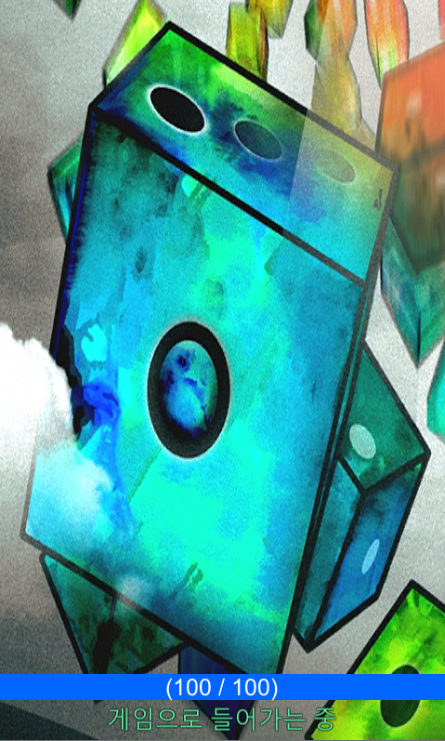
CopyProject -RandomDice

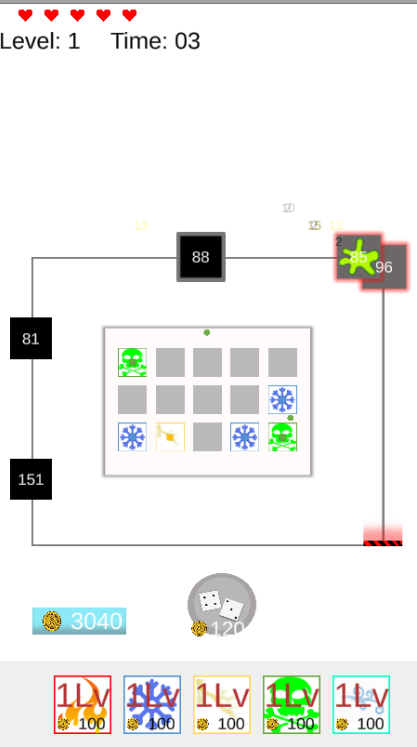
CopyProject 란? : 실제 존재하는 게임들을 분석하고 모방하여 1주 단위로 총3주 동안의 스프린트를 거쳐 해당 기간 내에 핵심 기능을 구현하는 것이 목표입니다.

대상 게임: [RandomDice](https://play.google.com/store/apps/details?id=com.percent.royaldice&hl=ko)

GitHub: <https://github.com/irispon/RandomDiceCopy>

YouTube: <https://youtu.be/s6gYw1HasOU>





프로젝트 설명:

1. **DataBase, Cache**

먼저 프로젝트를 시작함에 앞서 중요한 것은 데이터와 어플리케이션(프로그램)을 분리시키는 것이라 생각했습니다.

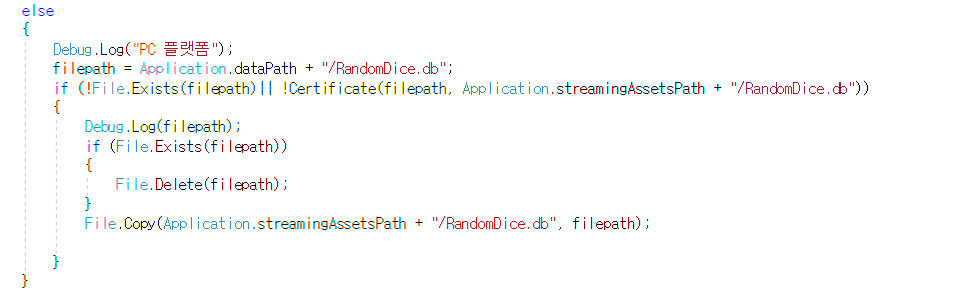
왜냐하면 데이터와 어플리케이션을 분리함으로써 분리된 데이터의 변경이 있어도 추가적인 컴파일이 필요하지 않기 때문입니다.

따라서 본 프로젝트에 사용된 데이터베이스는 비록 경로가 어플리케이션 내부, 혹은 컴퓨터 내부에 존재하고 있지만 서버에 존재하고 있다는 가정 하에 접근을 생각하게 되었습니다.

데이터베이스 접근은 필연적으로 지연시간을 가져오게 됩니다. 서버에 연결하는 시간, 데이터 베이스에서 필요한 데이터를 불러오는 시간 등이 필요합니다.

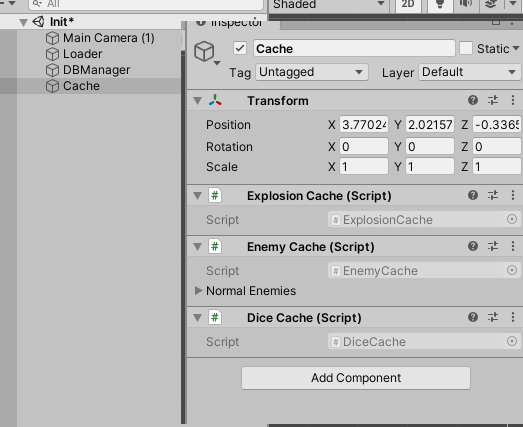
그래서 저는 두 가지의 방법을 통해 이 문제를 해결하고자 했습니다. 하나는 서버의 데이터 베이스를 로컬에 복사하는 것. 또 다른 하나는 처음 실행 시 필요한 데이터들을 메모리에 올려놓는 것. 이었습니다.





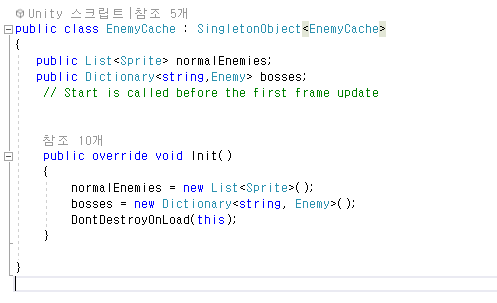
[그림. 1] Database복사 코드

처음 프로젝트를 시작할 때 안드로이드 플랫폼이 아닌 PC플랫폼에서 시작을 했었고 원본 데이터베이스 파일 자체는 Asset이 아닌 컴퓨터 내 다른 경로에 데이터베이스 파일을 두고 테스트를 진행했었습니다. 후에 안드로이드 플랫폼도 호환을 되게 하면서 원본 데이터 베이스 경로를 Streaming Assets 폴더에 두게 되었습니다.



[그림. 2] Cache

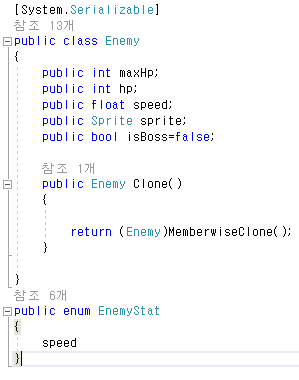
다음은 데이터 캐시 관련 오브젝트입니다. 데이터에 빠른 접근이 가능하기 때문에 실제 Cache는 아니지만Cache라 명명했습니다. 게임이 시작될 때 DontDestroyOnLoad를 통해 계속해서 유지되며 각 캐시 스크립트들은 Singleton Pattern을 적용하여 하나의 객체를 통해 접근이 가능하게 되어있습니다.



[그림. 2-1] Cache의 구조. 직접 만든 SingletonObject<T>를 상속하여 Singleton pattern을 적용했다.

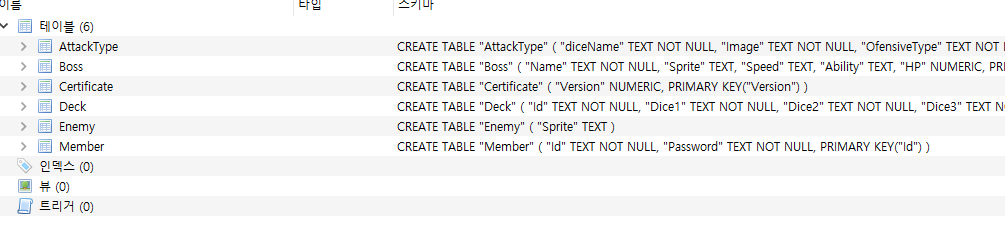
해당 Cache들의 데이터들은 실행될 때 DataParser 스크립트를 통해 Database에 있는 특정 데이터들을 저장하게 됩니다. 즉 데이터가 필요할 경우 해당 Cache 객체를 통해 빠른 접근이 가능하게 됩니다.





[그림. 3] DataPaser와 Data Class

DataPaser는 처음 초기화 시 실행되며 DB에 접근하여 필요한 데이터들을 데이터 클래스, 혹은 구조체로 데이터를 정리하여 Cache 객체에 저장시키는 역할을 합니다. 원래 목적은 Asset내부가 아닌 외부의 데이터(Sprite, Animation 등) 또한 불러올 수 있게 만들었으나 Android로 변경하게 되면서 부분적으로만 외부에서 불러올 수 있게 했습니다.

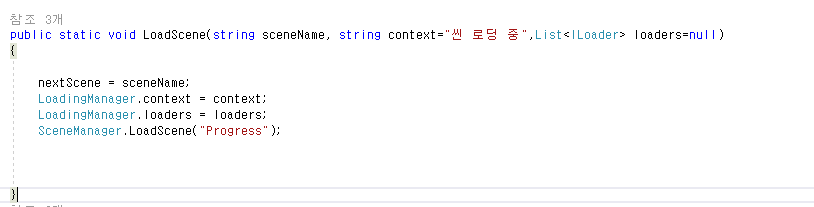


[그림. 3-1] 데이터 구조. Member와 Certificate 테이블을 제외하면 parsing되는 테이블이다.

1. **Progress Scene, Loader**

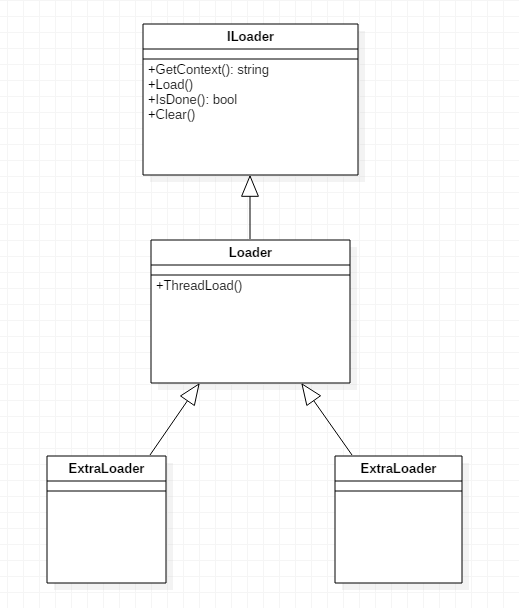
예전 Android 과목을 수강할 때 들었던 말이 있습니다. 프로그램 혹은 어플리케이션의 사용자는 반응에 대하여 일정 시간 안에 응답이 있어야 된다는 내용이었습니다. 즉 그 이상의 시간이 지체되면 사용자는 이상을 느끼거나 좋지 않다는 인식을 주게 된다는 내용이었습니다.

따라서 초기화 시 데이터베이스를 복사하거나 불러올 때, Scene과 Scene을 넘어갈 때 Progress Scene이 필요하다 느꼈고 Progress Scene을 만들게 되었습니다.

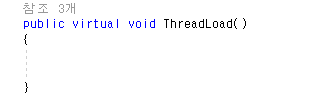
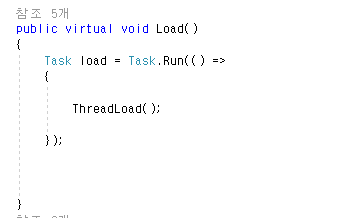


[그림. 4] Progress Sceane의 호출

Progress Scene의 로딩은 Coroutine내에서 이루어집니다. 또한 로딩 중에 미리 필요한 작업들(시간이 필요한 작업)을 Progress Scene에서 처리하고자 ILoader 인터페이스를 상속받은 Class들을 통해 작업을 할 수 있게 만들었습니다.



[그림. 4-1] Loader의 구조



[그림. 4-1-1] Loader의 구조

Loader의 대략적인 구조는 이렇게 됩니다. ILoader를 Loader가 구현했으며 또한 Loader는 ThreadLoad라는 메서드를 통해 상속받은 기타 Loader들이 Thread에서 작업할 수 있게 구현하였습니다.



[그림. 4-1] Loader의 호출

이전 Scene에서 넘겨받은 Loader들은 Progress Scene의 Coroutine 내에서 작업을 호출되게 됩니다. 기본적으로 Loader 내부는 스레드 작업도 포함되기 때문에 해당 작업들은 앞선 Loader들의 작업들이 끝날 때까지 대기하게 됩니다.

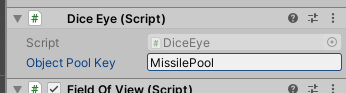
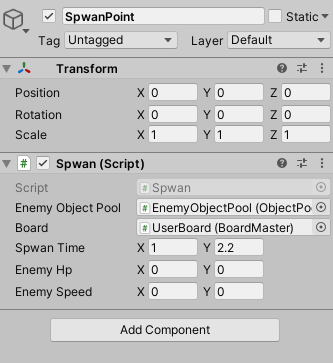
여기서 사용한 대기 방법은 While을 통한 대기이지만 이는 별로 좋은 방법은 아니기 때문에 추후 CallBack을 통해 개선할 예정입니다.

1. **ObjectPool**



[그림. 5-1] ObjectPool

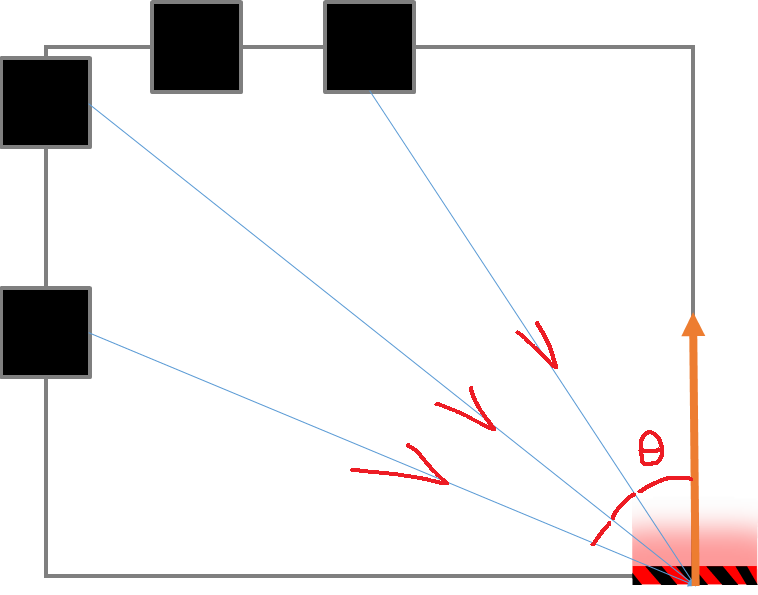
탄환 발사, 피해량 표시, 적의 생성 등에는 빈번한 오브젝트 사용이 있기 때문에 ObjectPool을 직접 구현하였습니다. ObjectPool은 생성시 ObjectPoolManager라는 Singleton Object에 등록되며 해당 Manager를 통해 Object Pool에 접근하거나 Objectool을 직접 Object에 등록해서 접근할 수 있게 하였습니다.



[그림. 5-2] 오브젝트 등록을 통한 접근, 오브젝트 매니저를 통한 접근(key를 이용한 접근)

1. **타겟 설정 알고리즘**

타겟을 설정하는데 있어서 중요한 것은 목적지와 가장 가까운 적을 타겟으로 삼는 것이었습니다. 처음에는 OverlapBoxAll을 통해 주사위 눈이 가진 사거리 내에서 타겟을 설정했으나 타겟이 적절하게 설정되지 않아 직접 타겟 알고리즘을 만들게 되었습니다.



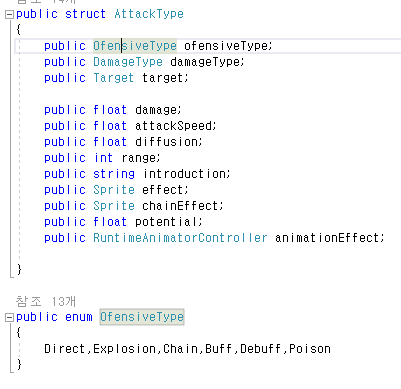
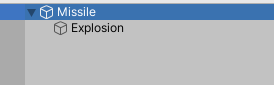
[그림. 6] 타겟 알고리즘

적 오브젝트가 가까워질수록 목적 지점의 수직 방향 벡터와 적과 목적 지점의 방향 벡터의 각이 점점 작아지는 점, 또한 각이 0이 될 때는 목적지와 적의 거리를 통해 타겟을 설정할 수 있다는 점을 통해 알고리즘을 만들었습니다.



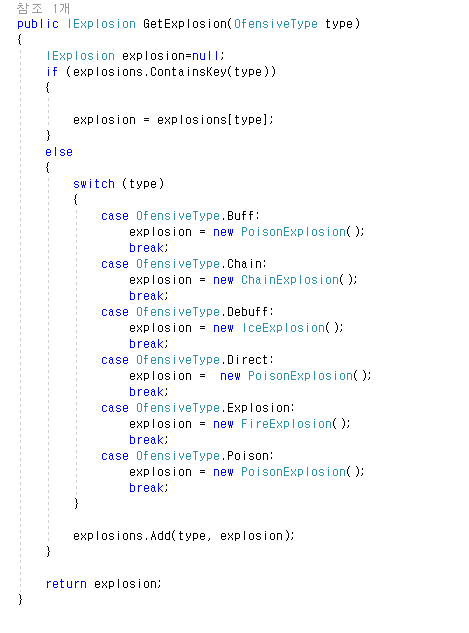
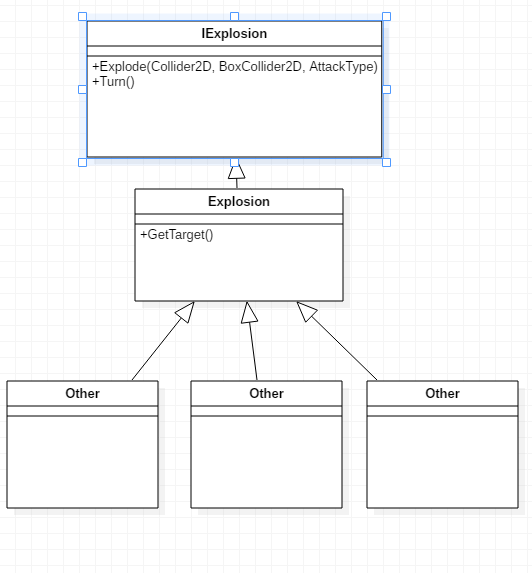
[그림. 6-1] 해당 사항을 구현한 알고리즘. 거리는 소수점으로 두어 각이 0이 되었을 때 거리가 의미가 있도록 구현하였다.

1. **탄, 효과, 폭발**



[그림. 7] 탄의 구조와 탄이 가지는 정보

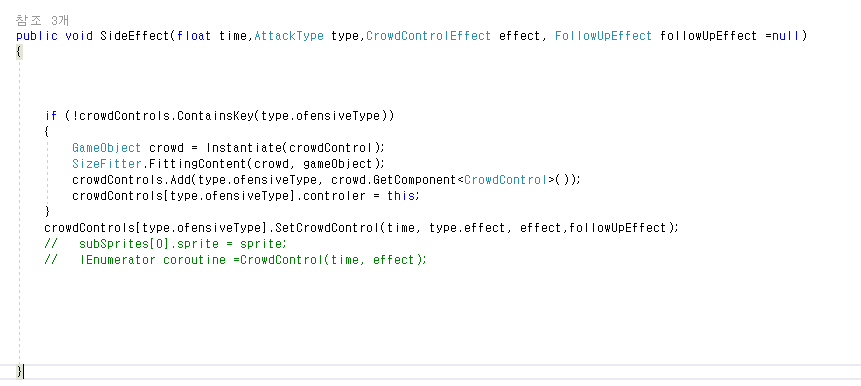
주사위의 눈이 발사하는 탄환은 주사위의 속성에 따라 발사하는 탄환이 달라지므로 받는 AttackType에 따라 탄환의 속성을 변경할 수 있게 만들었습니다. 기본적인 알고리즘은 탄환이 적에게 부딪치면 피해를 주고 OfensiveType에 따라 추가적인 효과(폭발, 상태이상)을 일으키도록 하였습니다.



[그림. 7] 후속 효과의 구조와 캐시

상태 이상이 발생한 경우 Delegate를 통해 효과를 기술한 메서드를 넘겨주게 됩니다. 이후 상태 효과를 받은 대상은 효과가 중복될 경우 지속 시간과 효과를 갱신하고 새로운 효과인 경우 대상은 새로운 상태 효과 오브젝트를 생성하여 적용됩니다.

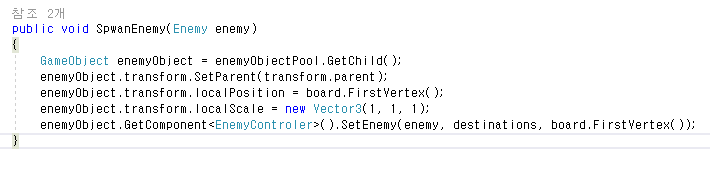




[그림. 8] Explosion을 상속한 IceExplosion의 코드와 적에게 상태이상을 부여하는 메서드

1. **몬스터의 생성과 이동**

몬스터의 생성은 Spawn이라는 오브젝트에서 이루어지게 됩니다. Spawn은 Enemy 클래스의 스테이터스과 외형을 무작위로 생성하고 레벨 스케일링 시스템에 따라 전체적으로 생성되는 값이 높아지게 디자인했습니다. Crouton을 통해 일정 시간 내에서 무작위로 지속적으로 생성되며 Monster가 이동할 경로를 지정해주는 역할을 합니다.





[그림. 9] 몬스터 생성 로직

몬스터의 이동은 [그림. 10] 에서 보이는 보드의 3개의 좌표를 통해 움직이게 됩니다. Coroutine에서 Vector의 MoveToVector를 통해 움직이며 좌표에 도달하게 되면 다음 좌표로 넘어가 이동을 하게 됩니다. 최종적으로 목표지점에 도달하게 되면 플레이어의 체력을 감소시키고 ObjectPool로 돌아가게 됩니다.

[그림. 10] 게임 보드

1. **아쉬운 점과 개선 사항**

7-1. 이펙트 효과

상태이상 효과의 경우 그림으로 그려져 있어 적이 사각형과 일정 크기가 아니라면 이펙트가 틀어지는 현상이 발생했습니다. 처음에는 대상 스프라이트 사이즈에 따라 스케일을 줄여보고자 했으나 모양이 다른 경우의 해결책이 아니었기 때문에 반려하게 되었습니다.

다음 프로젝트에서는 OutLine을 이용하여 스프라이트 외곽에 이펙트를 구현할 예정입니다.

7-2. 안드로이드 외부 리소스 로딩(DB관련)

안드로이드로도 실행이 가능하게 프로젝트를 전환함에 따라 Database를 내부에 두게 되었습니다. 외부에서 불러오는 방식의 경우 권한을 얻어와야하는 것으로 알고 있어서 다음 프로젝트에서는 실제 서버를 구축하고 외부 리소스를 불러오게 할 것입니다.