

게임 제목: 키메라 시뮬레이터

장르: 육성 시뮬레이션, 수집

주요 시스템: 키메라 발생, 키메라 전투

모티브가 된 게임: 우마무스메 프리티더비, 포켓몬스터

유니티버전: 유니티6, URP

플랫폼: 스팀



모티브 게임과 주요 시스템 관계



사이게임즈.

현재 플레이중인 게임.

육성하려는 '우마무스메(캐릭터)'를 선택하고, 정해진 '기간'동안, 훈련을 '선택'하여 능력치를 증가시키고, 순차적으로 목표를 달성하여 최종적으로 육성을 완료. 이때 특정 시나리오 목표에 따라 굿 엔딩과 배드 엔딩으로 나뉨.

시나리오별 스토리 진행.

->

육성하려는 '키메라'를 선택하고, 유전자를 '선택'하고 삽입하여 능력치 증가 및 스킬 선택, 최종적으로 굿 엔딩을 확인.

진행한 시간에 따라 TimeLine 스토리 개방.

직급에 따라 ResearcherRank 스토리 개방.

모티브 게임과 주요 시스템 관계



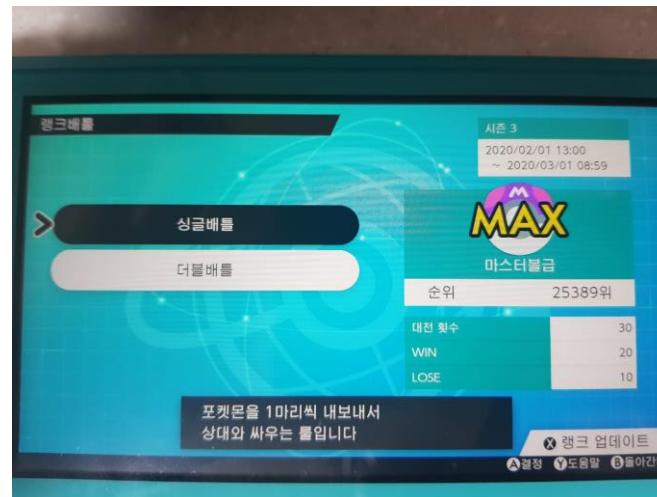
닌텐도 스위치, 게임프리크

재밌게, 열심히 플레이했던 게임(통신 배틀, 싱글 배틀 기준 마스터볼 랭크 달성).

싱글배틀 기준, 두 플레이어의 포켓몬이 턴 제로 기술을 사용하여 주고받음.

->

키메라는 전투(상호 테스트)가 진행되는 동안 서로의 상태(공격, 피격 등)를 주고 받음.



이 게임의 개발 계기 및 경험 목표

저는 과학관련 유튜브를 자주 봅니다. 영상에서 유전과 관련된 내용이 나왔고 그 내용을 기반으로 게임을 만들면 좋겠다는 생각이 들어 개발을 시작했습니다. 현재 해당 게임을 1차적으로 완료 및 출시 직전 상태(2025.02.20) 기준, 유니티를 운용해본 기간이 정확히 1년인 상태로, 지금까지 배운 프로그래밍 지식을 체화해보고 싶었습니다.

게임 개발자를 꿈꾸게 되면서 지금까지 플레이했던 게임들 중 재미있게 플레이했던 게임들의 주요 기능을 구현하기 위해 여러가지 방법을 시도하고 직접 구현하면서 기초적인 개발의 감을 익혀보았습니다. 특히 UGUI를 많이 다뤄본 것 같습니다.

활용할 수 있는 자원을 효율적으로 사용하기 위해 가진 자원의 질을 분석하고 직접 사용하여 좋은 결과를 유도하는 것이 이 게임의 제작 목표였고 플레이어가 최대한 직접 선택을 하게 하여 플레이어의 선택에 의한 결과임을 느끼게 하고 싶었습니다.

TimeLine스토리와 로딩 씬, 키메라 설명 등 일부 문구를 통해서 무분별하게 키메라를 발생하고, 그 키메라들이 어떻게 사용이 되는지 노출시켜서 플레이어를 세계관에 몰입시키고 싶었습니다.

게임 컨셉 및 플롯

전설의 동물인 용이 실존했고, 우주로부터 지구로 이동하여 정확히 1년 뒤에 지구를 파괴(작 중 정화)할 것을 예고했으며 인류는 이를 막기 위해 여러가지 방안을 강구한다. 모든 과학 분야가 용의 지구 침공을 대비하기 위해 온 힘을 쓸기 시작했고, 지구에 머무르는 용 자체에 대한 연구와 무기 개발이 주를 이루게 된다.

특히 생물학 분야는 직접 용 자체에 대한 연구를 집중적으로 진행하고 그 결과를 공유하기 시작했다. 발생학과 유전학의 어떤 학자들은 이 때를 노려 연구윤리를 무시하고 유전자 변형체인 키메라를 발생시키기 시작한다. 이 무분별하게 발생한 키메라들이 의도치 않게 연구소를 탈출하게 되면 어떤 사회적 혼란을 초래할지 알 수 없기 때문에 남태평양의 바다 한 가운데의 무인도에 연구소를 설립, 이전하게 된다.

이곳에서 연구원들은 자유롭게 키메라를 발생하고 용들에게 대항할 키메라를 창조해야 한다.

플레이어는 이 연구원들 중 한 명이며 키메라 발생, 상호 테스트를 통해 성과를 남기고 인간 키메라에 대한 연구소에 합류하는 것이 목표다.

이 게임의 진행 목표

주 목표: 최종 직급 달성을 통한 굿 엔딩(스토리) 확인.

부가 목표: 원 종 키메라의 변종(Mutant)를 수집.

부가 목표: 오피셜테스트 최종 클리어.

직급 - (최초)Junior, Senior, Principal, (최종)Director. 연구 성과에 따라서 직급 상승.

원 종 키메라 - 플레이어가 처음부터 발생할 수 있는 키메라.

변종 - 플레이어가 특정 목표를 달성하여 발생 권한을 획득하면 발생할 수 있는 키메라

* 특정 목표: Principal 직급의 일부 NPC는 변종을 다루며, 해당 NPC에게서 승리하면 그 NPC가 다루는 변종의 발생 권한을 획득

오피셜 테스트 - 낮은 직급 순으로 정해진 다른 NPC 연구원들의 키메라들을 상호 테스트.

게임 내 주요 시스템

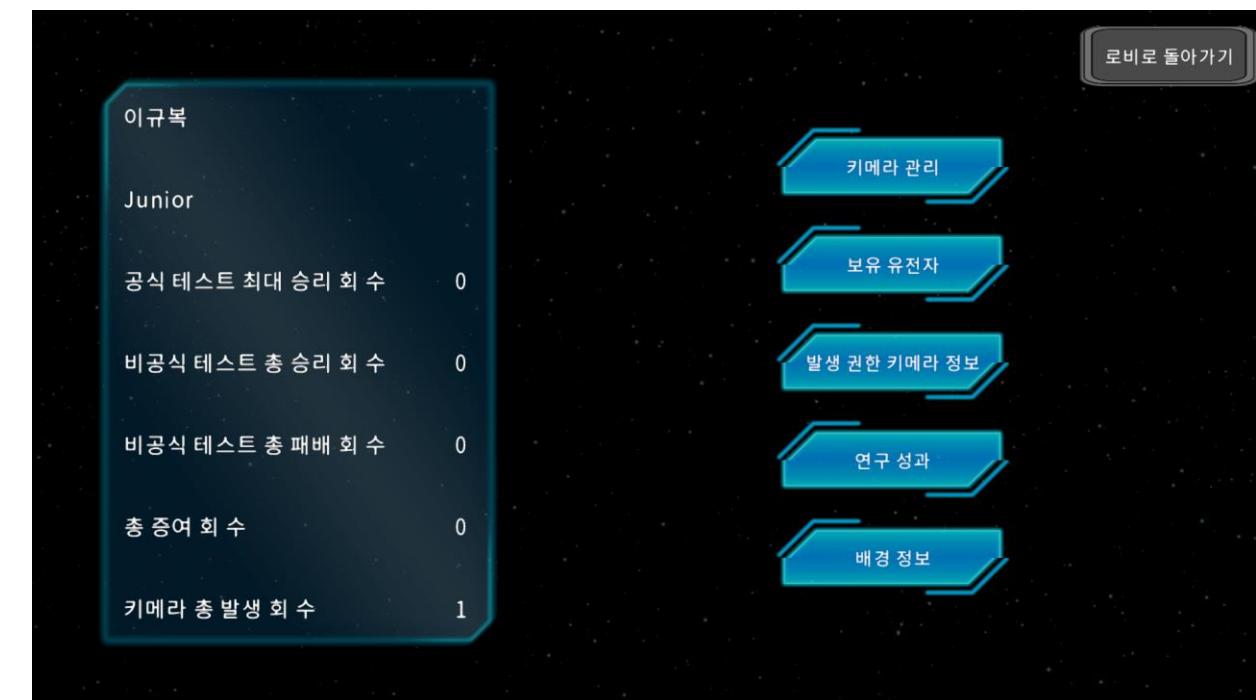
1. 직급과 업적
2. 키메라(키메라, 키메라 데이터 스크립터블 오브젝트, Dna, Gene, 발생, Status, Skill)
3. 전투(공식/비공식 테스트, 적응형 NPC)
4. 인 게임 스토리(Json활용)
5. 그 외(씬 전환, 다 회 차 플레이 등)

1. 직급과 업적

플레이어는 게임을 시작하면 가장 낮은 직급인 Junior부터 시작합니다.

직급은 총 4개로 이루어져 있으며 플레이어는 Junior, Senior, Principal, Director 순으로 직급을 상승시킬 수 있습니다.
직급은 플레이어의 여러가지 활동을 통한 연구 성과에 따라 상승합니다.

1. 로비 – 내 권한 화면



2. 내 권한 – 연구 성과 화면



1. 직급과 업적(전체 코드-관련 인터페이스, 클래스, 매니저)

```
public interface IAchieveClear
{
    □ 5 implementations
    public bool IsCleared { get; }

    □ 5 implementations
    public bool IsRewarded { get; }

    □ 5 implementations
    public void SetCleared();

    □ 5 implementations
    public void GetReward();
}

□ 8 usages □ 1 exposing API
public class OfficialTestAchieveInfo : IAchieveClear{...}

□ 8 usages □ 1 exposing API
public class NonOfficialTestVictoryAchieveInfo : IAchieveClear{...}

□ 8 usages □ 1 exposing API
public class NonOfficialTestDefeatAchieveInfo : IAchieveClear{...}

□ 8 usages □ 1 exposing API
public class ReallocationAchieveInfo : IAchieveClear{...}

□ 8 usages □ 1 exposing API
public class DevelopmentChimeraAchieveInfo : IAchieveClear{...}

/* public class ResearcherRankInfo ... */

□ 39 usages □ 7 exposing APIs
public enum ResearcherRank
{
    Public,
    Junior,
    Senior,
    Principal,
    Director
}
```

```
155     public class AchieveManager : ImmortalObject<AchieveManager>, IDataReset
156     {
157         private const int AchieveCount = 10;
158
159         □ Frequently called □ 18 usages
160         public ResearcherRank PlayerRank { get; private set; } = ResearcherRank.Junior;
161
162         □ Frequently called □ 9 usages
163         public int HighestOfficialTestClearStage { get; private set; } = 0;
164         □ Frequently called □ 7 usages
165         public int TotalNonOfficialTestVictoryCount { get; private set; } = 0;
166         □ Frequently called □ 7 usages
167         public int TotalNonOfficialTestDefeatCount { get; private set; } = 0;
168         □ Frequently called □ 7 usages
169         public int TotalReallocationCount { get; private set; } = 0;
170         □ Frequently called □ 7 usages
171         public int TotalDevelopmentChimeraCount { get; private set; } = 0;
172
173         private int _currentOfficialTestAchieveInfoKey;
174         private int _currentNonOfficialTestVictoryAchieveInfoKey;
175         private int _currentNonOfficialTestDefeatAchieveInfoKey;
176         private int _currentReallocationAchieveInfoKey;
177         private int _currentDevelopmentChimeraAchieveInfoKey;
178
179         □ Frequently called □ 10 usages
180         public Dictionary<int, OfficialTestAchieveInfo> OfficialTestAchieveInfos { get; private set; }
181         □ Frequently called □ 10 usages
182         public Dictionary<int, NonOfficialTestVictoryAchieveInfo> NonOfficialTestVictoryAchieveInfos { get; private set; }
183         □ Frequently called □ 10 usages
184         public Dictionary<int, NonOfficialTestDefeatAchieveInfo> NonOfficialTestDefeatAchieveInfos { get; private set; }
185         □ Frequently called □ 10 usages
186         public Dictionary<int, ReallocationAchieveInfo> ReallocationAchieveInfos { get; private set; }
187         □ Frequently called □ 10 usages
188         public Dictionary<int, DevelopmentChimeraAchieveInfo> DevelopmentChimeraAchieveInfos { get; private set; }
189
190         □ Frequently called □ 6 usages
191         public OfficialTestAchieveInfo NextOfficialTestAchieve { get; private set; }
192         □ Frequently called □ 6 usages
193         public NonOfficialTestVictoryAchieveInfo NextNonOfficialTestVictory { get; private set; }
194         □ Frequently called □ 6 usages
195         public NonOfficialTestDefeatAchieveInfo NextNonOfficialTestDefeat { get; private set; }
196         □ Frequently called □ 6 usages
197         public ReallocationAchieveInfo NextReallocationAchieve { get; private set; }
198         □ Frequently called □ 6 usages
199         public DevelopmentChimeraAchieveInfo NextDevelopmentChimera { get; private set; }
200
201         private float _totalAchieveInfo = 0;
202     }
```

1. 직급과 업적(코드)

```
public interface IAchieveClear
{
    □ 5 implementations
    public bool IsCleared { get; }
    □ 5 implementations
    public bool IsRewarded { get; }
    □ 5 implementations
    public void SetCleared();
    □ 5 implementations
    public void GetReward();
}
```

모든 업적 항목들은 IAchieveClear 인터페이스를 구현합니다. 이 인터페이스는 업적의 클리어 여부와 보상 획득 여부에 대한 구현을 강제합니다.

대부분의 게임들은 기본적인 플레이를 통해서 업적을 달성을 할 수 있습니다. 플레이어는 달성한 업적에 대한 보상이 있는 경우, 그 보상을 수령하지 않은 상태에서도 게임을 플레이할 수 있습니다.

이를 통해 업적 시스템의 업적 달성과 보상 획득과정은 구분되어 있음을 알 수 있고 달성과 획득에 대한 관리를 나누어 구현했습니다.

```
8 usages □ 1 exposing API
public class OfficialTestAchieveInfo : IAchieveClear{...}

8 usages □ 1 exposing API
public class NonOfficialTestVictoryAchieveInfo : IAchieveClear{...}

8 usages □ 1 exposing API
public class NonOfficialTestDefeatAchieveInfo : IAchieveClear{...}

8 usages □ 1 exposing API
public class ReallocationAchieveInfo : IAchieveClear{...}

8 usages □ 1 exposing API
public class DevelopmentChimeraAchieveInfo : IAchieveClear{...}
```

현재 게임에 구현되어 있는 업적에 대한 분류는 5개입니다.

1. 공식 테스트의 클리어 수준.
2. 비공식 테스트의 승리 회수.
3. 비공식 테스트의 패배 회수.
4. 키메라 증여 회수.
5. 키메라 발생 회수.

이 업적들은 플레이어가 특정한 활동을 만족하면 갱신됩니다.

1. 직급과 업적(코드)

```
public class OfficialTestAchieveInfo : IAchieveClear
{
    ⚡ Frequently called [ 5 usages
    public int NeedClearStageIndex { get; }

    ⚡ Frequently called [ 3 usages
    public bool IsCleared { get; private set; }

    ⚡ Frequently called [ 4 usages
    public bool IsRewarded { get; private set; }

    ⚡ Frequently called [ 2 usages
    public OfficialTestAchieveInfo(int stage)
    {
        NeedClearStageIndex = stage + 1;
        IsCleared = false;
        IsRewarded = false;
    }

    ⚡ Frequently called [ 2 usages
    public void SetCleared()
    {
        if (IsCleared) return;
        IsCleared = true;
    }

    [ 1 usage
    public void GetReward()
    {
        if (IsRewarded) return;
        GameImmortalManager.Instance.AddAccountGene(new Gene((int)AchieveManager.Instance.PlayerRank));
        IsRewarded = true;
    }
}
```

업적 항목 중 공식테스트 최종 클리어 수준에 해당하는 코드입니다.
업적을 생성할 때, 인자로 정수 값을 필요로 하며 이 정수 값은 플레이어가
클리어 해야 할 단계를 의미합니다. 다른 모든 업적 항목들은 이와 같은 방
식으로 생성되며, 인자로 받는 정수 값을 여러 방식으로 계산하여 플레이어
가 달성해야 할 수치를 정의합니다.

모든 업적에 대해서, 한 번 클리어 된 업적은 다시 클리어할 수 없고 보상도
중복수령이 불가능합니다.
보상은 플레이어의 직급에 영향을 받는 무작위 GeneType의 Gene입니다.

1. 직급과 업적(코드)

```
public class AchieveManager : ImmortalObject<AchieveManager>, IDataReset
{
    private const int AchieveCount = 10;

    ⚡ Frequently called 18 usages
    public ResearcherRank PlayerRank { get; private set; } = ResearcherRank.Junior;

    ⚡ Frequently called 9 usages
    public int HighestOfficialTestClearStage { get; private set; } = 0;
    ⚡ Frequently called 7 usages
    public int TotalNonOfficialTestVictoryCount { get; private set; } = 0;
    ⚡ Frequently called 7 usages
    public int TotalNonOfficialTestDefeatCount { get; private set; } = 0;
    ⚡ Frequently called 7 usages
    public int TotalReallocationCount { get; private set; } = 0;
    ⚡ Frequently called 7 usages
    public int TotalDevelopmentChimeraCount { get; private set; } = 0;

    private int _currentOfficialTestAchieveInfoKey;
    private int _currentNonOfficialTestVictoryAchieveInfoKey;
    private int _currentNonOfficialTestDefeatAchieveInfoKey;
    private int _currentReallocationAchieveInfoKey;
    private int _currentDevelopmentChimeraAchieveInfoKey;
```

상속되어 있는 부모클래스와 인터페이스는 뒤에서 다루겠습니다.
아카이브 매니저는 런 타임 동안 항상 존재합니다. 플레이어가 업적 정보를
갱신하면 그 정보를 추적하고 반영해야 합니다.

공식 테스트 업적을 제외한 나머지 업적들은 총 10개의 단계로 나뉩니다.
플레이어의 시작 직급은 주니어로 자동 설정됩니다.

플레이어가 게임을 진행하는 동안 플레이어가 연구에 기여한 모든 활동을
기록합니다.

1. 가장 높은 공식 테스트 클리어 수준
2. 비공식 테스트 총 승리 회 수
3. 비공식 테스트 총 패배 회 수
4. 키메라 총 증여 회 수
5. 키메라 총 발생 회 수

현재 달성한 업적 단계입니다. 이는 클리어한 업적 개수를 의미하며 추후
에 플레이어의 직급 상승 여부를 결정할 때 사용됩니다.

1. 직급과 업적(코드)

```
↳ Frequently called [ 10 usages
public Dictionary<int, OfficialTestAchieveInfo> OfficialTestAchieveInfos { get; private set; }

↳ Frequently called [ 10 usages
public Dictionary<int, NonOfficialTestVictoryAchieveInfo> NonOfficialTestVictoryAchieveInfos { get; private set; }

↳ Frequently called [ 10 usages
public Dictionary<int, NonOfficialTestDefeatAchieveInfo> NonOfficialTestDefeatAchieveInfos { get; private set; }

↳ Frequently called [ 10 usages
public Dictionary<int, ReallocationAchieveInfo> ReallocationAchieveInfos { get; private set; }

↳ Frequently called [ 10 usages
public Dictionary<int, DevelopmentChimeraAchieveInfo> DevelopmentChimeraAchieveInfos { get; private set; }

↳ Frequently called [ 6 usages
public OfficialTestAchieveInfo NextOfficialTestAchieve { get; private set; }

↳ Frequently called [ 6 usages
public NonOfficialTestVictoryAchieveInfo NextNonOfficialTestVictory { get; private set; }

↳ Frequently called [ 6 usages
public NonOfficialTestDefeatAchieveInfo NextNonOfficialTestDefeat { get; private set; }

↳ Frequently called [ 6 usages
public ReallocationAchieveInfo NextReallocationAchieve { get; private set; }

↳ Frequently called [ 6 usages
public DevelopmentChimeraAchieveInfo NextDevelopmentChimera { get; private set; }

private float _totalAchieveInfo = 0;
```

모든 업적은 딕셔너리로 관리되며 목표로 설정된 다음 업적 항목을 구분하여 할당합니다.

업적 관리를 노드를 활용하는 방식도 고려해보았습니다. 하지만 딕셔너리로도 충분히 구현이 가능했고 조금 더 쉬운 방법이라고 생각되어 딕셔너리로 구현했습니다.

UI를 통해 플레이어가 다음 목표 업적을 노출시킬 때, 딕셔너리에서 바로 값을 읽어 노출시킬 수 있지만, 가독성을 더 높이기 위해 따로 다음 업적 항목을 할당할 수 있는 변수를 선언하였습니다.

마지막 변수는 플레이어가 달성한 모든 업적을 종합한 점수입니다. 종합 점수로 플레이어의 직급이 상승될 수 있습니다.

1. 직급과 업적(코드)

```
230     public void SetOfficialTestAchieveInfo(int clearedStage)
231         //인자로 받는 int는 클리어한 스테이지. ex) 7단계 클리어시 clearedStage == 7;
232         if (clearedStage > HighestOfficialTestClearStage)
233         {
234             HighestOfficialTestClearStage = clearedStage;
235         }
236         else
237         {
238             return;
239         }
240
241
242         //인자로 받은 최종 클리어 스테이지가 다음 요구 클리어 스테이지보다 크거나 같다면
243         if (HighestOfficialTestClearStage >= NextOfficialTestAchieve.NeedClearStageIndex)
244         {
245             //다음 업적을 클리어하고
246             NextOfficialTestAchieve.SetCleared();
247             //키값을 증가시킨 후
248             int nextInfoKey = _currentOfficialTestAchieveInfoKey + 1;
249             if (OfficialTestAchieveInfos.TryGetValue(nextInfoKey, out OfficialTestAchieveInfo info))
250             {
251                 _currentOfficialTestAchieveInfoKey = nextInfoKey;
252                 NextOfficialTestAchieve = info;
253                 SetResult();
254             }
255             else
256             { if (_currentOfficialTestAchieveInfoKey == OfficialTestManager.TotalResearchers)
257             {
258                 return;
259             }
260             _currentOfficialTestAchieveInfoKey = OfficialTestManager.TotalResearchers;
261             SetResult();
262             return;
263         }
264         //그 다음 업적을 실행하고
265         //다시비교한다
266
267         // 만약 새로 출증한 단계의 요구 클리어 스테이지보다 인자로 받은 최종 클리어 스테이지가 여전히 크거나 같다면
268         if (HighestOfficialTestClearStage > NextOfficialTestAchieve.NeedClearStageIndex)
269         {
270             //재설정한다.
271             SetOfficialTestAchieveInfo(HighestOfficialTestClearStage);
272             SetResult();
273             return;
274         }
275     }
276     /* foreach (var pair in OfficialTestAchieveInfos) ... */
```

공식 테스트 수준 업적에 관련된 수치를 갱신할 때 실행되는 코드입니다. 공식 테스트는 현재 플레이어의 최종 기록을 갱신해야 실행되며 그 여부를 먼저 판단합니다(다른 업적들은 회수를 추가하기 때문에 해당 로직이 생략되어 있습니다).

인자로 받은 최종 클리어 수준이 현재의 수준보다 높은 경우 아래의 코드가 실행됩니다.

새로 전달받은 최종 클리어 수준이 현재 목표로 설정된 업적의 요구치를 상회한다면, 현재 업적을 클리어 처리한 뒤, int형 지역 변수를 선언하고 현재 단계에 1을 더한 값을 할당하여 다음 업적 단계가 있는지 딕셔너리를 통해 확인합니다. 다음 업적이 있다면 지역 변수의 값을 현재 목표로 설정할 업적의 단계로 할당하고 다음 업적을 현재 목표 업적으로 설정합니다. 이 후 플레이어의 직급 상승 여부를 결정합니다(SetResult()).

만약 다음 업적이 없다면 모든 업적을 완료한 것이므로 현재 업적 단계를 최대 값(상호 테스트할 수 있는 총 연구원 수)으로 설정할 지 그 여부를 결정합니다. 이미 그 값이라면 함수를 종료하고, 아니라면 해당 값을 설정한 뒤 플레이어의 직급 상승 여부를 결정하고 함수를 종료합니다(다른 업적들의 최대 개수는 10개입니다).

공식 테스트는 한 번에 여러 단계를 건너뛸 수 있기 때문에 현재 목표 업적이 변경된 뒤에도 인자로 받아 최종 클리어 수준으로 할당한 값이 현재 목표 업적이 요구하는 수치보다 크다면 이 함수를 재귀적으로 호출합니다.

이 로직은 다른 업적들의 수치 갱신 시 실행되는 코드에서 동일하게 사용됩니다.

1. 직급과 업적(코드)

```
⌚ Frequently called 17 usages
private void SetResult()
{
    _totalAchieveInfo = _currentReallocationAchieveInfoKey + _currentDevelopmentChimeraAchieveInfoKey +
        _currentOfficialTestAchieveInfoKey + _currentNonOfficialTestDefeatAchieveInfoKey +
        _currentNonOfficialTestVictoryAchieveInfoKey;

    int info = (int)(_totalAchieveInfo / 10);
    switch (info)
    {
        case 0:
        case 1:
            PlayerRank = ResearcherRank.Junior;//19개까지 주니어
            break;
        case 2:
            PlayerRank = ResearcherRank.Senior;//29까지 시니어
            break;
        case 3:
            PlayerRank = ResearcherRank.Principal;//39까지 프린시펄
            break;
        case 4:
            PlayerRank = ResearcherRank.Director;//40개부터 디렉터
            break;
        default:
            PlayerRank = ResearcherRank.Director;
            break;
    }
}
```

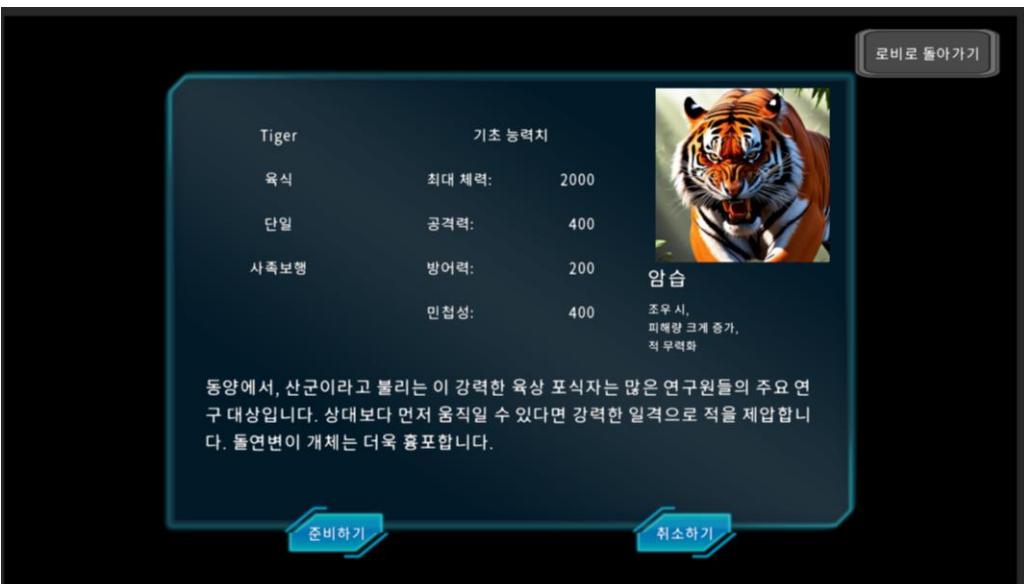
플레이어의 직급을 상승시킬 지, 그 여부를 결정하는 함수입니다.
현재까지 플레이어가 달성한 업적의 개 수를 모두 더한 뒤 그 값을 10으로 나눈
값을 기준으로 합니다. 플레이어는 19개의 업적을 달성할 때까지 Junior직급으로
고정되며 20개를 달성한 시점에서 Senior가 됩니다.

이 후 10개를 달성할 때마다 직급이 상승됩니다. 이를 통해 플레이어는 자신의
직급이 상승되는 시점과 남은 업적의 개수를 토대로 다음 직급 달성까지의 요구
업적 개 수를 추측할 수도 있습니다.

2. 키메라

직급 상승을 위해 요구되는 연구 성과들은 모두 키메라가 연관되어 있습니다. 키메라를 발생하고, 증여 해야 합니다. 또는 키메라를 통해 공식, 비공식 테스트를 진행하여 플레이어가 발생한 키메라가 얼마나 강력한지, 다른 NPC연구원들에게 증명해야 합니다. 그리고 그 증명과정에서, 플레이어는 변종 키메라의 발생권한을 획득함으로써 더 강한 키메라를 발생할 수 있는 기회를 얻을 수 있습니다.

1. 발생할 키메라 선택 시 화면



2. 변종(좌측)과 원 종의 능력치 차이



2. 키메라(전체코드 중 변수)

```
public class Chimera : MonoBehaviour
{
    [SerializeField] private bool isMutant; * "true"
* 11 usages
    public bool IsMutant => isMutant;
    [SerializeField] private GeneType geneType; * Changed in 12 assets
* 44 usages
    public GeneType GeneType => geneType;
* 1 usage
    public MainDna MainDna { get; private set; }
* Frequently called 5 usages
    public DnaMainSkill MainSkill { get; private set; }
* Frequently called 16 usages
    public List<DnaSubSkill> SubSkills { get; private set; }
* Frequently called 2 usages
    public float MaxHealthPoint { get; private set; }
* Frequently called 9 usages
    public float CurrentHealthPoint { get; private set; }
* Frequently called 17 usages
    public float AttackPoint { get; private set; }
* Frequently called 8 usages
    public float DefencePoint { get; private set; }
* Frequently called 15 usages
    public float AgilityPoint { get; private set; }

* Frequently called 7 usages
    public IChimeraState CurrentState { get; private set; }
* 1 usage
    public IChimeraState LastState { get; private set; }
* Frequently called 9 usages
    public StandingState StandingState { get; private set; }
* Frequently called 5 usages
    public BasicAttackState BasicAttackState { get; private set; }
* 6 usages
    public FirstSkillState FirstSkillState { get; private set; }
* 4 usages
    public SecondSkillState SecondSkillState { get; private set; }
* 2 usages
    public ThirdSkillState ThirdSkillState { get; private set; }
* 2 usages
    public MainSkillState MainSkillState { get; private set; }
```

이 키메라가 변종인지 확인합니다.

이 키메라의 유전자 종류를 의미합니다.

이 키메라의 MainDna를 의미합니다.

이 키메라의 MainSkill을 의미합니다.

이 키메라의 SubSkill들을 의미합니다.

이 키메라의 능력치인 최대체력, 현재체력, 공격력, 방어력, 민첩성을 의미합니다.

이 스킬과 능력치, 아래의 상태들은 이후 키메라의 전투에서 활용됩니다(링크).

상태패턴을 통해 키메라의 상태를 관리하고 추적합니다. 포켓몬스터의 배틀처럼 키메라들이 애니메이션을 통해 움직이고, 역동적인 카메라 무빙을 활용할 생각으로 상태패턴을 구현하였습니다.
하지만 제가 원하는 에셋들을 구할 수 없었고, 어울리지 않는 애니메이션을 보여주는 것은 원치 않았기 때문에 각 상태의 내부는 구현되지 않았습니다.

```
* Frequently called 3 usages
public GroggyState GroggyState { get; private set; }

* Frequently called 12 usages
public VulnerableState VulnerableState { get; private set; }

* Frequently called 3 usages
public DodgeState DodgeState { get; private set; }

* 1 usage
public SurvivalState SurvivalState { get; private set; }

* Frequently called 2 usages
public DeathState DeathState { get; private set; }

* Frequently called 4 usages
public SpawnAndWaitState SpawnAndWaitState { get; private set; }

* Frequently called 16 usages
public Dictionary<IDnaSkill, IChimeraState> SkillStates { get; private set; }

public bool IsPoisoned; * Unchanged

private float _decreasedAttackPoint;
private float _decreasedDefencePoint;
private float _decreasedAgilityPoint;
```

위 이미지 내의 모든 변수들은 키메라의 전투에서 활용됩니다.

SkillStates는 스킬을 Key로 스테이트를 가져올 수 있습니다.

특정 키메라는 전투 간 적 키메라의 능력치를 감소시킬 수 있습니다. 이를 제어하기 위해 4개의 변수를 선언하여 상태 이상 여부와 감소된 능력치를 따로 저장합니다.

2. 키메라(전체코드 중 변수)

```
public void Initialize(ChimeraData chimeraData)
{
    MainDna = chimeraData.MainDna;
    MainSkill = chimeraData.MainSkill;
    SubSkills = chimeraData.SubSkills;

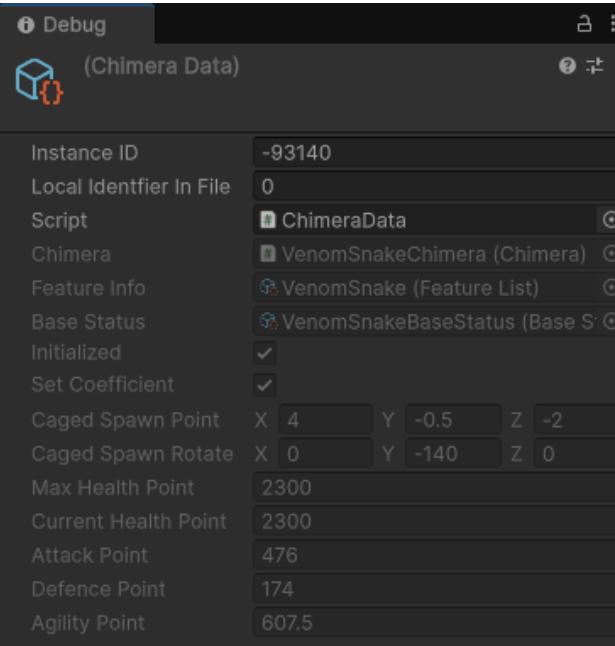
    MaxHealthPoint = chimeraData.MaxHealthPoint;
    CurrentHealthPoint = chimeraData.CurrentHealthPoint;
    AttackPoint = chimeraData.AttackPoint;
    DefencePoint = chimeraData.DefencePoint;
    AgilityPoint = chimeraData.AgilityPoint;

    SpawnAndWaitState = new SpawnAndWaitState( chimera: this );
    StandingState = new StandingState( chimera: this );
    BasicAttackState = new BasicAttackState( chimera: this );
    SkillStates = new Dictionary<IDnaSkill, IChimeraState>();
    switch (SubSkills.Count)
    {
        case 0:
            break;
        case 1:...
        case 2:
            FirstSkillState = new FirstSkillState( chimera: this, SubSkills[0] );
            SkillStates[SubSkills[0]] = FirstSkillState;
            SecondSkillState = new SecondSkillState( chimera: this, SubSkills[1] );
            SkillStates[SubSkills[1]] = SecondSkillState;
            break;
        case 3:
            FirstSkillState = new FirstSkillState( chimera: this, SubSkills[0] );
            SkillStates[SubSkills[0]] = FirstSkillState;
            SecondSkillState = new SecondSkillState( chimera: this, SubSkills[1] );
            SkillStates[SubSkills[1]] = SecondSkillState;
            ThirdSkillState = new ThirdSkillState( chimera: this, SubSkills[2] );
            SkillStates[SubSkills[2]] = ThirdSkillState;
            break;
    }

    MainSkillState = new MainSkillState( chimera: this );
    SkillStates[MainSkill] = MainSkillState;

    GroggyState = new GroggyState( chimera: this );
    VulnerableState = new VulnerableState( chimera: this );
    DodgeState = new DodgeState( chimera: this );
    SurvivalState = new SurvivalState( chimera: this );

    DeathState = new DeathState( chimera: this );
}
```



키메라 오브젝트는 상단의 스크립터블 오브젝트(이하 키메라 데이터)로 변환되어 관리됩니다.

좌측의 함수는 키메라 오브젝트 내에 있는 함수로 키메라 데이터에 저장된 키메라의 정보를 읽어서 이전 장의 키메라가 사용해야 할 변수들을 할당해줍니다.

키메라는 최대 3개의 SubSkill을 가질 수 있습니다.

```
#region OnCombat
    # Frequently called [ 11 usages
    public void TakeDamage(float damage, float skillCoefficient = 0){...}

    # Frequently called [ 1 usage
    public void Refresh(){...}

    # Frequently called [ 1 usage
    public void SetPoisoned(float strength){...}

    # Frequently called [ 1 usage
    private void ResetPoisoned(){...}

    # Frequently called [ 2 usages
    public void SetAgility(float agility){...}
#endregion
```

위 이미지 내의 모든 함수들은 키메라 전투에서 사용되는 함수들입니다.

2. 키메라(키메라 데이터 스크립터블 오브젝트 전체 코드 중 변수)

```
public class ChimeraData : ScriptableObject
{
    // Frequently called [8 usages]
    public Chimera Chimera { get; private set; }

    // Frequently called [1 usage]
    public FeatureList FeatureInfo { get; private set; }

    // Frequently called [14 usages]
    public BaseStatus BaseStatus { get; private set; }

    private bool _initialized = false;
    private bool _setCoefficient = false;

    private Vector3 _cagedSpawnPoint = new (4f, -0.5f, -2f);
    private Vector3 _cagedSpawnRotate = new (0f, -140f, 0f);

    // Frequently called [64 usages]
    public MainDna MainDna { get; private set; }

    // Frequently called [2 usages]
    public DnaMainSkill MainSkill { get; private set; }

    // Frequently called [2 usages]
    public List<DnaSubSkill> SubSkills { get; private set; }

    // Frequently called [9 usages]
    public float MaxHealthPoint { get; private set; }

    // Frequently called [3 usages]
    public float CurrentHealthPoint { get; private set; }

    // Frequently called [7 usages]
    public float AttackPoint { get; private set; }

    // Frequently called [7 usages]
    public float DefencePoint { get; private set; }

    // Frequently called [7 usages]
    public float AgilityPoint { get; private set; }
```

키메라 데이터는 키메라의 주요 정보들을 저장합니다.

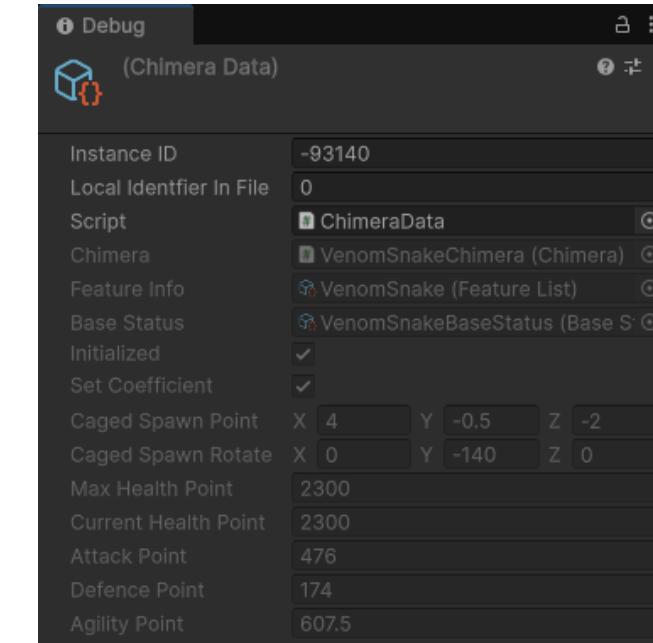
어떤 키메라인지, 이 키메라의 특성, 기초 능력치를 저장합니다.

키메라 정보의 설정 여부와 능력치 계수의 적용 여부를 확인합니다.

특정 씬에서 이 키메라가 어떤 위치, 회전으로 인스턴스화 될 지 결정합니다.

이 키메라의 MainDna와 스킬 정보들을 저장합니다.

이 키메라의 모든 능력치를 할당합니다. 키메라는 이 정보(우측 이미지)들을 바탕으로 인스턴스화됩니다.



2. 키메라(키메라 데이터 스크립터블 오브젝트 전체 코드 중 함수)

```
public void SetChimeraData(Chimera chimera, FeatureList featureInfo, BaseStatus baseStatus)
{
    if (!initialized)
    {
        if (chimera == null) return;
        if (featureInfo == null) return;
        if (baseStatus == null) return;
        Chimera = chimera;
        FeatureInfo = featureInfo;
        BaseStatus = baseStatus;

        MainDna = new MainDna(chimera.GeneType);
        MainSkill = MainDna.DnaMainSkill;
        SubSkills = MainDna.DnaSubSkills;

        MaxHealthPoint = BaseStatus.MaxHealthPoint;
        CurrentHealthPoint = MaxHealthPoint;
        AttackPoint = BaseStatus.AttackPoint;
        DefencePoint = BaseStatus.DefencePoint;
        AgilityPoint = BaseStatus.AgilityPoint;

        _initialized = true;
    }

    ⚡ Frequently called 3 usages
    public void SetCoefficientToStatus()
    {
        if (!_setCoefficient)
        {
            MaxHealthPoint += MaxHealthPoint * MainDna.TotalHealthCoefficient/100;
            CurrentHealthPoint = MaxHealthPoint;
            AttackPoint += AttackPoint * MainDna.TotalAttackCoefficient/100;
            DefencePoint += DefencePoint * MainDna.TotalDefenceCoefficient/100;
            AgilityPoint += AgilityPoint * MainDna.TotalAgilityCoefficient/100;
            _setCoefficient = true;
        }
    }

    ⚡ 1 usage
    public Chimera CagedInstantiateChimera(){...}
    ⚡ 4 usages
    public Chimera InstantiateChimera(Vector3 spawnPoint = default){...}
}
```

키메라 데이터는 최초에 키메라의 데이터를 할당받을 때, 한 번만 할당받습니다. 알 수 없는 버그로 재할당되는 것을 막으려는 의도입니다.

마찬가지로 능력치 계수를 적용하여 최종 능력치를 계산할 때도 동일한 의도입니다.

키메라의 발생 시, 최초에 키메라의 기초 데이터를 할당받으며 SetChimeraData 함수가 실행됩니다.

이 후 유전자 삽입 과정이 종료되어 발생을 종료할 때, SetCoefficientToStatus 함수가 실행되어 최종적인 능력치를 설정합니다.

우측의 첫 번째 이미지는 이 키메라의 기초 능력치이며, 두 번째 이미지는 유전자를 삽입함으로써 증가한 능력치입니다.

마지막 두 함수는 특정 씬에서 키메라가 어떤 위치에서, 어떤 회전 값으로 인스턴스화 될 지 결정해주는 함수입니다.



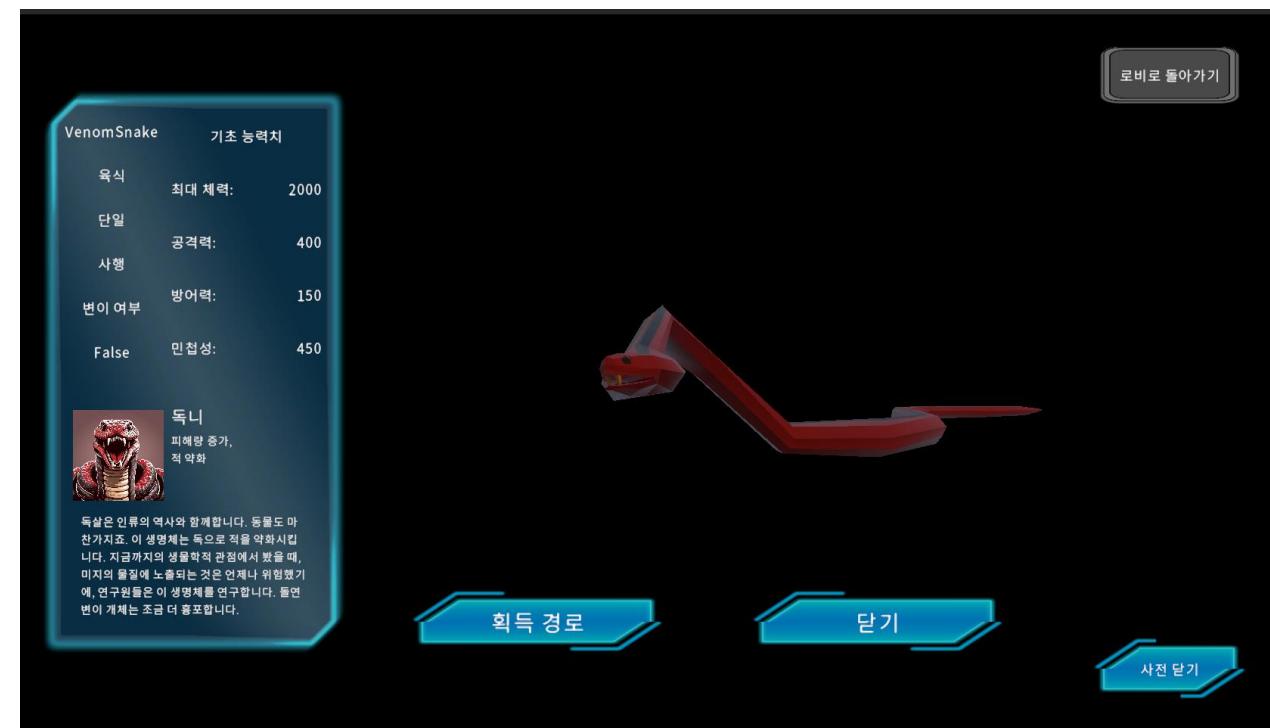
2. 키메라(Dna)

모든 키메라들은 고유의 Dna를 가지고 있으며, 이 Dna는 키메라의 외형, MainSkill, 기본 능력치를 결정합니다. 플레이어는 발생하려는 키메라의 정보를 읽고 원하는 키메라를 선택하여 발생할 수 있습니다.

1. 구현된 키메라(유전자/키메라 사전 - 키메라)



2. 1에서 VenomSnake 클릭 시



2. 키메라(Dna 전체 코드 중 변수)

```
public class MainDna
{
    ⚡ Frequently called [7 usages]
    public GeneType GeneType { get; private set; }

    ⚡ Frequently called [19 usages]
    public DnaMainSkill DnaMainSkill { get; private set; }

    ⚡ Frequently called [13 usages] More...
    public FeatureList DnaFeatureList { get; private set; }

    ⚡ Frequently called [2 usages]
    public static int MaxGeneCapacity { get; } = 20;

    ⚡ Frequently called [12 usages]
    public List<Gene> GeneList { get; private set; }

    ⚡ Frequently called [7 usages]
    public Sprite Sprite { get; private set; }

    ⚡ Frequently called [14 usages]
    public List<DnaSubSkill> DnaSubSkills { get; private set; }

    ⚡ Frequently called [2 usages]
    public static int MaxSubSkillsCount { get; } = 3;

    ⚡ Frequently called [6 usages]
    public int TotalHealthCoefficient { get; private set; } = 0;
    ⚡ Frequently called [6 usages]
    public int TotalAttackCoefficient { get; private set; } = 0;
    ⚡ Frequently called [6 usages]
    public int TotalDefenceCoefficient { get; private set; } = 0;
    ⚡ Frequently called [6 usages]
    public int TotalAgilityCoefficient { get; private set; } = 0;

    ⚡ Frequently called [9 usages]
    public Dictionary<Gene, int> FeatureCoefficientByGene { get; private set; }

    ⚡ Frequently called [8 usages]
    public string MainDnaDescription { get; private set; }
```

MainDna에는 유전자 종류와 그에 따른 MainSkill, 특성(Featurlist)를 갖습니다.

최대 Gene 수용량은 20으로, 이는 Gene 최대 20개 수용할 수 있음을 의미합니다.

삽입된 Gene GeneList에 할당됩니다.

키메라의 스킬 자체는 MainDna에 저장됩니다. 이는 생물학의 '전사(외부 링크)'라는 현상에서 차안했습니다.

최대 SubSkill 개 수는 3개로 제한하고, 모든 능력치의 계수도 MainDna에 저장됩니다.

FeatureCoefficientByGene 딕셔너리는 MainDna에 삽입된 Gene간의 특성 유사도를 검사하여 그 수치에 따라 능력치 계수를 추가합니다. 각 Gene마다 유사도가 다를 가능성이 매우 높아 어떤 Gene이 어느 정도의 유사도를 가졌는지 출력하기 위해 딕셔너리를 사용했습니다.

MainDna의 설명입니다. Feature에 저장되어 있는 각 키메라의 설명을 할당합니다.

2. 키메라(Dna 전체 코드 중 함수)

```
public MainDna(GeneType gene)
{
    GeneType = gene;
    GeneList = new List<Gene>();
    FeatureCoefficientByGene = new Dictionary<Gene, int>();
    Sprite = Resources.Load<Sprite>(path: $"MainSkillsImage/{GeneType.ToString()}");

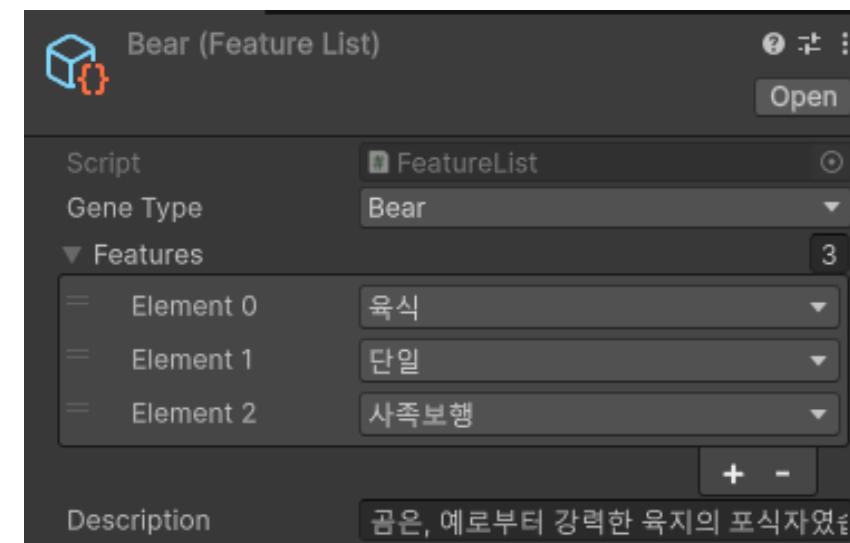
    //유사 팩토리
    switch (gene)
    {
        case GeneType.Wolf:...
        case GeneType.Bear:...
        case GeneType.VenomSnake:...
        case GeneType.Buffalo:...
        case GeneType.Tiger:...
        case GeneType.Sheep:
            DnaMainSkill = new SheepOverExSkill();
            DnaFeatureList = ImmortalScriptableObjectManager.Instance.FeaturesByGeneType[gene];
            MainDnaDescription = ImmortalScriptableObjectManager.Instance.FeaturesByGeneType[gene].Description;
            break;
    }

    DnaSubSkills = new List<DnaSubSkill>(MaxSubSkillsCount);
}
```

MainDna의 생성자입니다.
인자로 받는 GeneType에 따라 데이터들을 할당합니다. FeatureList는 스크립터를 오브젝트 형태로 데이터를 관리하는 매니저에게서 정보를 받아 할당합니다.

DnaSubSkills 리스트는 최대 값을 초기 용량으로 하여 할당합니다.

아래 이미지는 FeatureList 스크립터를 오브젝트에 저장된 데이터 예시입니다.



2. 키메라(Dna 전체 코드 중 함수)

```
//유전자 삽입가능 여부 확인하기
↳ Frequently called [ 6 usages
public bool TryInsertGene(Gene gene)
{
    //현재 수용량을 계산하여 해당 지역변수 선언
    int currentGeneCapacity = 0;
    //유전자 리스트에 1개라도 있으면
    if (GeneList.Count > 0)
    {
        //각각의 유전자에 대해서
        foreach (Gene g in GeneList)
        {
            //안정화정도를 현재 수용량에 더한다.
            currentGeneCapacity += g.StabilizationDegree;
        }

        //현재 수용량이 최대 수용량보다 작으면 == 전달받은 유전자를 넣을 가능성이 있다는 뜻
        if (currentGeneCapacity < MaxGeneCapacity)
        {
            //넓으려는 유전자의 안정화정도를 가지고온다.
            int geneSize = gene.StabilizationDegree;

            //새로운 수용량 지역변수를 선언하고 그 변수에 현재수용량과 넓으려는 유전자의 안정화정도를 더한다.
            int newCapacity = currentGeneCapacity + geneSize;
            //새로운 수용량이 최대 수용량보다 작거나, 같으면
            if (newCapacity <= MaxGeneCapacity)
            {
                return true;
            }
            //새로운 수용량이 최대 수용량보다 크면 리턴한다 == 들려줘야되는데?
            return false;
        }
        //현재 수용량이 최대 수용량과 같거나 그보다 크면
        return false;
    }
    //유전자 리스트에 아무것도 없으면 그냥 넣는다.
    return true;
}
```

MainDna에 Gene을 삽입하기 조건이 필요합니다.
이 Gene을 삽입해도 최대 수용량을 초과하지 않는지
확인해야합니다.

현재 수용량을 확인하고, 그 수용량에 삽입될 Gene의
안정도(요구 수용량)를 더해봅니다. 그 값에 따라 이
Gene을 삽입할 지 하지 않을지 결정합니다.

2. 키메라(Dna 전체 코드 중 함수)

```
public void InsertGene(Gene gene)
{
    GeneList.Add(gene);
    SetTotalCoefficient(gene);
    SetSubSkill(gene);
}

↳ Frequently called [2] 1 usage
private void SetTotalCoefficient(Gene gene)
{
    SetGeneCoefficient(gene);
    SetFeatureCoefficient(gene);
    TotalHealthCoefficient += FeatureCoefficientByGene[gene];
    TotalAttackCoefficient += FeatureCoefficientByGene[gene];
    TotalDefenceCoefficient += FeatureCoefficientByGene[gene];
    TotalAgilityCoefficient += FeatureCoefficientByGene[gene];
}

↳ Frequently called [2] 1 usage
private void SetGeneCoefficient(Gene gene, bool add = true)
{
    if (add)
    {
        if (!gene.IsActivation)
        {
            int hpCoefficient = gene.RandomStatusCoefficient[0];
            TotalHealthCoefficient += hpCoefficient;

            int atkCoefficient = gene.RandomStatusCoefficient[1];
            TotalAttackCoefficient += atkCoefficient;

            int defCoefficient = gene.RandomStatusCoefficient[2];
            TotalDefenceCoefficient += defCoefficient;

            int agilityCoefficient = gene.RandomStatusCoefficient[3];
            TotalAgilityCoefficient += agilityCoefficient;

            gene.SetActivationTrue();
        }
        else
        {
            if (gene.IsActivation){...}
        }
    }
}
```

Gene의 삽입이 확정되면 해당 Gene을 리스트에 추가한 뒤, 가지고 있는 능력치 계수를 적용합니다. MainDna의 GeneType에 맞는 FeatureList와 비교하여 그 항목을 검사하고 일치하는 항목 당 1씩 증가 시켜 최종적으로 유사도 계수를 설정합니다.

```
//20250128 - 유전자 개수가 늘어나면 피쳐코이피센트가 유전자 총개수 -1만큼 더 적용되는 현상. -> 가장 앞에 있는 것(0)이 활용됨
//SetFeatureCoefficient에서 TotalCoefficient에 한번 적용하고, SetTotalCoefficient에서 TotalCoefficient를 한번더 적용하면서 버그발생
//타인 유사도에 따른 추가 계수
↳ Frequently called [2] 1 usage
private void SetFeatureCoefficient(Gene gene, bool add = true)
{
    //추가여부?
    if (add)
    {
        //현재 피쳐리스트 가져오기
        FeatureList featureList = ImmortalScriptableObjectManager.Instance.FeaturesByGeneType[gene.GeneType];
        //유전자를 키로 사용량 추가하기
        FeatureCoefficientByGene.Add(gene, 0);
        //수용된 유전자와 피쳐리스트의 각 피쳐에 대해서
        foreach (var feature in featureList.Features)
        {
            //MainDna의 피쳐리스트를 가져온다.
            for (int i = 0; i < DnaFeatureList.Features.Length; i++)
            {
                //그 피쳐리스트의 각각의 피쳐와 비교한다.
                if (feature.Equals(DnaFeatureList.Features[i]))
                {
                    //일치하는게 있다면 값을 1 증가시킨다.
                    FeatureCoefficientByGene[gene]++;
                }
            }
        }
        else
        {
            FeatureCoefficientByGene.Remove(gene);
        }
    }
    //수용된 유전자 하니어 대한 피쳐리스트 가져오기
}
```

2. 키메라(Dna 전체 코드 중 함수)

```
//서브스킬 활성화 여부 확인하기
⌚ Frequently called 3 usages
public bool TrySetSubSkill(Gene gene)
{
    //공간이 남아있고
    if (DnaSubSkills.Count < MaxSubSkillsCount)
    {
        foreach (var subSkill in DnaSubSkills)
        {
            //같은 GeneType이 하나라도 있으면
            if (subSkill.GeneType.Equals(gene.GeneType))
            {
                //Debug.Log("Failed to set sub skill: Same gene type");
                return false;//종료
            }
        }
        //같은 GeneType이 하나도 없으면
        return true;
    }

    //Debug.Log("Failed to set sub skill: list is Full");
    return false;
}

⌚ Frequently called 1 usage
private void SetSubSkill(Gene gene)
{
    if (TrySetSubSkill(gene))
    {
        DnaSubSkills.Add(gene.DnaSkill);
        //Debug.Log(gene.GeneType);
    }
}
```

Gene의 삽입이 확정되면 해당 Gene의 SubSkill의 활성화 여부를 결정합니다. 이때 몇 가지 조건이 필요합니다.

1. SubSkill을 추가할 수 있는지?
2. 이미 같은 SubSkill이 추가되어 있는지?

SubSkill의 활성화는 가장 처음 들어가는, 서로 다른 GeneType을 기준으로 합니다. 같은 GeneType을 연속으로 삽입해도 해당 GeneType의 SubSkill은 한 개만 적용됩니다.

추가적인 업데이트 사항으로 같은 GeneType이 연속해서 들어간 경우, 그 GeneType의 스킬을 강화하는 것입니다. 현재 구상중인 일부 업데이트 사항에 포함되어 있습니다.

SetSubSkill함수에서 실질적으로 SubSkill의 활성화가 이루어지는 이유는 Ui로 플레이어에게 이 스킬의 삽입 여부를 알려주어야 했기 때문입니다.

키메라 발생 시, 이미 값을 저장해놓는 방법도 생각해 봤지만 불필요한 변수 선언이라고 생각했습니다. 이 부분은 최적화와 관련하여 공부가 필요한 내용입니다.

2. 키메라(gene)

더 강력한 키메라를 획득하는 방법은 주로 Gene의 삽입입니다. 플레이어는 Gene이 갖는 능력치 계수의 크기 뿐만 아니라 그 Gene과 발생하려는 키메라의 Dna의 GeneType이 유사한지, 그 여부도 따져야 합니다. 게다가 키메라가 사용할 스킬이 이 키메라에게 얼마나 도움이 될지도 고려해야 합니다. 키메라는 전투가 지속되는 동안, 자신의 생존을 위해 가용한 모든 자원을 사용할 것입니다.

1. 구현된 유전자(유전자/키메라 사전 – 유전자 – VenomSnake 클릭 시)



2. 이 키메라는 충각스킬을 절대 사용할 수 없습니다.



2. 키메라(gene 전체 코드 중 변수)

```
public enum GeneGrade
{
    None = 0,
    Crude,
    Processed,
    Advanced,
    Pure
}

✉ 83 usages ⚡ 25 exposing APIs
public enum GeneType
{
    Wolf,
    Bear,
    VenomSnake,
    Buffalo,
    Tiger,
    Sheep,
    Horse,
    Elephant,
    Rhino,
    Dragon = 999
}
```

```
public class Gene
{
    ⚡ Frequently called ✉ 26 usages
    public GeneType GeneType { get; private set; }

    ⚡ Frequently called ✉ 16 usages
    public DnaSubSkill DnaSubSkill { get; private set; }

    ⚡ Frequently called ✉ 8 usages
    public Sprite Sprite { get; private set; }

    ⚡ Frequently called ✉ 13 usages
    public GeneGrade GeneGrade { get; private set; }

    /// <summary>
    /// 0 == Health, 1 == Attack, 2 == Defense, 3 == Speed
    /// </summary>
    ⚡ Frequently called ✉ 40 usages
    public List<int> RandomStatusCoefficient { get; private set; } = new( capacity: 4 );

    ⚡ Frequently called ✉ 9 usages
    public int StabilizationDegree { get; private set; }

    ⚡ Frequently called ✉ 7 usages
    public int GeneID { get; private set; }

    ⚡ Frequently called ✉ 12 usages
    public static int DataId { get; private set; } = 0;

    ⚡ Frequently called ✉ 7 usages
    public bool IsActivation { get; private set; } = false;
    private static List<Gene> geneList = new( capacity: 500 );
}
```

첫 번째 이미지는 Gene의 등급과 구현된 종류입니다. 마지막 Dragon은 추후 업데이트 사항으로 모든 공식 테스트를 통과했을 때, 보스로 등장시킬 예정입니다.

이 Gene이 갖는 정보는 Dna와 비슷하지만 조금 다릅니다.

최초에 게임의 세이브와 로드를 지원하려는 생각으로, 생성과 삭제가 잣은 Gene을 효율적으로 관리할 필요가 있어 각 Gene에 고유한 ID를 부여했습니다.

Gene의 삽입과 추출에 대한 시스템을 고려했기 때문에 이 Gene이 삽입되어 활성화 중인지 그 여부도 검사해야했습니다.

마지막 _geneList는 사용되지 않는 Gene을 수거하여 저장합니다. 이는 플레이어가 게임을 종료하지 않고 여러 번 플레이를 진행할 때 자동으로 삭제됩니다.

2. 키메라(gene 전체 코드 중 함수)

```
public Gene(int playerResearcherRank = 0)
{
    //DragonType은 나오지 않음
    int randomIndex = Random.Range(0, Enum.GetValues(typeof(GeneType)).Length-1);
    GeneType = (GeneType)randomIndex;

    SetSubSkill(GeneType);

    RandomStatusCoefficient.Add(0);
    RandomStatusCoefficient.Add(0);

    //Gene 등급 추가
    var gradeSelector = 0;

    switch (playerResearcherRank)
    {
        case 0:
            // 나머지 두 개의 랜덤 값 추가 (4~10 사이)
            for (int i = 0; i < 2; i++)
            {
                int randomCoefficient = Random.Range(4, 11);
                RandomStatusCoefficient.Add(randomCoefficient);
                gradeSelector += randomCoefficient;
            }
            break;
        default:
            for (int i = 0; i < 2; i++)
            {
                int randomCoefficient = Random.Range(4+playerResearcherRank, 11);
                RandomStatusCoefficient.Add(randomCoefficient);
                gradeSelector += randomCoefficient;
            }
            break;
    }

    GeneGrade = (GeneGrade)((gradeSelector/4)-1);
    StabilizationDegree = Enum.GetValues(typeof(GeneGrade)).Length - (int)GeneGrade;

    // 리스트 섞기 (0의 위치를 무작위로)
    ShuffleList(RandomStatusCoefficient);
    GeneID = DataId;
    //JSONWriter.Instance.SaveGeneData(this);
    //GameImmortalManager.Instance.AddUserGene(this);
    Sprite = Resources.Load<Sprite>(path: $"GeneImage/{GeneType.ToString()}");
    DataId++;
}
```

Gene의 주요 생성자입니다. 플레이어의 직급에 Gene의 등급이 영향을 받게 함으로써 플레이어는 직급을 올릴 필요성을 느끼게 됩니다.

Dragon을 제외한 나머지 종류들 중 하나를 임의로 가져와 시작합니다. SubSkill을 할당하고 임의의 능력치 계수를 2개 추가합니다. 이 때 플레이어의 직급이 능력치를 결정하는 주요 변수로 적용됩니다.

Gene의 등급과 안정도(요구 수용량)을 설정하고 능력치 계수를 섞습니다(Fisher-Yates 알고리즘, 우측 상단 이미지).

SetSubSkill함수는 Switch문을 사용하여 GeneType에 맞는 SubSkill을 생성하도록 했습니다.

NPC들은 좌측과 같은 로직을 사용하여 그들의 키메라를 발생합니다. 단, 그들의 직급에 따라 키메라에 삽입되는 Gene의 총 양이 결정됩니다(우측 하단 이미지).

이 때문에 플레이어는 높은 직급의 NPC를 상대하기 위해 강력한 키메라를 발생해야 합니다.

```
private void ShuffleList(List<int> list)
{
    for (int i = list.Count - 1; i > 0; i--)
    {
        int randomIndex = Random.Range(0, i + 1);
        (list[i], list[randomIndex]) = (list[randomIndex], list[i]);
    }

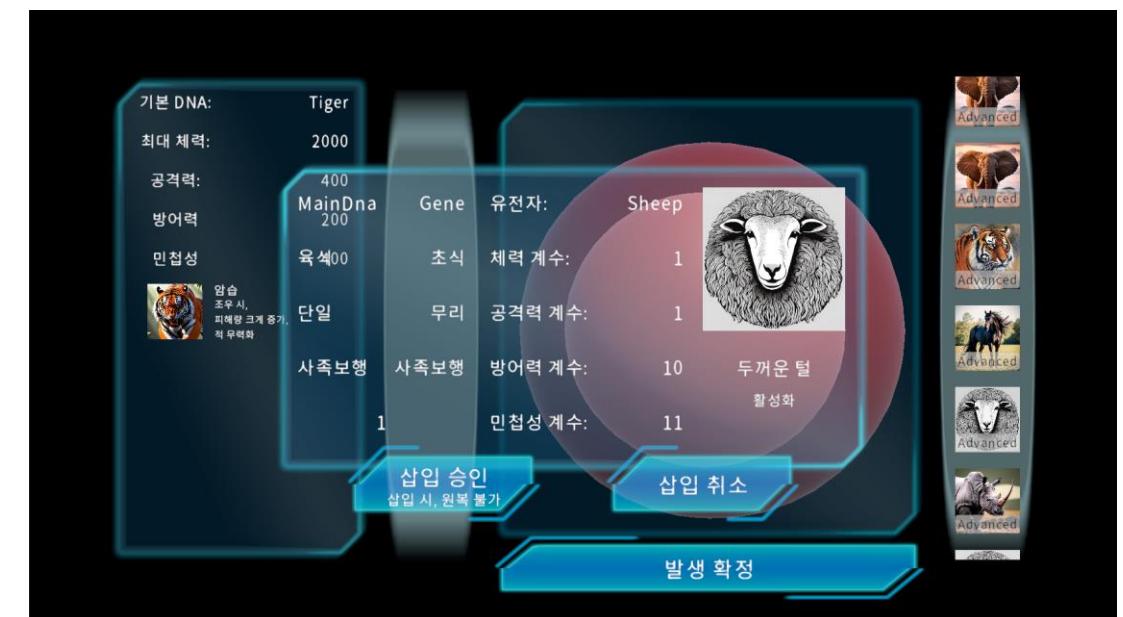
    //이거 위와 동일하세요.
    /*int randomIndex = Random.Range(0, i + 1);
    int temp = list[i];
    list[i] = list[randomIndex];
    list[randomIndex] = temp;*/}
}
```

```
public Gene(ResearcherRank rank)
{
    switch (rank)
    {
        case ResearcherRank.Public:
        case ResearcherRank.Junior://최소 5개
            for (int i = 0; i < 2; i++)
            {
                randomCoefficient = Random.Range(5, 9);
                RandomStatusCoefficient.Add(randomCoefficient);
                gradeSelector += randomCoefficient;
            }
            break;
        case ResearcherRank.Senior:
        case ResearcherRank.Principal://최소 10개
            for (int i = 0; i < 2; i++)
            {
                randomCoefficient = Random.Range(8, 11);
                RandomStatusCoefficient.Add(randomCoefficient);
                gradeSelector += randomCoefficient;
            }
            break;
        case ResearcherRank.Director://20개
            for (int i = 0; i < 2; i++)
            {
                randomCoefficient = 10;
                RandomStatusCoefficient.Add(randomCoefficient);
                gradeSelector += randomCoefficient;
            }
            break;
    }
}
```

2. 키메라(발생)

키메라는 발생시키지 않으면 배아 상태로 존재합니다. 플레이어는 키메라를 선택하여 발생해야 테스트에 활용할 수 있습니다. 키메라에 삽입될 Gene을 선택할 때에 고려사항으로 Gene의 등급, SubSkill 그리고 GeneType의 유사도입니다.

같은 Advanced 등급이지만 GeneType유사도에 따라 총 능력치는 크게 차이가 날 수 있습니다(30 vs 23).



2. 키메라(발생 전체 코드)

```
public class ChimeraCreator : MortalManager<ChimeraCreator>
{
    [SerializeField] private Embryo embryoPrefab; // TestEmbryo (Embryo)

    // Frequently called [ 3 usages
    public Syringe Syringe { get; private set; }

    private ChimeraData _targetChimeraData;
    [ 8 usages
    public ChimeraData TargetChimeraData => _targetChimeraData;

    private readonly WaitForSeconds _delay = new WaitForSeconds(3.5f);
    // Start is called once before the first execution of Update after the MonoBehaviour is created
    // Event function
    void Start()
    {
        Syringe = FindFirstObjectByType<Syringe>();
    }

    [ 1 usage
    public void SetTargetChimeraData(Chimera chimeraPrefab)
    {
        _targetChimeraData = ScriptableObject.CreateInstance<ChimeraData>();
        if (chimeraPrefab.IsMutant)
        {
            _targetChimeraData.SetChimeraData(chimeraPrefab,
                ImmortalScriptableObjectManager.Instance.FeaturesByGeneType[chimeraPrefab.GeneType],
                ImmortalScriptableObjectManager.Instance.MutantStatusesByGeneType[chimeraPrefab.GeneType]);
        }
        else
        {
            _targetChimeraData.SetChimeraData(chimeraPrefab,
                ImmortalScriptableObjectManager.Instance.FeaturesByGeneType[chimeraPrefab.GeneType],
                ImmortalScriptableObjectManager.Instance.NormalStatusesByGeneType[chimeraPrefab.GeneType]);
        }
        Instantiate(embryoPrefab);
    }
}
```

embryoPrefab 은 배아 오브젝트입니다.
Syringe는 주사바늘 오브젝트입니다.
이들은 애니메이션을 담당합니다. 이를
통해 플레이어는 직접 Gene을 삽입하는
듯 한 느낌을 받을 수 있습니다.

먼저 _targetChimeraData를 생성하여 플
레이어가 발생을 결정한 키메라 데이터를
받습니다. 이 키메라는 변종일 수 있고,
그렇지 않을 수 있습니다. 선택한 키메라
의 배아를 보여줍니다.

2. 키메라(발생 전체 코드)

```
public void InsertGene(Gene gene)
{
    if (_targetChimeraData == null)
    {
        return;
    }
    Syringe.MoveToEmbryo();
    _targetChimeraData.MainDna.InsertGene(gene);
    GameImmortalManager.Instance.AccountUseAbleGene.Remove(gene);
    StartCoroutine(routine: MoveAwayFromEmbryo());
}

④ 1 usage
public ChimeraData DevelopmentChimera()
{
    ChimeraData chimeraData = _targetChimeraData;
    //Debug.Log(chimeraData.SubSkills.Count);
    //chimeraData.SetChimeraData(chimera);
    _targetChimeraData = null;
    return chimeraData;
}

⌚ Frequently called ④ 1 usage
private IEnumerator MoveAwayFromEmbryo()
{
    yield return _delay;
    Syringe.MoveAwayFromEmbryo();
}
```

플레이어가 Gene을 선택하고 삽입하면 주사바늘이 배아를 향해 움직이는 연출이 진행됩니다. InsertGene함수는 사전에 Gene의 삽입이 가능한 경우에만 실행됩니다. Gene이 삽입되면 플레이어가 삽입 할 수 있는 GeneList에서 해당 Gene을 제거합니다.

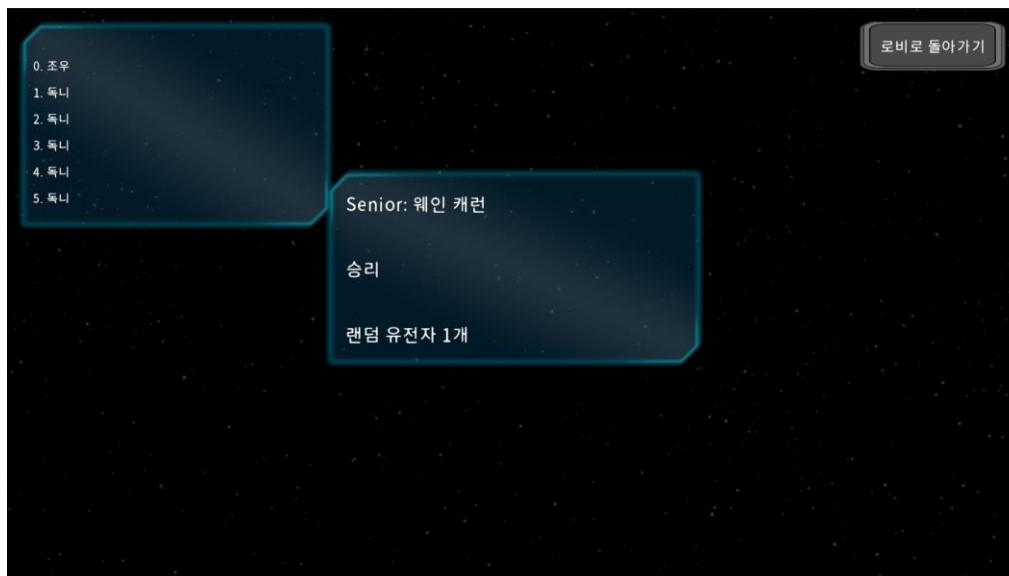
코루틴은 주사바늘이 배아에서 멀어지는 연출입니다.

이 후, 플레이어가 발생 버튼을 누르면 DevelopmentChimera 함수가 실행됩니다. 모든 설정이 끝난 키메라 데이터를 반 환합니다. 반환된 데이터는 플레이어의 발생 키메라 리스트에 저장되는 로직을 거치게 됩니다.

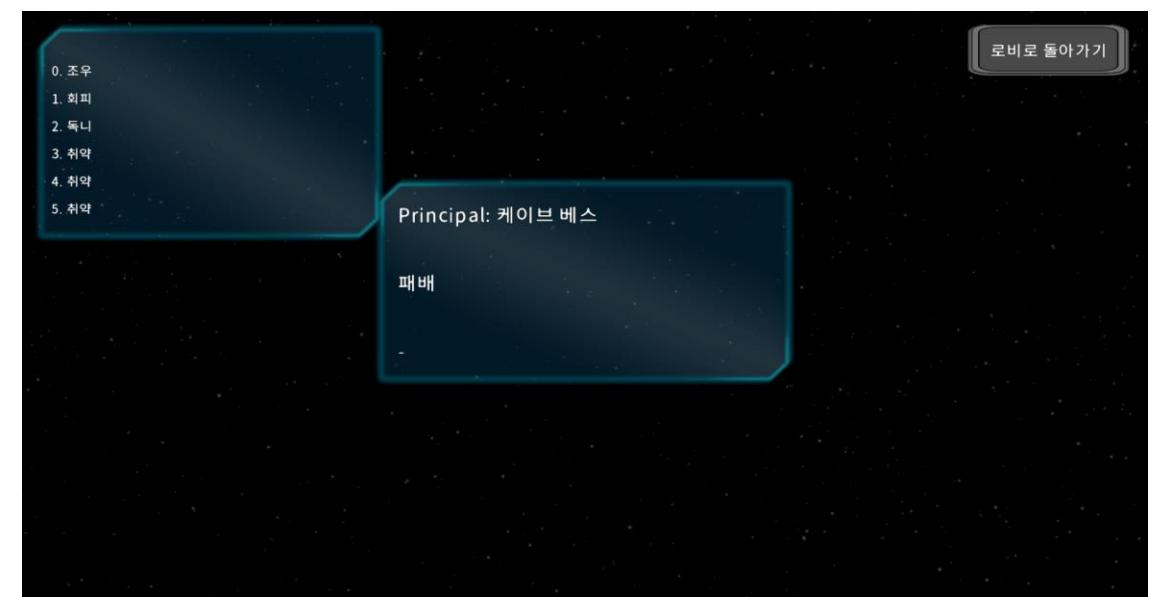
2. 키메라(Status)

발생한 키메라로 테스트를 시작하면, 키메라는 스스로 자신의 상태를 전환하며 상대 키메라로부터 살아남기 위해 사력을 다합니다. 그들의 전투 기록은 좌상단의 로그에 기록되어 플레이어에게 알려줍니다. 플레이어는 제한적인 정보를 새로운 키메라를 발생할 때 고려하여 더 강한 키메라를 발생하기 위해 고민합니다. 로그는 키메라의 주요 상태를 기록하며, 스킬을 사용한 경우 스킬 이름을 기록합니다. 플레이어는 상대 키메라의 상태를 확인할 수 없습니다. 이 게임에서 원하는 키메라는, 대부분의 상황에서 강한 면모를 보이는 키메라이기 때문입니다.

1. 승리 시 예시



2. 패배 시 예시



2. 키메라(Status 활용 방안)

상태패턴을 통해 키메라의 상태를 관리하고 추적합니다. 포켓몬스터의 배틀처럼 키메라들이 애니메이션을 통해 움직이고, 역동적인 카메라 무빙을 활용할 생각으로 상태패턴을 구현하였습니다.
하지만 제가 원하는 에셋들을 구할 수 없었고, 어울리지 않는 애니메이션을 보여주는 것은 원치 않았기 때문에 각 상태의 내부는 구현되지 않았습니다.

1. 포켓몬스터 배틀 예시 이미지(배틀 영상 링크)



2. 키메라(Skill)

키메라는 전투가 진행되는 동안 자신이 사용할 수 있는 스킬들을 적재적소에 활용하려고 합니다. 키메라의 생존은 플레이어의 성과에 반영되며 플레이어는 키메라의 생존을 위해 능력치를 상승시키고 활용가능성이 높은 스킬들을 설정해주어야 합니다. 강한 키메라는 살아남은 키메라입니다.

1. 스킬 설정이 잘못된 예시



2. 좋은 스킬 설정의 예시 및 결과



2. 키메라(Skill)

스킬(DnaMainSkill, DnaSubSkill)의 기본 구조입니다.

```
public abstract class DnaMainSkill : IDnaSkill
{
    7 usages 6 overrides
    public abstract string SkillName { get; }

    6 usages 6 overrides
    public abstract string SkillDescription { get; }

    0+16 usages 6 overrides
    public abstract int NeedInstinctPoint { get; protected set; }
}
```

```
public interface IDnaSkill
{
    17 implementations
    public string SkillName { get; }

    17 implementations
    public string SkillDescription { get; }

    Frequently called 16 usages 17 implementations
    public int NeedInstinctPoint { get; }
}
```

```
public abstract class DnaSubSkill : IDnaSkill
{
    private Image _mainSkillImage;
    8 usages 9 overrides
    public abstract string SkillName { get; }

    4 usages 9 overrides
    public abstract string SkillDescription { get; }

    0+16 usages 9 overrides
    public abstract int NeedInstinctPoint { get; }

    Frequently called 4 usages
    public GeneType GeneType { get; }

    Frequently called 9 usages
    protected DnaSubSkill(GeneType geneType)
    {
        GeneType = geneType;
    }
}
```

큰 틀을 만들고, 세분화 과정이 필요했습니다. 스킬들은 기본적으로 스킬명, 스킬 설명 그리고 스킬 사용을 위해 요구되는 자원을 갖습니다.

MainSkill은 Dna자체에 GeneType을 갖기 때문에 DnaSubSkill과 다르게 GeneType을 따로 구현하지 않습니다.

DnaSubSkill은 GeneType을 따로 구현합니다. 이는 추후 업데이트 방향에 Gene에 다른 스킬을 할당하는 것을 고려하고 있기 때문입니다.

2. 키메라(Skill)

스킬(공격 스킬, 방어 스킬)의 기본 구조입니다.

```
public interface IAttackSkill : IDnaSkill
{
    ⚡ Frequently called 29 usages 9 implementations
    public float DamageCoefficient { get; }

    ⚡ Frequently called 1 usage 9 implementations
    public bool IsOpponentOnState(IChimeraState lastOpponentState);

    ⚡ Frequently called 4 usages 9 implementations
    public IChimeraState AdditionalEffectToOpponentChimera(Chimera chimera);

    ⚡ Frequently called 1 usage 9 implementations
    public bool IsDodgeAble();
}
```

공격 스킬은 피해 계수, 요구하는 상대 키메라의 마지막 상태, 스킬 사용시 강제로 적용할 상대 키메라의 상태, 회피 가능 여부를 구현하도록 강제합니다.

```
public interface IDefendSkill : IDnaSkill
{
    ⚡ Frequently called 3 usages 6 implementations
    public float DefendCoefficient { get; }

    ⚡ Frequently called 1 usage 6 implementations
    public bool IsOpponentOnState(IChimeraState currentOpponentState);

    ⚡ Frequently called 4 usages 6 implementations
    public IDnaSkill AdditionalEffectToMySelf(Chimera chimera);
}
```

방어 스킬은 피해 감소 계수, 요구하는 상대 키메라의 마지막 상태, 스킬 사용시 자신에게 적용할 효과를 구현하도록 강제합니다. 자신에게 적용할 효과의 경우 스킬을 리턴합니다.

2. 키메라(Skill)

스킬(공격 스킬, 특정 상태 요구)의 예시입니다.

```
public class TigerOverExSkill : DnaMainSkill, IAttackSkill
{
    // 0+7 usages
    public override string SkillName { get; } = "암습";

    // 0+6 usages
    public override string SkillDescription { get; } = "조무 시, \n피해량 크게 증가, \n적 무력화";

    // 0+16 usages
    public override int NeedInstinctPoint { get; protected set; } = 14;

    // 0+9 usages
    public float DamageCoefficient { get; } = 1.4f;

    // Frequently called 0+1 usages
    public bool IsOpponentOnState(IChimeraState lastOpponentState)
    {
        return lastOpponentState is StandingState;
    }

    // Frequently called 0+4 usages
    public IChimeraState AdditionalEffectToOpponentChimera(Chimera defender)
    {
        return defender.GroggyState;
    }

    // Frequently called 0+1 usages
    public bool IsDodgeAble()
    {
        return false;
    }
}
```

구현된 키메라 중 하나인 호랑이의 MainDnaSkill입니다.

이 스킬은 최초로 전투에 진입 시(StandingState) 1회만 발동하도록 설계되어 있습니다. 높은 피해량으로 적을 무력화(그로기) 상태로 강제하여 최소 2번의 공격을 강제로 성공시킬 수 있습니다(하단 코드 이미지).

단, 선공권을 가진 경우에만 발동하도록 설계되어 먼저 공격할 수 없다면 그 성능이 떨어집니다.

```
//사용될 공격스킬이 회피할 수 없다면
if (!attackerStateInfo.Item2.IsDodgeAble()){...}

//만약 디펜더의 직전 상태가 그로기상태라면
if (_combatChimerasStatesDictionary[defender][^1] is GroggyState)
{
    defender.TakeDamage(attacker.AttackPoint * attackerStateInfo.Item2.DamageCoefficient * 2f);
    if (CheckEndCombat(attacker, defender)) return;

    _combatChimerasInstinctPointsDictionary[_attacker] += 10;
    _combatChimerasInstinctPointsDictionary[_defender] += 10;
    _combatChimerasStatesDictionary[defender].Add(attackerStateInfo.Item3);
    return;
}
```

2. 키메라(Skill)

스킬(공격 스킬, 공용)의 예시입니다.

```
public class HorseDnaSubSkill : DnaSubSkill, IAttackSkill
{
    // Frequently called 1 usage
    public HorseDnaSubSkill(GeneType geneType) : base(geneType)
    {
        GeneType = geneType;
    }

    // 0+8 usages
    public override string SkillName { get; } = "발길질";
    // 0+4 usages
    public override string SkillDescription { get; } = "피해량 증가";
    // 0+16 usages
    public override int NeedInstinctPoint { get; } = 11;
    // Frequently called 1+3 usages
    public sealed override GeneType GeneType { get; protected set; }
    // 0+9 usages
    public float DamageCoefficient { get; } = 1.2f;
    // Frequently called 0+2 usages
    public bool IsOpponentOnState(IChimeraState lastOpponentState)
    {
        return true;
    }

    // Frequently called 0+4 usages
    public IChimeraState AdditionalEffectToOpponentChimera(Chimera chimera)
    {
        return chimera.VulnerableState;
    }

    // Frequently called 0+1 usages
    public bool IsDodgeAble()
    {
        return true;
    }
}
```

구현된 유전자 중 하나인 말의 DnaSubSkill입니다.

스킬의 이름은 Gene이나 키메라와 연상이 되도록 명명되어있습니다. 대부분의 공격 스킬은 취약 상태(VulnerableState)를 리턴합니다. 방어 키메라는 회피하지 못하면 계속 취약 상태로 남게 되고 끝내 사망에 이르게 됩니다.

2. 키메라(skill)

스킬(방어 스킬)의 예시입니다.

```
public class SheepOverExSkill : DnaMainSkill, IDefendSkill
{
    public override string SkillName { get; } = "양털";
    public override string SkillDescription { get; } = "기본 공격 피해 크게 감소";
    public override int NeedInstinctPoint { get; protected set; } = 14;
    public float DefendCoefficient { get; } = 0.5f;
    public bool IsOpponentOnState(IChimeraState currentOpponentState)
    {
        return currentOpponentState is BasicAttackState;
    }
    public IDnaSkill AdditionalEffectToMyself(Chimera chimera)
    {
        return this as IDnaSkill;
    }
}
```

구현된 키메라 중 하나인 양의 MainDnaSkill입니다.

이 스킬은 상대의 기본 공격 피해를 매우 크게 감소시키는 스킬입니다. 특정 상태를 저격하여 효과를 발동하기 때문에 더 큰 비율로 피해를 감소하도록 설계했습니다.

```
public class BuffaloOverExSkill : DnaMainSkill, IDefendSkill
{
    public override string SkillName { get; } = "두꺼운 가죽";
    public override string SkillDescription { get; } = "모든 피해 크게 감소";
    public override int NeedInstinctPoint { get; protected set; } = 14;
    public float DefendCoefficient { get; } = 0.7f;
    public bool IsOpponentOnState(IChimeraState currentOpponentState)
    {
        return true;
    }
    public IDnaSkill AdditionalEffectToMyself(Chimera chimera)
    {
        return this as IDnaSkill;
    }
}
```

구현된 키메라 중 하나인 물소의 MainDnaSkill입니다.

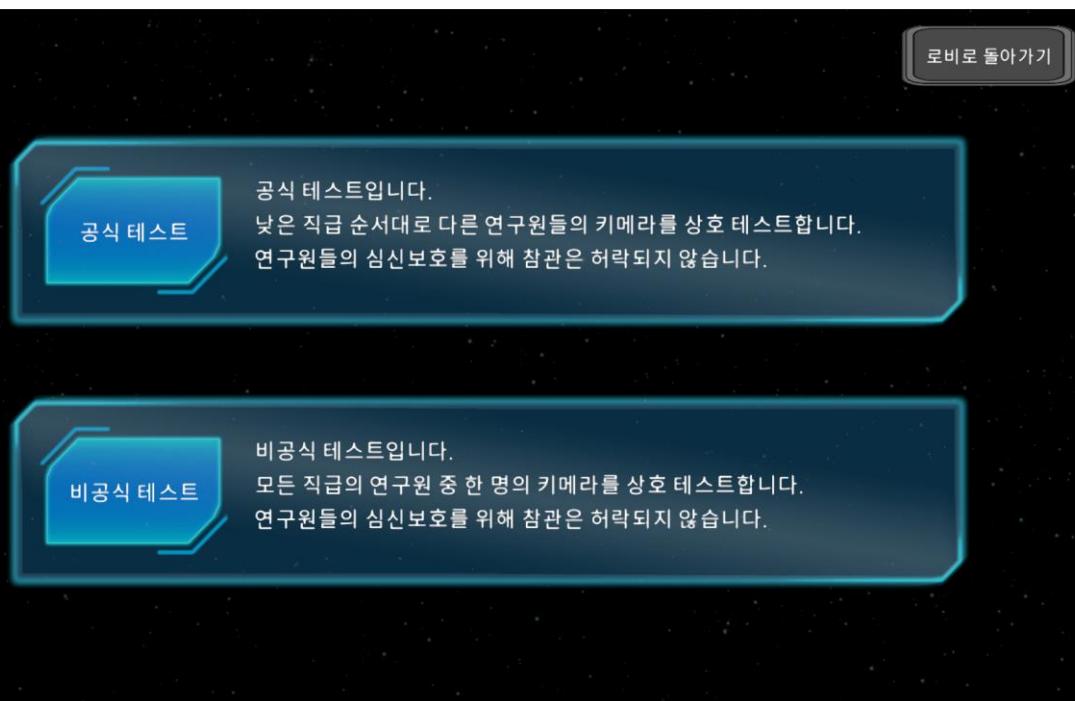
이 스킬은 상대의 공격 피해를 크게 감소시키는 스킬입니다. 모든 상태에 대해서 발동하기 때문에 양 보다 더 낮은 비율로 피해를 감소시키도록 설계했습니다.

플레이어는 스킬 설명을 읽고 스킬의 효율을 분석해야 합니다. 특정 조건에서만 발동되는 스킬은 그 조건에서 더 큰 효율을 발휘한다는 힌트를 제공할 뿐 구체적인 수치는 보여주지 않음으로써 플레이어를 고민하게 합니다.

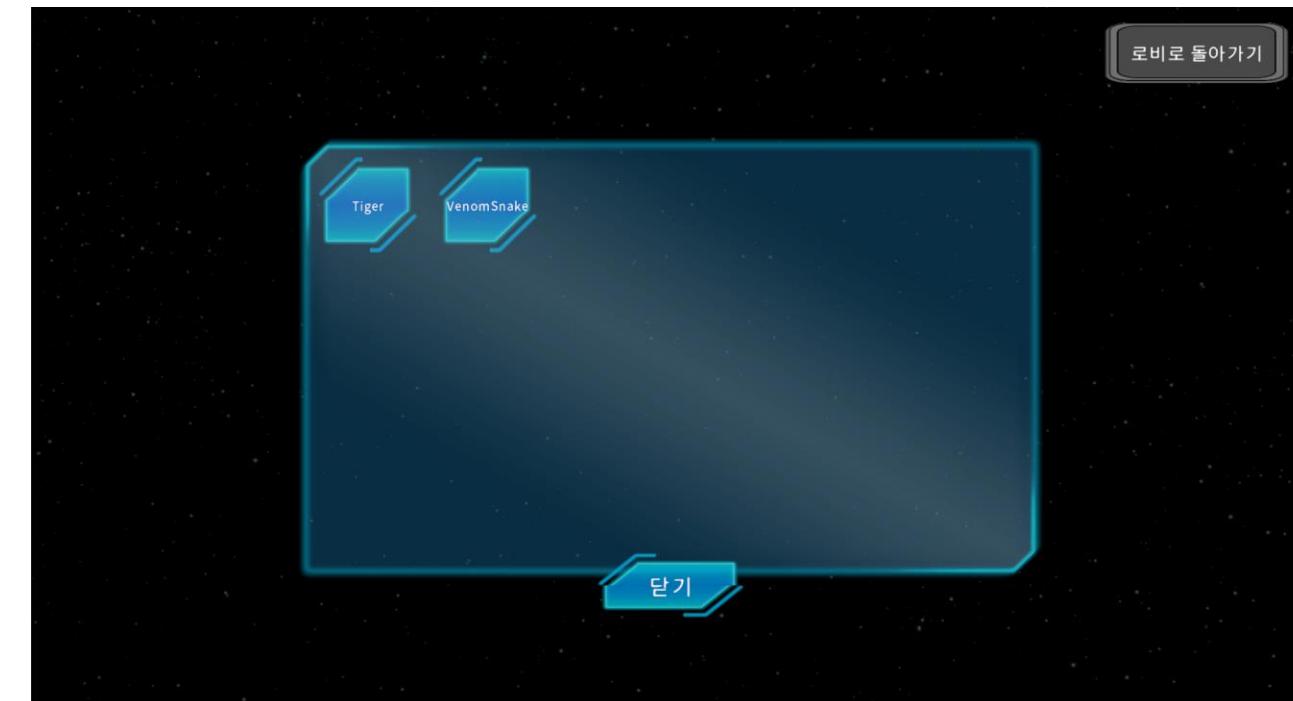
3. 공식/비공식 테스트

플레이어는 육성을 완료하여 제어권을 획득한 키메라를 테스트할 수 있습니다. 비공식 테스트는 모든 직급을 대상으로 도전할 수 있으며 1회만 테스트를 진행합니다. 공식 테스트는 낮은 직급부터 높은 직급 순으로 임의의 연구원들과 상호 테스트를 진행하며 플레이어의 키메라가 사망할 때까지 진행합니다. 승리시 직급에 영향을 받는 임의의 유전자를 획득할 수 있으며 테스트 형식에 관계없이 변종 전문 연구원에게서 승리하면 해당 연구원이 다루는 변종 키메라에 대한 발생 권한을 획득합니다.

1. 진입 테스트 선택 화면



2. 임의의 테스트 선택 시 테스트할 키메라 선택 화면



3. 공식/비공식 테스트

플레이어는 테스트 종료 후 화면 좌 상단의 기록을 통해 내 키메라의 전투 기록을 확인할 수 있습니다.

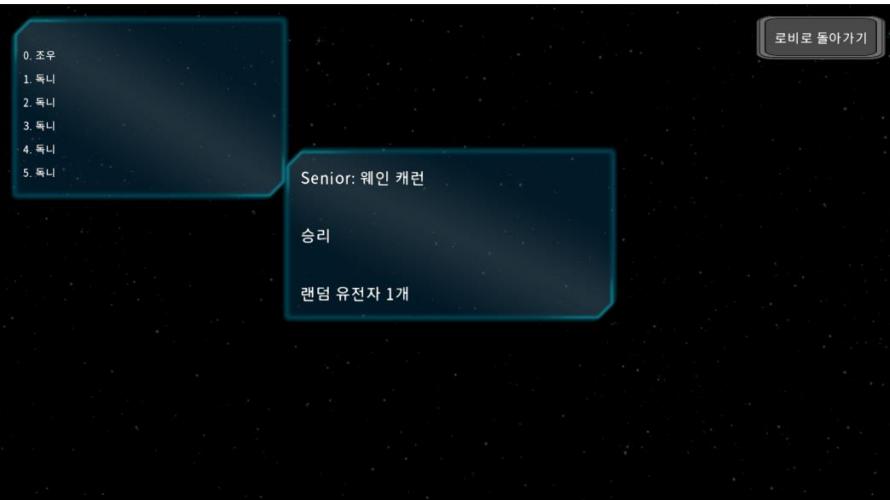
3. 테스트 할 키메라 선택 화면



4. 테스트 진행 화면



5. 승리 시 화면



6. 패배 시 화면



3. 공식/비공식 테스트(전체 코드 중 변수)

```
public class CombatMachine
{
    private Chimera _playerChimera;
    private Chimera _opponentChimera;

    private Chimera _attacker;
    private Chimera _defender;

    private List<IChimeraState> _playerChimeraStates;
    private List<IChimeraState> _opponentChimeraStates;

    private Dictionary<Chimera, List<IChimeraState>> _combatChimerasStatesDictionary;
    private Dictionary<Chimera, int> _combatChimerasInstinctPointsDictionary;

    ITestManager _testManager;

    private int _attackerAgilityPoint;
    private int _defenderAgilityPoint;

    private bool _endCombat = false;
    private bool _switchAttacker = false;

    private int _encounterCount = 0;
```

CombatMachine(이하 CM)은 이 게임에서 전투 시스템을 담당합니다. CM은 키메라만 필요합니다. 플레이어와 NPC의 키메라를 할당받고, 선공 키메라를 결정하는 함수를 통해 _attacker와 _defender를 각각 할당합니다.

결과 출력을 위해 플레이어와 NPC의 키메라의 상태를 추적하며 기록합니다.

_attacker와 _defender로 공격 키메라와 방어 키메라의 상태를 정확히 추적하기 위해 딕셔너리를 활용하며, 각 키메라가 현재 가용한 자원이 얼마나 남았는지 체크하기 위해 마찬가지로 딕셔너리를 활용합니다.

_testManager 변수는 공식/비공식 테스트 매니저를 타입에 구애받지 않고 할당하기 위해 두 매니저가 구현중인 인터페이스로 선언합니다.

_AgilityPoint 는 두 키메라가 누가 먼저 움직일지, 회피여부 등을 결정하기 위해 사용합니다.

두 bool 변수는 전투의 종료와 공격권의 교대 발생시 사용합니다.

낮은 확률로 전투가 끝나지 않는 경우를 대비하여 최대 상태기록을 초과하는 경우 전투를 종료하기 위해 두 키메라의 전투 기록 개수를 기록합니다.

[2. 키메라로 돌아가기](#)

3. 공식/비공식 테스트(전체 코드 중 함수)

```
public CombatMachine(Chimera player, Chimera opponent, ITestManager testManager)
{
    _playerChimera = player;
    _opponentChimera = opponent;
    _testManager = testManager;

    _playerChimeraStates = new List<IChimeraState>();
    _opponentChimeraStates = new List<IChimeraState>();

    _combatChimerasStatesDictionary = new Dictionary<Chimera, List<IChimeraState>>()
    {
        {_playerChimera, _playerChimeraStates},
        {_opponentChimera, _opponentChimeraStates}
    };
    _combatChimerasInstinctPointsDictionary = new Dictionary<Chimera, int>()
    {
        { _playerChimera, 20 },
        { _opponentChimera, 20 }
    };
}
```

CombatMachine(이하 CM)은 이 게임에서 전투 시스템을 담당합니다. CM은 전투 기록을 위해 필요한 변수들을 할당하며 생성됩니다.

최초에 두 키메라가 사용할 수 있는 자원의 양은 20으로 스킬들은 특정한 값을 요구하며, 이 값을 충족하지 않으면 스킬을 사용할 수 없습니다.

```
/// <summary>
/// officialTest Only
/// </summary>
/// <param name="player"></param>
/// <param name="opponent"></param>
[usage]
public void RestartCombat(Chimera player, Chimera opponent)
{
    _endCombat = false;
    _encounterCount = 0;
    _playerChimera = player;
    _opponentChimera = opponent;

    _playerChimeraStates ??= new List<IChimeraState>();
    _playerChimeraStates.Clear();
    _opponentChimeraStates ??= new List<IChimeraState>();
    _opponentChimeraStates.Clear();
```

비공식 테스트는 1회만 진행되지만 공식 테스트의 경우 플레이어가 패배할 때 까지 진행합니다. 그렇기에 사용했던 변수들을 새롭게 할당해야 할 필요가 있습니다.

3. 공식/비공식 테스트(전체 코드 중 함수)

```
/// <summary>
/// first is player's, second is researcher's
/// </summary>
/// <returns></returns>
↳ Frequently called [2 usages]
public (List<IChimeraState>, List<IChimeraState>) GetChimeraStatesChain()
{
    Chaining();
    for (int i = 0; i < _playerChimeraStates.Count; i++)
    {
        switch (_testManager)
        {
            case OfficialTestManager:
                OfficialTestUiManager.Instance.AddTestLog($"{i}. {_playerChimeraStates[i].StateName}");
                break;
            case NonOfficialTestManager:
                NonOfficialTestUiManager.Instance.AddTestLog($"{i}. {_playerChimeraStates[i].StateName}");
                break;
        }
    }

    /*for (int i = 0; i < _opponentChimeraStates.Count; i++)
    {
        Debug.Log($"{i}. {_opponentChimeraStates[i].GetType()} / Cost: {_combatChimerasInstinctPointsDictionary[_opponentChimera]}");
    }*/
}

return (_playerChimeraStates, _opponentChimeraStates);
}
```

CombatMachine(이하 CM)은 이 게임에서 전투 시스템을 담당합니다. CM은 Chaining함수를 통해 두 키메라의 상호 전투 기록을 저장합니다. 또한 플레이어에게 전투 기록을 출력하기 위해 키메라의 전투 상태를 공유합니다.

전투 상태는 강 상태의 StateName이며 스킬을 사용한 경우, 스킬 명이 StateName에 할당됩니다.

```
public class FirstSkillState : IChimeraState
{
    ↳ 1 usage
    public Chimera Chimera { get; }
    public Animator Animator { get; }
    ↳ 1+2 usages
    public string StateName { get; }
    ↳ 1 usage
    public DnaSubSkill SubSkill { get; }

    ↳ 3 usages
}
```

```
public class GroggyState : IChimeraState
{
    ↳ 1 usage
    public Chimera Chimera { get; }
    public Animator Animator { get; }
    ↳ 0+2 usages
    public string StateName { get; } = "무력화";
}
```

```
public class MainSkillState : IChimeraState
{
    ↳ 1 usage
    public Chimera Chimera { get; }
    public Animator Animator { get; }

    ↳ 2 usages
    public DnaMainSkill MainSkill { get; }

    ↳ 1+2 usages
    public string StateName { get; }

    ↳ 1 usage
    public MainSkillState(Chimera chimera)
    {
        Chimera = chimera;
        MainSkill = chimera.MainSkill;
        StateName = MainSkill.SkillName;
        //Animator = chimera.Animator;
    }
}
```

3. 공식/비공식 테스트(전체 코드 중 함수)

```
private void Chaining()
{
    //선공권 결정 후 공격시도
    SetFirstStates(); //<- 리스트의 최초 상태를 채움.(공격, 회피 또는 공격, 회피, 회피시 공 수 교대)
    while (!_endCombat)
    {
        Encounter(_attacker, _defender);
        _encounterCount++;
        /* Debug.Log($"{_encounterCount}: P: {_playerChimeraStates[^1].GetType()}/{_combatChimeras[_encounterCount].GetType()}"); */
        if (_endCombat)
        {
            break;
        }
        if (_switchAttacker)
        {
            _switchAttacker = false;
        }

        //avoid while infinity loop
        if (_encounterCount > 50)
        {
            if (_playerChimera.CurrentHealthPoint >= _opponentChimera.CurrentHealthPoint)
            {
                _testManager.WinnerIs(_playerChimera);
            }
            else
            {
                _testManager.WinnerIs(_opponentChimera);
            }
            _endCombat = true;
        }
    }
}
```

CM의 Chaining함수는 선공권을 결정한 후(SetFirstStates) 두 키메라의 전투가 종료될 때까지 Encounter함수를 반복합니다.

만약 두 키메라의 전투 기록이 50회를 넘게 되면 전투를 강제 종료하고 잔여 체력 상황에 따라 승리한 키메라를 정합니다. 플레이어에게 더 우호적으로 판단될 수 있도록 등호를 추가하였습니다.

승리 키메라는 테스트매니저의 WinnerIs함수의 인자로 전달되어 플레이어에게 적절한 결과를 출력하고 반영합니다.

3. 공식/비공식 테스트(전체 코드 중 함수)

```
//시작 시 선공 키메라 정하기. Agility 비교하여 같으면 랜덤하기
⌚ Frequently called [ 1 usage]
private void SetFirstStates()
{
    if (_playerChimera.AgilityPoint > _opponentChimera.AgilityPoint)
    {
        _attacker = _playerChimera;
        _defender = _opponentChimera;
        _combatChimerasStatesDictionary[_attacker].Add(_attacker.StandingState);
        _combatChimerasStatesDictionary[_defender].Add(_defender.StandingState);
    }
    else if (_playerChimera.AgilityPoint < _opponentChimera.AgilityPoint)
    {
        _attacker = _opponentChimera;
        _defender = _playerChimera;
        _combatChimerasStatesDictionary[_attacker].Add(_attacker.StandingState);
        _combatChimerasStatesDictionary[_defender].Add(_defender.StandingState);
    }
    else
    {
        int rand = UnityEngine.Random.Range(0, 2);
        switch (rand)
        {
            case 0:
                _attacker = _playerChimera;
                _defender = _opponentChimera;
                _combatChimerasStatesDictionary[_attacker].Add(_attacker.StandingState);
                _combatChimerasStatesDictionary[_defender].Add(_defender.StandingState);
                break;
            case 1:
                _attacker = _opponentChimera;
                _defender = _playerChimera;
                _combatChimerasStatesDictionary[_attacker].Add(_attacker.StandingState);
                _combatChimerasStatesDictionary[_defender].Add(_defender.StandingState);
                break;
        }
    }
}
```

CM의 SetFirstStates함수는 선공권을 가질 키메라를 결정합니다. 이 때 두 키메라의 민첩성 포인트(AgilityPoint)를 기준으로 결정합니다. 만약 두 키메라의 민첩성 포인트가 같다면, 임의로 결정합니다.

두 키메라는 최초에 조우 상태로 전투에 진입하며 선공권을 가진 키메라가 _attacker에 할당되어 공격을 시작합니다.

3. 공식/비공식 테스트(전체 코드 중 함수)

```
private void Encounter(Chimera attacker, Chimera defender)
{
    //AttackerSkillState, CanDodge, defenderState
    (IChimeraState, IAttackSkill, IChimeraState) attackerStateInfo = AttackerUseSkill();

    _combatChimerasStatesDictionary[attacker].Add(attackerStateInfo.Item1);

    //공격 스킬을 사용했다면
    if (attackerStateInfo.Item2 != null)
    {
        //사용된 공격스킬이 회피할 수 없다면
        if (!attackerStateInfo.Item2.IsDodgeable())
        {
            //공격 스킬이 방어 키메라의 상태를 강제하는 경우
            if (attackerStateInfo.Item3 != null)
            {
                if (_combatChimerasStatesDictionary[defender][^1] is GroggyState)
                {
                    defender.TakeDamage(attacker.AttackPoint * attackerStateInfo.Item2.DamageCoefficient * 2f);
                    if (CheckEndCombat(attacker, defender)) return;

                    _combatChimerasInstinctPointsDictionary[_attacker] += 10;
                    _combatChimerasInstinctPointsDictionary[_defender] += 10;
                }
                else
                {
                    defender.TakeDamage(attacker.AttackPoint * attackerStateInfo.Item2.DamageCoefficient);
                    if (CheckEndCombat(attacker, defender)) return;

                    _combatChimerasInstinctPointsDictionary[_attacker] += 10;
                    _combatChimerasInstinctPointsDictionary[_defender] += 10;
                }
                _combatChimerasStatesDictionary[defender].Add(attackerStateInfo.Item3);
                return;
            }
            (IChimeraState, IDefendSkill) defenderSkill = DefenderUseSkill();
            _combatChimerasStatesDictionary[defender].Add(defenderSkill.Item1);
            if (defenderSkill.Item2 != null)
            {
                defender.TakeDamage(attacker.AttackPoint * attackerStateInfo.Item2.DamageCoefficient * defenderSkill.Item2.DefendCoefficient);
            }
            else
            {
                defender.TakeDamage(attacker.AttackPoint * attackerStateInfo.Item2.DamageCoefficient);
            }
            if (CheckEndCombat(attacker, defender)) return;

            _combatChimerasInstinctPointsDictionary[_attacker] += 10;
            _combatChimerasInstinctPointsDictionary[_defender] += 10;
            return;
        }
        (IChimeraState, IDefendSkill) defenderSkill = DefenderUseSkill();
    }
}
```

CM의 Encounter함수는 두 키메라 간의 *합을 계산합니다. 공격 키메라는 AttackerUseSkill을 통해 공격 상태, 스킬, 방어 키메라의 상태 강제 여부를 계산하고, 공격 상태는 attacker키메라의 상태 추적용 딕셔너리의 벨류에 해당하는 리스트(이하 상태 추적 리스트)에 추가합니다. 이후, 공격 스킬과 그 성질에 따라 방어 키메라에게 피해를 입히거나 회피 여부를 결정하고 그 결과를 반영한 뒤 방어 키메라의 상태 추적 리스트에 추가합니다.

Encounter함수는 매 실행마다 전투 종료 여부를 계산하고 두 키메라가 사용할 수 있는 자원의 양을 증가시킵니다.

GroggyState는 무력화 상태로 피해를 무조건 입는 상태입니다. 이미지에는 나오지 않았지만 VulnerableState는 취약 상태로 회피가 가능한 상태입니다. 일반적으로, 방어 키메라는 피해를 입으면 취약상태가 됩니다.

회피 여부는 SetFirstStates 함수에서 사용한 로직과 동일합니다.
* 한 번 교차했다가 떨어지는 것, 고전소설에서 등장하는 단어

```
(IChimeraState, IDefendSkill) defenderSkill = DefenderUseSkill();
_combatChimerasStatesDictionary[defender].Add(defenderSkill.Item1);
if (defenderSkill.Item2 != null)
{
    defender.TakeDamage(attacker.AttackPoint * attackerStateInfo.Item2.DamageCoefficient * defenderSkill.Item2.DefendCoefficient);
}
else
{
    defender.TakeDamage(attacker.AttackPoint * attackerStateInfo.Item2.DamageCoefficient);
}
if (CheckEndCombat(attacker, defender)) return;

_combatChimerasInstinctPointsDictionary[_attacker] += 10;
_combatChimerasInstinctPointsDictionary[_defender] += 10;
return;
```

3. 공식/비공식 테스트(전체 코드 중 함수)

```
⌚ Frequently called ⌚ 8 usages
private bool CheckEndCombat(Chimera attacker, Chimera defender)
{
    if (defender.CurrentHealthPoint <= 0)
    {
        _combatChimerasStatesDictionary[defender].Add(defender.DeathState);
        _testManager.WinnerIs(attacker);
        _endCombat = true;
        return true;
    }

    return false;
}
```

```
⌚ Frequently called ⌚ 2 usages
private void SwitchAttacker()
{
    _switchAttacker = true;
    //회피시 공수 교대
    if (_combatChimerasStatesDictionary[_defender][^1] is DodgeState)
    {
        (_attacker, _defender) = (_defender, _attacker);
    }
}
```

방어 키메라의 현재 체력이 0이하로 떨어지면 방어 키메라의 상태에 사망(DeathState)을 추가하고 전투 종료를 알립니다. 전투가 종료되면 승리한 키메라의 소유자에 따라 다른 로직이 실행됩니다.

공수 교대는 방어 키메라가 회피한 경우(방어 키메라의 마지막 제출 상태가 DodgeState)에만 적용됩니다.

3. 공식/비공식 테스트(전체 코드 중 함수)

```
1. {  
    private (IChimeraState, IAttackSkill, IChimeraState) AttackerUseSkill()  
    {  
        //attacker의 모든 스킬 중  
        List<IDnaSkill> skills = new List<IDnaSkill>();  
        skills.AddRange(_attacker.SubSkills);  
        skills.Add(_attacker.MainSkill);  
  
        //공격 스킬들 찾고  
        List<IAttackSkill> attackSkills = new List<IAttackSkill>();  
        foreach (var skill in skills)  
        {  
            if (skill is IAttackSkill iAttackSkill)  
            {  
                attackSkills.Add(iAttackSkill);  
            }  
        }  
  
        //리턴 보기 1: 공격스킬 없음  
        if (attackSkills.Count == 0)  
        {  
            return (_attacker.BasicAttackState, (IAttackSkill) null, (IChimeraState) null);  
        }  
  
        //공격 스킬들 중 현재 자원량으로 사용 가능한 스킬들을 찾고  
        List<IAttackSkill> currentUseAbleAttackSkills = new List<IAttackSkill>();  
        foreach (var skill in attackSkills)  
        {  
            if (skill.NeedInstinctPoint <= _combatChimerasInstinctPointsDictionary[_attacker])  
            {  
                currentUseAbleAttackSkills.Add(skill);  
            }  
        }  
  
        //리턴 보기2: 공격스킬은 있으나 현재 자원으로 사용할 수 없음  
        if (currentUseAbleAttackSkills.Count == 0)  
        {  
            return (_attacker.BasicAttackState, (IAttackSkill) null, (IChimeraState) null);  
        }  
    }  
}
```

공격 키메라는 다음 조건을 통해 공격상태를 결정합니다.

1. 자신에게 공격 스킬이 있는지.
2. 그 공격 스킬들이 현재 자신의 자원량으로 사용가능한지.
만약, 1과2의 조건들을 만족하지 못한다면 기본 공격 상태를 제출합니다. 지역변수로 리스트를 선언하여 foreach문으로 확인합니다. 최대 4번을 반복하므로 시간 복잡도를 따로 고려하지 않았습니다.
3. 사용 가능한 공격스킬들 중 적의 특정 상태를 요구하는 스킬들과 그렇지 않은 스킬들을 분류

```
3. //자원량을 충족하는 스킬들 중 적의 특정 마지막 상태를 요구하는 스킬들을 찾는다.  
List<IAttackSkill> stateCheckedSkills = new List<IAttackSkill>();  
//자원량을 충족하는 스킬들 중 적의 특정 마지막 상태를 요구하지 않는 스킬들을 찾는다.  
List<IAttackSkill> noStateCheckedSkills = new List<IAttackSkill>();  
foreach (var skill in currentUseAbleAttackSkills)  
{  
    if (skill.IsOpponentOnState(_combatChimerasStatesDictionary[_defender][^1]))  
    {  
        stateCheckedSkills.Add(skill);  
    }  
  
    //케이지 쪽에서 사용되는 스테이트로 전투에서는 사용되지 않는 스테이트를 넣음.  
    //모든 노스테이트체크스킬들은 인자로 받는 상태에 관계 없이 true를 반환하기 때문  
    if (skill.IsOpponentOnState(_defender.SpawnAndWaitState))  
    {  
        noStateCheckedSkills.Add(skill);  
    }  
}
```

3. 공식/비공식 테스트(전체 코드 중 함수)

4. IAttackSkill attackSkill = null;

//리턴 분기3: 공격스킬이 있고, 현재 자원량으로 사용할 수 있으며 적의 특정 마지막 상태를 요구하는 스킬이 있고 그 스킬이 메인 스킬인 경우
foreach (IAttackSkill skill in stateCheckedSkills)

```
{  
    if (skill is DnaMainSkill)  
    {  
        _combatChimerasInstinctPointsDictionary[_attacker] -= skill.NeedInstinctPoint;  
        return (_attacker.SkillStates[skill], skill, skill.AdditionalEffectToOpponentChimera(_defender));  
    }  
}
```

//리턴 분기4: 공격스킬이 있고, 현재 자원량으로 사용할 수 있으나 적의 특정 마지막 상태를 요구하는 스킬이 아니고 그 스킬이 메인 스킬인 경우
foreach (IAttackSkill skill in noStateCheckedSkills)

```
{  
    if (skill is DnaMainSkill)  
    {  
        _combatChimerasInstinctPointsDictionary[_attacker] -= skill.NeedInstinctPoint;  
        return (_attacker.SkillStates[skill], skill, skill.AdditionalEffectToOpponentChimera(_defender));  
    }  
}
```

4. 분류된 두 개의 스킬 리스트 중 MainDnaSkill이 포함되어있다면 그 스킬을 실행.

MainDnaSkill이 사용할 수 있는 상황이라면 반드시 MainDnaSkill이 사용되도록 설계했습니다. 게임 컨셉 상, 야생성이 살아있는 키메라를 테스트에 사용하며 이들은 본능에 충실할 것이라고 생각했기 때문입니다.

3. 공식/비공식 테스트(전체 코드 중 함수)

5.

```
//리턴 분기5: 사용 가능한 분류된 스킬들 중 메인스킬이 없는 경우이면서 특정 상태를 요구하는 스킬이 있어 해당 스킬을 먼저 사용해야하는 경우
switch (stateCheckedSkills.Count)
{
    //상태 요구를 충족하는 스킬이 없어? -> 특정 상태를 요구하지 않는 스킬을 확인해야함.
    case 0:
        break;
    //상태 요구를 충족하는 스킬이 있어?
    case >= 1:
        attackSkill = stateCheckedSkills[0]; // 첫 번째 스킬을 기본으로 설정
        foreach (var skill in stateCheckedSkills)
        {
            if (skill.NeedInstinctPoint > attackSkill.NeedInstinctPoint)
            {
                attackSkill = skill;
            }
        }

        _combatChimerasInstinctPointsDictionary[_attacker] -= attackSkill.NeedInstinctPoint;
        return (_attacker.SkillStates[attackSkill], attackSkill, attackSkill.AdditionalEffectToOpponentChimera(_defender));
}
```

5. MainDnaSkill이 사용되지 않았다면, 특정 상태를 요구하는 스킬을 먼저 사용.

특정 상태를 요구하는 스킬은 모든 상태에서 사용할 수 있는 스킬보다 더 효율이 좋은 편입니다. 키메라는 본능적으로 그것을 활용할 줄 알고 있습니다. 또 만약, 같은 조건의 스킬이 여러 개라면, 요구하는 비용이 더 큰 것을 먼저 사용하도록 설계했습니다. 키메라에게는 생존과 직결되는 부분으로 가용한 자원을 남김없이 사용해야 합니다.

6번 분기는 남아있는 스킬을 사용하거나 기본공격을 최종적으로 리턴합니다. 6번 분기는 5번 분기와 같은 로직을 사용하지만 case 0: 에서 기본공격을 리턴합니다.

3. 공식/비공식 테스트(전체 코드 중 함수)

```
/// <summary>
/// </summary>
/// <returns>Defender(MainSkill)State, DefendSkill</returns>
Frequently called 3 usages
private (IChimeraState, IDefendSkill) DefenderUseSkill()
{
    //defender의 모든 스킬 중
    List<IDnaSkill> skills = new List<IDnaSkill>();
    skills.AddRange(_defender.SubSkills);
    skills.Add(_defender.MainSkill);

    //방어 스킬을 찾고
    List<IDefendSkill> defendSkills = new List<IDefendSkill>();
    foreach (var skill in skills){...}

    //리턴 보기1: 방어스킬이 없음
    if (defendSkills.Count == 0){...}

    //방어 스킬들 중 현재 자원량으로 사용 가능한 스킬들을 찾고
    List<IDefendSkill> currentUseAbleDefendSkills = new List<IDefendSkill>();
    foreach (var skill in defendSkills){...}

    //리턴 보기2: 방어스킬은 있으나 현재 자원으로 사용할 수 없음
    if (currentUseAbleDefendSkills.Count == 0){...}

    //자원량을 충족하는 스킬들 중 적의 특정 마지막 상태를 요구하는 스킬들을 찾는다.
    List<IDefendSkill> stateCheckedSkills = new List<IDefendSkill>();
    //자원량을 충족하는 스킬들 중 모든 상황에서 사용가능한 스킬들을 찾는다.
    List<IDefendSkill> noStateCheckedSkills = new List<IDefendSkill>();

    foreach (var skill in currentUseAbleDefendSkills){...}
```

방어 키메라는 공격 키메라가 공격 스킬을 결정하는 것과 같은 로직으로 방어 스킬을 결정합니다.

```
foreach (var skill in currentUseAbleDefendSkills){...}

//리턴 보기3: 사용 가능한 방어스킬 중 특정 적 상태를 요구하는 스킬이 메인스킬임
foreach (IDefendSkill skill in stateCheckedSkills){...}

//리턴 보기4: 사용 가능한 방어스킬 중 특정 적 상태를 요구하지 않는 스킬이 메인스킬
foreach (IDefendSkill skill in noStateCheckedSkills){...}

IDefendSkill defendSkill = null;

//리턴보기5: 사용가능한 방어스킬 중 메인스킬이 방어스킬이 아니고, 특정 적 상태를 요구하는 스킬이 있음.
switch (stateCheckedSkills.Count){...}

//리턴보기6: 사용가능한 방어스킬 중 메인스킬이 방어스킬이 아니고, 모든 상황에서 사용가능한 방어 스킬인 경우
switch (noStateCheckedSkills.Count){...}

//알 수 없는 버그로 여기까지 왔다면, 그냥 취약 리턴
//만약 인 게임이 아닌 다른 방식으로 로그를 저장할 수 있다면 여기서 로그를 보내면 될 듯..
return (_defender.VulnerableState, (IDefendSkill) null);
```

3. 공식/비공식 테스트(적응형 NPC)

```
WinChimeraData = PlayerChimeraData;

Gene gene = new Gene((int)AchieveManager.Instance.PlayerRank);
gene.SetActivationFalse();
GameImmortalManager.Instance.AddAccountGene(gene);

if (_researcherDataScriptableObject is MutantResearcherDataScriptableObject mutantResearcher)
{
    if (ChimeraManager.Instance.AddDevelopmentableChimera(mutantResearcher.RewardMutantChimera))
    {
        OfficialTestUiManager.Instance.SetResultText( opponent: $"{_researcherDataScriptableObject.ResearcherRank}: " +
                                                       $"{_researcherDataScriptableObject.ResearcherName}",
                                                       reward: "랜덤 유전자1개, 돌연변이 권한: " + mutantResearcher.RewardMutantChimera.GeneType, combatResult: "승리");
    }
    else
    {
        OfficialTestUiManager.Instance.SetResultText( opponent: $"{_researcherDataScriptableObject.ResearcherRank}: " +
                                                       $"{_researcherDataScriptableObject.ResearcherName}",
                                                       reward: "랜덤 유전자1개: " + mutantResearcher.RewardMutantChimera.GeneType, combatResult: "승리");
    }
}
else
{
    OfficialTestUiManager.Instance.SetResultText( opponent: $"{_researcherDataScriptableObject.ResearcherRank}: " +
                                                       $"{_researcherDataScriptableObject.ResearcherName}",
                                                       reward: "랜덤 유전자 1개", combatResult: "승리");
}

ResearcherManager.Instance.SetNewChimeraData(_researcherDataScriptableObject);
_totalWinCount++;
if (_researcherDataScriptableObjects.Count < _totalWinCount)
{
    OfficialTestUiManager.Instance.SetResultText( opponent: $"{_researcherDataScriptableObject.ResearcherRank}: {_researcherDataScriptableObject.ResearcherName}",
                                                AchieveManager.Instance.SetOfficialTestAchieveInfo(_totalWinCount);
    OfficialTestUiManager.Instance.SetNextMatchButton(false);
    OfficialTestUiManager.Instance.TurnOnGoMainButton();
    return;
}
```

플레이어에게 패배한 NPC는 새로운 키메라를 발생합니다. 플레이어는 한 번 이겼던 NPC를 다시 만나더라도 패배할 수 있으며 이를 통해 플레이어가 끝 없이 키메라를 발생시키도록 합니다.

1. 플레이어가 상호 테스트에서 승리한 경우, NPC연구원은 새로운 키메라를 발생시킵니다(사진 내 붉은 박스).

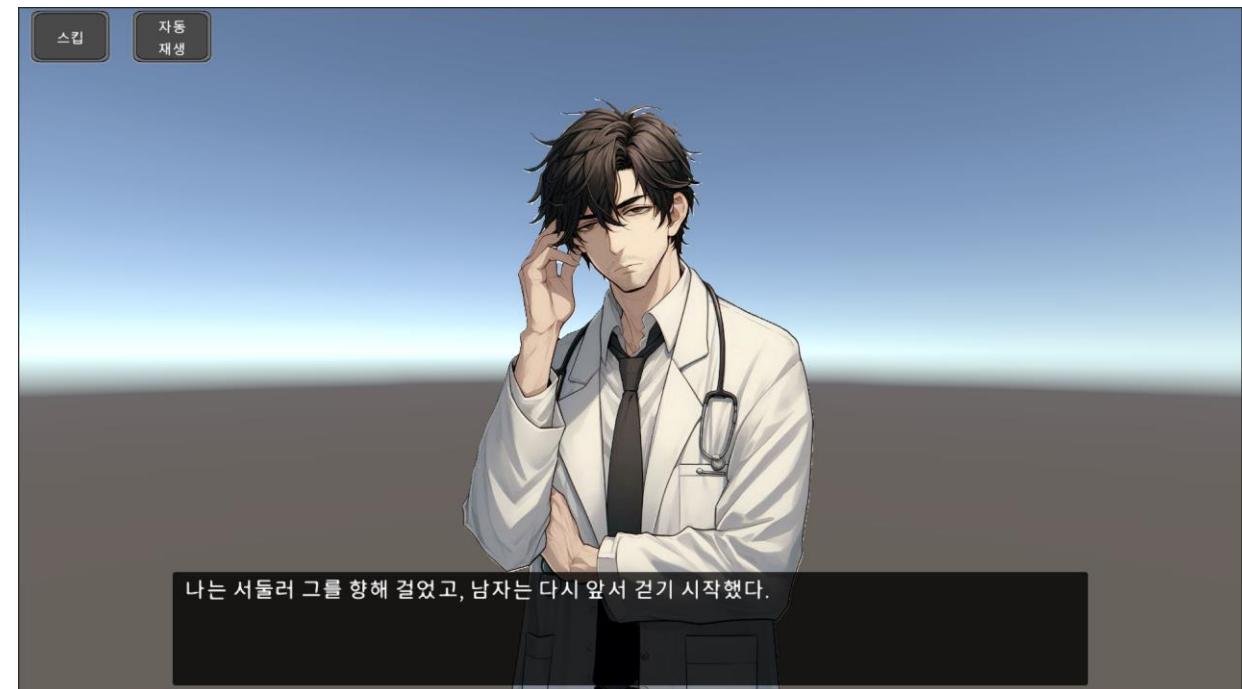
4. 인 게임 스토리(Json활용)

플레이어는 직급별로, 게임을 진행한 인 게임 내 기간별로 스토리를 열람하여 확인할 수 있습니다. 인 게임 내 기간별 스토리(TimeLine)는 매 15일마다 해방되며 직급별 스토리(ResearcherRank)는 플레이어가 특정 직급을 달성하면 해방됩니다. 내 권한 - 배경정보에서 해방된 스토리 중 원하는 스토리를 클릭하여 확인할 수 있습니다. *.Json으로 저장되어 스크립터를 오브젝트로 변환 후 사용됩니다.

1. 배경 정보 화면



2. 타임라인 스토리 예시 화면



* 현재 인 게임 스토리의 캐릭터 이미지는 AI를 통해 생성된 이미지이며 현재 일부 스토리만 캐릭터가 구현되어 있습니다(AI 이미지 생성 크레딧 이슈).

** 로그 기능은 출시 후 추가 예정입니다.

4. 인 게임 스토리(Json활용)

주어지는 선택지에 따라 일부 다른 대사가 출력됩니다. 스kip 버튼과 자동 재생버튼을 통해 해당 기능을 사용할 수 있으며 최초에 자동 재생 모드입니다.

1. 선택지 출력 예시



* 현재 인 게임 스토리의 캐릭터 이미지는 AI를 통해 생성된 이미지이며 현재 일부 스토리만 캐릭터가 구현되어 있습니다(AI 이미지 생성 크레딧 이슈).

** 로그 기능은 출시 후 추가 예정입니다.

4. 인 게임 스토리(Json활용)

주어지는 선택지에 따라 일부 다른 대사가 출력됩니다. 스킵 버튼과 자동 재생버튼을 통해 해당 기능을 사용할 수 있으며 최초에 자동 재생 모드입니다.

1. 선택지 1 예시



2. 선택지 2 예시



* 현재 인 게임 스토리의 캐릭터 이미지는 AI를 통해 생성된 이미지이며 현재 일부 스토리만 캐릭터가 구현되어 있습니다(AI 이미지 생성 크레딧 이슈).

** 로그 기능은 출시 후 추가 예정입니다.

4. 인 게임 스토리(Json활용 - 코드)

```
public enum BackGroundInfoType
{
    ByResearcherRank,
    ByTimeLine
}

[Serializable]public class ScenarioJson
{
    public List<Scenario> ScenarioBooks; // Serializable
}
```

```
[Serializable]public class TimelineScenario : ScenarioJson
{
    public int NeedTimeLineDate; // Serializable
    public TimelineScenario(int needTimeLineDate)
    {
        NeedTimeLineDate = needTimeLineDate;
        ScenarioBooks = new List<Scenario>();
    }
}
```

```
[Serializable]public class ResearcherRankScenario : ScenarioJson
{
    // 질문코드에 따른 시나리오 복 - 딕셔너리로 생성이 안되네
    // 이 데이터가 시간대 스토리인지? 직급 스토리인지? 0 -> 직급, 1 -> 시간대
    public int NeedResearcherRankInt; // Serializable
    public int SortOrder; // Serializable
    private static int Index;

    public ResearcherRankScenario(int needResearcherRankInt)
    {
        NeedResearcherRankInt = needResearcherRankInt;
        Index++;
        SortOrder = Index;
        ScenarioBooks = new List<Scenario>();
    }
}
```

최초에는 계정 데이터를 로컬 파일로 저장하는 것을 목적으로 Json을 활용하려 했지만 계정 데이터는 민감한 정보라고 판단하고 계정 데이터를 저장하는 대신 플레이어에게 게임의 세계관에 몰입하는 데 도움이 되었으면 하는 생각으로 스토리를 작성하였습니다. 최근 플레이하기 시작한 '니케:승리의 여신'의 '상담' 기능을 비슷하게 구현해보았습니다(작성일 기준, AI 이미지 생성 사이트의 크레딧 이유로 일부만 캐릭터 이미지를 구현했습니다).

실제로 출력되는 캐릭터들의 대사는 Scenario의 Sentences입니다. NextSceneNames의 요소들을 다음 Scenario를 결정하는 키로 사용하며 플레이어는 여러 개의 버튼이 출력될 때 특정 버튼을 클릭하여 일부 변경된 스토리를 확인할 수 있습니다.

```
[Serializable]public class Scenario
{
    // 현재 시나리오 이름
    public string SceneName; // Serializable
    // 보여줄 이미지
    public string ImagePath; // Serializable
    // 이게 보기인자 아닌지? 선택지가 나온 때 사용함.
    public bool IsBranch; // Serializable
    // 이 시나리오의 대사를
    public List<string> Sentences; // Serializable
    // 현재 시나리오에서 할 수 있는 다음 시나리오들. 시나리오들의 개수 만큼 버튼 생성
    public List<string> NextSceneNames; // Serializable
}
```

4. 인 게임 스토리(Json활용 - 코드)

```
/*//샘플 파일이 필요한 경우 실행
private void Start()
{
    string folderPath = Path.Combine(Application.persistentDataPath, "ScenarioJson");

    for (int i = 0; i < (int)300/7; i++)
    {
        string filePath = Path.Combine(folderPath, $"{BackGroundInfoType.ByTimeLine.ToString()}_{i}_ScenarioData.json");
        if (IsExistJsonFile(filePath))
        {
            Debug.Log(ReadTimelineScenarioData(i).ScenarioBooks.Count);
            //GameImmortalManager.Instance.SetBackgroundInfoData(ReadTimelineScenarioData(i));
            continue;
        }

        TestTimeLineScenario(i);
        //GameImmortalManager.Instance.SetBackgroundInfoData(ReadTimelineScenarioData(i));
    }

    for (int i = 0; i < 4; i++)
    {
        string filePath = Path.Combine(folderPath, $"{BackGroundInfoType.ByResearcherRank.ToString()}_{i+1}_{i}_ScenarioData.json");
        if (IsExistJsonFile(filePath))
        {
            Debug.Log(ReadResearcherScenarioData(i).ScenarioBooks.Count);
            //GameImmortalManager.Instance.SetBackgroundInfoData(ReadResearcherScenarioData(i));
            continue;
        }

        TestResearcherRankScenario(i);
        //GameImmortalManager.Instance.SetBackgroundInfoData(ReadResearcherScenarioData(i));
    }
}
```

먼저 기본 틀로 사용할 Json파일을 임의로 생성(사진)하고, 그 내부를 직접 채우는 방식으로 구현하였습니다.

Json파일들은 기본적으로 설정된 경로로 저장됩니다. 스토리유형으로 최초로 구분하고 ByTimeLine이라면 해금되는 시기를(300일 기준, 15일이 지날 때마다 해방), ByResearcherRank라면 enum타입의 요구 랭크에 해당하는 int값_정렬순서 규칙으로 저장합니다.

사진의 코드들은 초기에 Json파일 내부 틀을 생성하기 위해 사용했고 더 이상 사용하지 않기 때문에 주석 처리하였습니다.

4. 인 게임 스토리(Json활용 – *.Json)

.idea	2025-02-20 오후 7:51	파일 블더	
ByResearcherRank_0_1_ScenarioData.json	2025-02-19 오후 8:57	JSON File	3KB
ByResearcherRank_1_2_ScenarioData.json	2025-02-19 오후 8:57	JSON File	3KB
ByResearcherRank_2_3_ScenarioData.json	2025-02-19 오후 8:57	JSON File	3KB
ByResearcherRank_3_4_ScenarioData.json	2025-02-20 오전 8:50	JSON File	3KB
ByTimeLine_00_ScenarioData.json	2025-02-19 오후 9:04	JSON File	5KB
ByTimeLine_01_ScenarioData.json	2025-02-20 오후 1:18	JSON File	11KB
ByTimeLine_02_ScenarioData.json	2025-02-19 오후 6:00	JSON File	3KB
ByTimeLine_03_ScenarioData.json	2025-02-19 오후 12:06	JSON File	4KB
ByTimeLine_04_ScenarioData.json	2025-02-19 오후 12:06	JSON File	5KB
ByTimeLine_05_ScenarioData.json	2025-02-20 오전 8:40	JSON File	4KB
ByTimeLine_06_ScenarioData.json	2025-02-19 오후 12:06	JSON File	6KB
ByTimeLine_07_ScenarioData.json	2025-02-19 오후 12:06	JSON File	5KB
ByTimeLine_08_ScenarioData.json	2025-02-19 오후 12:07	JSON File	5KB
ByTimeLine_09_ScenarioData.json	2025-02-19 오후 12:07	JSON File	5KB
ByTimeLine_10_ScenarioData.json	2025-02-19 오후 12:07	JSON File	3KB
ByTimeLine_11_ScenarioData.json	2025-02-19 오전 11:41	JSON File	6KB
ByTimeLine_12_ScenarioData.json	2025-02-19 오전 11:41	JSON File	3KB
ByTimeLine_13_ScenarioData.json	2025-02-19 오전 11:41	JSON File	2KB
ByTimeLine_14_ScenarioData.json	2025-02-19 오전 11:41	JSON File	3KB
ByTimeLine_15_ScenarioData.json	2025-02-19 오전 11:41	JSON File	3KB
ByTimeLine_16_ScenarioData.json	2025-02-19 오전 11:41	JSON File	7KB
ByTimeLine_17_ScenarioData.json	2025-02-19 오전 11:41	JSON File	4KB
ByTimeLine_18_ScenarioData.json	2025-02-20 오전 8:39	JSON File	4KB
ByTimeLine_19_ScenarioData.json	2025-02-19 오전 11:37	JSON File	3KB
ByTimeLine_20_ScenarioData.json	2025-02-20 오전 9:01	JSON File	6KB
ByTimeLine_21_ScenarioData.json	2025-02-20 오후 7:32	JSON File	6KB

틀을 만들고 저장한 뒤, 실제로 테스트를 진행해봤습니다.
이 때, 단순히 0,1,2 ~ 10,11.. 순으로 저장하면(서식을 정해주지
않으면) 순서가 꼬이는 것을 확인하고 형식을 일치시켰습니다.

실제 Json의 내부 형식은 아래와 같습니다.

```
1  {
2   "ScenarioBooks": [
3     {"SceneName": "헬기에서 내린다.", "ImagePath": "Kang", "IsBranch": false, "Sentences": ["여자의 목소리가 들리자마자 승강기 문이 천천히 달렸다."], "NextSceneNames": []}, {"SceneName": "\"네, 맞습니다.\\"", "ImagePath": "Kang", "IsBranch": false, "Sentences": ["여자의 목소리가 들리자마자 승강기 문이 천천히 달렸다."], "NextSceneNames": []}, {"SceneName": "\"아니요.\\"", "ImagePath": "Kang", "IsBranch": false, "Sentences": ["여자의 목소리가 들리자마자 승강기 문이 천천히 달렸다."], "NextSceneNames": []}, {"SceneName": "\"생물학입니다.\\"", "ImagePath": "Kang", "IsBranch": false, "Sentences": ["여자의 목소리가 들리자마자 승강기 문이 천천히 달렸다."], "NextSceneNames": []}, {"SceneName": "\"예, 맞습니다.\\"", "ImagePath": "Kang", "IsBranch": false, "Sentences": ["여자의 목소리가 들리자마자 승강기 문이 천천히 달렸다."], "NextSceneNames": []}, {"SceneName": "\\"그럼 무문을 빌게요.\\"", "ImagePath": "Kang", "IsBranch": false, "Sentences": ["여자의 목소리가 들리자마자 승강기 문이 천천히 달렸다."], "NextSceneNames": []}, {"SceneName": "\\"ImagePath": "Kang", "IsBranch": false, "Sentences": ["여자의 목소리가 들리자마자 승강기 문이 천천히 달렸다."], "NextSceneNames": []}, {"SceneName": "\\"IsBranch": false, "IsBranch": false, "Sentences": ["여자의 목소리가 들리자마자 승강기 문이 천천히 달렸다."], "NextSceneNames": []}, {"SceneName": "\\"Sentences": [", "NextSceneNames": []}, {"SceneName": "\\"NeedTimeLineDate": 0}], "NeedTimeLineDate": 0}
```

5. 그 외(씬 전환)

```
private IEnumerator SetActiveSceneAsync(string additiveSceneName, Scene unloadingScene)
{
    loadingSceneObject.SetActive(true); //<- 로딩용 캔버스

    // 이전 씬 언로드
    yield return StartCoroutine(routine: UnloadSceneAsync(unloadingScene)); //언로딩을 먼저 해야해요!

    // 비동기 로드 시작- 어싱크로오레이션은 현재 씬이 얼마나 로드되었는지 알려주는 객체다
    AsyncOperation uiSceneOperation = SceneManager.LoadSceneAsync(additiveSceneName, LoadSceneMode.Additive);

    //사용자의 입력 막기
    sceneEventSystem.gameObject.SetActive(false);
    //sceneEventSystem = null;

}
}
```

```
//Debug.Log(unloadingScene.name);
// 새 씬 활성화
Scene additiveScene = SceneManager.GetSceneByName(additiveSceneName);
SceneManager.SetActiveScene(additiveScene);

// EventSystem 재설정
//sceneEventSystem = FindFirstObjectOfType<EventSystem>();
sceneEventSystem.gameObject.SetActive(true);
//Debug.Log(SceneManager.GetActiveScene().name);

//NotifyObservers(additiveSceneName);
loadingSceneObject.SetActive(false);
```

AdditiveScene, (Un)LoadSceneAsync를 활용한 씬 전환을 구현했습니다.

씬 전환이 일어날 때, 절대로 사라지지 않는 ImmortalScene의 로딩 씬 용 캔버스에서 로딩 씬 이미지를 켜고 씬을 언로드하는 동안 대기하며, 언로드가 끝나면 이어서 로드를 진행합니다. 로드가 끝나면 SetActiveScene함수로 로드 된 씬을 ActiveScene으로 설정한 뒤 로딩 씬 용 캔버스를 끕니다(처음과 마지막 부분).

5. 그 외(다회 차 플레이)

```
private void ResetGameResources()
{
    RemainedDay = BasicRemainDay;
    GeneSupplyTokenCount = 0;
    ChimeraSupplyTokenCount = 0;

    AccountUseAbleGene.Clear(); //이건 정상작동

    AllGenes.Clear();
    AllChimeras.Clear();
    DictionaryChimera.Clear();

    for (int i = 0; i < Enum.GetValues(typeof(GeneType)).Length - 1; i++)
    {
        AllGenes.Add(new Gene((GeneType)i));
    }
}
```

```
for (int i = 0; i < Enum.GetValues(typeof(GeneType)).Length - 1; i++)
{
    AllGenes.Add(new Gene((GeneType)i));
}

ChimeraManager.Instance.DataReset();
for (int i = 0; i < basicChimeras.Count; i++)
{
    DictionaryChimera.Add(basicChimeras[i].GeneType, basicChimeras[i]);
    if (!ChimeraManager.Instance.AddDevelopmentableChimera(basicChimeras[i])) break;
}

AllChimeras.AddRange(basicChimeras);
AllChimeras.AddRange(rewardChimera);

//여기는 왜 안 정상작동? -> 중간 for문에 return있었음...
Gene.DataReset();
AchieveManager.Instance.DataReset();

}
```

게임을 종료하지 않고 재시작하는 경우 데이터들을 리셋합니다.

런타임에서 스크립터블 오브젝트를 임의로 삭제할 수 없는 것을 확인했습니다(데이터 손실 방지를 위해 삭제할 수 없다는 에러로그 확인).

업데이트(또는 리팩토링)할 때 메모리 관리를 위해 수정해야 할 사항입니다. 이는 증여에서도 같을 것으로 생각합니다(증여는 플레이어의 키메라 데이터를 리스트에서 Remove합니다).

5. 그 외(매니저 구분)

```
public class UiSoundManager : ImmortalObject<UiSoundManager>
{
    public class AchieveUiManager : MortalManager<AchieveUiManager>, IGoMain
    {
        [SerializeField] private Button achieveButton; // AchieveButton (Button)

        [SerializeField] private Transform officialTestArchiveContent; // Content (Transform)
        [SerializeField] private Transform nonOfficialTestVictoryArchiveContent; // Content
```

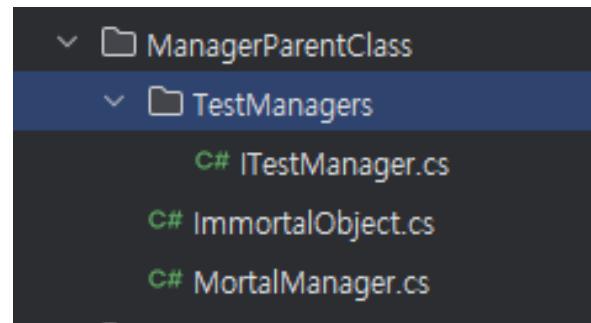
```
public class ImmortalObject<T> : MonoBehaviour where T : MonoBehaviour
{
    // Frequently called 306 usages More...
    public static T Instance { get; private set; }

    // Event function 11 usages 11 overrides
    protected virtual void Awake()
    {
        if (Instance == null)
        {
            // 컴파일 시점에서 T가 어떤 클래스인지 알 수 없다.
            // 이 오브젝트에 들어있는 스크립트의 T(ImmortalObject<T>)는 가지고온다.
            Instance = GetComponent<T>();
            if (Instance == null)
            {
                Instance = gameObject.AddComponent<T>();

                DontDestroyOnLoad(gameObject);
            }
            else
            {
                DestroyImmediate(gameObject);
            }
        }
        //Debug.Log(Instance.gameObject.name);
    }
}
```

```
No asset usages 19 usages 18 inheritors 1 exposing API
public class MortalManager<T> : MonoBehaviour where T : MortalManager<T>
{
    // Frequently called 41 usages
    public static T Instance {get; private set;}

    // Event function 13 usages 13 overrides
    protected virtual void Awake()
    {
        if (Instance == null)
        {
            Instance = GetComponent<T>();
            if (Instance == null)
            {
                Instance = gameObject.AddComponent<T>();
            }
            //Debug.Log(Instance.gameObject.name);
        }
    }
}
```



싱글톤 패턴을 활용하여 절대로 사라지지 않는 매니저와 UI, 특정 씬에서만 작동하는, 사라져도 되는 매니저를 구분하여 구현했습니다. 각 부모 클래스는 싱글톤의 형식만 구현하고 있습니다.

테스트 매니저는 공식/비공식 테스트 매니저를 구현 시 사용됩니다.

```
public interface ITestManager
{
    // 2 implementations
    public ChimeraData PlayerChimeraData { get; }
    // 2 implementations
    public ChimeraData OpponentChimeraData { get; }
    // 2 implementations
    public ChimeraData WinChimeraData { get; }
    // 2 implementations
    public Chimera PlayerChimera { get; }
    // 2 implementations
    public Chimera OpponentChimera { get; }
    // 2 implementations
    public CombatMachine CombatMachine { get; }
    // Frequently called 3 usages 2 implementations
    public void WinnerIs(Chimera chimera);
}
```

5. 그 외(스크립터블 오브젝트)

스크립터블 오브젝트를 데이터컨테이너로 사용하면서 동적으로 생성되는 스크립터블 오브젝트의 경우 런타임에서 실행해야 할 함수를 추가하여 활용했습니다(키메라데이터, NPC데이터).

```
[CreateAssetMenu(fileName = "BaseStatus", menuName = "Scriptable Obj
[CreateAssetMenu(fileName = "FeatureList", menuName = "Scriptable
public class BaseStatus : ScriptableObject
{
    [SerializeField] private int maxHealthPoint; &gt; 2000
    public int MaxHealthPoint => maxHealthPoint;
    [SerializeField] private int attackPoint; &gt; Changed in 12 assets
    public int AttackPoint => attackPoint;
    [SerializeField] private int defencePoint; &gt; Changed in 12 assets
    public int DefencePoint => defencePoint;
    [SerializeField] private int agilityPoint; &gt; Changed in 12 assets
    public int AgilityPoint => agilityPoint;
}

public class FeatureList : ScriptableObject
{
    [SerializeField] private GeneType geneType; &gt; Changed in 9 assets
    public GeneType GeneType => geneType;
    [SerializeField] private Feature[] features = new Feature[3];
    public Feature[] Features => features;
    [SerializeField] private string description; &gt; Unchanged
    public string Description => description;
}
```

```
[CreateAssetMenu(fileName = "ChimeraData", menuName = "
public class ChimeraData : ScriptableObject
{
    public Chimera Chimera { get; private set; }
    public FeatureList FeatureInfo { get; private set; }
    public BaseStatus BaseStatus { get; private set; }

    public void SetChimeraData(Chimera chimera, FeatureList featureInfo, BaseStatus baseStatus)
    {
        chimera.chimeraData = this;
        chimera.featureInfo = featureInfo;
        chimera.baseStatus = baseStatus;
    }

    public void SetCoefficientToStatus()
    {
        foreach (var coefficient in coefficients)
        {
            if (coefficient != null)
            {
                coefficient.SetCoefficient();
            }
        }
    }

    public Chimera CagedInstantiateChimera()
    {
        return Instantiate(chimera);
    }

    public Chimera InstantiateChimera(Vector3 spawnPoint = default)
    {
        return Instantiate(chimera, spawnPoint);
    }
}
```

```
[CreateAssetMenu(fileName = "ResearcherDataScriptableObject", menuName = "Scr
public class ResearcherDataScriptableObject : ScriptableObject
{
    [SerializeField] private GeneType chimeraGeneType; &gt; Changed in 28+ assets
    public GeneType ChimeraGeneType => chimeraGeneType;
    [SerializeField] private string researcherName; &gt; Unchanged
    public string ResearcherName => researcherName;
    [SerializeField] private ResearcherRank researcherRank; &gt; Changed in 28+ assets
    public ResearcherRank ResearcherRank => researcherRank;

    public void SetChimeraData(bool isMutantResearcher = false)
    {
        if (isMutantResearcher)
        {
            SetChimeraData((Chimera)chimeraGeneType);
        }
        else
        {
            SetChimeraData((Chimera)chimeraGeneType);
        }
    }
}
```

```
[CreateAssetMenu(fileName = "MutantResearcherDataScriptableObject", menuName = "Scri
public class MutantResearcherDataScriptableObject : ResearcherDataScriptableObject
{
    [SerializeField] private Chimera rewardMutantChimera; &gt; Changed in 6 assets
    public Chimera RewardMutantChimera => rewardMutantChimera;
}
```

개발기간

약 1달 정도를 소요했습니다.

최초 기획 후 첫 2주 동안 스크립트를 작성했습니다. 구현에 대한 생각이 잘 떠오르지 않으면 공책을 활용하여 추가 기획 및 기획 수정을 위한 생각을 정리하는 과정을 주기적으로 거쳤습니다.

3주차부터 에셋 탐색, 구매 및 적용과 디버깅을 병행하고 4주차부터 테스트와 디버깅을 병행하면서 편의성을 개선하기 시작했습니다. 결과적으로 1월20일에 시작하여 2월21일에 해당 문서를 작성하게 되었습니다.

 키메라(chimera) 시뮬레이터.pptx	2025-02-20 오후 9:14	Microsoft PowerP...	3,278KB
 키메라(chimera) 시뮬레이터 - 복사본 (24).pptx	2025-02-19 오후 11:13	Microsoft PowerP...	3,278KB
 키메라(chimera) 시뮬레이터 - 복사본 (23).pptx	2025-02-18 오후 10:30	Microsoft PowerP...	3,276KB
 키메라(chimera) 시뮬레이터 - 복사본 (22).pptx	2025-02-16 오후 10:32	Microsoft PowerP...	3,274KB
 키메라(chimera) 시뮬레이터 - 복사본 (21).pptx	2025-02-15 오후 10:13	Microsoft PowerP...	3,273KB
 키메라(chimera) 시뮬레이터 - 복사본 (20).pptx	2025-02-14 오전 10:24	Microsoft PowerP...	3,270KB
 키메라(chimera) 시뮬레이터 - 복사본 (19).pptx	2025-02-13 오후 10:09	Microsoft PowerP...	3,270KB
 키메라(chimera) 시뮬레이터 - 복사본 (18).pptx	2025-02-13 오전 1:02	Microsoft PowerP...	3,267KB
 키메라(chimera) 시뮬레이터 - 복사본 (17).pptx	2025-02-11 오후 7:00	Microsoft PowerP...	3,267KB
 키메라(chimera) 시뮬레이터 - 복사본 (16).pptx	2025-02-10 오후 11:41	Microsoft PowerP...	3,266KB
 키메라(chimera) 시뮬레이터 - 복사본 (15).pptx	2025-02-09 오후 10:01	Microsoft PowerP...	3,265KB
 키메라(chimera) 시뮬레이터 - 복사본 (14).pptx	2025-02-07 오후 10:56	Microsoft PowerP...	3,263KB
 키메라(chimera) 시뮬레이터 - 복사본 (13).pptx	2025-02-05 오후 10:36	Microsoft PowerP...	3,285KB
 키메라(chimera) 시뮬레이터 - 복사본 (12).pptx	2025-02-04 오후 8:23	Microsoft PowerP...	3,251KB
 키메라(chimera) 시뮬레이터 - 복사본 (11).pptx	2025-02-03 오후 6:28	Microsoft PowerP...	3,245KB
 키메라(chimera) 시뮬레이터 - 복사본 (10).pptx	2025-02-03 오전 7:36	Microsoft PowerP...	3,243KB
 키메라(chimera) 시뮬레이터 - 복사본 (9).pptx	2025-02-02 오후 7:09	Microsoft PowerP...	3,243KB
 키메라(chimera) 시뮬레이터 - 복사본 (8).pptx	2025-02-01 오후 11:45	Microsoft PowerP...	3,241KB
 키메라(chimera) 시뮬레이터 - 복사본 (7).pptx	2025-01-30 오후 8:53	Microsoft PowerP...	3,223KB
 키메라(chimera) 시뮬레이터 - 복사본 (6).pptx	2025-01-28 오후 8:10	Microsoft PowerP...	3,214KB
 키메라(chimera) 시뮬레이터 - 복사본 (5).pptx	2025-01-24 오후 7:52	Microsoft PowerP...	3,208KB
 키메라(chimera) 시뮬레이터 - 복사본 (4).pptx	2025-01-24 오전 11:26	Microsoft PowerP...	3,202KB
 키메라(chimera) 시뮬레이터 - 복사본 (3).pptx	2025-01-22 오후 9:43	Microsoft PowerP...	3,189KB
 키메라(chimera) 시뮬레이터 - 복사본 (2).pptx	2025-01-21 오후 6:25	Microsoft PowerP...	2,367KB
 키메라(chimera) 시뮬레이터 - 복사본.pptx	2025-01-20 오후 9:25	Microsoft PowerP...	2,361KB

소감

최근에 개임 개발 프로세스와 관련된 책을 구매하고 읽으면서 보았던 개발자용 치트코드를 적용한 것이 마지막 4주차에 매우 유효했습니다(하단 이미지).

원래는 테스트하는 과정을 병행한다는 생각으로 개발을 진행했으나 많이 느리다는 것을 체감했던 기억이 납니다.

기획, 설계, 아트 등 다양한 분야의 전문가들이 필요하다는 것을 정말 절실하게 느꼈습니다. 기획과 설계는 정말 많은 생각을 요구하여 조금이라도 수정되면 실제로 구현할 때 많은 부분이 변경되게 되었습니다.

아트나 사운드의 경우에는 게임의 세계관에 어울리는 에셋들을 구하는 것이 힘들었습니다. 해당 분야에 종사하시는 분들이 프로젝트에 참여하여 오직 그 프로젝트를 위한 창작물을 만드는 것이 게임의 완성도에 크게 기여한다는 것을 느꼈습니다.

밸런스 부분에서도 꽤나 어려움을 겪었습니다.

이 게임에서 VenomSnake는 유일하게 적을 약화시키는 오브젝트인데 이 정도가 매우 강력해서 많게는 10개의 유전자 차이가 나는 경우에도 승리하는 모습을 종종 볼 수 있었기 때문입니다(해당 부분은 약화 정도를 낮춤으로써 해결했습니다).

```
if (AchieveManager.Instance.PlayerRank.Equals(ResearcherRank.Junior)&&Input.GetKeyDown(KeyCode.Q))
{
    if (Input.GetKeyDown(KeyCode.Semicolon))
    {
        RemainedDay = 100;
        for (int i = 0; i < 46; i++)
        {
            AchieveManager.Instance.SetReallocationAchieveInfo();
            AchieveManager.Instance.SetDevelopmentChimeraAchieveInfo();
            AchieveManager.Instance.SetNonOfficialTestVictoryAchieveInfo();
            AchieveManager.Instance.SetNonOfficialTestDefeatAchieveInfo();
        }

        for (int i = 0; i < 78; i++)
        {
            AddAccountGene(new Gene((int)AchieveManager.Instance.PlayerRank));
        }
        LobbyUiManager.Instance.WhenCheat(RemainedDay, GeneSupplyTokenCount, ChimeraSupplyTokenCount);
    }

    if (Input.GetKeyDown(KeyCode.W))
    {
        AchieveManager.Instance.SetReallocationAchieveInfo();
        AchieveManager.Instance.SetDevelopmentChimeraAchieveInfo();
        AchieveManager.Instance.SetNonOfficialTestVictoryAchieveInfo();
        AchieveManager.Instance.SetNonOfficialTestDefeatAchieveInfo();
    }
}

if (Input.GetKey(KeyCode.A))
{
    if (Input.GetKeyDown(KeyCode.Semicolon))
    {
        RemainedDay = 1;
        LobbyUiManager.Instance.WhenCheat(RemainedDay, GeneSupplyTokenCount, ChimeraSupplyTokenCount);
    }
}
```

이후계획

유니티 관련 국비교육을 이수하고 유니티를 활용하여 게임을 개발하기 시작한 것이 정확히 1년이 되었습니다. 해당 교육과정에서 주로 사용했던 컴포넌트들 외에도 사용하지 않았던 컴포넌트들을 적극 활용하여 유니티의 사용능력을 높일 예정입니다.

2월17일 가산에서 진행된 유니티6 데모 쇼케이스에 참여하여 추가된 기능들을 정리하고 해당 기능들을 위주로 사용해볼 계획입니다. 특히 잘 활용하지 않았던 Light를 다뤄 게임을 제작해보겠습니다.

그 외에도 쉐이더와 VFX를 다뤄볼 예정입니다.

해당 게임은 스팀에 무료로 업로드 예정이며 전체 코드는 깃에 업로드 되어있습니다. 현재 스팀에 심의 및 오류가 있어 깃에 플레이 가능한 파일을 같이 올렸습니다.

[깃 바로가기](#)

[플레이 영상 바로가기](#)