i = 0; i < buildingLocators.Length; i++)
    {
        Building building = GameObject.Instantiate(prefab);
        building.transform.position = buildingLocators[i].position;
        building.Destroyed += this.OnBuildingDestroyed;
        buildings.Add(building);
    }
}
```

앞에서 제가 계속 강조했던 것처럼, 이벤트를 바인딩한 다음에는 반드시 언바인딩 처리를 해 주어야 합니다. 이벤트 수신 함수인 OnBuildingDestroyed 에서 다음과 같이 언바인딩을 합니다.

```
BuildingManager.cs
void OnBuildingDestroyed(Building building)
{
    building.Destroyed -= this.OnBuildingDestroyed;
}
```

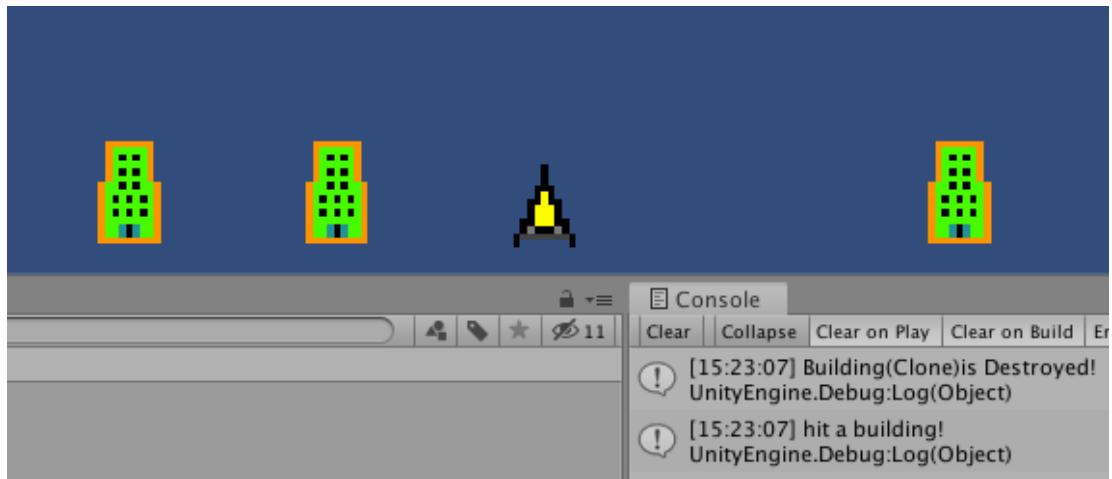
빌딩을 실제로 파괴하기 전에 현재 생성된 모든 빌딩들을 보관하고 있는 buildings 리스트에서 해당 빌딩(파괴될 빌딩)을 제거해야 합니다. 이를 위해 IndexOf() 명령을 이용해 파괴될 빌딩의 index 값을 알아내도록 하겠습니다.

```
void OnBuildingDestroyed(Building building)
{
    building.Destroyed -= OnBuildingDestroyed;
    int index = buildings.IndexOf(building);
}
```

그리고 이렇게 알아낸 인덱스를 이용하여 buildings 리스트에서 파괴될 빌딩을 제거합니다. 그리고 나서 빌딩을 실제로 파괴해야 하는데, 빌딩은 총알이나 미사일과 달리 재활용하지 않으므로 Destroy() 명령을 이용하여 바로 없애 버리겠습니다. 그런데 일반적인 유니티 스크립트에서 하듯 그냥 Destory() 명령을 사용하면 비쥬얼 스튜디오에서 경고 메시지를 보여 줍니다. BuildingManager.cs 가 MonoBehaviour 파생 클래스가 아니라 보통의 C# 클래스이기 때문입니다. 따라서 Destroy() 가 아니라 GameObject.Destroy() 를 사용해서 빌딩을 완전히 없애도록 하겠습니다.

```
void OnBuildingDestroyed(Building building)
{
    building.Destroyed -= this.OnBuildingDestroyed;
    int index = buildings.IndexOf(building);
    buildings.RemoveAt(index);
    GameObject.Destroy(building.gameObject);
}
```

이제 유니티 에디터로 가서 게임을 실행하고 테스트해 보겠습니다. 보시는 것처럼 미사일과 빌딩이 부딪치면 빌딩이 없어지는 것을 확인하실 수가 있습니다.



[동영상 예제 파일명: 066_create_building_destruction_event.mp4]

빌딩 파괴 이펙트 프리팹 만들기

현재 빌딩이 잘 없어지기만 하지만 아무런 파괴 효과 없이 그냥 사라지기 때문에 멋진 합니다. 따라서 미사일과 빌딩이 부딪치는 순간 파괴 이펙트(DestroyEffect)를 만들어 그 자리에 생성하도록 하겠습니다. 이를 위해서 먼저 DestroyEffect.cs라는 유니티 C# 클래스를 만든 다음, MonoBehaviour가 아니라 RecycleObject의 파생 클래스로 다음과 같이 지정해 주겠습니다.

DestroyEffect.cs

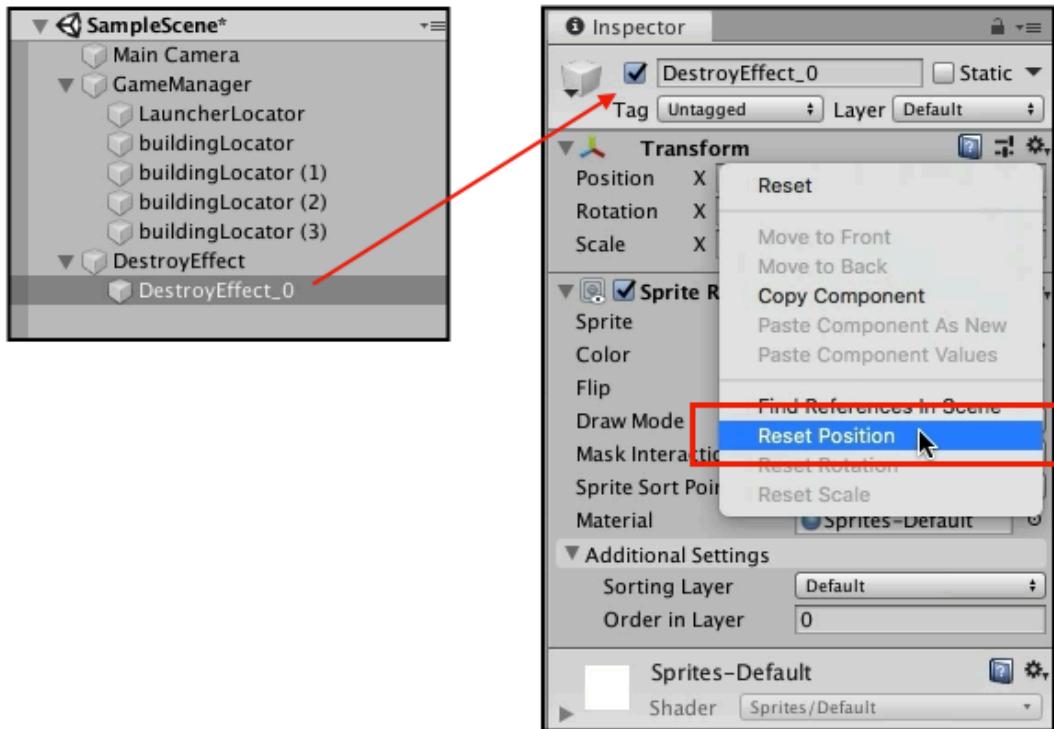
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class DestroyEffect : RecycleObject
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
    }
}
```

다음으로 해야 할 일은 실제 파괴 효과(DestroyEffect)의 프리팹을 만드는 일입니다. 이를 위해 유니티 에디터로 가서 씬에 빈 게임 오브젝트(Empty GameObject)를 하나 만들고 이름을 DestroyEffect 로 변경하도록 하겠습니다. 그리고 프로젝트 뷰에 있는 [Sprites] 폴더에서 DestroyEffect 라는 이름의 스프라이트를 찾아서 드래그한 다음, 앞에서 만든 DestroyEffect 게임 오브젝트 위에 드롭해서 자식 오브젝트로 만들겠습니다. 그리고 나서 이 스프라이트의 로컬 포지션 값이 (0,0,0)이 되도록 한 다음, 위에서 만든 DestroyEffect.cs 라는 스크립트 파일을 연결하도록 하겠습니다.



이제 이 게임 오브젝트를 드래그해서 프로젝트 뷰의 [Prefabs] 폴더에 갖다 놓으면 프리팹 제작이 모두 끝나게 됩니다. 이 상태에서 씬에 남아 있는 게임 오브젝트는 더 이상 필요 없으니 삭제하면 되겠습니다.

[동영상 예제 파일명: 067_destroy_effect_prefab_creation.mp4]

빌딩 파괴 효과(DestroyEffect) 전용 팩토리 만들기

이제 파괴 효과(DestroyEffect)를 실제 게임에서 생성하는 작업을 하겠습니다. 파괴 효과(DestroyEffect)도 팩토리를 이용하여 재활용할 예정이므로, 우선 다음과 같이 빌딩 매니저(BuildingManager.cs)에 파괴 효과를 전담해서 생성하고 관리할 effectFactory라는 팩토리 변수를 하나 선언해 주겠습니다.

```
BuildingManager.cs
public class BuildingManager
{
    Building prefab;
    Transform[] buildingLocators;
    Factory effectFactory;

    List<Building> buildings = new List<Building>();

    ...
}
```

다음으로는 생성자로 가서, 다음 예시와 같이 Factory 타입의 3번째 인자를 추가한 뒤, 이를 방금 만든 변수에 할당하는 코드를 작성하도록 하겠습니다.

```
BuildingManager.cs
public BuildingManager(Building prefab, Transform[] buildingLocators,
                      Factory effectFactory)
{
    this.prefab = prefab;
    this.buildingLocators = buildingLocators;
    this.effectFactory = effectFactory;

    Debug.Assert(this.prefab != null, "null building prefab!");
    Debug.Assert(this.buildingLocators != null, "null buildingLocators!");
}
```

또한 주입받은 effectFactory 가 null 인지 여부를 체크해서 null 일 경우 경고 메시지가 뜨도록 다음과 같이 코드를 작성하겠습니다.

```
BuildingManager.cs
public BuildingManager(Building prefab, Transform[] buildingLocators,
                           Factory effectFactory)
{
    this.prefab = prefab;
    this.buildingLocators = buildingLocators;
    this.effectFactory = effectFactory;

    Debug.Assert(this.prefab != null, "null building prefab!");
    Debug.Assert(this.buildingLocators != null, "null buildingLocators!");
    Debug.Assert(this.effectFactory != null, "null effectFactory!");
}
```

일단 코드는 제대로 작성했지만, 이 상태로는 에러가 날 것입니다. 게임 매니저에서 빌딩 매니저의 인스턴스를 만들 때, 생성자에 3번 째 인자를 전달하지 않았기 때문입니다. 이를 수정해 보겠습니다

[동영상 예제 파일명: 068_declare_effect_factory.mp4]

우선 GameManager.cs 로 가서 다음과 같이 파괴 효과(DestroyEffect) 프리팹을 참조 할 effectPrefab 이라는 이름의 변수를 만들고, [SerializeField] 어트리뷰트를 덧붙여서 유니티 에디터의 인스펙터에 노출되도록 하겠습니다.

GameManager.cs

```
public class GameManager : MonoBehaviour
{
    [SerializeField]
    BulletLauncher launcherPrefab;
    BulletLauncher launcher;

    [SerializeField]
    Transform launcherLocator;

    [SerializeField]
    Building buildingPrefab;

    [SerializeField]
    Transform[] buildingLocators;

    [SerializeField]
    Missile missilePrefab;

    [SerializeField]
    DestroyEffect effectPrefab;

    MouseGameController mouseController;
    BuildingManager buildingManager;
    TimeManager timeManager;
    MissileManager missileManager;

    ...
}
```

다음으로는 Start() 함수로 가서 빌딩 매니저의 인스턴스를 생성할 때, 파괴 효과 (DestroyEffect) 전담 팩토리를 만들어 주입하도록 다음과 같이 코드를 추가하겠습니다. 이 때, 오브젝트의 기본 개수를 2로 지정하겠습니다. 게임에 등장하는 빌딩의 개수

가 몇 개 되지 않기 때문에, 빌딩 파괴 효과를 미리 많이 만들어 둘 필요가 없기 때문입니다.

```
GameManager.cs
void Start()
{
    launcher = Instantiate(launcherPrefab);
    launcher.transform.position = launcherLocator.position;

    mouseController = gameObject.AddComponent<MouseGameController>();
    buildingManager = new BuildingManager(buildingPrefab,
        buildingLocators, new Factory(effectPrefab, 2));
    timeManager = gameObject.AddComponent<TimeManager>();
    missileManager = new MissileManager(new Factory(missilePrefab),
        buildingManager);

    BindEvents();

    timeManager.StartGame(1f);
}
```

이제 BuildingManager.cs 로 다시 가서 나머지 작업을 진행하도록 하겠습니다. 빌딩 파괴 효과는 빌딩이 파괴되는 순간에 사용되어야 하므로, 빌딩 파괴 순간에 호출될 이벤트 수신 함수인 OnBuildingDestroyed() 안에 코드를 추가하도록 하겠습니다.

제일 먼저 할 일은, 파괴될 빌딩이 회수되기 전에 이 빌딩의 위치를 로컬 변수에 미리 저장해 두는 것입니다. 다음 코드처럼 lastPosition 이라는 로컬 변수를 하나 만들고 여기에 파괴될 빌딩의 위치를 저장해 둡니다.

```
BuildingManager.cs
void OnBuildingDestroyed(Building building)
{
    Vector3 lastPosition = building.transform.position;
    building.Destroyed -= OnBuildingDestroyed;
    int index = buildings.IndexOf(building);
    buildings.RemoveAt(index);
```

```
        GameObject.Destroy(building.gameObject);
    }
```

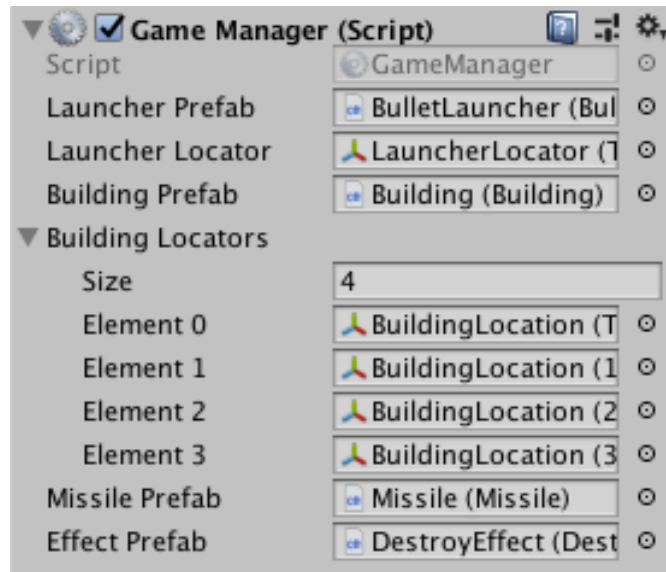
다음으로는 이펙트 팩토리(effectFactory)의 Get() 함수를 이용해서 빌딩 파괴 이펙트를 하나 가져 온 다음, Activate() 함수를 이용하여 아까 저장한 빌딩의 위치에 해당 이펙트를 위치시키도록 하겠습니다.

BuildingManager.cs

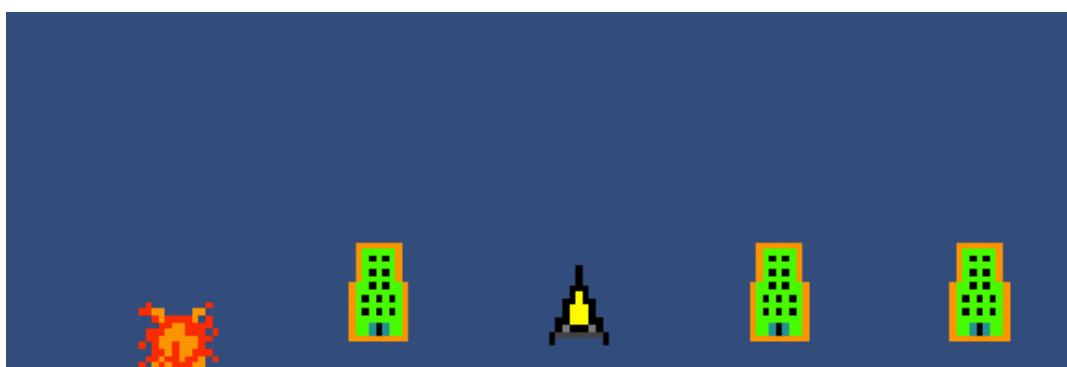
```
void OnBuildingDestroyed(Building building)
{
    Vector3 lastPosition = building.transform.position;
    building.Destroyed -= OnBuildingDestroyed;
    int index = buildings.IndexOf(building);
    buildings.RemoveAt(index);
    GameObject.Destroy(building.gameObject);

    RecycleObject effect = effectFactory.Get();
    effect.Activate(lastPosition);
}
```

이제 유니티 에디터로 가서 테스트해 보겠습니다. 플레이 버튼을 누르기 전에 먼저 GameManager 게임 오브젝트를 선택한 다음에, 인스펙터에서 다음과 같이 파괴 이펙트 프리팹을 effectPrefab 변수에 연결해 줍니다.



그리고 게임을 실행하면 미사일과 빌딩이 충돌할 때, 빌딩의 위치에 이펙트가 잘 생성되는 것을 확인하실 수 있습니다



그런데 빌딩 파괴 이펙트가 생성되는 데에는 문제가 없지만, 한번 생성된 이펙트가 사라지지 않고 있습니다. 또한 생성된 이펙트의 위치가 건물 바닥쪽으로 쏠려 있는 것도 어색해 보입니다. 이것은 현재 빌딩 파괴 이펙트를 건물의 위치에 생성하기 때문입니다. 현재 건물은 중심점이 가운데의 맨 아래쪽으로 설정되어 있습니다. 따라서 빌딩 파괴 이펙트를 건물 위치에 생성하면 지금 보시는 것처럼 이펙트가 너무 아래쪽 위치로 생겨난다는 느낌을 받게 됩니다.

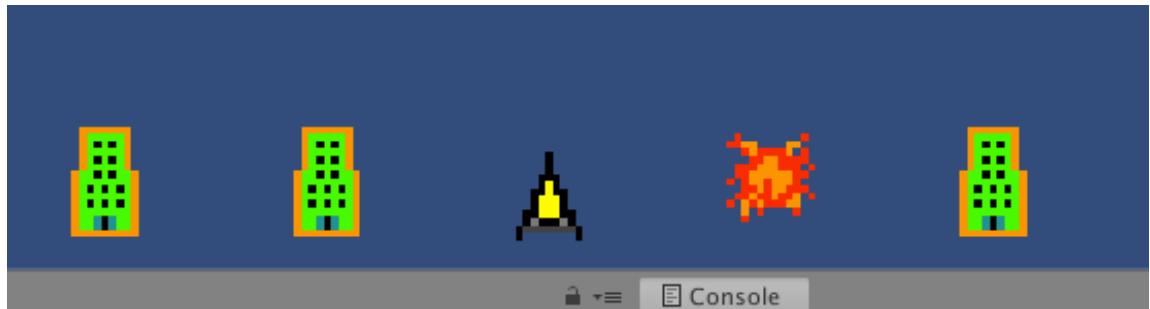
그럼 먼저 빌딩 파괴 이펙트가 생겨나는 위치부터 조정해 보도록 하겠습니다. 이를 위해서는 BuildingManager.cs 로 가서, 파괴된 빌딩의 위치를 저장했던 lastPosition 의 위치 값을 약간 위쪽으로 옮겨 주면 됩니다. 이 때, 옮겨줄 높이는 빌딩의 박스 콜라이더 사이즈의 절반만큼이면 됩니다. 따라서 다음과 같이 lastPosition.y 값에 (박스 콜라이더의 높이 * 0.5) 만큼을 더해 주면 되겠습니다.

BuildingManager.cs

```
void OnBuildingDestroyed(Building building)
{
    Vector3 lastPosition = building.transform.position;
    lastPosition.y += (building.GetComponent<BoxCollider2D>().size.y * 0.5f);
    building.Destroyed -= OnBuildingDestroyed;
    int index = buildings.IndexOf(building);
    buildings.RemoveAt(index);
    GameObject.Destroy(building.gameObject);

    RecycleObject effect = effectFactory.Get();
    effect.Activate(lastPosition);
}
```

이제 유니티 에디터로 가서 게임을 테스트해 보면, 미사일이 빌딩과 부딪칠 때 빌딩 파괴 효과가 정확하게 가운데 위치에 생성되는 것을 확인하실 수 있을 것입니다.



[동영상 예제 파일명: 069_show_destroy_effect_on_collision.mp4]

시간이 지나면 폭발 효과 사라지게 하자

현재는 빌딩 파괴 효과가 사라지지 않고 그대로 남아 있습니다. 따라서 이것을 수정해 보겠습니다.

우선 DestroyEffect.cs 로 가서, 다음과 같이 폭발 효과가 지속될 시간을 지정할 변수(effectTime)를 만들고 [SerializeField] 어트리뷰트를 이용해서 유니티 에디터의 인스펙터에서 값을 수정할 수 있도록 하겠습니다. 그리고 나서 실제로 시간이 얼마나 지났는지를 계산할 때 사용할 변수 elapsedTime 도 그 아래에 선언하겠습니다.

DestroyEffect.cs

```
public class DestroyEffect : RecycleObject
{
    [SerializeField]
    float effectTime = 0.5f;
    float elapsedTime = 0f;

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

다음으로는 Update() 함수로 가서, isActive가 false 일 때는 더 이상의 코드를 실행하지 않고 바로 중단하도록 조건문을 작성합니다.

```
void Update()
{
    if (!isActivated)
        return;
}
```

이후. 경과된 시간이 effectTime 을 넘었는지를 판단하는 조건문을 하나 작성하겠습니다.

```
void Update()
{
    if (!isActivated)
        return;

    elapsedTime += Time.deltaTime;
    if (elapsedTime >= effectTime)
    {
    }
}
```

그리고 조건문 내부에서 elapsedTime 을 다시 0으로 초기화하고, isActivated 를 false 로 바꾸는 코드를 추가하겠습니다. 앞에서 반복적으로 사용한 방법입니다. (이 코드 가 RecycleObject 의 파생클래스들 사이에 반복적으로 사용되고 있으므로, 이 코드 를 매번 이렇게 새로 작성하기보다는 부모 클래스인 RecycleObject 로 옮기는 것도 생각해 볼 수 있습니다. 여기에서는 이 부분이 별로 중요하지 않으므로 그냥 이대로 진행하겠습니다.)

```

void Update()
{
    if (!isActivated)
        return;

    elapsedTime += Time.deltaTime;
    if (elapsedTime >= effectTime)
    {
        elapsedTime = 0f;
        isActivated = false;
    }
}

```

그리고 나서 여기에서 Destroyed 액션 이벤트를 다음과 같이 발생시키면 됩니다.

```

void Update()
{
    if (!isActivated)
        return;

    elapsedTime += Time.deltaTime;
    if (elapsedTime >= effectTime)
    {
        elapsedTime = 0f;
        isActivated = false;

        Destroyed?.Invoke(this);
    }
}

```

이제 다시 BuildingManager.cs로 가서, 앞에서 호출한 Destroyed 이벤트를 수신할 수 있도록 OnEffectDestroyed라는 함수를 만들고, 이를 OnBuildingDestroyed() 함수 안에서 바인딩하도록 하겠습니다.

```

void OnBuildingDestroyed(Building building)
{
    Vector3 lastPosition = building.transform.position;
    lastPosition.y += (building.GetComponent<BoxCollider2D>().size.y * 0.5f);
    building.Destroyed -= OnBuildingDestroyed;
    int index = buildings.IndexOf(building);
    buildings.RemoveAt(index);
    GameObject.Destroy(building.gameObject);

    RecycleObject effect = effectFactory.Get();
    effect.Activate(lastPosition);
    effect.Destroyed += this.OnEffectDestroyed;
}

void OnEffectDestroyed(RecycleObject effect)
{
}

```

다음으로는 OnEffectDestroyed() 로 가서, 바인딩된 이벤트를 다음과 같이 언바인딩 시키겠습니다.

```

void OnEffectDestroyed(RecycleObject effect)
{
    effect.Destroyed -= this.OnEffectDestroyed;
}

```

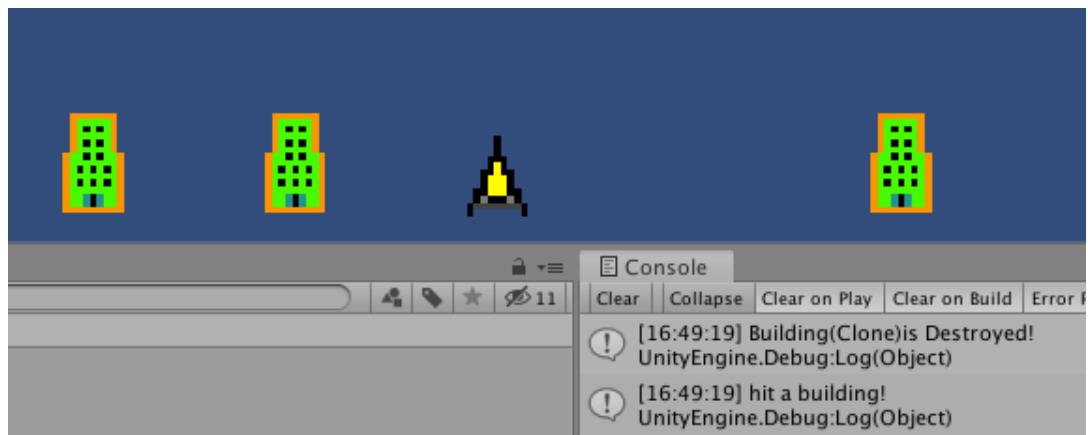
그리고 나서, 사용한 빌딩 폭발 이펙트를 effectFactory에 다시 반납하면 됩니다.

```

void OnEffectDestroyed(RecycleObject effect)
{
    effect.Destroyed -= this.OnEffectDestroyed;
    effectFactory.Restore(effect);
}

```

이제 유니티 에디터로 가서 게임을 테스트해 보겠습니다. 미사일이 빌딩과 부딪치면 빌딩 폭발 효과가 나왔다가 일정한 시간이 지나면 폭발 효과가 사라지는 것을 확인하실 수 있습니다.



[동영상 예제 파일명: 070_hide_destroy_effect.mp4]

미사일을 자동으로 생성하자

이제 미사일이 자동으로 계속 발사되도록 해 보겠습니다. 지정된 최대 개수의 미사일에 도달할 때까지 일정 시간 단위로 계속 미사일이 발사되는 로직을 만들 예정이며, 이렇게 만들어진 미사일은 남아 있는 빌딩 중 하나를 향해 날아가게 될 것입니다. 이를 위해 필요한 변수가 3개가 있는데 다음과 같습니다.

maxMissileCount	생성될 미사일의 최대 개수입니다.
currentMissileCount	현재까지 생성된 미사일의 개수입니다. 이 값이 maxMissileCount 에 도달하면 더 이상 미사일을 만들지 않습니다.
missileSpawnInterval	미사일을 하나 만들고 다음에 다시 만들 때까지의 대기 시간입니다.

그러면 먼저 MissileManager.cs 로 가서, 다음과 같이 3개의 변수를 선언하도록 하겠습니다.

```
MissileManager.cs
public class MissileManager : MonoBehaviour
{
    Factory missileFactory;
    BuildingManager buildingManager;

    bool isInitialized = false;

    int maxMissileCount = 20;
    int currentMissileCount;

    float missileSpawnInterval = 0.5f;

    public void Initialize(Factory missileFactory, BuildingManager buildingManager)
    {
        if (isInitialized)
```

```
        return;

    this.missileFactory = missileFactory;
    this.buildingManager = buildingManager;

    isInitialized = true;
}

...
}
```

다음으로는, 이 중에서 maxMissileCount와 missileSpawnInterval의 값을 외부에서 받을 수 있도록 Initialize()에 추가 인자를 두 개 추가해 주겠습니다. (이렇게 하는 이유는, MissileManager.cs를 처음부터 게임 오브젝트에 붙이는 것이 아니라 GameManager.cs의 Start() 함수에서 MissileManager의 인스턴스를 동적으로 생성하기 때문입니다. 동적으로 생성하기 때문에 유니티 에디터의 인스펙터에서 값을 수동으로 세팅할 수가 없습니다. 따라서 게임 매니저를 통해 이 값을 외부에서 전달 받도록 하려는 것입니다.)

MissileManager.cs

```
public void Initialize(Factory missileFactory, BuildingManager buildingManager,
                      int maxMissileCount, float missileSpawnInterval)
{
    if (isInitialized)
        return;

    this.missileFactory = missileFactory;
    this.buildingManager = buildingManager;

    isInitialized = true;
}
```

이제 앞에서와 마찬가지로 인자로 전달받은 두 개의 값을 MissileManager.cs 클래스 내부 변수에 다음과 같이 할당해 주겠습니다.

```
MissileManager.cs
public void Initialize(Factory missileFactory, BuildingManager buildingManager,
                      int maxMissileCount, float missileSpawnInterval)
{
    if (isInitialized)
        return;

    this.missileFactory = missileFactory;
    this.buildingManager = buildingManager;
    this.maxMissileCount = maxMissileCount;
    this.missileSpawnInterval = missileSpawnInterval;

    isInitialized = true;
}
```

다음으로는 SpawnMissile() 함수로 가서 미사일을 새로 생성할 때마다 currentMissileCount 를 증가시켜서, 현재 얼마나 많은 미사일이 새로 만들어졌는지를 체크할 수 있도록 하겠습니다.

```
MissileManager.cs
void SpawnMissile()
{
    Debug.Assert(this.missileFactory != null, "missile factory is null!");
    Debug.Assert(this.buildingManager != null, "building manager is null!");

    RecycleObject missile = missileFactory.Get();
    missile.Activate(GetMissileSpawnPosition(),
                     buildingManager.GetRandomBuildingPosition());
    missile.Destroyed += OnMissileDestroyed;

    currentMissileCount++;
}
```

[동영상 예제 파일명: 071_automatic_missile_creation_1.mp4]

다음으로는 간단히 코루틴을 이용해서 SpawnMissile() 를 일정 시간 단위로 호출하도록 해 보겠습니다. 이를 위해서 다음과 같이 AutoSpawnMissile() 이라는 이름의 코루틴 함수를 하나 만들도록 하겠습니다.

MissileManager.cs

```
public class MissileManager : MonoBehaviour
{
    ...
    public void OnGameStarted()
    {
        SpawnMissile();
    }

    IEnumerator AutoSpawnMissile()
    {
    }
}
```

그리고 나서 AutoSpawnMissile() 코루틴 함수 안에 while 루프를 만들고, 현재 생성된 미사일의 개수가 최대 미사일 개수보다 작을 때만 다음의 명령이 반복 실행되도록 하겠습니다.

MissileManager.cs

```
IEnumerator AutoSpawnMissile()
{
    while(currentMissileCount < maxMissileCount)
    {
    }
}
```

그리고 나서 yield return new WaitForSeconds() 코루틴 명령어를 이용하여, 일정한 시간(missileSpawnInterval)을 기다린 뒤, SpawnMissile() 을 호출하도록 하겠습니다.

MissileManager.cs

```
IEnumerator AutoSpawnMissile()
{
    while(currentMissileCount < maxMissileCount)
    {
        yield return new WaitForSeconds(missileSpawnInterval);
        SpawnMissile();
    }
}
```

다음으로 OnGameStarted() 함수로 가겠습니다. 이 함수는 게임이 처음 시작될 때 자동으로 실행되므로, 이 안에서 currentMissileCount 변수값을 0으로 초기화시키고, SpawnMissile() 함수의 호출 부분을 삭제해 줍니다. 앞으로 SpawnMissile() 함수는 코루틴 함수에서 호출할 것이기 때문입니다.

MissileManager.cs

```
public void OnGameStarted()
{
    currentMissileCount = 0;
    SpawnMissile();
}
```

그리고 나서 새로운 변수를 하나 선언하겠습니다. 이 변수는 실행한 코루틴을 참조하고 관리하기 위한 것이 목적입니다. 다음과 같이 데이터 타입은 Coroutine 으로 하고, 변수의 이름은 spawningMissile 이라고 해 주겠습니다.

MissileManager.cs

```
public class MissileManager : MonoBehaviour
{
    Factory missileFactory;
    BuildingManager buildingManager;
```

```
bool isInitialized = false;  
  
int maxMissileCount = 20;  
int currentMissileCount;  
  
float missileSpawnInterval = 0.5f;  
  
Coroutine spawningMissile;  
  
...  
}
```

다음으로는 OnGameStarted() 로 다시 가서, 앞에서 작성한 코루틴을 실행한 뒤, 이를 방금 만든 spawnMissile 코루틴 변수에 할당하겠습니다. 사실 이렇게 하지 않아도 코루틴이 작동하는 데에는 아무 문제가 없지만, 나중에 혹시라도 이 코루틴만을 따로 중단시킨다거나 하는 식으로 특별 관리해야 할 경우가 있을 것을 대비한 것입니다.

MissileManager.cs

```
public void OnGameStarted()  
{  
    currentMissileCount = 0;  
    spawningMissile = StartCoroutine(AutoSpawnMissile());  
}
```

이제 미사일 매니저(MissileManager)에서 필요한 작업을 다 마쳤으므로, 다음으로는 게임 매니저(GameManager.cs)를 이용하여 외부에서 maxMissileCount와 destroyedMissileCount 값을 세팅하고, 이것을 미사일 매니저의 인스턴스 (missileManager)에 전달하는 작업을 해 보겠습니다. 이를 위해 먼저 GameManager.cs 를 열고, 다음과 같이 두 개의 변수를 선언한 뒤, 유니티 에디터의 인스턴스에서 값을 변경할 수 있도록 [SerializeField] 어트리뷰트를 각각 붙여 주도록 하겠습니다.

```
public class GameManager : MonoBehaviour
{
    [SerializeField]
    BulletLauncher launcherPrefab;
    BulletLauncher launcher;

    [SerializeField]
    Transform launcherLocator;

    [SerializeField]
    Building buildingPrefab;

    [SerializeField]
    Transform[] buildingLocators;

    [SerializeField]
    Missile missilePrefab;

    [SerializeField]
    DestroyEffect effectPrefab;

    [SerializeField]
    int maxMissileCount = 20;

    [SerializeField]
    float missileSpawnInterval = 0.5f;

    ...
}
```

그리고 나서 Start() 함수로 가서, 미사일 매니저의 인스턴스를 초기화시키는 함수인
missileManager.Initialize()에 방금 만든 두 개의 변수 값을 인자로 추가해 주겠습니다.

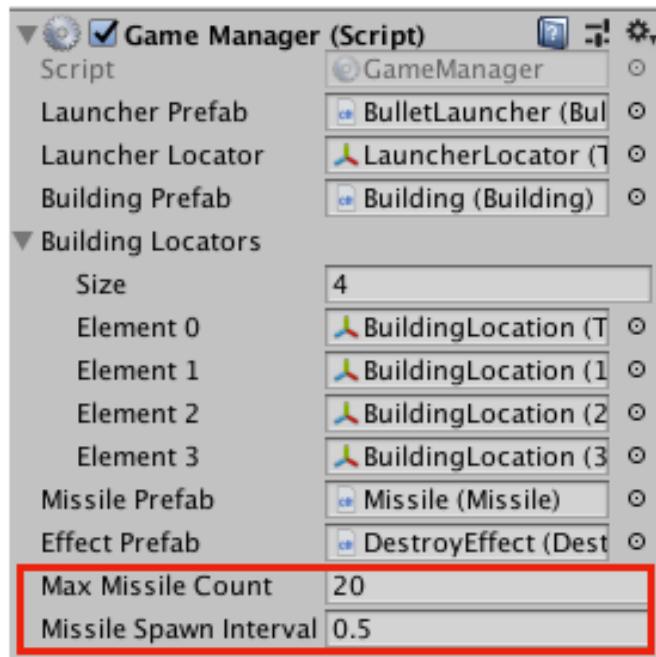
```
void Start()
{
    launcher = Instantiate(launcherPrefab);
    launcher.transform.position = launcherLocator.position;

    mouseController = gameObject.AddComponent<MouseGameController>();
    buildingManager = new BuildingManager(buildingPrefab,
                                           buildingLocators, new Factory(effectPrefab, 2));
    timeManager = gameObject.AddComponent<TimeManager>();
    missileManager = gameObject.AddComponent<MissileManager>();
    missileManager.Initialize(new Factory(missilePrefab), buildingManager,
                               maxMissileCount,missileSpawnInterval);

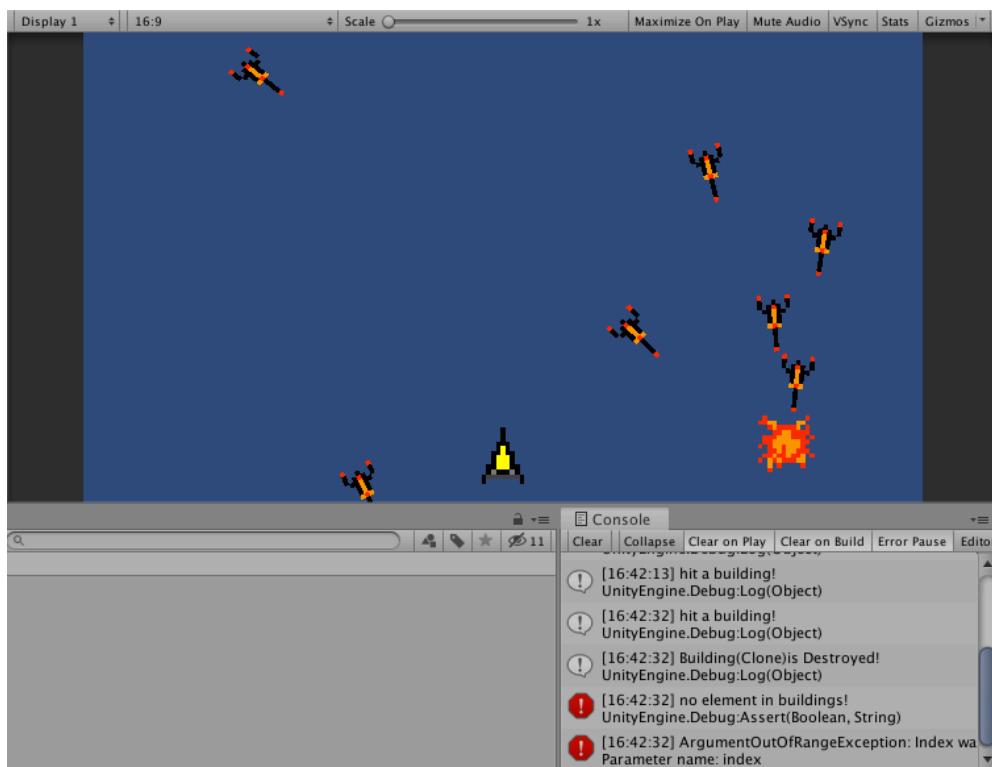
    BindEvents();

    timeManager.StartGame(1f);
}
```

이제 필요한 코딩 작업이 모두 끝났으므로, 유니티 에디터로 가서 테스트해 보도록 하겠습니다. 게임을 실행하기 전에 씬 뷰에서 GameManager 게임 오브젝트를 마우스로 선택하고 인스펙터 뷰를 보시면, 다음과 같이 maxMissileCount와 missileSpawnInterval 값을 변경할 수 있게 되어 있는 것을 확인하실 수 있습니다. (지금은 기본 값을 변경하지 않을 것이므로 확인만 하겠습니다.)



그럼 플레이버튼을 눌러서 게임을 실행해 보겠습니다. 이제 정해진 시간이 지날 때마다 상단의 랜덤 위치에서 미사일이 잘 생겨나지만, 빌딩이 모두 파괴되면 콘솔 창에 에러 메시지가 표시되는 것을 확인하실 수 있습니다. 이것은 빌딩이 모두 파괴된 다음에는 buildings 리스트에 남아 있는 데이터가 하나도 없어서 빌딩의 위치를 더 이상 가져올 수가 없기 때문입니다. 따라서 이 문제를 지금부터 고쳐 보도록 하겠습니다.



[동영상 예제 파일명: 072_automatic_missile_creation_2.mp4]

남아 있는 빌딩이 있는지 확인해 보자

존재하지 않는 빌딩의 위치를 가져 오려는 시도는 에러를 유발합니다. 따라서 이 문제를 해결하기 위해서는 현재 파괴되지 않고 남아 있는 남은 빌딩이 과연 있는지를 알 수 있어야 합니다. 이를 위해, BuildingManager.cs 로 가서 현재 남은 빌딩이 있는지 여부를 알려 주는 읽기 전용 프로퍼티(Property)를 하나 만들도록 하겠습니다.

읽기 전용 프로퍼티이기 때문에 get 만 사용할 것이며, 아래 예시와 같이 buildings.Count 가 0보다 큰 지 여부만을 리턴해 주도록 하면 됩니다.

```
BuildingManager.cs
public class BuildingManager
{
    Building prefab;
    Transform[] buildingLocators;
    Factory effectFactory;

    List<Building> buildings = new List<Building>();

    public bool HasBuilding { get {
        return buildings.Count > 0;
    }
}

...
}
```

이제 MissileManager.cs 로 가서 SpawnMissile() 함수를 보시면, 미사일을 생성하는 코드가 있을 것입니다. 그 앞에 파괴되지 않고 남아 있는 빌딩이 있는지를 확인하는 조건문을 만들고, 남은 빌딩이 없을 경우 더 이상 미사일을 더 생성하지 않도록 return 명령어로 함수를 여기에서 종료하겠습니다.

```

void SpawnMissile()
{
    Debug.Assert(this.missileFactory != null, "missile factory is null!");
    Debug.Assert(this.buildingManager != null, "building manager is null!");

    if (!buildingManager.HasBuilding)
    {
        return;
    }

    RecycleObject missile = missileFactory.Get();
    missile.Activate(GetMissileSpawnPosition(),
                     buildingManager.GetRandomBuildingPosition());
    missile.Destroyed += OnMissileDestroyed;

    currentMissileCount++;
}

```

그리고 return 앞에, “모든 빌딩이 다 파괴되었다(all buildings are destroyed)”는 경고 메시지를 표시하도록 Debug.LogWarning() 명령어를 추가 하겠습니다.

```

void SpawnMissile()
{
    Debug.Assert(this.missileFactory != null, "missile factory is null!");
    Debug.Assert(this.buildingManager != null, "building manager is null!");

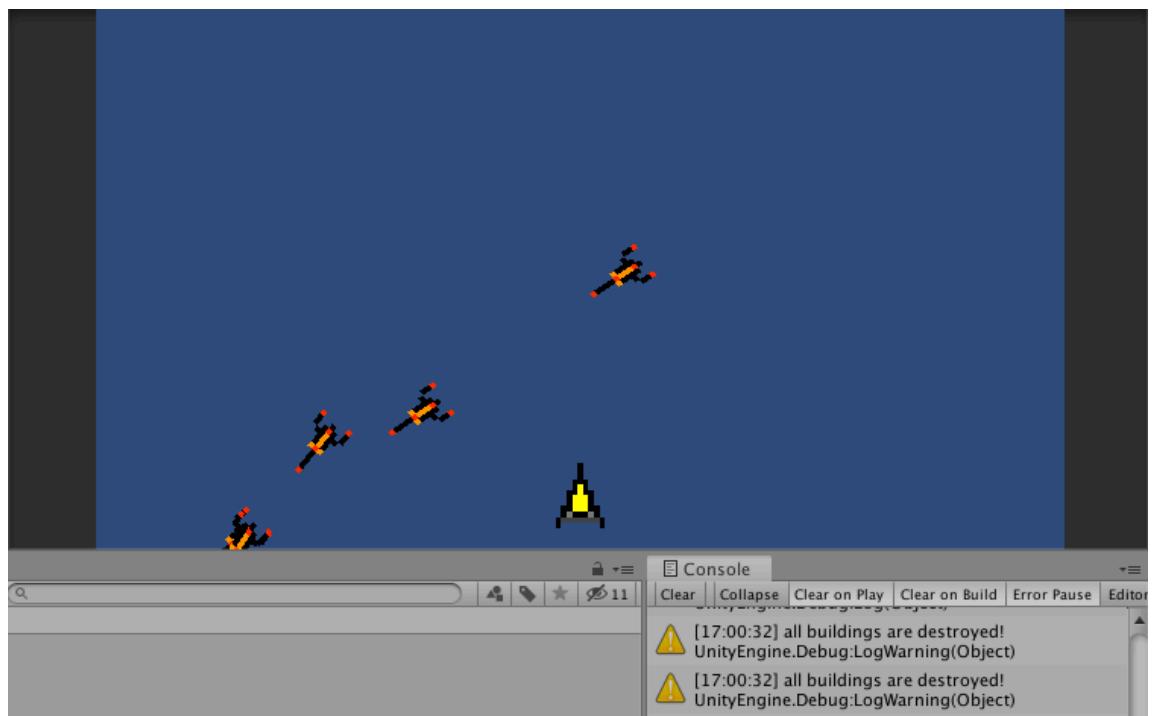
    if (!buildingManager.HasBuilding)
    {
        Debug.LogWarning("all buildings are destroyed!");
        return;
    }

    RecycleObject missile = missileFactory.Get();
    missile.Activate(GetMissileSpawnPosition(),
                     buildingManager.GetRandomBuildingPosition());
    missile.Destroyed += OnMissileDestroyed;

    currentMissileCount++;
}

```

이제 유니티 에디터로 가서 테스트해 보도록 하겠습니다. 게임을 실행하고 기다려 보시면, 빌딩이 다 파괴되고 나면 더 이상 미사일을 생성하지 않는 것을 알 수 있습니다. 하지만 콘솔창을 보시면 “all buildings are destroyed!”라는 경고 메시지는 중단되지 않고 계속 표시되고 있습니다. 이는 코루틴 함수가 계속 실행되고 있기 때문입니다. 이것을 고쳐 보겠습니다.



[동영상 예제 파일명: 073_check_if_building_exists.mp4]

현재, 남아 있는 빌딩이 더 이상 존재하지 않아서 미사일을 추가로 만들지 않는다고 해도 코루틴은 계속 반복됩니다. 이는 잔여 미사일을 확인하는 조건문이 SpawnMissile() 함수에 들어 있기 때문입니다. 따라서 이 조건문을 SpawnMissile() 함수가 아니라 코루틴 함수에서 실행하도록 코드를 수정하겠습니다. 이를 위해 먼저 SpawnMissile() 함수안에 있는 조건문을 다음과 같이 삭제해 줍니다. (삭제하기 보다는 잘라내기 – 붙이기를 하셔도 됩니다.)

MissileManager.cs

```
void SpawnMissile()
{
    Debug.Assert(this.missileFactory != null, "missile factory is null!");
    Debug.Assert(this.buildingManager != null, "building manager is null!");

    if (!buildingManager.HasBuilding)
    {
        Debug.LogWarning("all buildings are destroyed!");
        return;
    }

    RecycleObject missile = missileFactory.Get();
    missile.Activate(GetMissileSpawnPosition(),
                      buildingManager.GetRandomBuildingPosition());
    missile.Destroyed += OnMissileDestroyed;

    currentMissileCount++;
}
```

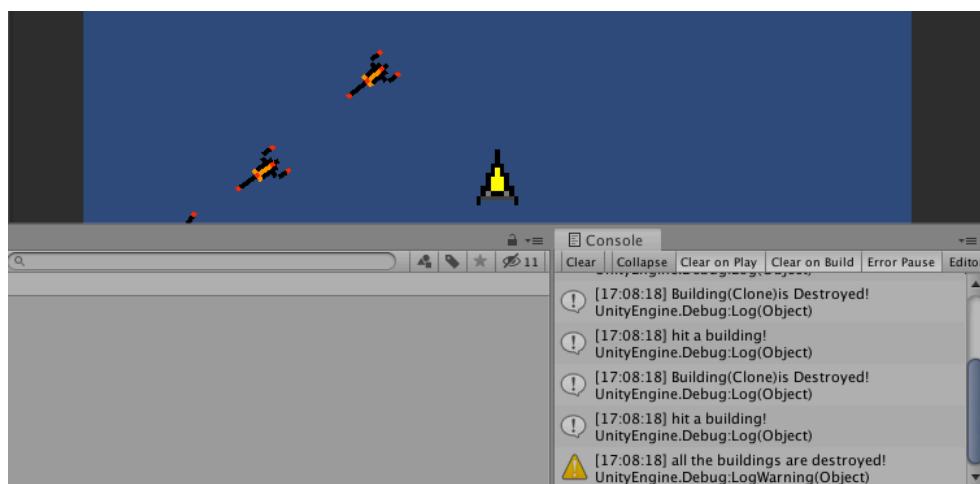
다음으로는 AutoSpawnMissile() 코루틴 함수로 가서 앞의 조건문을 이곳에 작성합니다. 다만, 앞의 조건문과 다른 부분이 한 곳 있습니다. 코루틴 함수를 중단시키기 위해서는 yield break 구문을 사용해야 하기 때문입니다. 따라서 원래 있던 return; 대신에 yield break; 를 이용하여 아래와 같이 코드를 작성하시면 됩니다.

```
IEnumerator AutoSpawnMissile()
{
    while(currentMissileCount < maxMissileCount)
    {
        yield return new WaitForSeconds(missileSpawnInterval);

        if (!buildingManager.HasBuilding)
        {
            Debug.LogWarning("all the buildings are destroyed!");
            yield break;
        }

        SpawnMissile();
    }
}
```

이제 다시 유니티 에디터로 가서 게임을 플레이해 보시면, 모든 빌딩이 파괴되었을 때의 경고 메시지가 단 1번만 나타나는 것을 알 수 있습니다. 이는 코루틴 자체가 중단되었기 때문입니다.



[동영상 예제 파일명: 074_stop_coroutine_if_no_buildings.mp4]

미사일과 총알의 폭발효과의 충돌 처리

미사일과 빌딩의 충돌을 구현했으니, 이제 미사일과 총알의 폭발 이펙트의 충돌시 미사일이 파괴되도록 하겠습니다. 역시 빌딩과 충돌했을 경우와 마찬가지로, 미사일(Missile.cs) 쪽에서 OnTriggerEnter2D() 유니티 이벤트 함수를 사용하는 방식으로 구현하겠습니다.

빌딩과 충돌했을 때는 충돌 관련한 로직을 미사일과 빌딩 양쪽 모두에서 구현해야 했지만, 총알 폭발 효과는 시간이 지나면 어차피 자동으로 사라지므로 신경 쓸 필요가 없고, 오직 미사일(Missile.cs) 쪽에서만 관련된 코딩을 하면 됩니다.

그럼 Missile.cs 의 OnTriggerEnter2D() 함수로 가 보겠습니다. 빌딩과의 충돌 여부를 판단할 때와 마찬가지 방법으로, 충돌 대상에게 Explosion이라는 컴포넌트가 붙어 있는지를 확인하는 조건문을 기존의 조건문 하단에 새로 추가해 줍니다.

Missile.cs

```
void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.GetComponent<Building>() != null)
    {
        DestroySelf();
    }

    if (collision.GetComponent<Explosion>() != null)
    {
    }
}
```

그런데 첫번 째 조건문에서 미사일과 충돌한 대상이 빌딩이라는 것이 확인되면, 그 다음 조건문을 실행할 필요가 없습니다. 왜냐하면 충돌 대상이 빌딩이면서 동시에 폭발 이펙트일 수는 없기 때문입니다. 따라서 빌딩과 충돌했는지를 체크하는 첫번 째 조건문에 다음과 같이 return 명령문을 넣어서, 빌딩과 충돌한 경우에는 여기에서 함수 종료시키겠습니다.

Missile.cs

```
void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.GetComponent<Building>() != null)
    {
        DestroySelf();
        return;
    }

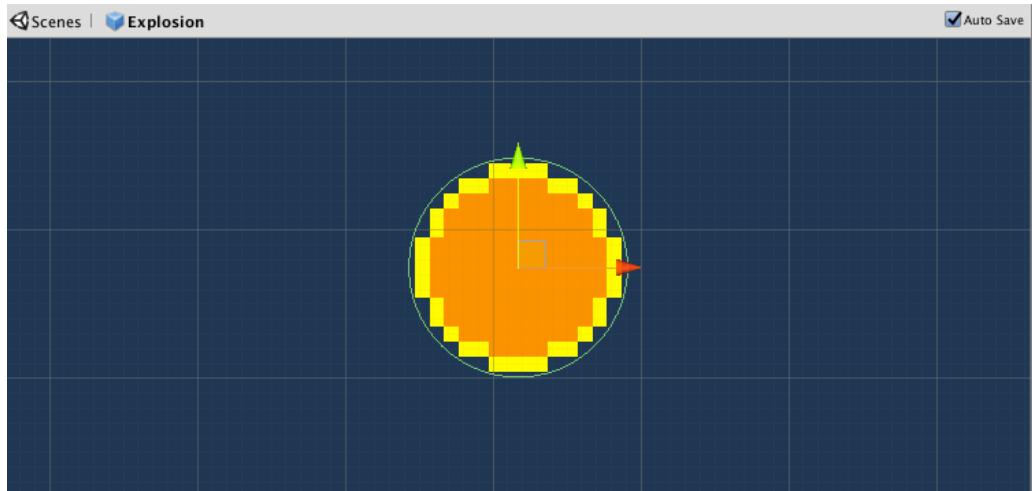
    if (collision.GetComponent<Explosion>() != null)
    {
    }
}
```

이제 두번 째 조건문을 작성할 차례입니다. 다음과 같이 “hit by a explosion!”이라는 디버그 메시지와 함께 DestroySelf() 함수를 실행하고, 하단에 return 명령어를 넣어 함수를 종료합니다.

```
void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.GetComponent<Building>() != null)
    {
        DestroySelf();
        return;
    }

    if (collision.GetComponent<Explosion>() != null)
    {
        Debug.Log("hit by a explosion!");
        DestroySelf();
        return;
    }
}
```

이제 유니티 에디터로 가서 테스트해 보겠습니다. 먼저 [prefabs] 폴더에서 explosion 프리팹을 찾아 편집모드로 들어간 다음, CircleCollider2D 의 radius 값을 조금 키워서 스프라이트보다 충돌 영역이 약간 커지게 세팅합니다.



이제 프리팹 편집 모드를 빠져 나와 플레이 버튼을 눌러 게임을 실행합니다. 날아오는 미사일들을 겨냥하여 총알을 발사하면, 총알이 폭발하면서 생겨난 폭발 효과들이 미사일과 충돌하는 순간 콘솔창에 “hit by a explosion!”라는 메시지가 표시되면서 미사일이 사라지는 것을 확인하실 수 있습니다.



[동영상 예제 파일명: 075_collision_missile_explosion.mp4]

ScoreManager 로 점수를 계산하자

이제 기본적인 게임 로직은 다 구현하였습니다. 다음으로는 게임을 통해 스코어(점수)가 변하는 부분을 처리해 보겠습니다. 역시 이번에도 하나의 클래스가 하나의 임무를 담당한다는 원칙에 따라 스코어 계산만을 전담하는 스코어 매니저(ScoreManager)라는 클래스를 만들겠습니다. 여기에서는 플레이어의 활동에 따른 스코어(점수)의 변화를 처리할 예정입니다.

참고로 스코어 매니저의 경우에도 외부와의 커뮤니케이션은 이벤트로 할 것입니다. 적 미사일 파괴와 같은 스코어 변동 조건이 발생할 때마다 관련된 이벤트를 수신해서 게임의 스코어를 업데이트하고, 이 결과를 다시 이벤트를 통해 전송하여 게임 UI 등에서 사용할 수 있도록 할 예정입니다. 우선 가장 기본적인 기능부터 구현하도록 하겠습니다.

우선 유니티에서 ScoreManager.cs 라는 이름의 C# 스크립트를 만들겠습니다. 그리고 다음과 같이 MonoBehaviour 관련된 것들을 모두 지워서 일반 클래스로 만들도록 하겠습니다. (꼭 일반 클래스로 만들어서 사용해야 하냐고 물으신다면 그렇지 않습니다. Monobehaviour 의 파생 클래스로 만들어도 상관 없습니다. 다만 저는 생성자를 사용하고 싶기 때문에 일반 클래스로 만든 것입니다.)

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ScoreManager : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}

```

이제 미사일을 하나 파괴할 때마다 얻을 기본 점수와 게임이 끝나고 남아 있는 벌딩 개수에 따른 보너스 점수를 지정할 변수들을 다음과 같이 선언하도록 하겠습니다.

```

public class ScoreManager
{
    int scorePerMissile;
    int scorePerBuilding;
}

```

이들 점수들은 모두 private로 선언되었습니다. 따라서 외부에서 값을 전달하기 위해서는 생성자를 사용하거나 아니면 프로퍼티(property) 등을 사용해야 할 것입니다. 여기에서 저는 생성자를 통해서만 외부에서 값을 전달 받도록 할 예정입니다. 생성자를 통해 값을 전달 받는다는 것은, 게임이 진행되는 동안 단 한번만 이 값을 업데이트할

수 있다는 의미가 됩니다. 왜냐하면 생성자는 클래스의 인스턴스를 만들 때 단 한번만 실행되기 때문입니다.

또한, 이 변수들의 값은 생성자를 통해 한번 업데이트 한 다음에는 게임 실행 도중 중간에 절대로 바뀌는 일이 없도록 할 예정입니다. 이런 경우에 사용할 수 있는 키워드가 바로 `readonly` 입니다. 이 키워드를 붙인 변수는 변수를 처음 선언할 때와, 생성자를 실행할 때 외에는 절대로 값을 변경할 수가 없습니다. 따라서 형태는 변수이지만 사실상 상수처럼 사용할 수 있는 것입니다.

```
ScoreManager.cs
public class ScoreManager
{
    readonly int scorePerMissile;
    readonly int scorePerBuilding;
}
```

다음으로는 실제 게임의 스코어를 저장할 변수를 다음과 같이 생성하도록 하겠습니다.

```
ScoreManager.cs
public class ScoreManager
{
    readonly int scorePerMissile;
    readonly int scorePerBuilding;

    int score;
}
```

이제 생성자를 만들고, 이 클래스의 인스턴스가 만들어지는 순간 클래스 외부로부터 ‘미사일을 한개 파괴할 때의 점수(scorePerMissile)’와 ‘파괴되지 않고 남아 있는 빌딩 1개당 보너스 점수(scorePerBuilding)’를 지정받도록 하겠습니다.

ScoreManager.cs

```
public class ScoreManager
{
    readonly int scorePerMissile;
    readonly int scorePerBuilding;

    int score;

    public ScoreManager(int scorePerMissile = 50, int scorePerBuilding = 5000)
    {
    }

}
```

다음으로는 생성자로부터 전달받은 값들을 내부 변수들에 할당하겠습니다. 앞에서도 말씀 드린 것처럼, Readonly로 지정된 변수들은 생성자에서만 단 한번 값을 변경할 수 있습니다.

ScoreManager.cs

```
public class ScoreManager
{
    readonly int scorePerMissile;
    readonly int scorePerBuilding;

    int score;

    public ScoreManager(int scorePerMissile = 50, int scorePerBuilding = 5000)
    {
        this.scorePerMissile = scorePerMissile;
        this.scorePerBuilding = scorePerBuilding;
    }
}
```

만약 readonly로 지정된 변수의 값을 생성자가 아닌 다른 함수에서 변경하려고 하면 어떻게 될까요? 아래 스크린샷처럼 다른 함수에서 scorePerMissile 값을 변경하려고 하면 비쥬얼 스튜디오에서 경고 메시지를 보여줄 것입니다. (참고로 아래 이미지의 함수는 설명을 위해 임시로 만든 예시용입니다. 여러분은 아래 함수를 코드에 작성해 넣으시면 안됩니다.)

```
void UpdateScoreMissile()
{
    scorePerMissile = 100;
}
F (필드) readonly int ScoreManager.scorePerMissile
읽기 전용 필드에는 할당할 수 없습니다. 단 생성자 또는 변수 초기화에서 예외입니다.
```

다음으로는 스코어를 변경하기 위한 함수를 작성하겠습니다. 게임 스코어가 갱신되기 위한 첫번 째 조건은 적의 미사일이 파괴되는 경우입니다. 이를 위해서는 미사일이 파괴되었다는 것을 스코어 매니저(ScoreManager)가 알 수 있어야 합니다. 현재까지 미사일이 파괴되었다는 것을 알 수 있는 존재는 미사일(Missile) 자신과 미사일 매니저(MissileManager)입니다. 이 중 누구를 통해서 미사일 파괴 이벤트를 수신하게 될지는 모르겠지만, 어쨌건 미사일 파괴 이벤트가 발생하는 경우 이를 수신하여 score를 업데이트하기 위한 함수를 다음과 같이 만들어 보겠습니다.

ScoreManager.cs

```
public class ScoreManager
{
    readonly int scorePerMissile;
    readonly int scorePerBuilding;

    int score;

    public ScoreManager(int scorePerMissile = 50, int scorePerBuilding = 5000)
    {
        this.scorePerMissile = scorePerMissile;
        this.scorePerBuilding = scorePerBuilding;
    }

    public void OnMissileDestroyed()
    {

    }
}
```

OnMissileDestroyed 이벤트 함수가 호출되었다는 것은 적의 미사일이 파괴되었다는 이야기입니다. 따라서 이 함수가 호출될 때마다 score 값을 scorePerMission에서 지정된 값만큼 증가시키는 코드를 작성해 넣도록 하겠습니다.

```
public void OnMissileDestroyed()
{
    score += scorePerMissile;
}
```

다음으로는 스코어가 변경되면 이를 외부에 송신하는 Action 이벤트를 생성하도록 하겠습니다. Action 을 사용하기 위해서는 using System; 이 필요합니다. 이것을 다음과 같이 네임스페이스 최상단에 추가하고, 그 아래에 ScoreChanged 라는 Action<int> 타입의 액션 이벤트를 선언하도록 하겠습니다. (Action<int> 를 사용하는 이유는 이벤트가 발생할 때마다 score 를 이벤트에 담아서 전송하기 위해서입니다.)

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ScoreManager
{
    readonly int scorePerMissile;
    readonly int scorePerBuilding;

    int score;

    public Action<int> ScoreChanged;

    public ScoreManager(int scorePerMissile = 50, int scorePerBuilding = 5000)
    {
        this.scorePerMissile = scorePerMissile;
        this.scorePerBuilding = scorePerBuilding;
    }

    public void OnMissileDestroyed()
    {
        score += scorePerMissile;
    }
}
```

이제 OnMissileDestroyed() 함수로 가서, 미사일 파괴로 인해 스코어가 변경되면 이를 방금 만든 ScoreChanged 이벤트에 담아서 송신하는 코드를 추가하도록 하겠습니다. 아래와 같이 score 를 함께 전송하는 것을 잊지 마시기 바랍니다.

ScoreManager.cs

```
public void OnMissileDestroyed()
{
    score += scorePerMissile;
    ScoreChanged?.Invoke(score);
}
```

그 아래에 테스트를 위해 Debug.Log() 를 이용하여 현재 변경된 점수를 콘솔창에 표시할 수 있도록 하겠습니다.

ScoreManager.cs

```
public void OnMissileDestroyed()
{
    score += scorePerMissile;
    ScoreChanged?.Invoke(score);
    Debug.Log("Score: " + score);
}
```

[동영상 예제 파일명: 076_score_manager_class_creation.mp4]

이제 스코어 매니저(ScoreManager.cs)의 기본적인 코딩이 완료되었으니, 다음으로는 게임 매니저(GameManager.cs)로 가서 스코어 매니저의 인스턴스를 생성해야 합니다. 이를 위해서 먼저 인스턴스를 저장할 변수를 하나 선언해 줍니다.

GameManager.cs

```
public class GameManager : MonoBehaviour
{
    [SerializeField]
    BulletLauncher launcherPrefab;
    BulletLauncher launcher;

    [SerializeField]
    Transform launcherLocator;

    [SerializeField]
    Building buildingPrefab;

    [SerializeField]
    Transform[] buildingLocators;

    [SerializeField]
    Missile missilePrefab;

    [SerializeField]
    DestroyEffect effectPrefab;

    [SerializeField]
    int maxMissileCount = 20;

    [SerializeField]
    float missileSpawnInterval = 0.5f;

    MouseGameController mouseController;
    BuildingManager buildingManager;
    TimeManager timeManager;
    MissileManager missileManager;
    ScoreManager scoreManager;

    ...
}
```

게임 매니저에서 스코어 매니저의 인스턴스를 만들 때 ‘미사일 1개 파괴 당 점수’와 ‘남아 있는 건물 1개’당 점수를 생성자에 전달해야만 합니다. 이렇게 전달할 값을 게임 매니저의 인스펙터를 통해 설정할 수 있도록 다음과 같이 관련 변수들을 만들고 [SerializeField] 어트리뷰트를 덧붙여 유니티 에디터의 인스펙터에서 값을 수정할 수 있도록 하겠습니다.

GameManager.cs

```
public class GameManager : MonoBehaviour
{
    ...
    [SerializeField]
    int scorePerMissile = 50;

    [SerializeField]
    int scorePerBuilding = 5000;

    MouseGameController mouseController;
    BuildingManager buildingManager;
    TimeManager timeManager;
    MissileManager missileManager;
    ScoreManager scoreManager;

    ...
}
```

다음으로는 Start()에서 스코어 매니저의 인스턴스를 생성하면서 다음과 같이 scorePerMissile과 scorePerBuilding을 인자로 전달하도록 하겠습니다.

```
void Start()
{
    launcher = Instantiate(launcherPrefab);
    launcher.transform.position = launcherLocator.position;

    mouseController = gameObject.AddComponent<MouseGameController>();
    buildingManager = new BuildingManager(buildingPrefab,
                                           buildingLocators, new Factory(effectPrefab, 2));
    timeManager = gameObject.AddComponent<TimeManager>();
    missileManager = gameObject.AddComponent<MissileManager>();
    missileManager.Initialize(new Factory(missilePrefab), buildingManager,
                               maxMissileCount, missileSpawnInterval);
    scoreManager = new ScoreManager(scorePerMissile, scorePerBuilding);

    BindEvents();

    timeManager.StartGame(1f);
}
```

[동영상 예제 파일명: 077_create_score_manager_instance.mp4]

다음으로 해야 할 일은, 미사일이 파괴되었다는 것을 스코어 매니저(ScoreManager)가 알 수 있도록 하는 것입니다. 이를 위해서는 미사일을 관리하는 미사일 매니저에서 미사일과 관련된 주요 이벤트들을 발생시키도록 해야 합니다. 따라서 MissileManager.cs 로 가서 필요한 작업을 하겠습니다.

현재 미사일 매니저(MissileManager.cs)에서는 미사일을 자동으로 생성하기는 하지만, 이렇게 만들어진 미사일들을 추적, 관리하지는 못하고 있습니다. 따라서 만들어진 모든 미사일들의 상태를 추적할 수 있도록 미사일들을 리스트에 담아 관리해야만 합니다. 이를 위해 다음과 같이 MissileManager.cs 로 가서, 미사일을 담기 위한 RecycleObject 리스트를 하나 만들겠습니다.

MissileManager.cs

```
public class MissileManager : MonoBehaviour
{
    Factory missileFactory;
    BuildingManager buildingManager;

    bool isInitialized = false;

    int maxMissileCount = 20;
    int currentMissileCount;
    int destroyedMissileCount;

    float missileSpawnInterval = 0.5f;

    Coroutine spawningMissile;

    List<RecycleObject> missiles = new List<RecycleObject>();

    ...
}
```

다음으로는 미사일을 실제로 만들어내는 함수인 `SpawnMissile()`로 가서, 미사일을 새로 만들 때마다 방금 만든 `missiles` 리스트에 추가해서 관리할 수 있도록 하겠습니다.

다음과 같이 `missiles.Add()`를 이용하면 됩니다.

MissileManager.cs

```
void SpawnMissile()
{
    Debug.Assert(this.missileFactory != null, "missile factory is null!");
    Debug.Assert(this.buildingManager != null, "building manager is null!");

    RecycleObject missile = missileFactory.Get();
    missile.Activate(GetMissileSpawnPosition(),
                     buildingManager.GetRandomBuildingPosition());
    missile.Destroyed += OnMissileDestroyed;
    missiles.Add(missile);

    currentMissileCount++;
}
```

다음으로는 미사일이 파괴된 경우에 대한 처리를 업데이트하겠습니다.

`OnMissileDestroyed()`로 가서 미사일이 파괴되면 `missileFactory`에 반납하기 전에 파괴된 미사일을 `missiles` 리스트에서 제거해야 합니다. 앞에서 했던 것과 마찬가지 방법으로 반납할(파괴된) 미사일의 인덱스를 가져온 다음, `RemoveAt()` 함수를 이용하여 이를 `missiles` 리스트에서 제거하면 됩니다.

MissileManager.cs

```
void OnMissileDestroyed(RecycleObject missile)
{
    missile.Destroyed -= OnMissileDestroyed;
    int index = missiles.IndexOf(missile);
    missiles.RemoveAt(index);
    missileFactory.Restore(missile);
}
```

다음으로는 미사일이 파괴되었을 때 이를 외부에 알리는 이벤트를 발생시켜야 합니다. 물론 개별 미사일들이 자신이 파괴되었다는 것을 Action 이벤트로 알려 주고 있지만, 현재 개별 미사일들과 직접 커뮤니케이션할 수 있는 것은 미사일 매니저(MissileManager) 뿐입니다. 따라서 개별 미사일이 파괴될 때마다 미사일 매니저에서 이 사실을 외부에 이벤트로 알려 주도록 하면, 스코어 매니저(ScoreManager)와 같은 다른 매니저들은 개별 미사일에 대해서는 전혀 상관하지 않고 오로지 미사일 매니저(MissileManager)를 통해 미사일이 파괴되었다는 사실을 알 수 있습니다. 이것을 위한 새 Action 을 미사일 매니저에 만들도록 하겠습니다.

앞에서와 마찬가지로, Action 을 사용하기 위해서는 Using System; 을 네임 스페이스 맨 위에 추가해야 합니다. 이렇게 하신 다음에 그 아래에 MissileDestroyed 라는 인자 없는 Action 을 하나 선언해 줍니다. (앞에서 개별 미사일들은 Destroyed 라는 액션에 자기 자신을 인자로 담아서 전달했습니다. 이 때문에 Action<RecycleObject>를 사용했습니다. 하지만 미사일 매니저의 경우 그냥 미사일이 파괴되었다는 사실만을 외부에 알리면 될 뿐, 구체적으로 어떤 미사일이 파괴되었는지까지 외부에 전달할 필요는 없습니다. 이 때문에 여기에서 Action<RecycleObject>가 아니라 보통의 Action 만을 사용한 것입니다.)

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MissileManager : MonoBehaviour
{
    Factory missileFactory;
    BuildingManager buildingManager;

    bool isInitialized = false;

    int maxMissileCount = 20;
    int currentMissileCount;
    int destroyedMissileCount;

    float missileSpawnInterval = 0.5f;

    Coroutine spawningMissile;

    List<RecycleObject> missiles = new List<RecycleObject>();

    public Action MissileDestroyed;

    ...
}
```

이제 OnMissileDestroyed() 함수로 가서, 미사일이 파괴되어 Factory에 회수될 때마다 미사일이 파괴되었다는 이벤트를 발생시켜 줍니다.

```

void OnMissileDestroyed(RecycleObject missile)
{
    missile.Destroyed -= OnMissileDestroyed;
    int index = missiles.IndexOf(missile);
    missiles.RemoveAt(index);
    missileFactory.Restore(missile);

    MissileDestroyed?.Invoke();
}

```

그런데 이 때 주의하셔야 할 점이 있습니다. Action을 사용하기 위해 using System; 을 네임 스페이스 맨 앞에 추가하면 GetMissileSpawnPosition() 의 Random.Range() 사용 부분(아래 예제에서 초록색으로 표시한 부분)에 경고 표시되는 것을 확인하실 수가 있습니다. 이것은 Random.Range()라는 이름의 함수가 System 네임 스페이스에도 있고 UnityEngine 네임 스페이스에도 있기 때문입니다. 어떤 네임스페이스에 속한 Random.Range() 함수를 사용해야 할지 알 수가 없기 때문에 이대로 게임을 실행하면 에러가 발생할 것입니다. 비쥬얼 스튜디오에서 경고를 표시하는 것은 바로 그 때문입니다.

```

Vector3 GetMissileSpawnPosition()
{
    Vector3 spawnPosition = Vector3.zero;
    spawnPosition.x = Random.Range(0f, 1f);
    spawnPosition.y = 1f;

    spawnPosition = Camera.main.ViewportToWorldPoint(spawnPosition);
    spawnPosition.z = 0f;
    return spawnPosition;
}

```

따라서 어떤 Random.Range() 함수를 사용할 것인지 분명하게 해 주어야만 합니다.

다음과 같이 코드를 수정해서 UnityEngine 네임 스페이스에 있는 Random.Range()

를 사용할 것임을 명확하게 한다면 더 이상의 에러 경고는 사라질 것입니다.

MissileManager.cs

```
Vector3 GetMissileSpawnPosition()
{
    Vector3 spawnPosition = Vector3.zero;
    spawnPosition.x = UnityEngine.Random.Range(0f, 1f);
    spawnPosition.y = 1f;

    spawnPosition = Camera.main.ViewportToWorldPoint(spawnPosition);
    spawnPosition.z = 0f;
    return spawnPosition;
}
```

이제 미사일 매니저 쪽의 작업이 모두 끝났으니, 다음으로는 게임 매니저

(GameManager.cs)의 BindEvents() 함수로 이동하겠습니다. 이곳에서 다음과 같이

미사일 매니저와 스코어 매니저의 이벤트를 다음과 같이 바인딩해 주겠습니다.

GameManager.cs

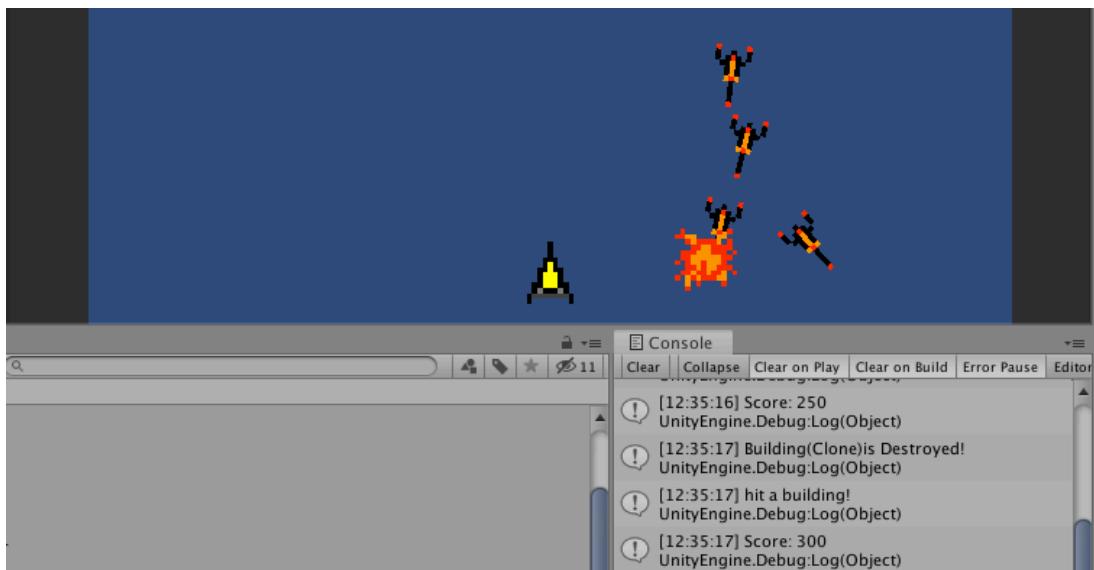
```
void BindEvents()
{
    mouseController.FireButtonPressed += launcher.OnFireButtonPressed;
    timeManager.GameStarted += buildingManager.OnGameStarted;
    timeManager.GameStarted += launcher.OnGameStarted;
    timeManager.GameStarted += missileManager.OnGameStarted;
    missileManager.MissileDestroyed += scoreManager.OnMissileDestroyed;
}
```

그리고 이 때 UnBindEvents()에서 다시 언바인딩해 주는 것을 잊으시면 안됩니다.

```
GameManager.cs
```

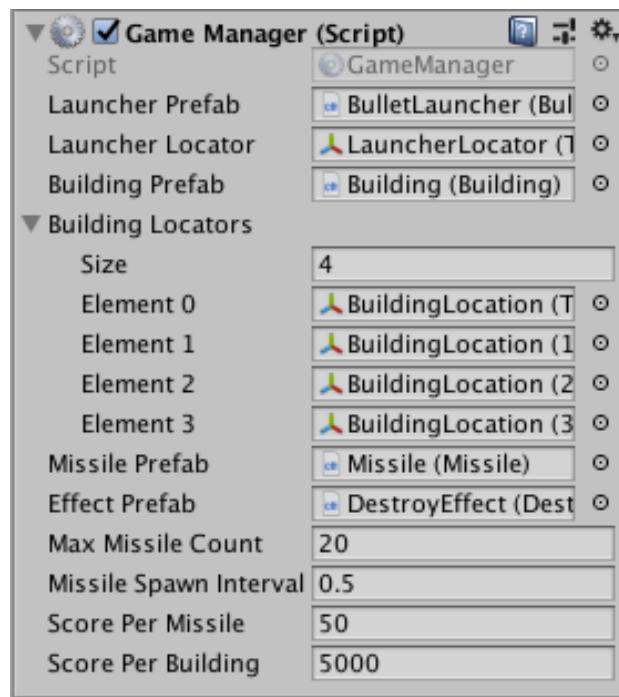
```
void UnBindEvents()
{
    mouseController.FireButtonPressed -= launcher.OnFireButtonPressed;
    timeManager.GameStarted -= buildingManager.OnGameStarted;
    timeManager.GameStarted -= launcher.OnGameStarted;
    timeManager.GameStarted -= missileManager.OnGameStarted;
    missileManager.MissileDestroyed -= scoreManager.OnMissileDestroyed;
}
```

이제 유니티 에디터로 가서 테스트를 해 보겠습니다. 게임을 실행하고 총알을 발사해서 미사일을 파괴할 때마다 콘솔창에 스코어 표시가 잘 나타나는지 확인해 보시기 바랍니다. 적의 미사일이 하나 파괴될 때마다 스코어가 50 단위로 늘어나는 것을 보실 수 있을 것입니다.



[동영상 예제 파일명: 078_create_missile_destruction_event.mp4]

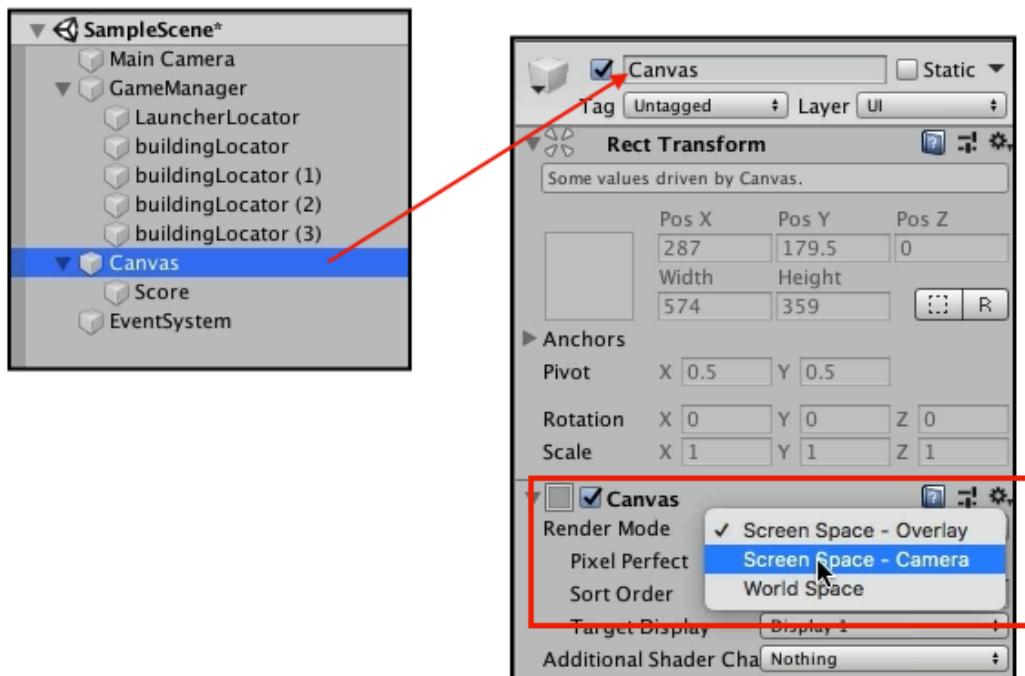
현재는 미사일 하나당 50점 씩 증가하고 있습니다. 만약 점수를 수정하고 싶으면 GameManager에서 원하는 점수로 바꾸시면 됩니다. 다만 런타임 환경(게임이 실행되고 있는 도중)에서는 스코어를 바꿔도 소용이 없습니다. 이 점수는 게임이 시작될 때, 다시 말해서 스코어 매니저의 인스턴스가 생성되는 순간 단 한번만 적용되기 때문입니다. (생성자를 통해 스코어가 세팅된다는 것을 기억해 보시기 바랍니다.)



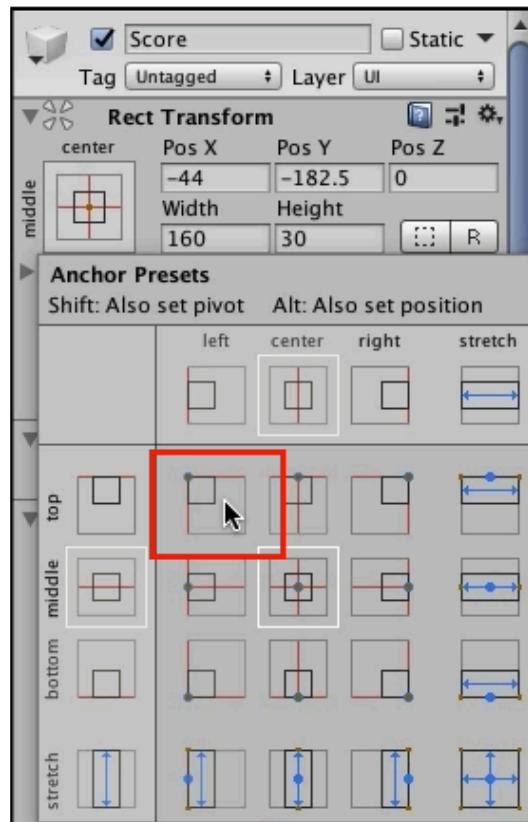
간단한 UI 를 만들어 점수를 표시하자

이제 스코어가 변할 때마다 화면에 이 점수를 표시하도록 하겠습니다. 이를 위해 간단한 Text UI 를 만들겠습니다. 이 책에서는 게임 개발 로직에 집중하고 있으므로 UI 를 멋지게 꾸미거나 하지는 않고, 단순한 기능만 수행하도록 할 예정입니다.

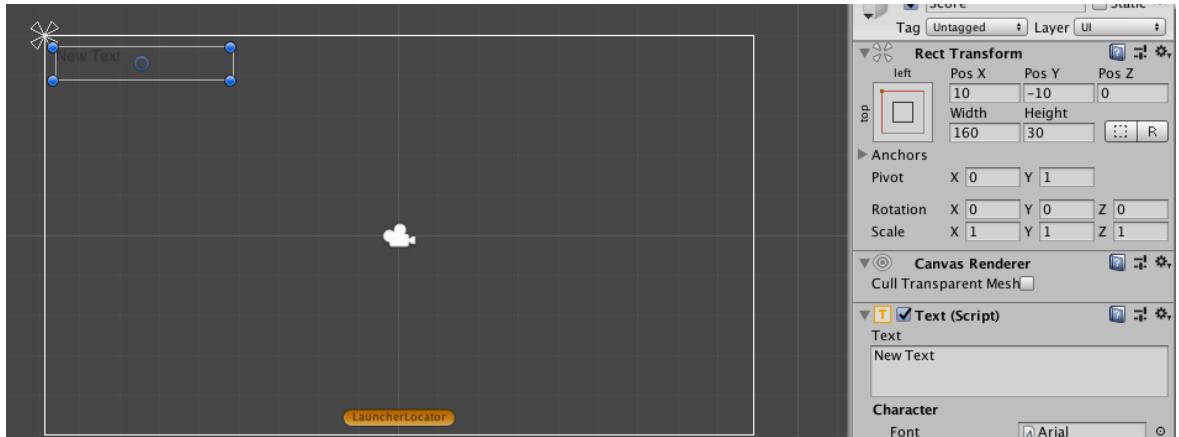
우선 유니티 에디터에서 UI Text 를 하나 생성하고 이름을 Score 라고 바꾸겠습니다. 이렇게 UI 요소를 만들면 Canvas 가 자동으로 생성될 것입니다. 마우스로 Canvas 를 선택하고 인스펙터에서 Render 모드를 ScreenSpace – Camera 로 바꾸겠습니다. 그리고 나서 하이어라키 뷰 목록에 있는 메인 카메라(Main Camera)를 드래그해서 다음과 같이 Canvas 인스펙터의 Render Camera 에 갖다 놓겠습니다.



다음으로는 Score의 앵커(Anchor)를 설정하겠습니다. 먼저 하이어라키 뷰에서 Canvas의 자식 오브젝트로 만들어져 있는 Score를 선택하고 인스펙터의 앵커 설정 부분을 마우스로 클릭합니다. 이 때 SHIFT + ALT (윈도우즈) 또는 SHIFT + OPTION (맥) 키를 동시에 누른 상태에서 앵커 프리셋(Anchor Presets)을 다음과 같이 top-left로 지정해 주면, 게임 화면 사이즈와 상관 없이 Score 텍스트 UI가 화면 왼쪽 상단의 일정한 영역에 표시되도록 할 수 있습니다.



다음으로는 Score의 Rect Transform에서 Pos X를 10, Pos Y를 -10으로 해 주겠습니다. 그러면 하단 스크린샷처럼 Score 표시 텍스트가 게임 화면 왼쪽 상단에서 적당히 떨어진 위치에 표시되는 것을 확인하실 수 있습니다. 이 상태에서 마지막으로 텍스트가 잘 보이도록 글자 색을 흰색으로 바꾸도록 하겠습니다.



[동영상 예제 파일명: 079_score_ui_creation.mp4]

UIRoot.cs 스크립트를 만들자

이제 UI에 표시되는 내용을 관리하기 위한 C# 스크립트를 새로 하나 추가해서 만들겠습니다. 먼저 다음과 같이 UIRoot.cs라는 이름의 새 MonoBehaviour 클래스를 만들겠습니다. 그리고 나서 필요 없는 함수를 삭제합니다. 보통의 유니티 입문서에서는 UI의 표시 내용을 갱신하기 위해 Update() 함수를 사용하는 경우가 많은데, 이 책에서는 이벤트 방식을 사용해서 기능을 구현할 예정입니다. 따라서 Update()는 쓸 일이 없으니 지워 주도록 하겠습니다.

UIRoot.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class UIRoot : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

다음으로는 앞에서 만든 score라는 Text UI를 할당할 변수를 선언하겠습니다. 이를 위해서는 다음과 같이 using UnityEngine.UI;라는 네임 스페이스를 붙여 주어야 합니다. 따라서 다음과 같이 코드를 작성해 줍니다. (scoreUI가 유니티 에디터의 인스펙터에 노출될 수 있도록 [SerializeField]를 scoreUI 변수 위에 붙여 줍니다)

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class UIRoot : MonoBehaviour
{
    [SerializeField]
    Text scoreUI;

    // Start is called before the first frame update
    void Start()
    {
    }

}

```

다음으로는 Start() 함수로 가서 다음과 같이 스코어 표시 UI를 처음에는 보이지 않게 비활성화합니다.

```

public class UIRoot : MonoBehaviour
{
    [SerializeField]
    Text scoreUI;

    // Start is called before the first frame update
    void Start()
    {
        scoreUI.gameObject.SetActive(false);
    }

}

```

이렇게 하는 것은 빌딩(Building)이나 총알 발사대(BulletLauncher)와 마찬가지로 처음에는 화면에 아무 것도 나타나지 않다가, 타임 매니저(TimeManager)에 의해 게임이 시작되면 그 때 스코어 표시 UI 가 화면에 나타나도록 하기 위해서입니다. 따라서

타임 매니저의 GameStarted라는 이벤트를 수신하기 위한 함수를 UIRoot.cs에서도 작성해야 합니다. 다음과 예제와 같이 OnGameStarted()라는 함수를 작성해 주겠습니다.

```
UIRoot.cs
```

```
public class UIRoot : MonoBehaviour
{
    [SerializeField]
    Text scoreUI;

    // Start is called before the first frame update
    void Start()
    {
        scoreUI.gameObject.SetActive(false);
    }

    public void OnGameStarted()
    {

    }
}
```

게임 시작 이벤트가 발생하면 제일 먼저 해야 할 일은 아까 Start() 함수에서 보이지 않게 감추었던 scoreUI를 활성화하는 것입니다. 다음과 같이 scoreUI를 활성화시키겠습니다.

```
UIRoot.cs
```

```
public class UIRoot : MonoBehaviour
{
    [SerializeField]
    Text scoreUI;

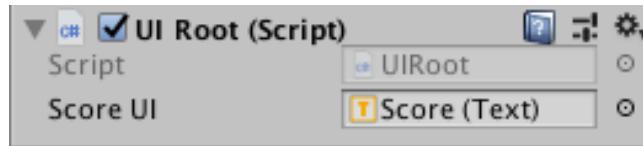
    ...

    public void OnGameStarted()
    {
        scoreUI.gameObject.SetActive(true);
    }
}
```

그리고 기본으로 표시될 텍스트 내용을 다음과 같이 string.Format() 을 이용하여 지정해 주겠습니다. 이렇게 하면 게임이 시작될 때 “Score: 0”이라는 텍스트가 화면에 표시될 것입니다.

```
UIRoot.cs  
public void OnGameStarted()  
{  
    scoreUI.gameObject.SetActive(true);  
    scoreUI.text = string.Format("Score: {0}", 0);  
}
```

이제 유니티 에디터로 가서, 하이어라키 뷰에서 Canvas 를 선택하고 방금 만든 UIRoot.cs 스크립트 파일을 연결해 주겠습니다. 그리고 나서 인스펙터에 노출된 scoreUI 변수에 Canvas의 자식 오브젝트인 Score 를 드래그해서 할당해 줍니다.



[동영상 예제 파일명: 080_uiroot_class_creation.mp4]

이제 기본적인 준비가 끝났으니, 다음에는 게임 매니저로 가서, 방금 만든 UIRoot를 타임 매니저의 이벤트와 연동해 주어야 합니다.

먼저 GameManager.cs로 가서 다음과 같이 UIRoot를 담을 변수를 생성하고 [SerializeField] 어트리뷰트를 덧붙여서 유니티 에디터의 인스펙터에 노출되도록 하겠습니다.

GameManager.cs

```
public class GameManager : MonoBehaviour
{
    ...
    [SerializeField]
    UIRoot uIRoot;

    MouseGameController mouseController;
    BuildingManager buildingManager;
    TimeManager timeManager;
    MissileManager missileManager;
    ScoreManager scoreManager;
    ...
}
```

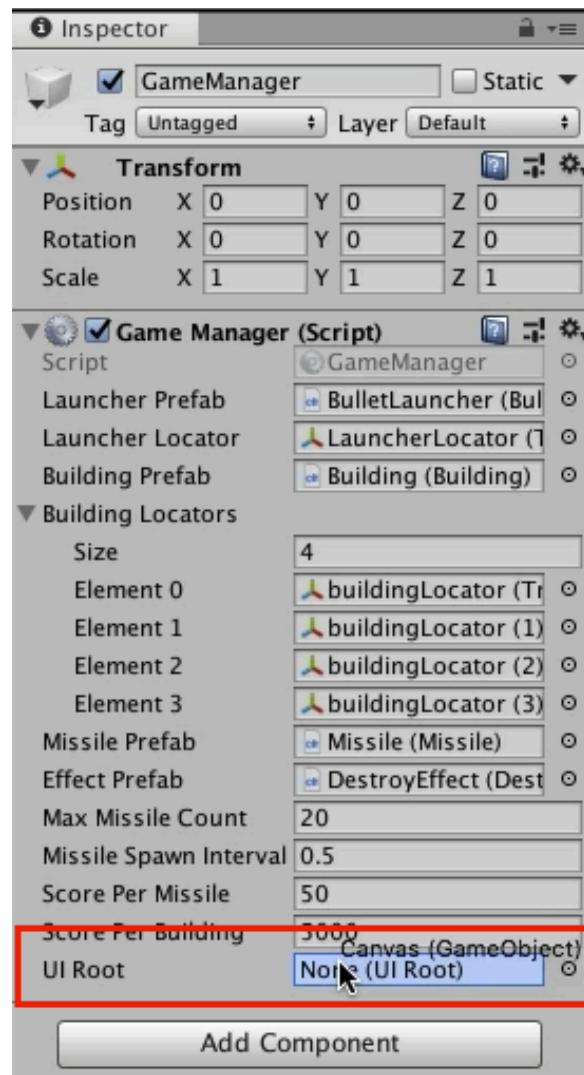
이제 타임 매니저에서 발생시키는 GameStarted 이벤트와 방금 UIRoot에서 만든 OnGameStarted 이벤트 수신 함수를 바인딩해 주도록 하겠습니다. 이를 위해 BindEvents() 함수로 가서 다음과 같이 코드를 작성해 줍니다.

```
void BindEvents()
{
    mouseController.FireButtonPressed += launcher.OnFireButtonPressed;
    timeManager.GameStarted += buildingManager.OnGameStarted;
    timeManager.GameStarted += launcher.OnGameStarted;
    timeManager.GameStarted += missileManager.OnGameStarted;
    timeManager.GameStarted += uIRoot.OnGameStarted;
    missileManager.MissileDestroyed += scoreManager.OnMissileDestroyed;
}
```

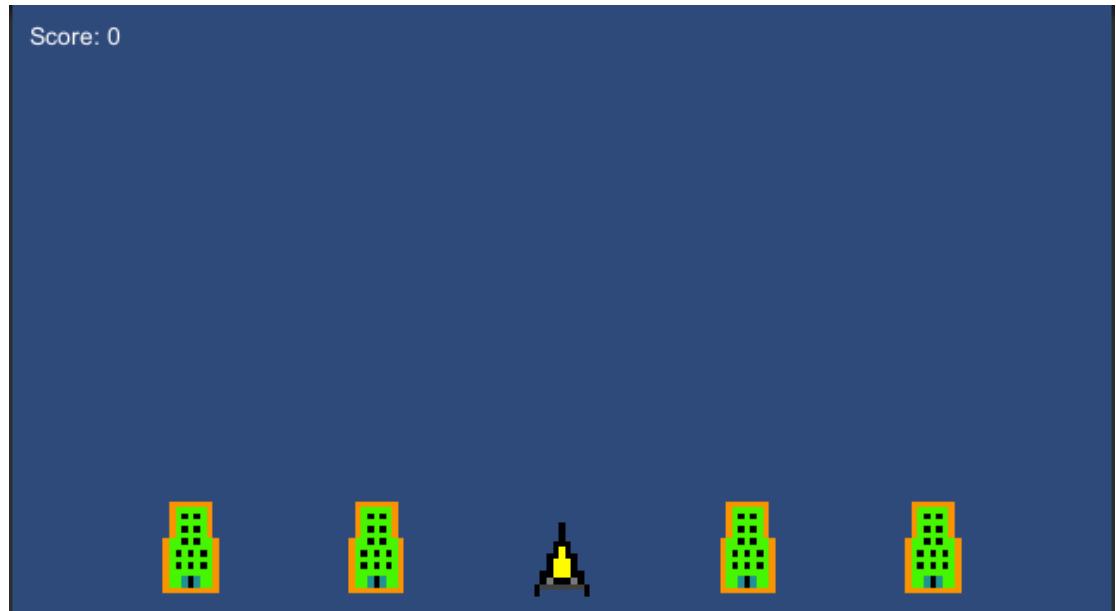
이번에도 UnBindEvents()에서 언바인딩하는 것을 잊지 마시기 바랍니다.

```
void UnBindEvents()
{
    mouseController.FireButtonPressed -= launcher.OnFireButtonPressed;
    timeManager.GameStarted -= buildingManager.OnGameStarted;
    timeManager.GameStarted -= launcher.OnGameStarted;
    timeManager.GameStarted -= missileManager.OnGameStarted;
    timeManager.GameStarted -= uIRoot.OnGameStarted;
    missileManager.MissileDestroyed -= scoreManager.OnMissileDestroyed;
}
```

이제 유니티 에디터로 이동하겠습니다. 하이어라키 뷰에서 GameManager 게임 오브젝트를 선택하고, Canvas를 드래그해서 인스펙터에 노출된 UIRoot 변수에 연결하면 모든 준비 작업이 끝나게 됩니다.



그럼 게임을 실행해서 테스트해 보겠습니다. 그러면 처음에는 화면에 아무 것도 보이지 않다가 게임이 시작되어 빌딩과 총알 발사대가 화면에 등장하면, 그와 동시에 Score: 0이라는 UI도 화면에 나타나는 것을 보실 수가 있을 것입니다.



[동영상 예제 파일명: 081_bind_uiroot_time_manager.mp4]

UI에서 스코어(score)를 업데이트하자

이제 스코어가 변경될 때마다 UIRoot.cs에서 이를 인지하여 화면에 표시되는 스코어를 자동으로 바꿀 수 있도록 해 보겠습니다. 원리는 간단합니다. 스코어매니저(ScoreManager)에서는 스코어가 변경될 때마다 ScoreChanged라는 이벤트와 함께 현재의 스코어를 전달해 주는데, 이 이벤트를 수신할 때마다 UI에 표시되는 스코어를 최신의 것으로 변경해 주기만 하면 되는 것입니다.

이런 식으로 스코어 표시를 바꾸는 방법은 이벤트 기반이라 매우 효율적입니다. 점수가 바뀔 때만 UI가 변경되기 때문입니다. 그러면 이 작업을 위해 다음과 같이 UIRoot.cs에 OnScoreChanged라는 이벤트 수신 함수를 작성하겠습니다. 스코어 매니저(ScoreManager)에서 발생하는 ScoreChanged 이벤트가 Action<int> 형식이므로, 지금 작성할 OnScoreChanged() 함수는 int 타입의 인자를 받아들이도록 만들어야 합니다. 코드는 다음과 같습니다.

```
UIRoot.cs
```

```
public class UIRoot : MonoBehaviour
{
    ...
    public void OnGameStarted()
    {
        ...
    }

    public void OnScoreChanged(int score)
    {
        ...
    }
}
```

다음으로는 이 함수 안에, 이벤트 송신자로부터 전달 받은 score 값을 UI에 표시하는 구문을 작성하겠습니다.

UIRoot.cs

```
public void OnScoreChanged(int score)
{
    scoreUI.text = string.Format("Score: {0}", score);
}
```

이제 GameManager.cs로 가서 다음과 같이 scoreManager의 ScoreChanged 이벤트와 uIRoot의 OnScoreChanged 이벤트를 바인딩해 주겠습니다.

GameManager.cs

```
void BindEvents()
{
    mouseController.FireButtonPressed += launcher.OnFireButtonPressed;
    timeManager.GameStarted += buildingManager.OnGameStarted;
    timeManager.GameStarted += launcher.OnGameStarted;
    timeManager.GameStarted += missileManager.OnGameStarted;
    timeManager.GameStarted += uIRoot.OnGameStarted;
    missileManager.MissileDestroyed += scoreManager.OnMissileDestroyed;
    scoreManager.ScoreChanged += uIRoot.OnScoreChanged;
}
```

역시 UnBindEvents()에서 이벤트를 언바인딩하는 것을 잊으시면 안됩니다.

GameManager.cs

```
void UnBindEvents()
{
    mouseController.FireButtonPressed -= launcher.OnFireButtonPressed;
    timeManager.GameStarted -= buildingManager.OnGameStarted;
    timeManager.GameStarted -= launcher.OnGameStarted;
    timeManager.GameStarted -= missileManager.OnGameStarted;
    timeManager.GameStarted -= uIRoot.OnGameStarted;
    missileManager.MissileDestroyed -= scoreManager.OnMissileDestroyed;
    scoreManager.ScoreChanged -= uIRoot.OnScoreChanged;
}
```

이제 유니티 에디터로 가서 게임을 실행해 보겠습니다. 그러면 미사일이 파괴될 때마다 화면 왼쪽의 스코어가 올라가는 것을 확인하실 수 있습니다.



[동영상 예제 파일명: 082_update_score_on_ui.mp4]

스크린 바깥으로 나간 미사일들을 회수하자

현재, 파괴된 미사일들은 missileFactory 에 의해 자동으로 회수되고 있습니다. 하지만 파괴되지 않고 화면 아래로 빠져 나간 미사일들은 끝없이 아래로 움직일 뿐입니다. 이들은 missileFactory에 의해 회수되지 않으므로 재활용되지도 않습니다. 따라서 이들이 화면 아래로 빠져 나가는 순간 다시 회수하는 작업을 해 주어야만 합니다.

이를 위해서는 미사일이 화면 아래로 사라졌을 때, 이를 알려 주는 Action 이벤트가 필요합니다. 이를 위해 미사일(Missile.cs)의 부모 클래스인 RecycleObject.cs 에 오브젝트가 화면 바깥으로 빠져 나갔다는 것을 알려 주기 위한 Action 을 하나 추가하겠습니다. 이 게임에서는 화면 바깥으로 나가는 것이 미사일 하나 뿐이므로 미사일(Missile.cs)에 이 Action을 추가해도 되지만, 나중에 다른 클래스에서도 이런 이벤트가 필요한 상황이 발생할 수 있으므로 부모 클래스인 RecycleObject.cs 에 관련된 액션을 추가하기로 하겠습니다.

먼저 ReycleObject.cs 로 가서 다음과 같이 OutOfScreen이라는 액션을 하나 추가하겠습니다.

```
ReycleObject.cs
```

```
public class RecycleObject : MonoBehaviour
{
    protected bool isActiveated = false;

    public Action<RecycleObject> Destroyed;
    public Action<RecycleObject> OutOfScreen;
    protected Vector3 targetPosition;
```

```
public virtual void Activate(Vector3 position)
{
    isActivated = true;
    transform.position = position;
}

public virtual void Activate(Vector3 startPosition, Vector3 targetPosition)
{
    ...
}
```

RecycleObject.cs 쪽에서 할 작업은 이것으로 끝입니다. 이제부터는 미사일 쪽에서 작업을 계속하겠습니다. Missile.cs 로 가서 다음과 같이 CheckOutOfScreen() 라는 이름의 함수를 하나 만들겠습니다. 이 함수는 미사일의 현재 위치를 추적하다가, 미사일이 화면 아래 쪽으로 이동해서 완전히 보이지 않게 되면 앞에서 만든 OutOfScreen 이라는 Action 이벤트를 발생시킬 것입니다.

Missile.cs

```
public class Missile : RecycleObject
{
    ...

    void DestroySelf()
    {
        isActivated = false;
        Destroyed?.Invoke(this);
    }

    void CheckOutOfScreen()
    {
    }
}
```

이 함수에서는 미사일이 화면 아래로 빠져 나갔는지 체크할 예정입니다. 이를 위해서는 현재 게임 화면에서 미사일의 Y 좌표가 어디에 위치해야 미사일이 완전히 보이지 않게 되는 지에 대한 계산이 필요합니다. 따라서 미사일이 완전히 보이지 않게 되는 지점의 Y 좌표를 저장하기 위한 bottomY라는 이름의 변수를 Missile.cs에 하나 만들도록 하겠습니다.

Missile.cs

```
public class Missile : RecycleObject
{
    BoxCollider2D box;
    Rigidbody2D body;

    [SerializeField]
    float moveSpeed = 3f;

    float bottomY;

    void Awake()
    {
        ...
    }
```

다음으로는 bottomY의 값을 알아 내야 합니다. 이를 위해 Start()에서 bottomY 위치를 구하기 위한 간단한 작업을 하겠습니다. 우선 화면의 맨 아래쪽의 좌표가 어떻게 되는지부터 알아 보도록 하겠습니다. 화면 좌측 하단을 나타내는 좌표는 뷰포트 (Viewport) 기준으로 (0,0)입니다. 따라서 먼저 이 위치를 월드 좌표로 변환시킨 값을 다음과 같이 구해 보도록 하겠습니다.

Missile.cs

```
void Start()
{
    Vector3 bottomPosition =
        Camera.main.ViewportToWorldPoint(new Vector2(0, 0));
}
```

그런데 이렇게 구한 좌표(bottomPosition)는 우리가 원하는 것이 아닙니다. 이 좌표에 미사일을 위치시키면 미사일이 완전히 사라지는 것이 아니라 일부가 보이게 될 것입니다. 이 좌표에서 미사일의 크기만큼을 더 아래로 옮겨 주어야 미사일이 화면 바깥으로 완전히 사라져서 보이지 않게 됩니다.

다시 말해서, 화면 바닥을 나타내는 Y 좌표는 bottomPosition.y 인데, 여기에 미사일의 높이(충돌체의 높이로 알아내겠습니다)를 더 빼주면 미사일이 화면 바깥으로 빠져 나갈 때의 y 좌표를 구할 수 있게 됩니다. (정확하게는 충돌체 높이의 절반을 빼 주면 되지만, 좀 더 확실하게 미사일이 사라지는 순간을 위해 충돌체 높이 자체를 빼 주었습니다.) 계산은 다음과 같이 하면 됩니다.

Missile.cs

```
void Start()
{
    Vector3 bottomPosition =
        Camera.main.ViewportToWorldPoint(new Vector2(0, 0));
    bottomY = bottomPosition.y - box.size.y;
}
```

이제 bottomY의 값을 구했으니, CheckOutOfScreen() 함수로 가서 미사일의 현재 y 위치가 bottomY보다 작아졌는지, 다시 말해서 미사일이 화면 밑으로 완전히 빠져 나가서 더 이상 보이지 않게 되었는지를 체크하는 조건문을 다음과 같이 작성하겠습니다.

```
void CheckOutOfScreen()
{
    if (transform.position.y < bottomY)
    {
        }
    }
}
```

그리고 이 경우, isActive 를 false로 바꿔서 이 미사일을 팩토리에서 회수할 준비를 하고, 앞에서 만든 OutOfScreen 이벤트를 발생시키겠습니다.

```
void CheckOutOfScreen()
{
    if (transform.position.y < bottomY)
    {
        isActive = false;
        OutOfScreen?.Invoke(this);
    }
}
```

그리고 나서 CheckOutOfScreen() 함수를 Update() 에서 실행해서, 미사일이 움직이는 동안 계속 위치를 체크할 수 있도록 하겠습니다.

```
void Update()
{
    if (!isActivated)
        return;

    transform.position += transform.up * moveSpeed * Time.deltaTime;
    CheckOutOfScreen();
}
```

[동영상 예제 파일명: 083_outofscreen_event_creation.mp4]

이제 미사일 매니저(MissileManager.cs)로 이동해서, 미사일이 화면 바깥으로 빠져나가는 경우에 대한 이벤트 수신 함수를 다음과 같이 만들겠습니다.

```
MissileManager.cs
public class MissileManager : MonoBehaviour
{
    ...
    void OnMissileDestroyed(RecycleObject missile)
    {
        ...
    }

    void OnMissileOutOfScreen(RecycleObject missile)
    {
        ...
    }
    ...
}
```

그리고 나서 이벤트를 바인딩해 줍니다. 미사일 매니저(MissileManager)와 미사일(Missile)을 바인딩하는 곳은 SpawnMissile() 함수이므로, 이곳에서 다음과 같이 코드를 작성하시면 됩니다.

```
MissileManager.cs
void SpawnMissile()
{
    Debug.Assert(this.missileFactory != null, "missile factory is null!");
    Debug.Assert(this.buildingManager != null, "building manager is null!");

    RecycleObject missile = missileFactory.Get();
    missile.Activate(GetMissileSpawnPosition(),
                     buildingManager.GetRandomBuildingPosition());
    missile.Destroyed += this.OnMissileDestroyed;
    missile.OutOfScreen += this.OnMissileOutOfScreen;
    missiles.Add(missile);

    currentMissileCount++;
}
```

다음으로는 OnMissileOutOfScreen() 함수로 가서, 미사일이 화면 바깥으로 사라졌을 경우 미사일을 missileFactory로 다시 회수하기 위한 코드를 추가하겠습니다. 여기에 서 해야 할 일은 다음과 같습니다.

1. 미사일에 바인딩된 모든 이벤트를 언바인딩
2. 회수할 미사일을 missiles 리스트에서 제거
3. missileFactory.Restore() 함수를 이용해서 미사일을 회수

그런데 잘 보시면, 이 3가지 작업은 앞에서 OnMissileDestroyed() 함수에서도 똑같이 수행하였습니다. 이 이야기는 OnMissileDestroyed() 함수와 OnMissileOutOfScreen() 함수에 중복된 명령어들이 많다는 의미가 됩니다. 따라서 똑같은 코드를 두 번 작성하기보다는, 이 공통된 부분을 뽑아서 하나의 함수를 만들어 사용하는 것이 낫습니다. 따라서 다음과 같이 RestoreMissile()이라는 새로운 함수를 하나 만들도록 하겠습니다.

MissileManager.cs

```
...
void OnMissileOutOfScreen(RecycleObject missile)
{
}

void RestoreMissile(RecycleObject missile)
{
}
```

그리고 나서 OnMissileDestroyed()에 있는 코드 중에서 이벤트 송신 부분 (MissileDestroyed?.Invoke())를 빼고 나머지 코드를 모두 새로 만든 RestoreMissile() 함수로 옮기겠습니다.

```
MissileManager.cs
```

```
void OnMissileDestroyed(RecycleObject missile)
{
    missile.Destroyed -= this.OnMissileDestroyed;
    int index = missiles.IndexOf(missile);
    missiles.RemoveAt(index);
    missileFactory.Restore(missile);

    MissileDestroyed?.Invoke();
}

void OnMissileOutOfScreen(RecycleObject missile)
{
}

void RestoreMissile(RecycleObject missile)
{
    missile.Destroyed -= this.OnMissileDestroyed;
    int index = missiles.IndexOf(missile);
    missiles.RemoveAt(index);
    missileFactory.Restore(missile);
}
```

그리고 나서 OutOfScreen 이벤트를 언바인딩하는 코드를 여기에 새로 추가하겠습니다.

```
MissileManager.cs
```

```
void RestoreMissile(RecycleObject missile)
{
    missile.Destroyed -= this.OnMissileDestroyed;
    missile.OutOfScreen -= this.OnMissileOutOfScreen;
    int index = missiles.IndexOf(missile);
    missiles.RemoveAt(index);
    missileFactory.Restore(missile);
}
```

이제 공통된 명령어들을 모아 둔 `RestoreMissile()`이라는 함수가 완성되었으므로, 이를 `OnMissileDestroyed()`에서 다음과 같이 호출해 주겠습니다.

MissileManager.cs

```
void OnMissileDestroyed(RecycleObject missile)
{
    RestoreMissile(missile);
    MissileDestroyed?.Invoke();
}
```

또한 `OnMissileOutOfScreen()`에서도 똑같은 방식으로 `RestoreMissile()`을 호출합니다.

MissileManager.cs

```
void OnMissileOutOfScreen(RecycleObject missile)
{
    RestoreMissile(missile);
}
```

`OnMissileOutOfScreen()`에서는 그냥 `RestoreMissile()`을 호출하고 끝낼 뿐이라 사실 두 함수를 따로 만들 필요는 없습니다. 하지만 게임을 계속 개발하다 보면 나중에 `OnMissileOutOfScreen()` 함수 안에 다른 코드를 추가해야 하는 경우가 생길 수 있으므로 이렇게 두 개의 함수를 따로 분리해서 사용하는 것으로 하겠습니다.

이제 유니티 에디터로 가서 게임을 실행해 보겠습니다. 그러면 파괴되지 않고 화면 아래로 빠져 나간 미사일들도 비활성화 되는 것을 알 수 있습니다. 즉, 팩토리로 회수된 것입니다.



[동영상 예제 파일명: 084_bind_outofscreen_event.mp4]

게임 오버 처리하기

이제 게임에서 승리하거나 패배했을 때의 처리를 구현하도록 하겠습니다. 미사일 커맨더(Missile Commander)에서 게임 오버의 조건은 다음의 2가지 중 하나입니다.

1. 빌딩이 모두 파괴될 경우 – 무조건 패배
2. 빌딩이 하나라도 남아 있고, 생성된 모든 미사일이 다 회수되었을 때 – 승리

두 개의 게임 오버 조건 중에서 가장 우선 순위가 높은 조건은 빌딩이 모두 파괴되었을 때입니다. 마지막 미사일과 마지막 빌딩이 충돌해서 게임에서 생성된 미사일과 빌딩 모두가 파괴되었을 경우, 어쨌건 남아 있는 빌딩이 하나 없기 때문에 게임에서 패배하게 되는 것입니다. 따라서 제일 먼저 체크해야 할 조건은 빌딩이 모두 파괴되었는가 하는 것입니다.

이를 위해 우선 BuildingManager.cs 로 가서 다음과 같이 모든 빌딩이 파괴되었을 때 발생시킬 이벤트 액션을 추가해 주겠습니다. 역시 Action 을 사용해야 하므로 using System; 을 네임 스페이스 맨 위에 추가합니다.

```

using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BuildingManager
{
    Building prefab;
    Transform[] buildingLocators;
    Factory effectFactory;

    List<Building> buildings = new List<Building>();

    public bool HasBuilding { get {
        return buildings.Count > 0;
    }
}

public Action AllBuildingsDestroyed;

    ...
}

```

그런데, 이렇게 using System; 을 추가하면 비주얼 스튜디오에서 또 다시 경고 메시지를 표시합니다. 이것은 앞에서도 한 번 일어났던 일입니다.

GetRandomBuildingPosition() 함수에서 사용한 Random.Range() 가 UnityEngine 네임스페이스에 있는 것인지 System 에 있는 것인지 불분명하게 되었기 때문입니다.

```

public Vector3 GetRandomBuildingPosition()
{
    Debug.Assert(buildings.Count > 0, "no element in buildings!");

    Building building = buildings[Random.Range(0, buildings.Count)];
    return building.transform.position;
}

```

따라서 다음과 같이 UnityEngine 의 Random.Range() 를 사용한다는 점을 명확하게 해 주겠습니다.

BuildingManager.cs

```
public Vector3 GetRandomBuildingPosition()
{
    Debug.Assert(buildings.Count > 0, "no element in buildings!");

    Building building =
        buildings[UnityEngine.Random.Range(0, buildings.Count)];
    return building.transform.position;
}
```

이제 OnBuildingDestroyed() 함수로 가서, 빌딩이 하나 파괴될 때마다 남은 빌딩 개수 체크하는 로직을 추가하도록 하겠습니다. 이렇게 빌딩이 파괴되다가 최후에 남은 빌딩이 하나도 없다면, 다시 말해서 buildings.Count 의 값이 0이라면 AllBuildingsDestroyed 라는 이벤트를 발생시켜 줍니다.

BuildingManager.cs

```
void OnBuildingDestroyed(Building building)
{
    Vector3 lastPosition = building.transform.position;
    lastPosition.y += (building.GetComponent<BoxCollider2D>().size.y * 0.5f);
    building.Destroyed -= OnBuildingDestroyed;
    int index = buildings.IndexOf(building);
    buildings.RemoveAt(index);
    GameObject.Destroy(building.gameObject);

    RecycleObject effect = effectFactory.Get();
    effect.Activate(lastPosition);
    effect.Destroyed += OnEffectDestroyed;

    if (buildings.Count == 0)
    {
        AllBuildingsDestroyed?.Invoke();
    }
}
```

이제 이벤트를 송신했으니, 다음에는 이벤트 수신자 측의 코드를 작성해야 합니다. 게임 매니저(GameManager.cs)로 가서 다음과 같이 OnAllBuildingDestroyed()라는 이벤트 수신 함수를 작성하겠습니다.

GameManager.cs

```
void OnDestroy()
{
    UnBindEvents();
}

void OnAllBuildingDestroyed()
{
}

// Update is called once per frame
void Update()
{
}
```

일단 제대로 된 코드를 작성하기 전에, Debug.Log()를 이용하여 게임에서 패배했다는 메시지를 콘솔창에 표시하도록 하겠습니다.

GameManager.cs

```
void OnAllBuildingDestroyed()
{
    Debug.Log("You Lost");
}
```

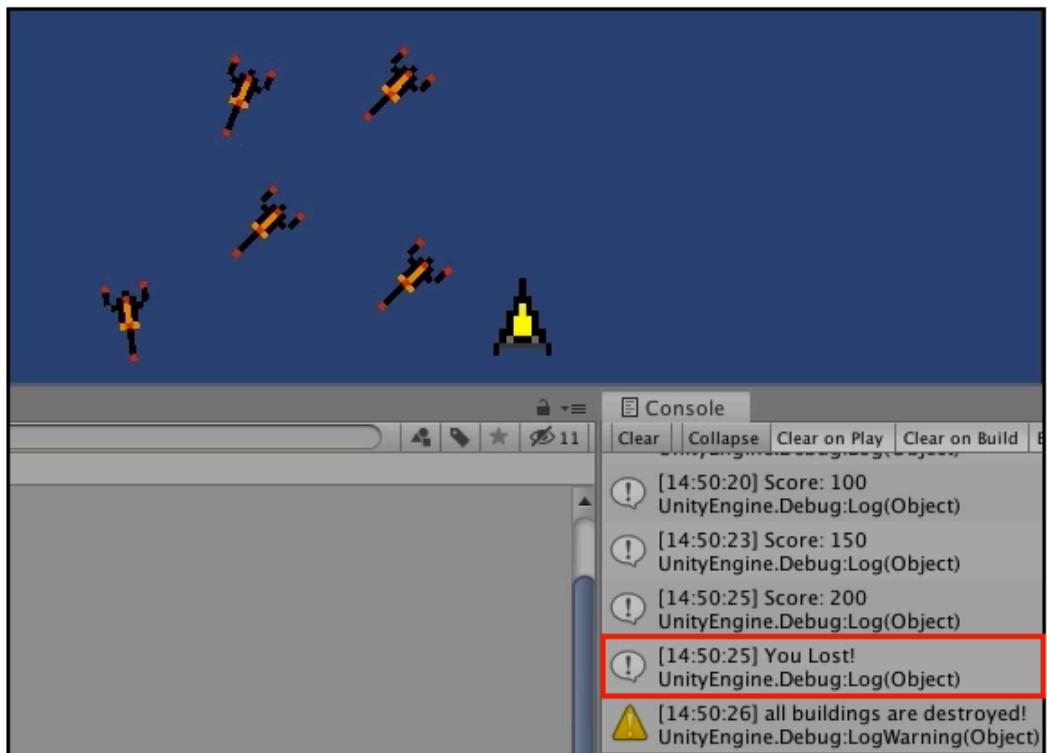
이제 BindEvents() 함수로 가겠습니다. 이곳에서 빌딩 매니저의 AllBuildingsDestroyed 이벤트를 방금 만든 OnAllBuildingDestroyed() 함수와 바인딩시키겠습니다. (가독성을 위해 this라는 키워드를 붙여도 됩니다. 여기에서는 그냥 this 없이 코드를 작성했습니다.)

```
void BindEvents()
{
    mouseController.FireButtonPressed += launcher.OnFireButtonPressed;
    timeManager.GameStarted += buildingManager.OnGameStarted;
    timeManager.GameStarted += launcher.OnGameStarted;
    timeManager.GameStarted += missileManager.OnGameStarted;
    timeManager.GameStarted += uIRoot.OnGameStarted;
    missileManager.MissileDestroyed += scoreManager.OnMissileDestroyed;
    scoreManager.ScoreChanged += uIRoot.OnScoreChanged;
    buildingManager.AllBuildingsDestroyed += OnAllBuildingDestroyed;
}
```

이벤트 바인딩 후에는 반드시 언바인딩해 주어야 합니다. UnBindEvents() 함수에서
이 작업을 해 줍니다.

```
void UnBindEvents()
{
    mouseController.FireButtonPressed -= launcher.OnFireButtonPressed;
    timeManager.GameStarted -= buildingManager.OnGameStarted;
    timeManager.GameStarted -= launcher.OnGameStarted;
    timeManager.GameStarted -= missileManager.OnGameStarted;
    timeManager.GameStarted -= uIRoot.OnGameStarted;
    missileManager.MissileDestroyed -= scoreManager.OnMissileDestroyed;
    scoreManager.ScoreChanged -= uIRoot.OnScoreChanged;
    buildingManager.AllBuildingsDestroyed -= OnAllBuildingDestroyed;
}
```

이제 유니티 에디터로 가서 게임을 플레이해 보겠습니다. 게임이 시작되고 가만히 기다리시면 모든 빌딩이 파괴될 것입니다. 그리고 빌딩이 다 파괴되는 순간 콘솔창에 “You Lost”라는 메시지가 표시되는 것을 확인하실 수가 있습니다.



[동영상 예제 파일명: 085_all_buildings_destroyed_event.mp4]

승리 조건을 체크하자

앞에서 패배 조건을 체크하는 로직을 만들었습니다. 남은 미사일의 개수와 상관 없이 빌딩이 모두 파괴되었다면 무조건 패배로 간주하였습니다. 다음으로는 승리할 조건을 판정하는 로직을 만들어 보겠습니다.

승리 조건 역시 간단합니다. 미사일이 모두 만들어진 상태에서, 미사일 매니저가 가지고 있는 missiles 리스트에 남아 있는 미사일이 하나도 없으면 승리로 간주하면 됩니다. 이것은 생성된 모든 미사일이 파괴거나 빗나가서 화면 아래로 빠져 나갔다는 이야기이기 때문입니다.

그러면 이를 위해서는 어디에서 어떤 작업을 해야 할까요? 그것은 간단합니다. 미사일이 회수될 때마다 이 승리 조건을 체크하기만 하면 되는 것입니다.

이 작업을 위해 이제 MissileManager.cs 로 가서, 생성된 모든 미사일이 회수되었는지를 체크하기 위한 함수를 다음과 같이 만들고 이를 RestoreMissile() 함수에서 호출하도록 하겠습니다.

```

void RestoreMissile(RecycleObject missile)
{
    missile.Destroyed -= OnMissileDestroyed;
    missile.OutOfScreen -= OnMissileOutOfScreen;
    int index = missiles.IndexOf(missile);
    missiles.RemoveAt(index);
    missileFactory.Restore(missile);

    CheckAllMissileRestored();
}

void CheckAllMissileRestored()
{
}

Vector3 GetMissileSpawnPosition()
{
    ...
}

```

이제 모든 미사일 파괴된 것이 확인되면 발생시킬 또 다른 이벤트인

AllMissilesDestroyed를 다음과 같이 선언해 줍니다.

```

public class MissileManager : MonoBehaviour
{
    ...

    float missileSpawnInterval = 0.5f;

    Coroutine spawningMissile;

    List<RecycleObject> missiles = new List<RecycleObject>();

    public Action MissileDestroyed;
    public Action AllMissilesDestroyed;

    ...
}

```

다음으로는 승리를 위한 필요 조건이 충족되었는지를 체크해야 합니다.

CheckAllMissileRestored() 함수로 가서 다음과 같이 조건 판정을 수행합니다. 즉, 미사일이 최대치까지 만들어졌고 동시에 만들어진 모든 미사일이 회수되었다면 AllMissilesDestroyed 이벤트를 발생시킵니다.

MissileManager.cs

```
void CheckAllMissileRestored()
{
    if (currentMissileCount == maxMissileCount && missiles.Count == 0)
    {
        AllMissilesDestroyed?.Invoke();
    }
}
```

이제 미사일 매니저에 생성한 새로운 이벤트를 게임 매니저와 연동하겠습니다. 먼저 GameManager.cs 로 가서, 방금 만든 미사일 매니저의 이벤트를 수신할 함수, OnAllMissileDestroyed() 를 다음과 같이 작성하고, 그 안에 “You Win” 이라는 메시지를 콘솔창에 표시할 Debug.Log() 명령어를 실행합니다.

GameManager.cs

```
void OnAllBuildingDestroyed()
{
    Debug.Log("You Lost");
}

void OnAllMissileDestroyed()
{
    Debug.Log("You Win");
}

// Update is called once per frame
void Update()
{
}
```

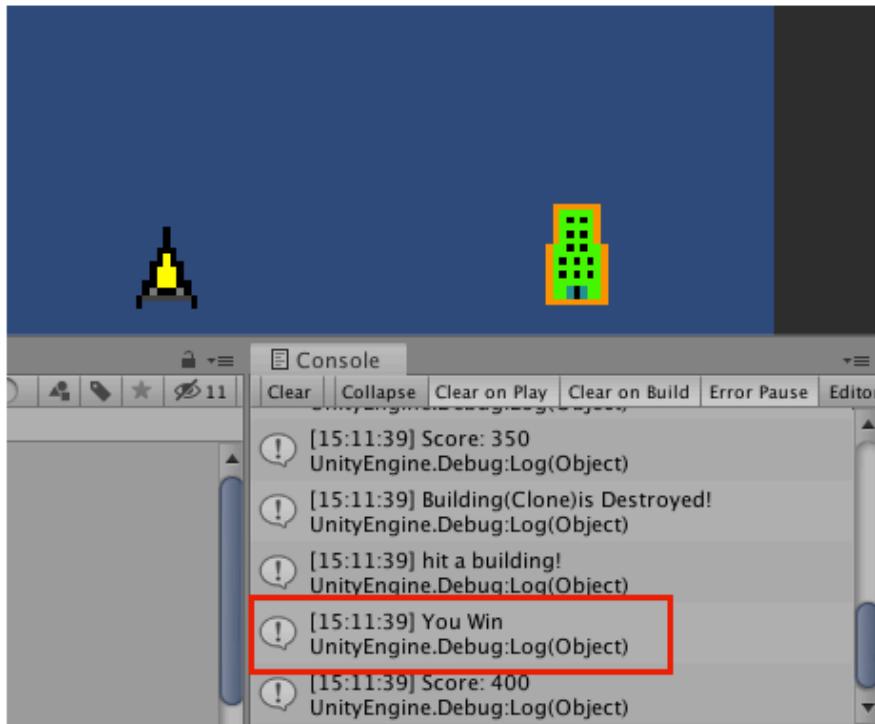
이제 BindEvents() 함수로 가서 missileManager의 새로운 이벤트를 다음과 같이 바인딩해 주겠습니다.

```
GameManager.cs
void BindEvents()
{
    mouseController.FireButtonPressed += launcher.OnFireButtonPressed;
    timeManager.GameStarted += buildingManager.OnGameStarted;
    timeManager.GameStarted += launcher.OnGameStarted;
    timeManager.GameStarted += missileManager.OnGameStarted;
    timeManager.GameStarted += uIRoot.OnGameStarted;
    missileManager.MissileDestroyed += scoreManager.OnMissileDestroyed;
    missileManager.AllMissilesDestroyed += OnAllMissileDestroyed;
    scoreManager.ScoreChanged += uIRoot.OnScoreChanged;
    buildingManager.AllBuildingsDestroyed += OnAllBuildingDestroyed;
}
```

역시 UnBindEvents()에서 이벤트를 언바인딩 해 주는 것도 잊지 마셔야 합니다.

```
GameManager.cs
void UnBindEvents()
{
    mouseController.FireButtonPressed -= launcher.OnFireButtonPressed;
    timeManager.GameStarted -= buildingManager.OnGameStarted;
    timeManager.GameStarted -= launcher.OnGameStarted;
    timeManager.GameStarted -= missileManager.OnGameStarted;
    timeManager.GameStarted -= uIRoot.OnGameStarted;
    missileManager.MissileDestroyed -= scoreManager.OnMissileDestroyed;
    missileManager.AllMissilesDestroyed -= OnAllMissileDestroyed;
    scoreManager.ScoreChanged -= uIRoot.OnScoreChanged;
    buildingManager.AllBuildingsDestroyed -= OnAllBuildingDestroyed;
}
```

이제 모든 작업이 끝났으므로 유니티 에디터로 가서 테스트를 해 보겠습니다. 원활한 게임 테스트를 위해 최대 미사일 개수를 4개로 줄이고 미사일 생성 인터벌 타임을 2초 정도로 늘린 뒤 테스트해 보겠습니다. 총알을 발사해서 날아 오는 미사일을 다 제거하면, 콘솔 창에 "You Win"이라는 메시지가 뜨는 것을 확인하실 수 있을 것입니다.



[동영상 예제 파일명: 086_check_winning_condition.mp4]

두 조건이 동시에 충족될 때의 승패 판정

앞에서 승리 패배 조건을 말씀 드릴 때, 미사일이 모두 파괴되었더라도 빌딩이 다 파괴되었다면 지는 것으로 간주한다고 말씀 드렸습니다. 그런데 마지막 미사일과 마지막 빌딩이 동시에 파괴되었을 경우에는 AllBuildingDestroyed 이벤트와 AllMissilesDestroyed 이벤트가 동시에 발생하기 때문에 승리와 패배 판정이 함께 발생할 우려가 있습니다.

정확한 승패 판정을 하기 위해서는 빌딩이 모두 파괴되었는지를 “먼저” 확인하고, 그렇지 않을 경우에만 미사일이 모두 파괴되었는지를 체크해야 합니다. 따라서 ‘미사일이 모두 파괴되었다’는 이벤트에 따른 승패 판정은 ‘빌딩이 모두 파괴되었다’는 이벤트에 따른 판정보다 한 템포 늦게 처리해야 합니다. 이를 위해 승패 판정 단계에서 코루틴(Coroutine)을 사용하겠습니다.

우선 GameManager.cs 로 가서, 모든 빌딩이 파괴되었는지를 체크하기 위한 bool 타입의 변수를 다음과 같이 선언하고 기본 값을 false 로 세팅하겠습니다.

```
GameManager.cs
```

```
public class GameManager : MonoBehaviour
{
    ...
    bool isAllBuildingDestroyed = false;

    MouseGameController mouseController;
    BuildingManager buildingManager;
    TimeManager timeManager;
    MissileManager missileManager;
```

```
ScoreManager scoreManager;  
}  
...
```

다음으로 OnAllBuildingDestroyed() 에 가서, isAllBuildingDestroyed 를 true 로 바꾸겠습니다. 모든 빌딩이 파괴되었다는 이벤트를 수신했기 때문입니다.

```
GameManager.cs  
void OnAllBuildingDestroyed()  
{  
    Debug.Log("You Lost");  
    isAllBuildingDestroyed = true;  
}
```

이제 게임 종료를 판단하기 위한 DelayedGameEnded() 라은 이름의 코루틴 함수를 작성합니다.

```
GameManager.cs  
void OnAllBuildingDestroyed()  
{  
    Debug.Log("You Lost");  
    isAllBuildingDestroyed = true;  
}  
  
void OnAllMissileDestroyed()  
{  
    Debug.Log("You Win");  
}  
  
IEnumerator DelayedGameEnded()  
{  
}
```

이 코루틴 함수에서는 딱 한 프레임만 기다리면 됩니다. 먼저, 모든 빌딩이 파괴되었을 경우에는 즉시 게임이 종료되면서 패배가 선언될 것입니다. 만약 그렇지 않다면

한 프레임 뒤에 ‘모든 미사일 파괴에 따른 판정’을 진행하면 됩니다. 최종 판정을 한 템포 늦게 내려야 하므로 다음과 같이 yield return null 을 추가하여 한 프레임의 대기 시간을 확보합니다.

GameManager.cs

```
IEnumerator DelayedGameEnded()
{
    yield return null;
}
```

이제 승리 선언을 하기 전에 먼저 isAllBuildingDestroyed 의 값이 true인지 false 인지를 체크하겠습니다. 만약에 isAllBuildingDestroyed 가 true 라면 모든 빌딩이 파괴된 것이므로 승리 판정을 내리면 안됩니다. 따라서 isAllBuildingDestroyed 의 값이 false 인 경우에만, 다시 말해서 모든 빌딩이 파괴되지 않았을 경우에만 승리를 선언할 수 있습니다. 이 경우에만 “You Win” 이라는 메시지가 콘솔창에 뜨도록 다음과 같이 조건문을 작성하겠습니다.

GameManager.cs

```
IEnumerator DelayedGameEnded()
{
    yield return null;

    if (!isAllBuildingDestroyed)
    {
        Debug.Log("You Win");
    }
}
```

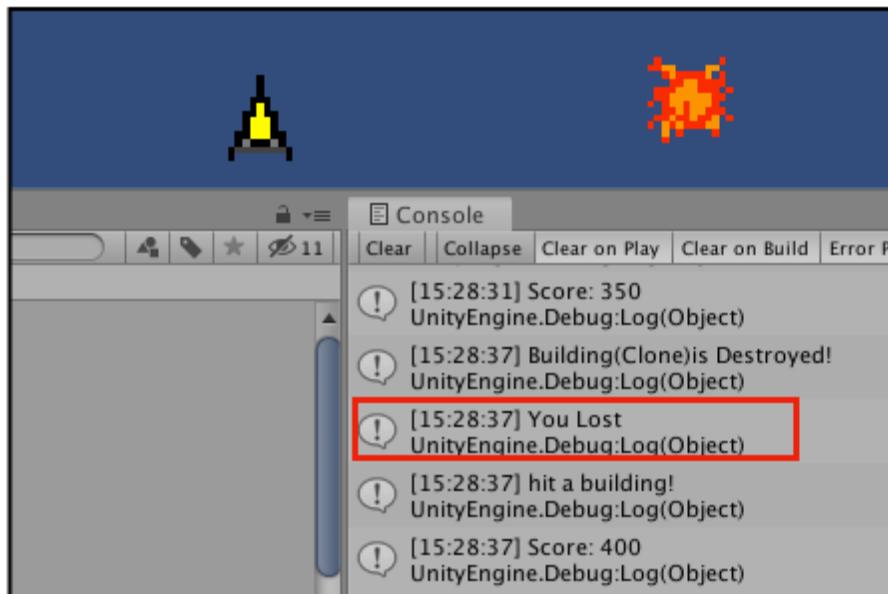
다음으로는 OnAllMissileDestroyed() 함수로 가서, 기존에 승리를 선언하는 코드를 삭제하고 방금 만든 코루틴 함수를 실행하겠습니다. 그러면 모든 미사일이 파괴되었을 때의 승리 판정은 빌딩의 파괴에 따른 판정보다 약간 늦게(한 프레임 늦게) 실행될

것입니다. (이렇게 할 경우 마지막 미사일과 마지막 빌딩이 동시에 파괴되는 경우는 패배 판정이 내려지게 됩니다.)

```
GameManager.cs
```

```
void OnAllMissileDestroyed()
{
    Debug.Log("You Win");
    StartCoroutine(DelayedGameEnded());
}
```

이제 유니티 에디터로 가서 테스트해 보겠습니다. 미사일 최대 개수와 생성 시간을 잘 조절해서 마지막 미사일이 마지막 빌딩과 충돌하는 상황을 만들어 보시기 바랍니다. 이 경우 패배만이 선언되는 것을 콘솔창에서 확인하실 수 있을 것입니다.



[동영상 예제 파일명: 087_update_win_lose_definition.mp4]

게임 종료 이벤트 처리

이제 게임이 종료되었다는 것을 알리기 위한 이벤트를 게임 매니저에 만들도록 하겠습니다. 이를 위해 GameManager.cs 로 가서 다음과 같이 GameEnded 라는 이름의 Action 이벤트를 선언해 줍니다. 이 때 GameEnded 액션은 다음과 같은 2개의 인자를 갖도록 해야 합니다.

1. bool 타입 : 승패의 결과를 전달 (승리일 경우 true, 패배일 경우 false)
2. int 타입: 게임이 끝났을 때 파괴되지 않고 남아 있는 빌딩의 개수를 전달 (보너스 점수 계산을 위해 사용됨)

이를 위해, 다음과 같이 네임스페이스 맨 앞에 using System; 을 추가하고 GameEnded 액션을 다음과 같이 선언해 주도록 하겠습니다.

GameManager.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class GameManager : MonoBehaviour
{
    ...

    public Action<bool, int> GameEnded;

    MouseGameController mouseController;
    BuildingManager buildingManager;
    TimeManager timeManager;

    ...
}
```

이 이벤트를 발생시키기 위해서는 남은 빌딩의 개수를 알 수 있어야 합니다. 남은 빌딩 개수를 알고 있는 것은 빌딩 매니저입니다. 따라서 BuildingManager.cs 로 가서 남은 빌딩 개수를 외부에서 알 수 있도록 읽기 전용 프로퍼티를 다음과 같이 생성하도록 하겠습니다.

BuildingManager.cs

```
public class BuildingManager
{
    Building prefab;
    Transform[] buildingLocators;
    Factory effectFactory;

    List<Building> buildings = new List<Building>();

    public bool HasBuilding { get {
        return buildings.Count > 0;
    }
}

public int BuildingCount { get { return buildings.Count; } }

public Action AllBuildingsDestroyed;

    ...
}
```

이제 다시 게임 매니저로 되돌아와서 게임이 종료되었음을 알리는 이벤트 발생 코드를 추가하도록 하겠습니다. GameManager.cs 의 OnAllBuildingDestroyed() 함수로 가서 테스트를 위해 작성했던 Debug.Log() 함수를 삭제합니다. 그리고 나서 함수 맨 아래 부분에서 게임이 종료했다는 GameEnded 이벤트를 발생시키는 코드를 다음 예제와 같이 작성합니다.

```

void OnAllBuildingDestroyed()
{
    Debug.Log("You Lost");
    isAllBuildingDestroyed = true;
    GameEnded?.Invoke(false, buildingManager.BuildingCount);
}

```

OnAllBuildingDestroyed() 함수가 실행될 때는 빌딩이 모두 파괴되었을 경우입니다. 이 경우에는 남은 미사일이 있는지 여부와 상관없이 게임이 무조건 패배로 종료되기 때문에 이곳에서 GameEnded 이벤트를 즉시 발생시키는 것입니다. 그리고 이벤트를 발생시킬 때, 첫 번째 인자로는 false를 전달합니다. 게임에서 패배했기 때문입니다. 그리고 두 번째 인자로 빌딩 매니저로부터 파괴되지 않고 남아 있는 빌딩의 개수를 전달합니다. 이 경우에는 무조건 0이 되겠지만 0이라는 값을 직접 입력하기보다는 buildingManager.BuildingCount라는 프로퍼티를 바로 전달해 주는 것이 더 좋은 습관입니다.

게임이 종료되는 순간을 처리하는 곳이 한 군데 더 있습니다. 바로 DelayedGameEnded() 코루틴 함수입니다. DelayedGameEnded() 함수로 이동해서 여기에서도 GameEnded 이벤트를 발생시키는 코드를 작성하도록 하겠습니다. 앞에서와 마찬가지로 Debug.Log() 명령문을 삭제하고 그 아래에서 GameEnded 이벤트를 발생시키되, 게임에서 승리했으므로 첫 번째 인자로 true를, 그리고 두 번째 인자로는 앞에서와 마찬가지로 빌딩 매니저로부터 남아 있는 빌딩의 개수를 가져 와서 전달하도록 하겠습니다.

```
IEnumerator DelayedGameEnded()
{
    yield return null;

    if (!isAllBuildingDestroyed)
    {
        Debug.Log("You Win");
        GameEnded?.Invoke(true, buildingManager.BuildingCount);
    }
}
```

[동영상 예제 파일명: 088_managing_game_end.mp4]

게임 종료 이벤트 수신 함수들을 만들자

이제 게임 매니저(GameManager) 쪽에서 게임 종료 이벤트(GameEnded)를 발생시키는 부분의 코드를 모두 작성하였습니다. 다음으로는 이 게임 종료 이벤트를 수신할 대상들을 찾아서 GameEnded 이벤트를 수신하기 위한 함수를 만들고, 이를 바인딩하는 작업을 하겠습니다. GameEnded 이벤트를 수신할 대상들은 다음과 같습니다.

BulletLauncher.cs	게임 종료시 더 이상 총알을 쓸 수 없도록 합니다.
MissileManager.cs	화면에 파괴되지 않은 미사일이 남아 있을 경우 모두 제거해야 합니다.
ScoreManager.cs	게임에서 승리했을 경우, 남아 있는 빌딩 개수를 근거로 보너스 점수를 반영합니다.
UIRoot.cs	게임이 종료되었을 때 승리/패배 메시지를 화면에 표시해야 합니다.

우선 총알 발사대의 GameEnded 이벤트 수신 함수를 생성하도록 하겠습니다.

BulletLauncher.cs 로 가셔서 다음과 같이 OnGameEnded 라는 이벤트 함수를 만들어 줍니다.

```
public class BulletLauncher : MonoBehaviour
{
    ...
    public void OnGameStarted()
    {
        isGameStarted = true;
    }

    public void OnGameEnded(bool isVictory, int buildingCount)
    {
    }
}
```

이 함수에서 해야 할 일은 한가지입니다. 마우스로 화면을 클릭했을 때 총알이 더 이상 발사되지 않도록 하는 것입니다. 총알 발사대에서는 isGameStarted 가 true인 경우에만 총알을 발사할 수 있게 되어 있으므로, 더 이상 총알을 발사하지 못하게 하려면 다음과 같이 isGameStarted를 false 로 바꿔 주기만 하면 됩니다.

```
public class BulletLauncher : MonoBehaviour
{
    ...
    public void OnGameStarted()
    {
        isGameStarted = true;
    }

    public void OnGameEnded(bool isVictory, int buildingCount)
    {
        isGameStarted = false;
    }
}
```

이제 게임 매니저(GameManager.cs)의 BindEvents() 함수로 가서, GameEnded 이벤트 송신자와 수신자를 바인딩해 주겠습니다. 그리고 다시 UnBindEvents() 함수로 가서 이 둘을 언바인딩하는 코드도 추가합니다.

GameManager.cs

```
void BindEvents()
{
    mouseController.FireButtonPressed += launcher.OnFireButtonPressed;
    timeManager.GameStarted += buildingManager.OnGameStarted;
    timeManager.GameStarted += launcher.OnGameStarted;
    timeManager.GameStarted += missileManager.OnGameStarted;
    timeManager.GameStarted += uIRoot.OnGameStarted;
    missileManager.MissileDestroyed += scoreManager.OnMissileDestroyed;
    missileManager.AllMissilesDestroyed += OnAllMissileDestroyed;
    scoreManager.ScoreChanged += uIRoot.OnScoreChanged;
    buildingManager.AllBuildingsDestroyed += OnAllBuildingDestroyed;

    this.GameEnded += launcher.OnGameEnded;
}

void UnBindEvents()
{
    mouseController.FireButtonPressed -= launcher.OnFireButtonPressed;
    timeManager.GameStarted -= buildingManager.OnGameStarted;
    timeManager.GameStarted -= launcher.OnGameStarted;
    timeManager.GameStarted -= missileManager.OnGameStarted;
    timeManager.GameStarted -= uIRoot.OnGameStarted;
    missileManager.MissileDestroyed -= scoreManager.OnMissileDestroyed;
    missileManager.AllMissilesDestroyed -= OnAllMissileDestroyed;
    scoreManager.ScoreChanged -= uIRoot.OnScoreChanged;
    buildingManager.AllBuildingsDestroyed -= OnAllBuildingDestroyed;

    this.GameEnded -= launcher.OnGameEnded;
}
```

이제 유니티 에디터로 가서 게임을 테스트해 보도록 하겠습니다. 게임을 플레이해 보면, 게임이 종료되었을 때 아무리 마우스 버튼을 클릭해도 총알이 나가지 않는 것을 확인하실 수가 있을 것입니다.



[동영상 예제 파일명: 089_bullet_launcher_game_ended.mp4]

다음으로는 미사일 매니저 차례입니다. MissileManager.cs 파일로 가서 다음과 같이 게임 종료 이벤트를 수신할 함수를 생성해 주도록 하겠습니다.

```
MissileManager.cs
public class MissileManager : MonoBehaviour
{
    ...
    Vector3 GetMissileSpawnPosition()
    {
        Vector3 spawnPosition = Vector3.zero;
        spawnPosition.x = UnityEngine.Random.Range(0f, 1f);
        spawnPosition.y = 1f;

        spawnPosition = Camera.main.ViewportToWorldPoint(spawnPosition);
        spawnPosition.z = 0f;
        return spawnPosition;
    }

    public void OnGameEnded(bool isVictory, int buildingCount)
    {
    }
}
```

이 함수에서 해야 할 일은 간단합니다. 게임이 종료되었을 때 화면에 파괴되지 않고 살아 있는 미사일들이 있으면 모두 Factroy로 회수해서 화면에서 제거해 주는 것입니다.

따라서 OnGameEnded() 이벤트 수신 함수가 실행되었을 때 제일 먼저 해야 할 일은 남은 미사일의 개수를 체크하는 것입니다. 이 때, 파괴되지 않고 남아 있는 미사일이 하나도 없다면 다음 코드를 실행할 필요가 없으므로 여기에서 바로 return 명령어를 이용하여 함수를 종료하도록 합니다.

```
public void OnGameEnded(bool isVictory, int buildingCount)
{
    if (missiles.Count == 0)
        return;
}
```

그렇지 않을 경우 foreach 구문을 이용하여 남아 있는 모든 미사일들을 missileFactory로 회수하도록 하겠습니다. (원래대로라면 미사일의 isActivated 값을 false로 바꿔 준 다음에 회수해야 하겠지만, 여기에서는 게임이 종료되는 시점이므로 그냥 회수해도 상관 없습니다.)

```
public void OnGameEnded(bool isVictory, int buildingCount)
{
    if (missiles.Count == 0)
        return;

    foreach (var missile in missiles)
    {
        missileFactory.Restore(missile);
    }
}
```

이제 미사일 매니저 쪽에서 이벤트 수신 함수 작성이 끝났으므로 게임 매니저 (GameManager.cs)의 BindEvents() 함수로 가서 GameEnded 이벤트의 송신자와 수신자를 바인딩해 주도록 하겠습니다. 이 때, UnBindEvents() 함수에서 언바인딩하는 것도 잊지 말아야 할 것입니다.

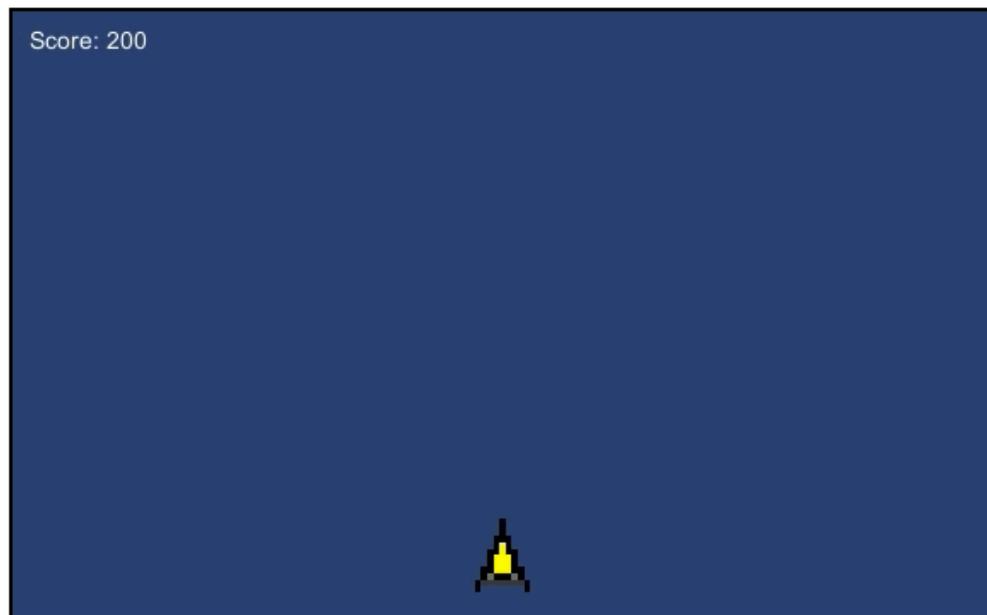
```
void BindEvents()
{
    mouseController.FireButtonPressed += launcher.OnFireButtonPressed;
    timeManager.GameStarted += buildingManager.OnGameStarted;
    timeManager.GameStarted += launcher.OnGameStarted;
    timeManager.GameStarted += missileManager.OnGameStarted;
    timeManager.GameStarted += uIRoot.OnGameStarted;
    missileManager.MissileDestroyed += scoreManager.OnMissileDestroyed;
    missileManager.AllMissilesDestroyed += OnAllMissileDestroyed;
    scoreManager.ScoreChanged += uIRoot.OnScoreChanged;
    buildingManager.AllBuildingsDestroyed += OnAllBuildingDestroyed;

    this.GameEnded += launcher.OnGameEnded;
    this.GameEnded += missileManager.OnGameEnded;
}

void UnBindEvents()
{
    mouseController.FireButtonPressed -= launcher.OnFireButtonPressed;
    timeManager.GameStarted -= buildingManager.OnGameStarted;
    timeManager.GameStarted -= launcher.OnGameStarted;
    timeManager.GameStarted -= missileManager.OnGameStarted;
    timeManager.GameStarted -= uIRoot.OnGameStarted;
    missileManager.MissileDestroyed -= scoreManager.OnMissileDestroyed;
    missileManager.AllMissilesDestroyed -= OnAllMissileDestroyed;
    scoreManager.ScoreChanged -= uIRoot.OnScoreChanged;
    buildingManager.AllBuildingsDestroyed -= OnAllBuildingDestroyed;

    this.GameEnded -= launcher.OnGameEnded;
    this.GameEnded -= missileManager.OnGameEnded;
}
```

이제 유니티로 가서 테스트를 해 보겠습니다. 그러면 게임을 실행하고 빌딩이 다 파괴되는 순간, 게임이 종료되면서 화면에 남아 있는 모든 미사일이 회수되어 사라지는 것을 확인하실 수 있을 것입니다.



[동영상 예제 파일명: 090_missile_manager_game_ended.mp4]

승리시의 보너스 점수 계산

다음으로는 스코어 매니저에 가서, 게임이 종료되었을 때 남아 있는 빌딩 개수를 근거로 보너스 점수를 계산하는 로직을 작성하도록 하겠습니다. 이를 위해 먼저 ScoreManager.cs 에 다음과 같이 게임 종료 이벤트를 수신할 함수를 작성하도록 하겠습니다.

```
ScoreManager.cs
public class ScoreManager
{
    ...
    public void OnMissileDestroyed()
    {
        score += scorePerMissile;
        ScoreChanged?.Invoke(score);
        Debug.Log("Score: " + score);
    }

    public void OnGameEnded(bool isVictory, int buildingCount)
    {
    }
}
```

여기에서 해야 할 것은 먼저 buildingCount 가 0 인가 여부를 체크하는 것입니다. buildingCount 가 0이라면 보너스 점수를 계산할 필요도 없고, 게임 스코어가 변경되었다는 이벤트를 발생시킬 필요가 없기 때문입니다. 따라서 다음과 같은 조건문을 하나 작성하도록 하겠습니다.

ScoreManager.cs

```
public void OnGameEnded(bool isVictory, int buildingCount)
{
    if (buildingCount == 0)
        return;

}
```

다음으로 남아 있는 빌딩 개수에 따라 보너스 스코어를 추가하고, 스코어가 변경되었다는 이벤트를 발생시키는 코드를 다음과 같이 작성합니다. “빌딩 1개당 보너스 점수 × 남은 빌딩 개수”를 구한 다음, 현재 스코어에 더해 주면 됩니다.

ScoreManager.cs

```
public void OnGameEnded(bool isVictory, int buildingCount)
{
    if (buildingCount == 0)
        return;

    score += scorePerBuilding * buildingCount;
    ScoreChanged?.Invoke(score);
}
```

이제 게임 매니저(GameManager.cs)의 BindEvents() 함수로 가서 이벤트 송신자와 수신자를 바인딩하고, 또한 UnBindEvents() 함수에 가서 다시 언바인딩하도록 하겠습니다.

```
void BindEvents()
{
    mouseController.FireButtonPressed += launcher.OnFireButtonPressed;
    timeManager.GameStarted += buildingManager.OnGameStarted;
    timeManager.GameStarted += launcher.OnGameStarted;
    timeManager.GameStarted += missileManager.OnGameStarted;
    timeManager.GameStarted += uIRoot.OnGameStarted;
    missileManager.MissileDestroyed += scoreManager.OnMissileDestroyed;
    missileManager.AllMissilesDestroyed += OnAllMissileDestroyed;
    scoreManager.ScoreChanged += uIRoot.OnScoreChanged;
    buildingManager.AllBuildingsDestroyed += OnAllBuildingDestroyed;

    this.GameEnded += launcher.OnGameEnded;
    this.GameEnded += missileManager.OnGameEnded;
    this.GameEnded += scoreManager.OnGameEnded;
}

void UnBindEvents()
{
    mouseController.FireButtonPressed -= launcher.OnFireButtonPressed;
    timeManager.GameStarted -= buildingManager.OnGameStarted;
    timeManager.GameStarted -= launcher.OnGameStarted;
    timeManager.GameStarted -= missileManager.OnGameStarted;
    timeManager.GameStarted -= uIRoot.OnGameStarted;
    missileManager.MissileDestroyed -= scoreManager.OnMissileDestroyed;
    missileManager.AllMissilesDestroyed -= OnAllMissileDestroyed;
    scoreManager.ScoreChanged -= uIRoot.OnScoreChanged;
    buildingManager.AllBuildingsDestroyed -= OnAllBuildingDestroyed;

    this.GameEnded -= launcher.OnGameEnded;
    this.GameEnded -= missileManager.OnGameEnded;
    this.GameEnded -= scoreManager.OnGameEnded;
}
```

이제 유니티로 가서 테스트해 보겠습니다. 게임이 끝나고 나면, 남은 빌딩에 따라 보너스 점수가 올라가는 것을 확인하실 수 있을 것입니다.



[동영상 예제 파일명: 091_score_manager_game_ended.mp4]

승패 결과를 화면에 표시하기

이제 게임의 승패를 화면에 표시하는 작업을 하겠습니다. 우선 UIRoot.cs로 가서 게임 결과를 표시할 UI 변수를 선언하고, [SerializeField] 어트리뷰트 붙여서 인스펙터에 노출되도록 하겠습니다.

```
UIRoot.cs
public class UIRoot : MonoBehaviour
{
    [SerializeField]
    Text scoreUI;

    [SerializeField]
    Text resultUI;

    ...
}
```

그런데 게임이 끝나기 전에는 이 UI가 화면에 보이지 않아야 합니다. 따라서 Start() 함수로 가서 이 UI가 보이지 않게 SetActive() 함수를 이용하여 비활성화시키도록 하겠습니다.

```
UIRoot.cs
void Start()
{
    scoreUI.gameObject.SetActive(false);
    resultUI.gameObject.SetActive(false);
}
```

그리고 다음과 같이 UIRoot.cs에도 게임 종료 이벤트를 수신할 함수를 작성해야 합니다. 다음과 같이 OnGameEnded라는 이름의 이벤트 수신자 함수를 작성하도록 하겠습니다.

```
public class UIRoot : MonoBehaviour
{
    ...
    public void OnScoreChanged(int score)
    {
        scoreUI.text = string.Format("Score: {0}", score);
    }

    public void OnGameEnded(bool isVictory, int buildingCount)
    {
    }
}
```

이 함수에서 해야 할 일은 간단합니다. 게임 매니저로부터 게임 종료 이벤트를 수신하면 앞에서 감추어 놓았던 resultUI 를 활성화시켜서 화면에 보이게 하고, 승리했을 경우(isVictory가 true)에는 “You Win!”이라는 텍스트를, 패배했을 경우(isVictory가 false일 경우)에는 “You Lose!”라는 텍스트를 표시하도록 하는 것입니다. 다음과 같이 코드를 작성해 줍니다.

```
public void OnGameEnded(bool isVictory, int buildingCount)
{
    resultUI.gameObject.SetActive(true);
    resultUI.text = isVictory ? "You Win!" : "You Lose";
}
```

이제 UIRoot.cs 와 관계된 코딩 작업이 모두 끝났으므로 게임 매니저(GameManager.cs)의 BindEvents() 함수로 가서 이벤트 송신자와 수신자를 바인딩해 주겠습니다. 그리고 다시 UnBindEvents() 함수로 가서 둘을 다시 언바인딩하는 코드를 작성하도록 하겠습니다.

```
void BindEvents()
{
    mouseController.FireButtonPressed += launcher.OnFireButtonPressed;
    timeManager.GameStarted += buildingManager.OnGameStarted;
    timeManager.GameStarted += launcher.OnGameStarted;
    timeManager.GameStarted += missileManager.OnGameStarted;
    timeManager.GameStarted += uIRoot.OnGameStarted;
    missileManager.MissileDestroyed += scoreManager.OnMissileDestroyed;
    missileManager.AllMissilesDestroyed += OnAllMissileDestroyed;
    scoreManager.ScoreChanged += uIRoot.OnScoreChanged;
    buildingManager.AllBuildingsDestroyed += OnAllBuildingDestroyed;

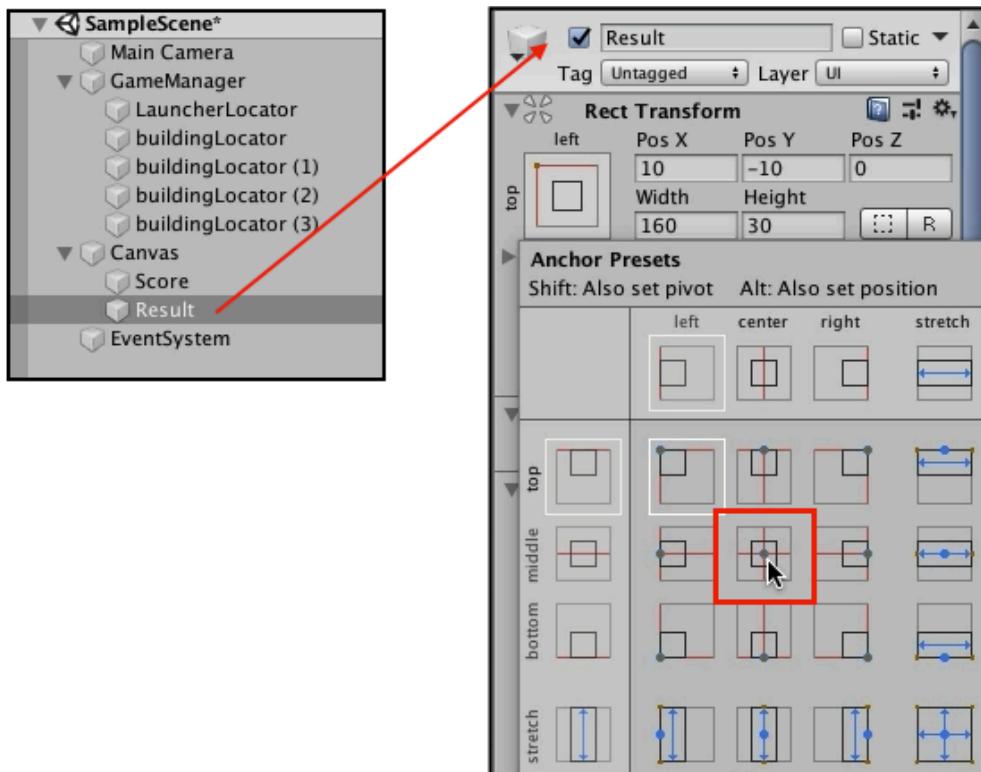
    this.GameEnded += launcher.OnGameEnded;
    this.GameEnded += missileManager.OnGameEnded;
    this.GameEnded += scoreManager.OnGameEnded;
    this.GameEnded += uIRoot.OnGameEnded;
}

void UnBindEvents()
{
    mouseController.FireButtonPressed -= launcher.OnFireButtonPressed;
    timeManager.GameStarted -= buildingManager.OnGameStarted;
    timeManager.GameStarted -= launcher.OnGameStarted;
    timeManager.GameStarted -= missileManager.OnGameStarted;
    timeManager.GameStarted -= uIRoot.OnGameStarted;
    missileManager.MissileDestroyed -= scoreManager.OnMissileDestroyed;
    missileManager.AllMissilesDestroyed -= OnAllMissileDestroyed;
    scoreManager.ScoreChanged -= uIRoot.OnScoreChanged;
    buildingManager.AllBuildingsDestroyed -= OnAllBuildingDestroyed;

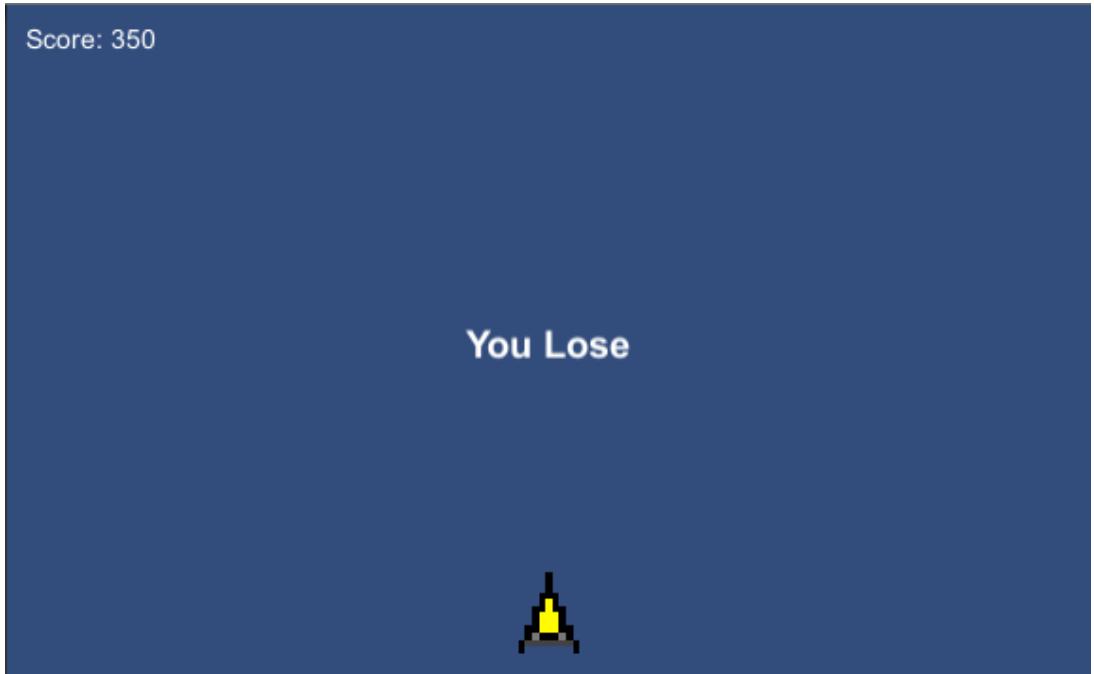
    this.GameEnded -= launcher.OnGameEnded;
    this.GameEnded -= missileManager.OnGameEnded;
    this.GameEnded -= scoreManager.OnGameEnded;
    this.GameEnded -= uIRoot.OnGameEnded;
}
```

이제 모든 작업이 끝났습니다. 유니티 에디터로 가서 실제 게임의 결과를 텍스트로 표시할 UI Text 를 유니티 씬에 만들도록 하겠습니다. 먼저 Canvas 의 자식 오브젝트인 Score를 복제하여 text UI를 하나 더 만들고 이름을 Result 라고 지어 주겠습니다. 그리고 텍스트의 Alignment를 가운데 정렬로 하고, 폰트 사이즈를 조금 크게 변경하도록 하겠습니다. 또한 화면에 잘 보이도록 텍스트의 색깔을 흰색으로 바꾸겠습니다.

그리고 나서 앵커 프리셋(Anchor Presets)를 선택해서 센터(Center)로 지정한 다음, 새로 만든 text 의 이름을 result 로 바꿔 주도록 하겠습니다. 이제 준비가 다 끝나면 result 게임 오브젝트를 드래그해서 Canvas 에 붙어 있는 UIRoot.cs 스크립트에 노출된 resultUI 하는 이름의 변수에 연결합니다.



이제 플레이 버튼을 눌러서 게임을 테스트해 보겠습니다. 게임이 종료되는 순간, 보시는 것처럼 “You Win!” 또는 “You Lose”라는 메시지가 화면에 잘 표시되는 것을 확인하실 수 있을 것입니다.



[동영상 예제 파일명: 092_uirroot_game_ended.mp4]

오디오 매니저(AudioManager)를 만드는 방법

지금까지 게임 플레이와 관련된 로직을 다루어 보았습니다. 현재 게임이 시작되고 종료될 때까지의 과정은 큰 문제 없지만, 아무런 소리가 나지 않아서 멋밋합니다. 따라서 게임 오디오를 처리하기 위해 AudioManager 를 만들어 보겠습니다. 앞에서 Factory 의 경우, 재활용 가능한 게임 오브젝트를 필요로 하는 클래스들이 각자 자신에게 필요한 Factory의 인스턴스를 만들어서 독립적으로 사용했습니다. 이 경우는 각각의 클래스들이 필요로 하는 게임 오브젝트(총알, 폭발 효과, 미사일 등)들이 서로 다르기 때문에 그렇게 할 수 있었습니다. 하지만 게임 오디오의 경우, 거의 모든 클래스들이 공통적으로 사용하고 있기 때문에 가급적이면 사용하기 쉬운 방법을 찾아 보는 것이 좋습니다.

사실 오디오 매니저(AudioManager)의 경우에도 이벤트 방식으로 구현하는 것이 이상적입니다. 각각의 클래스들이 “지금 미사일이 폭발한다!”고 이벤트를 발생시키면, 이를 받은 오디오 매니저가 그 이벤트 발생시 필요한 소리를 알아서 찾아 플레이하는 것입니다. 이렇게 하면 오디오 매니저와 다른 클래스간의 커플링을 최소화시키는 것이 가능하지만, 문제는 이 방식을 구현하는데 손이 많이 가고 불편하다는 것입니다. 오디오 매니저를 필요로 하는 모든 이벤트들과 바인딩/언바인딩을 하려면, 간단한 게임의 경우에는 큰 문제가 없겠지만 게임의 규모가 커져서 수 백~ 수천 개의 다른 사운드를 관리해야 할 경우에는 이벤트를 만들고 이를 바인딩하는 작업 자체만으로도 업무량이 너무 많아집니다. 따라서 이런 경우에는 싱글톤(Singleton)과 같은 글로벌 참조 방식을 사용하는 것이 효율면에서 나을 수 있습니다. 따라서 여기에서는 오디오 매

니저(AudioManager)를 싱글톤 형식으로 만들어 어디에서나 쉽게 사용하는 방식을 택하도록 하겠습니다.

많은 프로그래밍 아키텍처 책에서 싱글톤과 같은 글로벌 참조는 가급적 피하라고 권고하고 있지만, 이것은 권고일 뿐이지 반드시 따라야 할 법칙은 아닙니다. 게임을 만들다 보면 오디오 매니저의 경우와 마찬가지로, 글로벌 참조를 사용하는 경우가 더 생산성이 높을 경우가 있습니다. 특히 오디오 매니저의 경우에는 글로벌 참조를 통해 클래스 인스턴스의 상태를 변화시키거나 하는 일이 없고, 단지 어떤 소리를 내라고 명령만 내릴 뿐이므로, 싱글톤 패턴으로 편하게 사용하는 것도 괜찮은 방법입니다. 그럼 지금부터 오디오 매니저(AudioManager) 관련 작업을 시작하도록 하겠습니다.

스크립터블 오브젝트를 이용하여 오디오 파일을 관리하자

게임을 만들다 보면 수 많은 오디오 파일들을 관리해야 할 경우가 많습니다. 이를 위해서는 수 많은 오디오 파일들에 대한 참조 변수를 만들어야 하는데, 이를 씬(Scene)에 위치한 게임 오브젝트의 인스펙터를 통해 관리하기보다는, 스크립터블 오브젝트(ScriptableObject) 에셋을 만들어 관리하도록 하는 것이 더 편리합니다. 이 경우, 씬에 위치한 게임 오브젝트의 인스펙터에서는 다수의 개별 오디오 파일들이 아닌, 스크립터블 오브젝트 에셋 하나 만을 참조하도록 할 수 있습니다.

또 게임을 개발하다 보면, 기존의 사용하던 사운드 전체를 다른 분위기로 바꿔 본다거나 하는 식으로 테스트해야 할 경우가 발생할 수 있습니다. 이 경우 스크립터블 오브젝트 하나만을 교체함으로써 게임에서 사용하는 전체 사운드 파일들을 한번에 다른 것으로 바꿀 수 있습니다. 물론 새로운 사운드들이 마음에 들지 않을 경우 즉시 되돌리는 것도 가능합니다.

일단 스크립터블 오브젝트부터 하나 만들어 보겠습니다. 우선 AudioStorage.cs라는 이름의 유니티 C# 스크립트 파일을 생성합니다. 그리고 나서 다음과 같이 MonoBehaviour와 관련된 것을 모두 지우도록 하겠습니다.

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class AudioStorage : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}

```

그리고 나서 다음과 같이 이 클래스를 ScriptableObject 의 파생 클래스로 변경합니다. 그리고 클래스 선언부 위에 [CreateAssetMenu]를 붙여서, 앞으로 유니티 에디터의 [Create] 메뉴를 이용하여 AudioStorage의 에셋을 만들 수 있게 하겠습니다.

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[CreateAssetMenu]
public class AudioStorage : ScriptableObject
{
}

```

다음으로는 AudioStorage 클래스 바깥에 게임에 사용할 오디오들의 id 대신으로 사용할 enum 타입을 하나 선언하도록 하겠습니다. 다음 예제와 같이 SoundId 라는

enum 을 선언하고, 앞으로는 이것을 이용하여 게임에서 사용할 오디오 파일을 저장할 수 있도록 하겠습니다. 현재 미사일 커맨더 게임에서 사용할 오디오 파일은 4종류 이므로 아래와 같이 4개의 SoundId 타입을 세팅해 주겠습니다.

AudioStorage.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[CreateAssetMenu]
public class AudioStorage : ScriptableObject
{
}

public enum SoundId
{
    Shoot, BulletExplosion, BuildingExplosion, GameEnd
}
```

이제 각각의 오디오 파일과 위에서 설정한 SoundId 를 연동해 주어야 합니다. 이를 위해서 SoundSrc 라는 이름의 구조체를 만들도록 하겠습니다. 다음과 같이 구조체를 선언하시면 됩니다.

AudioStorage.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[CreateAssetMenu]
public class AudioStorage : ScriptableObject
{
}

public struct SoundSrc
{
}
```

```
public enum SoundId
{
    Shoot, BulletExplosion, BuildingExplosion, GameEnd
}
```

앞으로 우리는 이 구조체의 인스턴스들을 유니티 에디터의 인스펙터에 노출되도록 해서 필요한 오디오 클립 파일을 연결하고, SoundId 도 세팅할 예정입니다. 이처럼 구조체의 인스턴스들을 유니티의 인스펙터 뷰에 노출시킬 수 있게 하려면 다음과 같이 구조체 선언부 앞에 [Serializable]이라는 어트리뷰트를 붙여야 합니다. 그리고 이를 위해서는 다음과 같이 using System; 가 필요합니다. 네임스페이스 선언 부분 앞에 작성해 넣도록 하겠습니다.

AudioStorage.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[CreateAssetMenu]
public class AudioStorage : ScriptableObject
{

}

[Serializable]
public struct SoundSrc
{

}

public enum SoundId
{
    Shoot, BulletExplosion, BuildingExplosion, GameEnd
}
```

다음으로는 SoundSrc 구조체의 내용을 작성할 차례입니다. 다음과 같이 오디오 클립 파일을 참조할 soundFile이라는 이름의 변수와 SoundId를 지정할 변수를 선언하고, 각각에 대해 [SerializeField] 어트리뷰트를 붙여서 유니티 에디터의 인스펙터에 노출되도록 하겠습니다.

AudioStorage.cs

```
[Serializable]
public struct SoundSrc
{
    [SerializeField]
    AudioClip soundFile;

    [SerializeField]
    SoundId soundId;
}
```

현재 이 두개의 변수들은 모두 private로 되어 있습니다. 따라서 외부의 다른 클래스에서 접근할 수가 없습니다. 우리는 외부에서 이 변수의 값을 마음대로 바꾸는 것을 원하지 않으므로 읽기 전용 프로퍼티들을 만들어서 접근할 수 있도록 하겠습니다. 이를 위해서 먼저 SoundFile이라는 이름의 읽기 전용 프로퍼티를 만들고, 이것을 통해 soundFile을 읽어올 수 있게 하겠습니다. (아래 코드에서 대소문자에 주의하시기 바랍니다. 보통 변수명은 소문자로 시작하게 이름 짓고, 프로퍼티는 대문자로 시작하게 이름 짓습니다. 여기에서 프로퍼티는 대문자 'S'를 사용하고 있습니다.)

```
[Serializable]
public struct SoundSrc
{
    [SerializeField]
    AudioClip soundFile;

    [SerializeField]
    SoundId soundId;

    public AudioClip SoundFile { get { return soundFile; } }

}
```

그리고 나서, 그 아래에 Id 라는 이름의 읽기 전용 프로퍼티를 하나 더 만들겠습니다.

다음과 같이 이 프로퍼티를 통해 soundId 를 외부에서 읽을 수 있도록 하겠습니다.

```
[Serializable]
public struct SoundSrc
{
    [SerializeField]
    AudioClip soundFile;

    [SerializeField]
    SoundId soundId;

    public AudioClip SoundFile { get { return soundFile; } }

    public SoundId Id { get { return soundId; } }

}
```

[동영상 예제 파일명: 093_audiostorage_scriptable_object_1.mp4]

이제 AudioStorage 클래스로 이동합니다. AudioStorage 에서는 위에서 만든 구조체의 인스턴스들을 담을 배열을 선언하고 [SerializeField] 어트리뷰트를 이용해서 인스펙터에서 이 배열의 요소들에 접근할 수 있도록 해 주겠습니다.

AudioStorage.cs

```
[CreateAssetMenu]
public class AudioStorage : ScriptableObject
{
    [SerializeField]
    SoundSrc[] soundSrcs;

}

[Serializable]
public struct SoundSrc
{
    [SerializeField]
    AudioClip soundFile;

    [SerializeField]
    SoundId soundId;

    public AudioClip SoundFile { get { return soundFile; } }
    public SoundId Id { get { return soundId; } }
}

public enum SoundId
{
    Shoot, BulletExplosion, BuildingExplosion, GameEnd
}
```

다음으로 해야 할 작업은 외부에서 원하는 오디오 파일을 SoundId 를 이용하여 쉽게 찾을 수 있도록 하는 것입니다. 현재 AudioStorage 클래스가 가지고 있는 것은 soundSrc 라는 이름의 배열입니다. 따라서 필요한 오디오 파일을 찾기 위해 for 루프나 foreach 를 사용해야 합니다. 하지만 이 방법을 사용하게 되면 필요한 오디오 파일을 하나 찾으려 할 때마다 배열의 모든 요소를 일일이 다 뒤져야 합니다. 이렇게 하기

보다는, SoundId를 키(key)로 해서 필요한 오디오 파일을 단번에 찾을 수 있도록 딕셔너리를 미리 만들어 두는 것이 편리하고 성능면에서도 효율적입니다. 따라서 다음과 같이 주어진 배열을 이용해서 만들 딕셔너리를 선언해 주도록 하겠습니다. 이 때 키(key)로는 사운드 Id(SoundId), 그리고 값(value)으로는 AudioClip 타입을 사용하도록 하겠습니다.

```
AudioStorage.cs
public class AudioStorage : ScriptableObject
{
    [SerializeField]
    SoundSrc[] soundSrcs;

    Dictionary<SoundId, AudioClip> dicSounds =
        new Dictionary<SoundId, AudioClip>();
}
```

다음으로는 Dictionary 를 만들 때 사용할 함수를 작성하도록 하겠습니다. 다음과 같이 GenerateDictionary() 라는 이름의 함수를 하나 선언합니다.

```
AudioStorage.cs
public class AudioStorage : ScriptableObject
{
    ...
    void GenerateDictionary()
    {
    }
}
```

그리고 이 함수 안에 다음과 같이 soundSrcs 배열의 요소 개수만큼 루프를 돌면서 dicSounds 에 각각 id를 키로, soundFile 을 값으로 하는 딕셔너리 요소들을 추가하는 코드를 추가하겠습니다.

```
public class AudioStorage : ScriptableObject
{
    ...
    void GenerateDictionary()
    {
        for (int i = 0; i < soundSrcs.Length; i++)
        {
            dicSounds.Add(soundSrcs[i].Id, soundSrcs[i].SoundFile);
        }
    }
}
```

이제 딕셔너리가 완성되었으니, 다음으로는 외부에서 필요한 사운드를 가져 올 수 있도록 하는 Get()이라는 이름의 public 함수를 생성하겠습니다. 이 때 SoundId 타입의 인자를 받아, 원하는 오디오 파일을 바로 찾아낼 수 있도록 하겠습니다.

```
public AudioClip Get(SoundId id)
{
}
```

이 함수에서 제일 먼저 해야 할 일은, 만약에 soundSrcs 배열에 아무 것도 들어 있지 않으면 경고 메시지를 표시하도록 하는 것입니다. 다음과 같이 코드를 작성해 줍니다.

```
public AudioClip Get(SoundId id)
{
    Debug.Assert(soundSrcs.Length > 0, "No soundSource data!");
}
```

그리고 앞에서 만든 GenerateDictionary() 함수를 실행해서 딕셔너리를 처음으로 만드는 시점을 여기로 잡아 보았습니다. 다시 말해서 외부에서 오디오 파일을 처음 사용

하려고 할 때, 만약 딕셔너리가 만들어져 있지 않다면 그 때 딕셔너리를 새로 만들어서 사용하려는 것입니다.

이를 위해서 먼저 dicSounds라는 딕셔너리의 Count가 0인지를 체크한 뒤, 0일 경우에는 GenerateDictionary() 함수를 실행해서 필요한 딕셔너리를 만듭니다.(이런 식으로 일단 딕셔너리가 한번 만들어지고 나면 다시 만들어지지 않을 것입니다. 그 때는 dicSounds.Count가 0보다 클 것이기 때문입니다.)

AudioStorage.cs

```
public AudioClip Get(SoundId id)
{
    Debug.Assert(soundSrcs.Length > 0, "No soundSource data!");

    if (dicSounds.Count == 0)
    {
        GenerateDictionary();
    }
}
```

이렇게 딕셔너리가 다 만들어진 상태에서는 다음과 같이 함수의 인자로 전달된 Id를 키로 해서, 필요한 오디오 파일을 바로 찾아 리턴해 주면 됩니다.

AudioStorage.cs

```
public AudioClip Get(SoundId id)
{
    Debug.Assert(soundSrcs.Length > 0, "No soundSource data!");

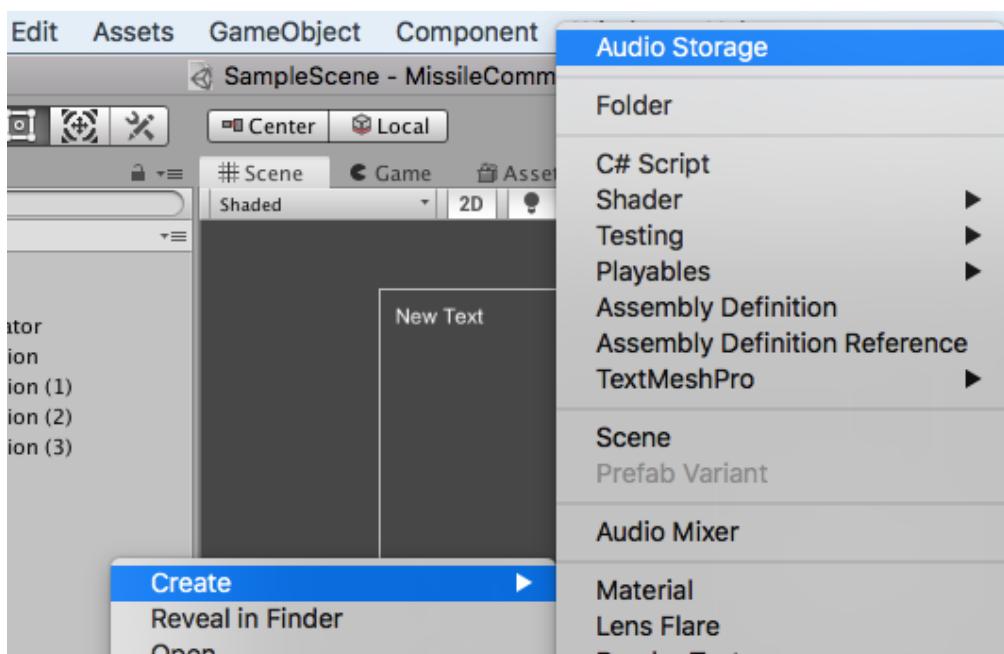
    if (dicSounds.Count == 0)
    {
        GenerateDictionary();
    }

    return dicSounds[id];
}
```

[동영상 예제 파일명: 094_audiostorage_scriptable_object_2.mp4]

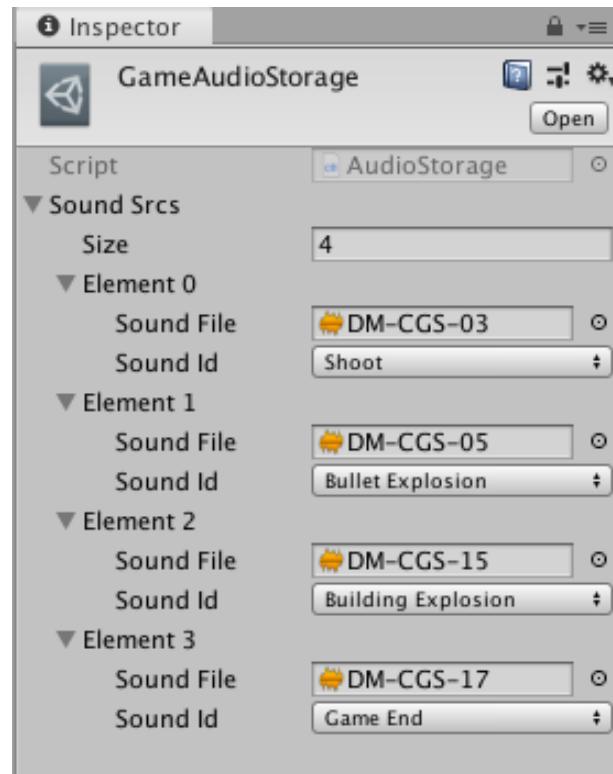
이제 AudioStorage 스크립터를 오브젝트에 필요한 코드를 모두 작성했습니다. 다음에는 이 AudioStorage를 이용하여 실제 스크립터를 오브젝트 애셋(Asset)을 만들어 사용하는 방법을 살펴 보도록 하겠습니다.

먼저 유니티 에디터로 가서 프로젝트 폴더에 [ScriptableObjects]라는 이름의 폴더를 만들겠습니다. 그리고 유니티 에디터 상단의 Create 메뉴를 클릭해 보면, 다음과 같이 새로운 AudioStorage 애셋을 만들 수가 있게 됩니다. 이렇게 애셋 파일을 하나 만들 어 [ScriptableObjects] 폴더에 넣은 뒤, 파일의 이름을 GameAudioStorage로 변경해 주도록 하겠습니다.



다음으로는 GameAudioStorage 애셋을 클릭한 뒤, 인스펙터에서 배열의 개수를 4로 지정하도록 하겠습니다. 그리고 다음과 같이 오디오 파일을 연결하고 각각에 대해 사

운드 ID 를 지정해 줍니다. 오디오 파일들은 [CausalGameSounds]라는 폴더에 들어 있습니다.



[동영상 예제 파일명: 095_game_audiostorage_creation.mp4]

오디오 매니저(AudioManager)를 만들자

이제 AudioManager.cs 유니티 스크립트를 작성하도록 하겠습니다. 먼저 MonoBehaviour 파생 클래스를 하나 만든 뒤, 이름을 AudioManager.cs라고 정해 줍니다. 이 때 Start()와 Update()는 사용하지 않을 것이므로 아래 예제와 같이 지워 주겠습니다. (MonoBehaviour는 지우지 않습니다)

AudioManager.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class AudioManager : MonoBehaviour
{
    //Start is called before the first frame update
    void Start()
    {

    }

    //Update is called once per frame
    void Update()
    {

    }
}
```

앞에서도 말씀 드렸듯이 AudioManager.cs는 글로벌 참조가 가능한 싱글톤(Singleton) 형식으로 만들 것입니다 이를 위해 다음과 같이 자기 자신(AudioManager)을 참조할 수 있는 instance라는 이름의 변수를 선언하고 이를 static으로 세팅해 주도록 하겠습니다.

```
public class AudioManager : MonoBehaviour
{
    public static AudioManager instance;
}
```

그리고 나서 다음과 같이 Awake() 함수를 만들겠습니다.

```
public class AudioManager : MonoBehaviour
{
    public static AudioManager instance;

    void Awake()
    {
    }
}
```

Awake() 함수에서는 먼저 instance에 자기 자신을 할당합니다.

```
public class AudioManager : MonoBehaviour
{
    public static AudioManager instance;

    void Awake()
    {
        instance = this;
    }
}
```

이렇게만 해도 글로벌 참조가 되므로, 그대로 사용할 수 있습니다. 하지만 이왕 싱글톤(Singleton) 패턴에 따라 코드를 작성하기로 했으니, 자기 자신의 인스턴스가 이미 있는지를 체크하는 조건문을 추가하기로 했습니다. 싱글톤을 사용한다는 것은 이 게임에서 AudioManager 클래스에 대하여 단 1개의 인스턴스만을 허용하겠다는 이야-

기입니다. 따라서 instance 가 null 인 경우에만 instance 에 자기 자신(this)를 할당하고, 그렇지 않은 경우(이미 AudioManager의 인스턴스가 있는 경우)에는 스스로를 파괴하도록 하겠습니다. 이렇게 하면, 게임을 플레이하는 동안 AudioManager 클래스의 인스턴스가 절대로 2개 이상이 될 수 없습니다.

AudioManager.cs

```
void Awake()
{
    if (instance == null)
        instance = this;
    else
        Destroy(this);
}
```

참고로 여기에서 this 는 게임 오브젝트가 아니라 AudioManager.cs 라는 클래스의 인스턴스입니다. 만약 AudioManager 클래스가 연결된 게임 오브젝트 자체를 삭제하고 싶으면 Destroy(gameObject)라고 해 주셔야 합니다. 하지만 이곳에서는 그럴 필요가 없으니 지금 이대로 진행하도록 하겠습니다.

다음으로는 아까 만든 스크립터를 오브젝트 에셋을 참조할 soundStorage라는 이름의 변수를 선언하겠습니다. 그리고 인스펙터에 노출되도록 [SerializeField] 어트리뷰트를 덧붙여 줍니다.

AudioManager.cs

```
public class AudioManager : MonoBehaviour
{
    public static AudioManager instance;

    [SerializeField]
    AudioStorage soundStorage;

    void Awake()
```

```
{  
    if (instance == null)  
        instance = this;  
    else  
        Destroy(this);  
}  
}
```

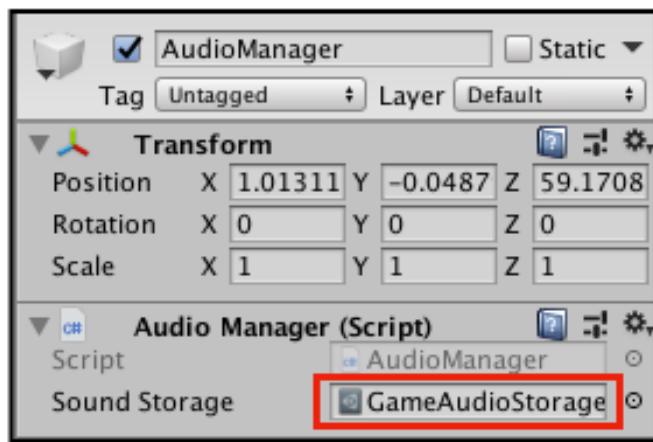
다음으로는 사운드 아이디(SoundId)를 이용해서 원하는 사운드를 플레이하는 함수를 만들도록 하겠습니다. 유니티에서 오디오를 플레이하기 위해서는 AudioSource 컴포넌트를 게임 오브젝트에 붙이거나, 아니면 AudioSource.PlayClipAtPoint() 라는 함수를 실행하면 됩니다. 여기에서는 후자의 방법을 사용할 예정입니다.

AudioSource.PlayClipAtPoint()의 첫번 째 인자로 오디오 클립(AudioClip)이 필요한데, 이것은 우리가 앞에서 만든 soundStorage 스크립터를 오브젝트에 있는 Get() 함수를 이용해서 가져올 수 있습니다. 또한 AudioSource.PlayClipAtPoint()의 두번 째 인자로는 소리가 날 게임 월드상의 위치를 Vector3 형식으로 전달해 주어야 하는데, 우리가 만드는 게임은 2D 게임이므로 소리가 날 위치는 단순히 Vector3.zero 로 지정하면 됩니다.

```
public class AudioManager : MonoBehaviour  
{  
    ...  
  
    public void PlaySound(SoundId id)  
    {  
        AudioSource.PlayClipAtPoint(soundStorage.Get(id), Vector3.zero);  
    }  
}
```

이제 모든 코딩 작업이 끝났으니 다음으로는 유니티 에디터로 이동하겠습니다. 씬 뷰에서 빈 게임 오브젝트(Empty GameObject)를 만들고 AudioManager 라고 이름을 변경한 다음에 방금 만든 AudioManager.cs 스크립트를 연결하도록 하겠습니다.

그리고 인스펙터에 노출된 SoundStorage 에 아까 만든 GameAudioStorage 스크립터를 오브젝트 에셋을 드래그해서 갖다 놓으면 됩니다.



[동영상 예제 파일명: 096_audio_manager_creation.mp4]

오디오 매니저(AudioManager)를 이용해서 소리를 내자

이제 AudioManager와 관련된 작업은 다 끝났습니다. 이제 게임에서 소리를 내야하는 곳을 찾아서 실제 오디오를 플레이해 보도록 하겠습니다.

우선 총알 발사대(BulletLauncher.cs)로가 보겠습니다. 총알을 발사하는 순간에 소리를 내야 하므로, OnFireButtonPressed() 함수를 찾아 다음과 같은 코드를 삽입해 줍니다.

BulletLauncher.cs

```
public void OnFireButtonPressed(Vector3 position)
{
    if (!isGameStarted)
        return;

    if (!canShoot)
        return;

    RecycleObject bullet = bulletFactory.Get();
    bullet.Activate(firePosition.position, position);
    bullet.Destroyed += OnBulletDestroyed;

    AudioManager.instance.PlaySound(SoundId.Shoot);

    canShoot = false;
}
```

이 상태로 유니티 에디터로 가서 게임을 플레이해 보시면, 마우스를 클릭해서 총알을 발사할 때마다 효과음이 나는 것을 확인하실 수 있을 것입니다.

다음으로는 총알이 폭발하는 순간의 오디오입니다. 역시 총알 발사대 (BulletLauncher.cs)로 가서 OnBulletDestroyed() 함수를 찾아 다음과 같이 폭발 효과를 내는 순간 소리가 나도록 하겠습니다.

BulletLauncher.cs

```
void OnBulletDestroyed(RecycleObject usedBullet)
{
    Vector3 lastBulletPosition = usedBullet.transform.position;
    usedBullet.Destroyed -= OnBulletDestroyed;
    bulletFactory.Restore(usedBullet);

    RecycleObject explosion = explosionFactory.Get();
    explosion.Activate(lastBulletPosition);
    explosion.Destroyed += OnExplosionDestroyed;

    AudioManager.instance.PlaySound(SoundId.BulletExplosion);
}
```

이제 벌딩이 파괴되는 순간의 사운드 처리입니다. 벌딩 매니저(BuildingManager.cs)로 가서 OnBuildingDestroyed() 함수를 찾으신 다음, 아래 예제와 같이 벌딩 폭발 소리가 나도록 명령어를 추가하겠습니다.

BuildingManager.cs

```
void OnBuildingDestroyed(Building building)
{
    AudioManager.instance.PlaySound(SoundId.BuildingExplosion);

    Vector3 lastPosition = building.transform.position;
    lastPosition.y += (building.GetComponent<BoxCollider2D>().size.y * 0.5f);
    building.Destroyed -= OnBuildingDestroyed;
    int index = buildings.IndexOf(building);
    buildings.RemoveAt(index);
    GameObject.Destroy(building.gameObject);

    ...
}
```

마지막으로 게임이 종료되었을 때(게임 오버) 소리가 나도록 해 보겠습니다. 게임 매니저로 가서, 게임이 종료되는 것을 처리하는 함수를 찾아서 관련된 코드를 삽입하도록 하겠습니다. 그런데 게임 매니저(GameManager.cs)에서 게임이 종료되었다는 이벤트(GameEnded)를 발생시키는 곳은 모두 두 군데입니다. 따라서 OnAllBuildingDestroyed() 함수와 DelayedGameEnded() 각각에 대해 아래와 같은 코드를 추가해 주도록 하겠습니다.

GameManager.cs

```
void OnAllBuildingDestroyed()
{
    isAllBuildingDestroyed = true;
    GameEnded?.Invoke(false, buildingManager.BuildingCount);
    AudioManager.instance.PlaySound(SoundId.GameEnd);
}

void OnAllMissileDestroyed()
{
    StartCoroutine(DelayedGameEnded());
}

IEnumerator DelayedGameEnded()
{
    yield return null;

    if (!isAllBuildingDestroyed)
    {
        GameEnded?.Invoke(true, buildingManager.BuildingCount);
        AudioManager.instance.PlaySound(SoundId.GameEnd);
    }
}
```

이제 유니티로 가서 게임을 플레이해 보겠습니다. 게임을 실행하고 플레이해 보시면, 게임 사운드가 잘 나는 것을 확인하실 수 있을 것입니다.

[동영상 예제 파일명: 097_play_sound_with_audio_manager.mp4]

오디오 매니저(AudioManager)와 같이 글로벌 참조를 이용하는 방법은 편리한 만큼 위험성을 내포하고 있다는 것을 명심하셔야 합니다. 우리가 지금 만든 오디오 매니저의 경우 다른 클래스에서 오디오 매니저의 상태를 변경하는 일이 없기 때문에 큰 문제는 없지만, 그렇지 않을 경우 게임 코드가 점점 많아지고 프로젝트에 참여한 프로그래머의 수가 늘어남에 따라 문제가 발생할 수 있습니다. 게임 실행 도중 어떤 코드에서 오디오 매니저의 상태를 마음대로 바꿔 버릴 경우, 그것을 추적해 내기가 어렵기 때문입니다. 따라서 싱글톤과 같은 글로벌참조는 정말 어쩔 수 없을 때만 신중하게 사용한다는 생각을 가지셔야 합니다.

이상으로 미사일 커맨더(Missile Commander) 게임 로직과 관련한 프로그래밍을 마무리 하겠습니다. 물론 제대로 된 게임을 만들기 위해서는 여기에서 더 해야 할 작업이 훨씬 더 많습니다. 하지만 이 책에서 다루고자 하는 것은 확장성과 유지보수성을 높이기 위한 코딩 방법을 살펴 보는 것이지 완성된 게임을 만드는 것이 아니므로, 더 이상 진행하지는 않고 여기에서 마무리하도록 하겠습니다.

마무리하며

지금까지 미사일 커맨더(Missile Commander)라는 간단한 게임의 로직을 만들어 보면서, 확장성과 유지보수성을 높이기 위해 어떤 고민을 해야 하는지에 대해 함께 살펴보았습니다. 이를 위해 인터페이스를 이용하는 방법, 디펜던시를 외부에서 주입하는 방법, 이벤트를 이용하여 클래스 인스턴스간에 서로의 존재를 모른 채 소통하는 방법 등 다양한 방법을 시도해 보았습니다.

또한 게임 매니저(GameManager.cs)라는 최상위 관리자 클래스가 빌딩 매니저나 미사일 매니저와 같은 중간 관리자급의 클래스 인스턴스들을 생성하고 디펜던시 인젝션, 이벤트 바인딩 같은 것들을 책임지도록 함으로써, 흔히 말하는 컴포지션 루트(Composition Root)라는 기법을 흥내내 보기도 하였습니다. 예제 코드에서 이벤트 기반의 커뮤니케이션 방법을 지나치게 강박적으로 사용했다는 문제가 있기는 하지만, 그 과정에서 클래스의 인스턴스들이 서로의 존재를 알지 못하는 가운데, 각자가 담당한 역할만 수행하는 것만으로도 전체 게임 로직이 문제 없이 작동할 수 있다는 점을 보여 드렸다는 점에서 나름 의미가 있는 작업을 했다는 생각이 듭니다.

유니티 프로그래밍 입문 단계를 벗어나 다음 단계로 나아가기 위해서는 단순히 유니티 엔진의 사용 방법을 익히는 차원이 아니라 소프트웨어 아키텍처에 대한 최소한의 이해가 필요합니다. 가급적 느슨하게 커플링된 구조를 만들도록 항상 신경을 써야 하고, 싱글톤과 같은 글로벌 참조를 남용하지 않으며, 필요할 때 기능을 추가하는 것은 쉬우면서도 이를 위해 기존의 코드를 수정할 필요는 없도록 프로그래밍 설계를 해야

합니다. 아마 여러분이 앞으로 다양한 수준의 프로그래머, 게임 디자이너, 레벨 디자이너와 함께 일하게 되면 아마 몸으로 깨달으실 수 있게 될 것입니다. 아마 그 때가 되어야 진짜 고민이 시작되겠지만, 그 이전에라도 연습을 통해 조금이나마 사전 훈련을 쌓아 둔다면, 진짜 그 문제가 현실감 있게 다가 올 때 큰 도움이 될 것이라 생각합니다.

이 책은 확장성과 유지 보수성을 고려한 코딩의 맛만 보여 드린 정도입니다. 또한 C# 프로그래밍과 유니티 입문서 정도만 읽으신 초급 프로그래머 독자 분들을 대상으로 했기 때문에 가급적 이론적인 용어를 직접적으로 사용하지 않으려 노력했습니다. 하지만 여기에서 다룬 내용을 충분히 몸으로 익히신 뒤, 소프트웨어 아키텍처나 디자인 패턴 관련 책들을 읽으신다면 이 책에서 다룬 몇 가지 방법이 특정한 디자인 패턴 (Design Pattern)과 원리(Principles)들을 참고했음을 아실 수 있을 것입니다. 시중에 이 분야에 대한 전문적인 책들이 많이 나와 있으므로 여러분이 의지를 가지고 공부를 계속하신다면 빠른 발전을 기대하실 수 있을 것입니다.

이제 저의 “유니티 C# 스크립트 프로그래밍 연습 시리즈” 중 첫번 째 책이 완성되었습니다. 앞으로 얼마나 더 많은 책을 쓸 수 있을지는 모르겠지만, 개인적으로도 이 책을 쓴 것이 계기가 되어 다양한 주제의 책들을 더 많이 출간할 수 있게 되기를 희망합니다. 여기까지 읽어 주신 분들께 감사드리며, 저는 이만 인사 드리겠습니다.

감사합니다.

저자 김옹남 드림