2D 포트폴리오 최종

제목 프로메테우스 미션
분류 시뮬레이션
작성자 김호곤

목차

가. 개요

Unity를 기반으로 제작한 테라포밍 시뮬레이션이다. 플레이어는 테라포밍 프로젝트를 총괄하는 한 우주 항공 기업의 AI 시점으로 플레이하는데, 승리 조건은 신 행성의 주권을 차지하는 것으로, 방법은 테라포밍을 완료하여 최다 공로를 인정받거나 다른 모든 경쟁 기업을 잠식하는 것이다.

나. 개발환경

1. Unity 2021.3.25.f1
2. 비쥬얼스튜디오 2022

다. 스크립트 구현도

1. 스크립트 폴더

스크립트는 용도별로 여러 폴더에 넣었다.

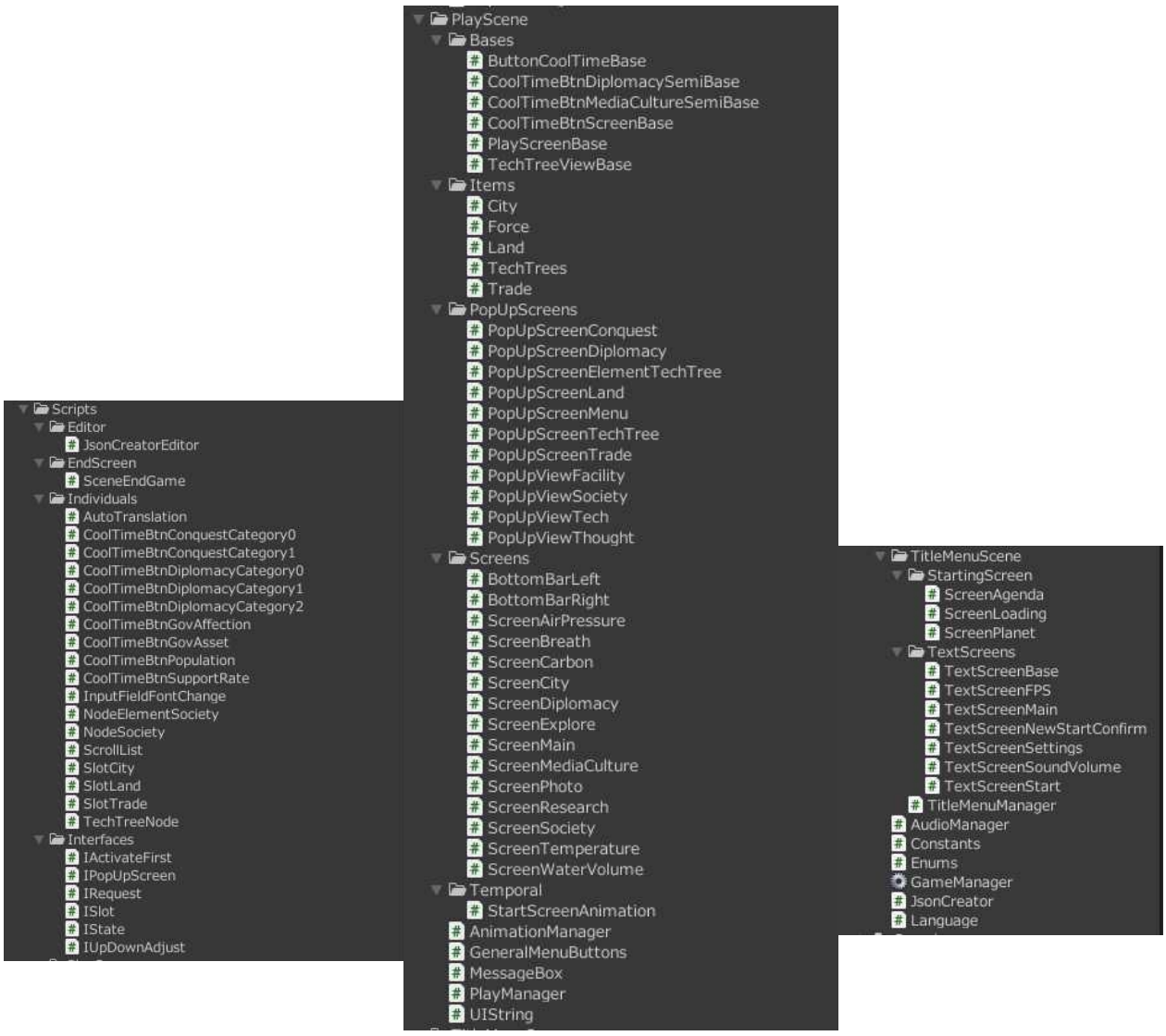

그림 다.1.0

스크립트 내의 각 구조는 아래와 같은 색으로 구분한다.

그림 다.1.1

2. Scripts\Interfaces

인터페이스를 모은 폴더다. C#은 클래스를 다중상속할 수 없기 때문에 여러 클래스에서 공통으로 사용할 동작은 인터페이스로 선언한다.

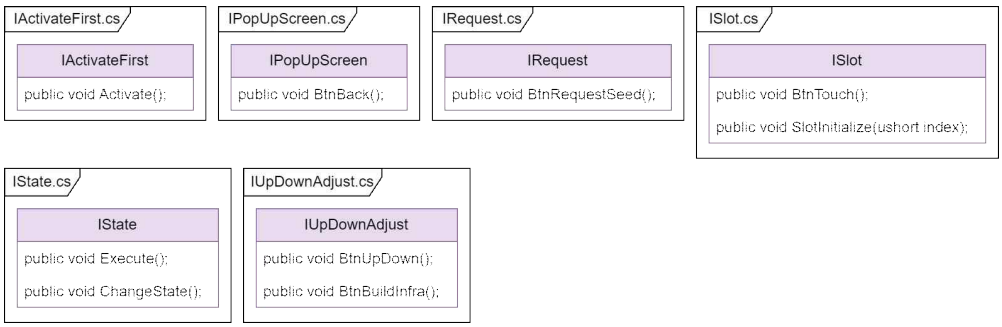

그림 다.2.0

가) IActivateFirst

MonoBehaviour를 상속받는 클래스 중에 게임 시작하자마자 동작해야 될 것이 있는 경우 이 인터페이스를 상속받는다. 그러면 해당 오브젝트가 비활성화된 상태여도 다른 클래스에서 Activate 함수를 대신 호출해주는 것으로 초기화 동작을 할 수 있다.

나) IPopUpScreen

기본적인 메뉴화면이 아니라 메뉴화면의 메뉴화면처럼 추가적인 화면을 표시하는 클래스는 이 인터페이스를 상속받는다. 이 기능에서 공통으로 필요한 동작은 아직까지 뒤로 가기 기능밖에 없다.

다) IRequest

게임 컨셉 상 모행성으로부터 생물 종자를 요청하는 클래스는 이 인터페이스를 상속받는다. 물론 이 인터페이스를 상속받는 클래스는 2개밖에 없기 때문에 이 인터페이스의 존재가 크게 유의미하지 않다.

라) ISlot

인벤토리의 슬롯처럼 슬롯의 동작을 하는 클래스는 이 인터페이스를 상속받는다. 이 게임에는 인벤토리라는 것은 없으나 그와 유사한 목록은 있다. BtnTouch 함수에서는 슬롯 클릭 시 동작이 정의될 것이고, SlotInitialize 함수에서는 슬롯을 초기화하는데 매개변수 index를 받는다. 슬롯 정보는 인덱스 번호로 결정될 것이기 때문에 이름을 index로 지었고, 인덱스 번호는 음수가 될 수 없으므로 unsigned short를 써서 int형을 쓸 때보다 메모리를 약간 아꼈다.

마) IState

상태패턴을 위한 인터페이스다. Execute 함수는 상태에 대한 동작을 할 것이고, ChangeState는 상태 변경을 할 것이다. 여러 유한상태기계에서 공통으로 사용할 인터페이스이므로 어떤 상태로 변경할 것인지는 매개변수로 받지 않고 각 상태 클래스가 스스로 정할 것이다.

바) IUpDownAdjust

게임 컨셉에서 어떤 물리량을 조절할 것인데, 그 동작을 할 클래스에서 상속받을 인터페이스다.

3. Scripts\TitleMenuScene

주 화면 관련 폴더다.

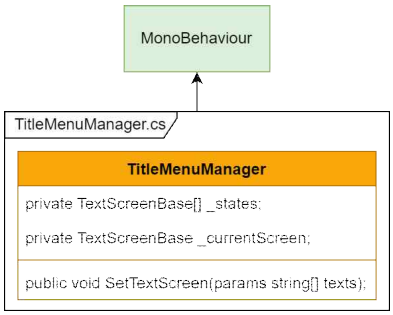

그림 다.3.0

가) TitleMenuManager

이 클래스는 상태기계가 될 것이다. TextScreen은 상태 클래스고, TitleMenuManager는 각 상태 클래스의 부모 형식을 참조한다. 상태 클래스는 상태에 대한 동작을 할 때 SetTextScreen 함수를 호출해서 원하는 화면을 표시한다.

4. Scripts\TitleMenuScene\TextScreen

주 화면의 상태 클래스를 담은 폴더다.

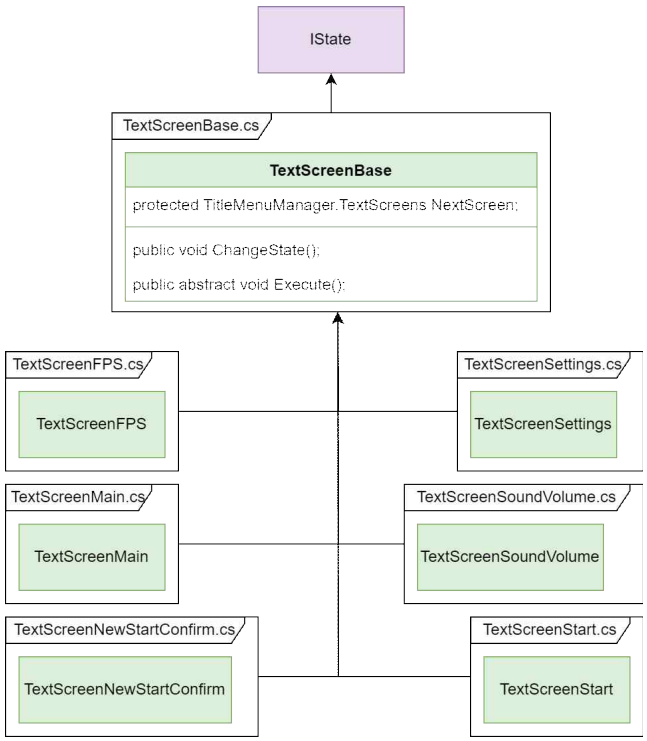

그림 다.4.0

가) TextScreenBase

모든 주 화면 상태 클래스의 부모클래스로 TitleMenuManager가 참조할 형식이다. ChangeState, Execute 함수는 인터페이스로부터 왔기 때문에 public이고 ChangeState는 이 클래스에서 정의할 수 있다. 대신 어느 상태로 전이할 것인지는 NextScreen으로 결정하고, 그 결정을 자식 클래스가 하기 위해 접근 제한자를 protected로 했다. NextScreen의 형식 TitleMenuManager.TextScreens는 TitleMenuManager에 있는 열거형이다.

나) TextScreen

주 화면의 상태 클래스다. 어떻게 화면을 표시할지와 어떤 동작을 할지 이 클래스에 정의되고 MainMenuManager는 부모 형식의 클래스를 참조하고 추상화 함수를 호출하는 것으로 객체지향 프로그래밍의 다형성이 적용된다.

5. Scripts\TitleMenuScene\StartingScreen

TextScreen은 옛날 콘솔 화면 같은 디자인을 위한 상태 클래스인데, 그런 디자인으로는 구현하기 불편한 기능이 있어서 콘솔 화면 디자인이 아닌 화면을 표시할 클래스를 만들었다.

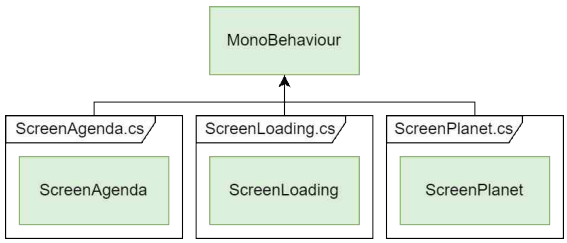

그림 다.5.0

각 클래스는 각자의 역할을 수행한다. ScreenLoading은 활성화됐을 때 자동으로 로딩화면 애니메이션을 보여주고 다음 씬을 불러온다.

6. Scripts\PlayScene

게임 플레이 관련 폴더다.

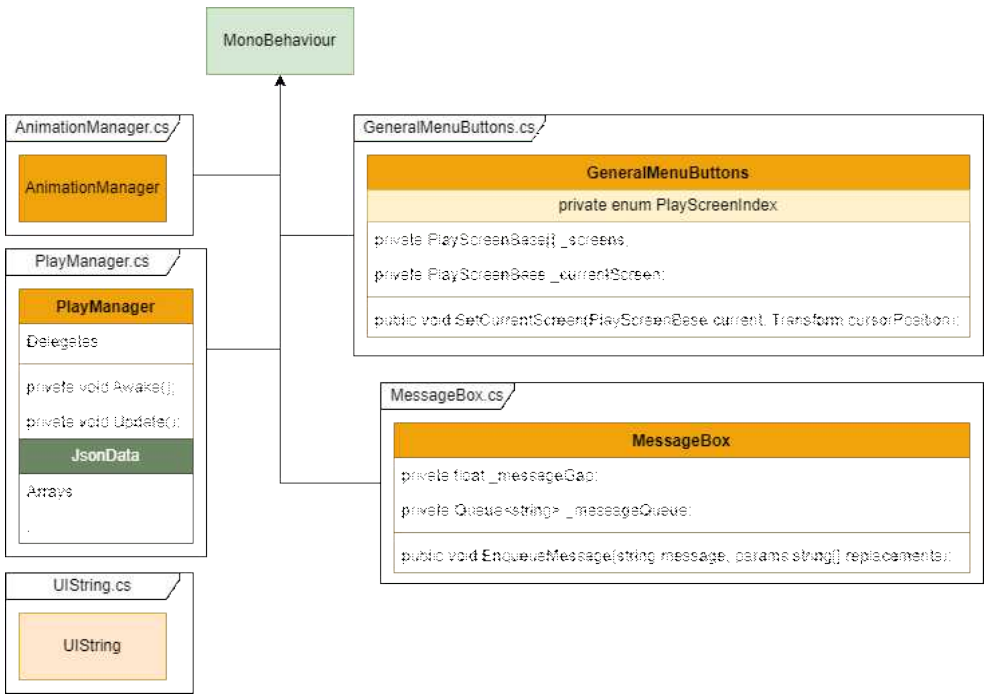

그림 다.6.0

가) AnimationManager

게임 플레이에 영향이 없는 애니메이션을 모아놓은 클래스다. 애니메이션 관련 값은 SerializeField를 통해 유니티 에디터에서 조절이 가능하다.

나) GeneralMenuButtons

플레이 씬에서 기본 화면의 상태기계다. 부모 형식인 PlayScreenBase를 참조하고 참조되는 상태 클래스는 SetCurrentScreen을 호출한다. 각 상태 클래스는 유니티 UI와 연결되어있기 때문에 화면의 동작을 스스로 처리하므로 주 화면에서와는 다르게 여기서는 다형성이 적용되지 않는다. 다만 상태 클래스가 스스로 처리할 수 있는 동작은 UI 동작밖에 없고, 상태패턴이라 부르기에 이 상태 클래스는 전이 조건, 전이할 상태를 스스로 결정할 수 없다. 그렇다보니 IState 인터페이스를 상속받아 Execute, ChageState 함수를 정의할 때도 각 상태 클래스 고유의 정의가 있을 수가 없어서 부모 클래스인 PlayScreenBase에서 정의할 수 있다. 결국 GeneralMenuButtons의 '상태기계'로써의 존재나 PlayScreen의 '상태'로써의 존재는 무의미하고 상태 패턴은 제거돼도 무방하나, 지금은 프로그래밍 연습이고, 또 코드를 약간 간략화할 수 있어서 형식적으로만 존재하고 있다. 그러나 다음 프로젝트에서는 이런 상황에 상태패턴을 적용할 것 같지 않다.

다) MessageBox

게임 상에서 사용자한테 정보 제공을 위한 메시지 창을 표시할 클래스다. 메시지가 한꺼번에 여러 개 표시되면 사용자가 확인을 못하므로, 메시지는 EnqueueMessage 함수를 통해 _messageQueue라는 큐 자료구조에 추가되고, 한 번 메시지 출력 후에 다음 메시지는 _messageGap의 값만큼 기다렸다가 표시한다.

라) PlayManager

기본적인 게임 루프는 PlayManager의 Update 함수에서 구현된다. 외부 클래스는 PlayManager의 Update 함수를 사용하기 위해 PlayManager에서 제공하는 대리자를 사용할 것이다. 또한 공통으로 사용할 변수, 함수는 모두 PlayManager에 선언된다. PlayManager 내에 있는 JsonData 구조체는 게임 저장 시 저장해야 될 변수를 담는다. 이 구조체는 오직 PlayManager만 생성할 것이므로 private로 제한돼있고, 값변수로 저장되기 때문에 참조변수를 쓸 때보다는 참조 회수를 한 번 줄일 수 있다. 게임 컨셉 상 저장할 변수는 매우 많기 때문에 이 구조체는 모든 변수를 같은 자료형끼리 배열로 묶어서 저장한다. 그러면 이 구조체는 변수 구분이 필요 없고, 대신 PlayManager가 구분해준다.

마) UIString

사용자에게 물리량 같은 정보를 표시할 때, 그 정보는 여러 화면에 반복해서 표시할 수 있다. 그런 경우, 표시할 문자열을 각 화면마다 따로 생성하는 것보다 한 문자열을 생성해서 그 문자열을 참조하는 것이 더 효율적일 것으로 생각해서 만든 클래스다. 이 클래스는 공통으로 사용할 문자열을 관리한다. 예로 누군가가 어떤 문자열을 가져오고 싶을 때, UIString 클래스는 해당 정보를 읽어오고 그 정보를 문자열로 만들어 반환한다. 그리고 읽어온 정보를 저장하고 있다가 다음 누군가가 그 문자열을 가져오고 싶을 때, UIString은 그 정보 값이 변했는지 확인하고 변하지 않았으면 이전에 생성한 문자열을 반환한다.

7. Scripts\PlayScene\Bases

부모 클래스 형식을 모은 폴더다.

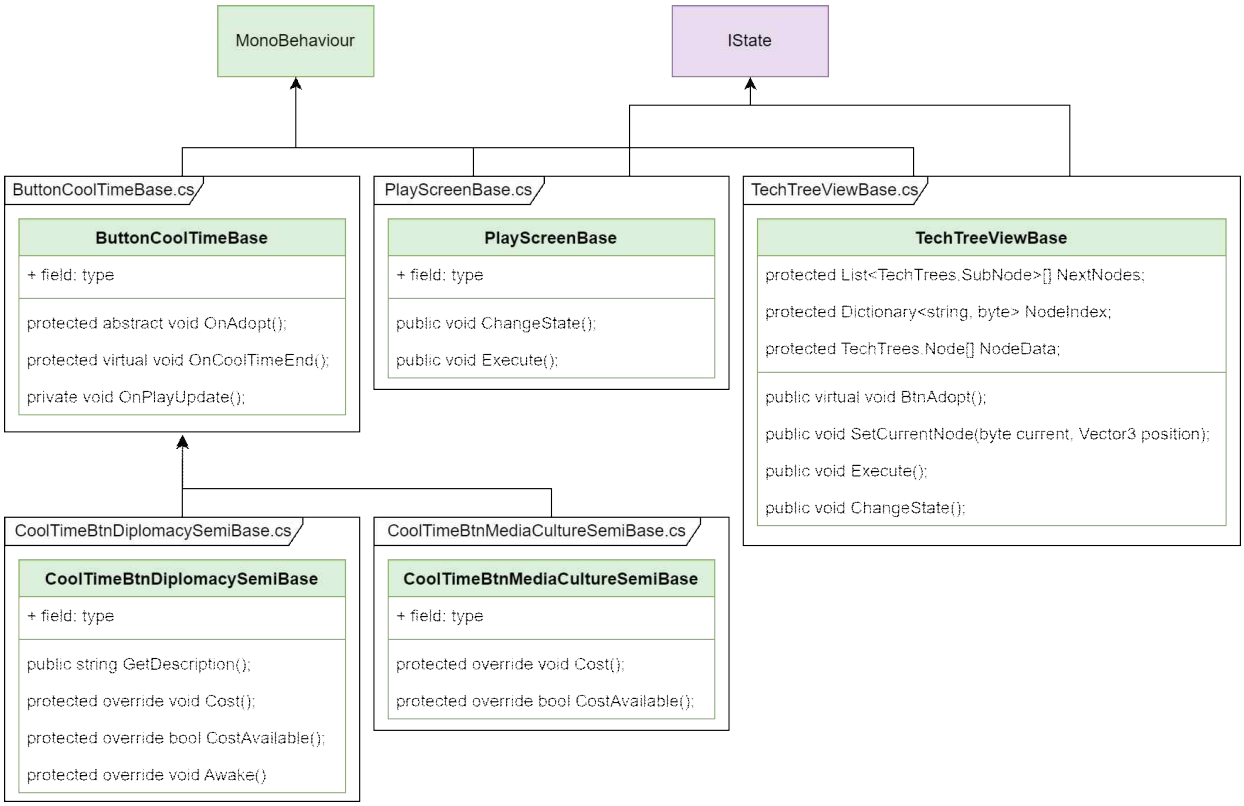

그림 다.7.0

가) ButtonCoolTimeBase

재사용 대기시간이 있는 버튼은 이 클래스를 상속받는다. OnAdopt 함수에서 클릭 시의 동작을 하고, OnCoolTimeEnd 함수에서 재사용 대기가 끝났을 때의 동작을 하고, OnPlayUpdate 함수는 PlayManager의 대리자에 등록해서 해당 오브젝트가 비활성화 상태여도 대기시간은 계속 흐르도록 한다.

나) PlayScreenBase

게임 플레이 화면 중 기본 화면 상태의 부모 형식이다. 각 화면은 유니티의 UI와 연결할 것이므로 Monobehaviour를 상속받는다. 따라서 상태 변경은 해당 화면 오브젝트의 활성화, 비활성화로 이루어진다.

다) TechTreeViewBase

테크트리 화면은 하나고, 그 한 화면에서 어떤 테크트리를 표시할 것인지는 TechTree 클래스로 결정하는데, 이 클래스는 TechTree 클래스의 부모 클래스다. 해당 테크트리에 들어갈 노드 배열은 NodeData에 저장하고, 해당 테크트리를 포함한 모든 테크트리 노드의 인덱스는 NodeIndex라는 해쉬테이블에서 찾을 수 있다. NextNodes는 다음 노드가 무엇인지 저장할 가변배열의 배열로, 배열 부분은 해당 테크트리의 각 노드와 대응할 것이므로 NodeData와 길이가 같고, 가변배열 부분은 다음 노드의 개수에 따라 길이가 달라지므로 가변배열로 선언했다. 이 세 자료구조는 이 클래스에서 직접 생성하는 것이 아니고 다른 클래스에서 생성한 것의 참조를 저장한다. 그러면 다른 클래스를 반복적으로 참조할 필요가 없어 참조 회수를 줄일 수 있다.

라) SemiBases

어떤 부모 클래스를 상속받는 또 다른 부모클래스다. 최초의 부모 클래스는 Base라는 이름을 붙였기 때문에 두 번째 부모 클래스는 SemiBase라는 이름을 붙였다. 각 SemiBase는 서로 다른 멤버변수와 멤버함수가 있다,

8. Scripts\PlayScene\Items

특정 기능을 수행하기보다 주로 데이터 저장 용도로 사용할 클래스를 모아놓은 폴더다.

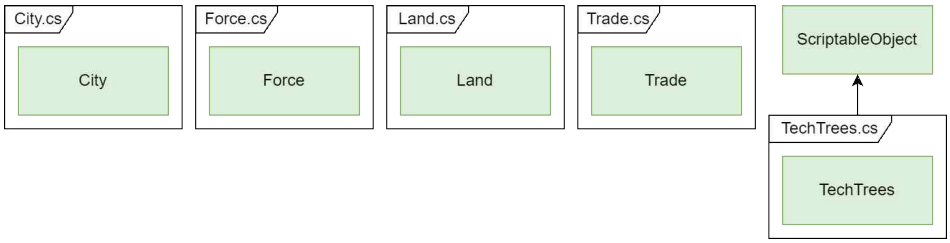

그림 다.8.0

각 클래스는 각각의 정보를 저장한다. 그 중 TechTrees는 테크트리의 노드 정보를 담을 것으로, 유니티 에디터에서 편집 가능하도록 ScriptableObject를 상속 받았다. 그 외 나머지 클래스는 동적으로 생성하게 된다.

9. Scripts\PlayScene\PopUpScreens

어떤 화면에서 또 다른 화면을 추가적으로 표시할 때 동작할 클래스를 담은 플더다.

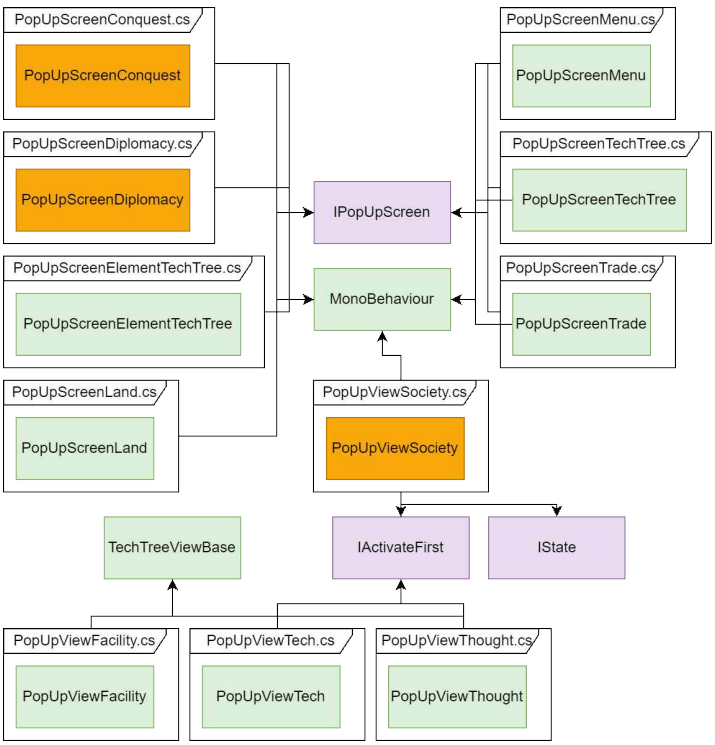

그림 다.9.0

가) PopUpScreen

화면 UI동작을 위한 클래스다. 공통적으로 IPopUpScreen 인터페이스를 상속받고 모두 각자의 역할을 수행한다. 구성하는 UI에 따라서는 유니티식 싱글턴패턴이 필요하기도 하다.

나) PopUpView

PopUpScreenTechTree 같은 테크트리 화면을 열었을 때 표시할 테크트리와 그에 관한 동작을 가지고 있다. 따라서 PopUpView에도 상태 패턴 적용이 가능하고, 부모 클래스인 TechTreeViewBase는 IState를 상속받는다.

다) PopUpViewSociety

이 클래스는 다른 테크트리와는 상이하기 때문에 같은 부모 클래스를 상속받지 않는다. 아직까지 이 클래스와 비슷한 동작을 하는 클래스는 없으므로, 이 클래스는 MonoBehaviour를 직접 상속받고 이 클래스만의 동작을 수행한다.

10. Scripts\PlayScene\Screens

게임 플레이 씬의 기본 화면의 동작을 수행할 클래스를 모은 폴더다.

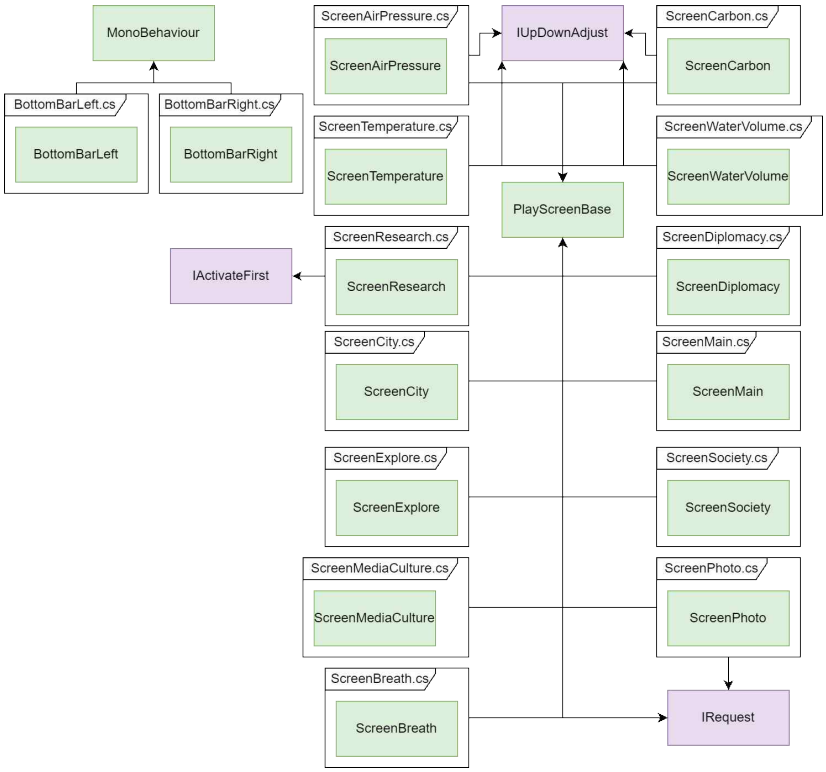

그림 다.10.0

각 클래스는 수행할 동작에 따라 필요한 클래스와 인터페이스를 상속받는다. 이 중 PlayScreenBase를 상속받는 클래스는 상태패턴이 적용된다.

11. Scripts

분류 없이 공통으로 사용할 클래스는 Scripts 폴더에 있다.

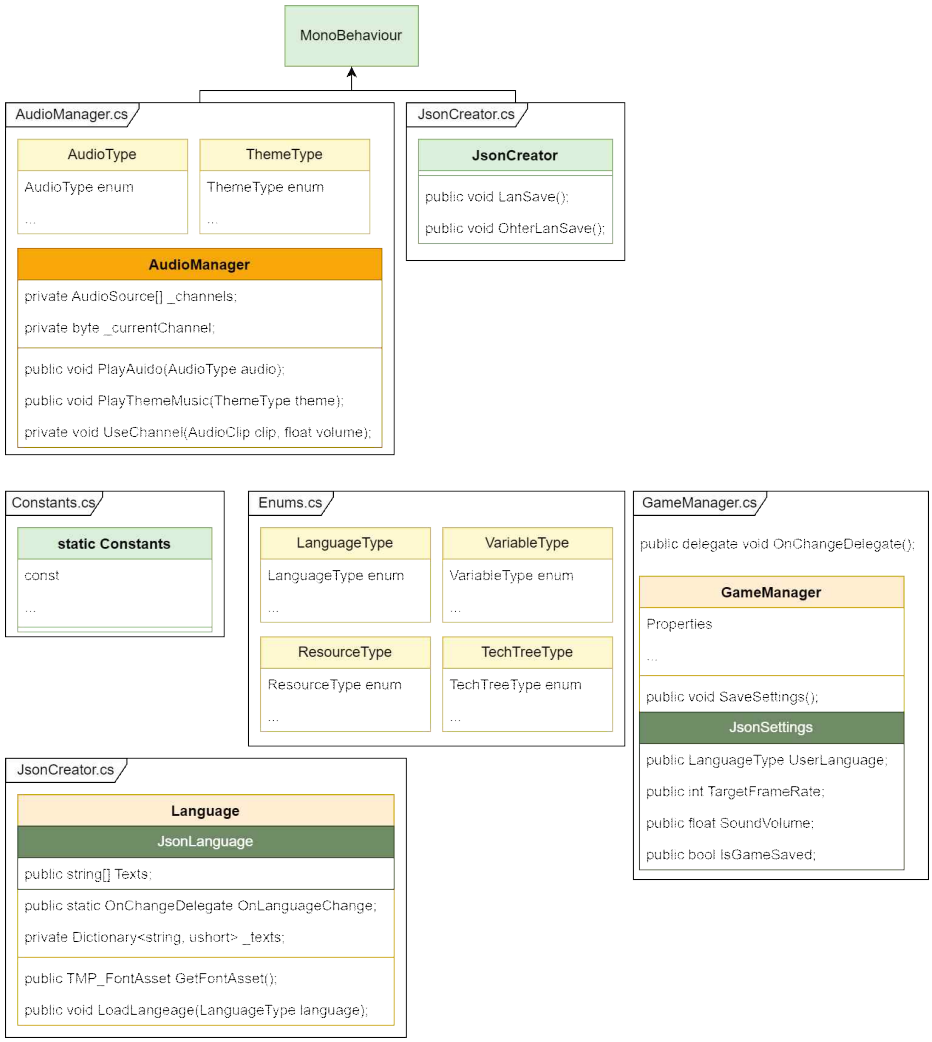

그림 다.11.0

가) AudioManager

모든 소리는 이 클래스가 담당한다. 씬이 변경되거나 같은 소리를 빠르게 연속으로 재생하면 재생되던 소리가 끊어지는 일이 생기는데, 이를 막기 위해 소리를 재생할 채널을 여러 개 생성하고 AudioManager에서 한 채널씩 재생할 소리를 할당하고 재생한다. 생성할 채널의 개수는 SerializeField를 통해 유니티 에디터에서 설정 가능하게 했고, 설정 값에 따라서 채널을 동적으로 생성한다. 씬을 변경할 때는 AudioManager와 소리 채널이 파괴되면 안 되기 때문에 DontDestroyOnLoad로 씬을 변경해도 파괴되지 않게 한다. 이때 AudioManager는 유니티 에디터에서 씬에 미리 생성해둔 것이 아니라 AudioManager가 없을 때만 동적으로 생성되는 것이다. AudioManager는 유니티식 싱글턴패턴이 적용되는데, AudioManager의 참조를 저장할 정적변수가 null일 때만 AudioManager가 동적으로 생성된다. 그러면 씬을 변경해도 AudioManager의 생성, 파괴를 반복하지 않아도 된다.

나) Constants

중요하거나 여러 클래스에서 공통으로 사용할 상수는 모두 Constants 클래스에 담는다. Constants는 정적 클래스로 인스턴스 생성 없이 멤버변수에 접근 가능하다. 게임 플레이에 중요한 변수를 한 클래스에 모았으므로 상수 관리가 용이하다.

다) Enums

이 스크립트에는 클래스가 없고 열거형만 있다. 이 열거형은 여러 클래스에서 공통으로 사용할 것인데, 만약 이 열거형이 어떤 클래스에 소속되어있으면 이 열거형에 접근할 때마다 소속 클래스를 써야 해서 일부러 전역 공간으로 빼뒀다. 각 열거형은 주로 특정 배열에 접근할 때 정수형 숫자 대신에 사용하게 될 것인데, 열거형을 사용하면 정수형 숫자를 쓸 때보다는 어떤 의미의 인덱스에 접근하는지 알기 쉽다. 특정 배열에 접근하기 위한 용도인데 이 열거형을 스크립트 파일을 따로 두어 전역 공간에 선언한 것은 그만큼 사용 빈도가 높거나 중요도가 높은 경우다. 예로 이 중 한 열거형은 참조된 회수가 99개가 넘는다.

열거형의 요소 개수는 배열의 길이가 될 것이다. 각 열거형의 가장 마지막 요소는 End라는 이름이 붙고, 각 배열은 End라는 이름의 요소로 배열 길이를 정한다. 따라서 작업 중 배열 길이를 수정해야 되는 상황에서도 열거형에서 End라는 이름의 요소를 가장 마지막으로 고정하면 배열 길이를 직접 수정할 필요 없다. 반복문을 사용하는 경우에도 End 요소가 반복 회수가 되므로 개발자는 열거형만 수정하면 나머지 배열, 반복문은 수동으로 수정할 필요 없다.

라) GameManager

씬이 변경된 후에도 유지해야 될 정보가 있으면 GameManager 클래스에 전달한다. 그러면 변경된 씬에서 다른 클래스가 GameManager 클래스에 있는 정보에 접근한다. GameManager가 담는 정보는 게임 플레이 관련 정보 뿐만 아니라 언어 설정 등 전반적인 설정 정보도 담고 있고, 이러한 전반적인 설정은 게임 종료하더라도 다음에 게임 실행 시 그 설정을 유지하기 위해 json 파일로 저장한다. json으로 저장될 정보는 JsonSetting이라는 구조체에 담겨있고, 이 구조체는 오직 GameManager만 생성할 것이기 때문에 private로 제한한다. 또한 구조체라서 값변수에 저장되므로, 클래스로 선언해서 참조변수를 사용할 때보다 참조 회수를 한 번 줄일 수 있다.

GameManager의 용도를 보았을 때 좀 더 어울릴 만한 이름은 SettingManager나 비슷한 이름 정도가 될 것인데, 굳이 GameManager라고 이름 지은 이유는 유니티 에디터에서 GameManager라는 이름을 가진 스크립트는 아래 그림 다.11.1과 같은 아이콘으로 표시되기 때문이다.

그림 다.11.1

게임 플레이에 관한 동작을 할 클래스는 이미 PlayManager라고 이름지었기 때문에 이 게임에는 톱니바퀴 아이콘에 어울릴만한 클래스가 없다. 그래서 전반적인 게임 설정을 담을 클래스가 GameManager라는 이름을 대신 가져갔다.

마) JsonCreator

PlayManager, GameManager의 구조체를 json 파일로 저장할 때와는 다르게, Language 클래스의 구조체를 json 파일로 저장하는 것은 게임 플레이 중에 발생하지 않는다. 즉, 아무도 Language 클래스의 언어 파일 저장 함수를 호출해주지 않는다. 따라서 오직 유니티 에디터에서만 사용할 용도의 JsonCreator 클래스를 만들었다. JsonCreator는 에디터 스크립트와 연결돼있어서 에디터 인스펙터 창에 있는 버튼으로 간편하게 json 파일을 생성할 수 있다. 유니티에 에디터 스크립트 기능이 없었으면 아마도 따로 콘솔 프로그램을 만들어서 수행했을 것이다.

바) Language

여러 언어로 번역할 것으로 상정하고 만든 클래스다. 어떤 단어를 표시할 때마다 분기문을 만드는 것보다는 언어 설정에 따라 그에 맞는 문자열이 반환되는 형태인 것이 개발자 입장에서는

더 편할 것이다. 게임을 처음 실행하거나 언어 설정이 변경되면 Language 클래스는 언어 파일을 읽어오고, 언어 문자열은 배열에 저장된다. 이때 어느 인덱스에 원하는 문자열이 존재하는지는 _texts라는 해쉬테이블에서 찾을 수 있다. 이 해쉬테이블은 한국어 '키'로 접근하면 인덱스 '값'을 반환한다. 예를 들어, 누군가가 어떤 번역된 단어 혹은 문장을 가져오고 싶을 때, Language 클래스에 해당 한국어 '키'를 전달하면 Language 클래스는 그 '키'로 인덱스 '값'을 찾고, 그 '값'으로 언어 배열의 요소에 접근해 원하는 문자열을 반환해주는 것이다. 따라서 문자열을 요청한 쪽의 입장에서는 Language 클래스에 한국어를 전달하면 번역된 문자열이 반환되는 것이다. 이는 물론 언어 설정이 한국어로 돼 있어도 똑같이 동작한다. 분기문을 통해 언어 설정이 한국어인지만 확인하고 한국어인 경우에는 Language 클래스를 거치치 않도록 만들어도 무방할 것이나, 그렇게 하면 개발자 입장에서는 코드가 보기 안 좋아질 것이고, 한국어를 사용하지 않는 사용자 입장에서는 언어 설정 확인만 반복적으로 계속하는 꼴이 된다. 또한 앞서 얘기했듯이 해쉬테이블과 배열의 검색 속도는 모두 O(1)이기 때문에 Language 클래스를 거친다고 해서 큰 손해가 있지는 않을 것이다.

라. 코드 설명

1. MainMenuManager

가) 주 화면 상태기계의 화면 변경 동작

```
/// <summary>
/// 메뉴 화면 이동
/// </summary>
public void MoveScreen(TextScreens screen)
{
    // 화면 비운다.
    _textScreen.text = null;
    _textScreenBuilder.Clear();
    SetButtons();

    // 다음 화면
    _currentScreen = _states[(int)screen];
    _currentScreen.Execute();
}
```

위는 MainMenuManager의 멤버함수 중 하나다. 주 화면의 기본적인 화면의 상태와 동작을 관리하는 여러 TextScreen 클래스 중 하나가 상태를 변경할 때 이 MoveScreen 함수를 호출한다. TextScreens는 화면 종류 열거형이고 이것을 매개변수로 받아 _currentScreen에 현재 화면 상태를 저장하고 Execute 함수를 호출한다. Execute 함수에서는 화면 표시 애니메이션을 위한 함수를 포출할 것이다.

나) 시간에 따른 애니메이션 동작

```
private void Update()
{
    if (_screenAnimation)
    {
        if (Constants.TEXT_SCREEN_SPEEDMULT <= _timer)
        {
            if (_textLength > _phase)
            {
                // 소리 재생
```

```
                    AudioManager.Instance.PlayAuido(AudioType.Touch);

                    // 화면 차례대로 표시
                    _textScreenBuilder.Append(_texts[_phase]);
                    _textScreen.text = _textScreenBuilder.ToString();
                    ++_phase;
                    _timer -= Constants.TEXT_SCREEN_SPEEDMULT;
                }
                else
                {
                    // 표시 완료
                    _currentScreen.SetButtons();
                    _screenAnimation = false;
                }
            }
            else
            {
                // 시간 경과
                _timer += Time.deltaTime;
            }

            //여기서 함수 종료
            return;
        }
        else if (_firstStart)
        {
            if (0 >= _timer)
            {
                // 밝아지는 중
                _warningText.color = new Color(1.0f, 1.0f, 1.0f, Mathf.Cos(_timer) * 0.5f + 0.5f);
            }
            else if (Constants.DOUBLE_PI < _timer)
            {
                // 더이상 필요 없음
                Destroy(_warningText.gameObject);
                _firstStart = false;
                GameManager.Instance.IsApplicationFirstStarted = false;
                MoveScreen(TextScreens.Main);
            }
            else if (Constants.PI < _timer)
            {
                // 어두워지는 중
                _warningText.color = new Color(1.0f, 1.0f, 1.0f, Mathf.Cos(_timer - Constants.PI)
* 0.5f + 0.5f);
            }

            // 시간 경과
            _timer += Time.deltaTime * Constants.STARTING_TEXT_SPEEDMULT;

            // 여기서 함수 종료
            return;
        }

        // 단축키 동작
        if (Input.GetKeyUp(KeyCode.Escape))
        {
```

```
            _currentScreen.OnEscapeBtn();
        }
    }
```

Update 함수에서는 애니메이션 동작과 단축키 동작이 이루어진다. 화면 상태 클래스의 Execute 함수를 호출하면 _screenAnimation는 참이 되고 화면 표시 애니메이션이 실행된다. 표시할 화면은 문자열의 배열로 존재하는데, _phase를 1씩 증가시키면서 일정 시간 간격으로 한 줄씩 화면에 표시한다. 이때 화면에 표시할 문자열은 한 줄씩 _textScreenBuilder에 계속해서 추가되는데, 배열의 길이만큼 문자열 편집이 있으므로 편집할 때마다 string을 생성하는 것보다는 StringBuilder 클래스를 사용하는 것이 더 좋다. 애니메이션의 한 루프가 끝나면 return으로 함수가 반환되는데, 이는 애니메이션이 실행되는 동안 Update 함수 맨 아래쪽에 있는 단축키 동작을 생략하기 위함이다.

_firstStart는 이 프로그램이 처음 실행된 것인지 확인하고 처음 실행 시 안내 문구를 표시하기 위함이다. 시간이 경과하면서 안내 문구가 나타났다가 자동으로 사라지는 형태다. 이때 시간 경과를 측정하는 _timer 변수의 값이 0 이하일 때 '밝아지는 중' 동작이 이루어지는데, 이는 _timer가 0에서부터 시작하지 않기 때문이다. 0부터 시작하지 않은 이유는 안내 문구가 밝아지고 어두워지는 정도는 Cos 함수를 따랐기 때문이다.

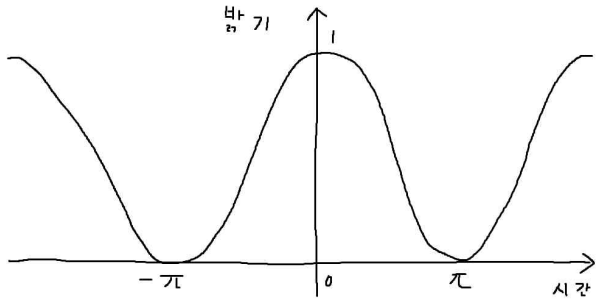

그림 라.1.0

안내 문구의 밝기 정도 그래프가 위 그림 라.1.0과 같다고 할 때, 처음에는 밝기가 밝아지는 것으로 시작해야 되므로 시작 시 _timer는 $-\pi$에서부터 시작한다. 그러나 이처럼 밝기 그래프를 Cos 함수에 그대로 따르게 하면 안내 문구 밝기가 1이 되자마자 곧바로 어두워지는 꼴이 된다. 따라서 실제 안내 문구의 밝기 그래프는 아내 그림 라.1.1과 같아야 된다.

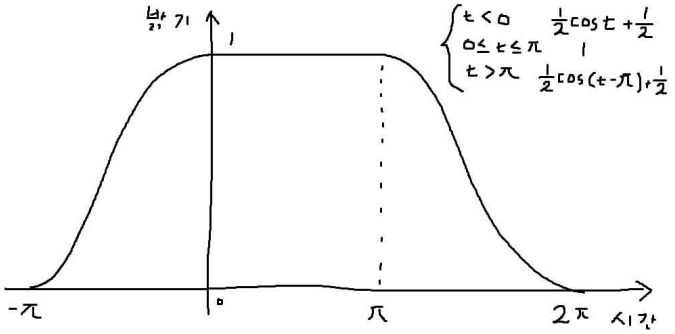

그림 라.1.1

이 그래프는 분기문을 사용해서 만들 수 있다. 여기서 _timer가 0 이상 π 이하인 구간은 밝기가 1로 고정되는데, Update 함수에서 참, 거짓을 계속 확인하면서 밝기 1을 대입하는 것은 불필요하다. 필요한 동작은 $-\pi$ 이상 0 미만 구간에서 밝아지는 동작, π 초과 2π 이하 구간에서 어두워지는 동작, 2π 초과 구간에서 애니메이션이 끝나는 동작이다. 여기서 다시 한번 확인이 불필요한 조건을 제거하면, 0 미만 구간에서의 동작, π 초과 구간에서의 동작, 2π 초과 구간에서의

동작이 된다. 물론 여기서는 π 초과 구간과 2π 초과 구간에는 겹치는 구간이 존재한다. 그러므로 코드로 작성할 때는 2π 초과 구간인지 먼저 확인하고 그것이 거짓이면 그때서 π 초과 구간인지 확인하도록 작성했다.

2. MessageBox

가) 큐에 메시지 문자열 추가

```
/// <summary>
/// 메세지 큐에 추가
/// </summary>
public void EnqueueMessage(string message, params string[] replacements)
{
    if (0 < replacements.Length)
    {
        _messageQueue.Enqueue(AddDate(message, replacements));
    }
    else
    {
        _messageQueue.Enqueue(AddDate(message));
    }
}
```

어떤 클래스든 사용자에게 어떤 메시지를 표시하고 싶을 때 위 함수를 호출한다. 메시지는 먼저 들어온 순서대로 표시할 것이므로 표시할 메시지를 담는 _messageQueue는 문자열을 담는 큐다. 이 큐에 문자열이 저장되면 이후 일정 시간 간격으로 사용자에게 표시되고 표시된 문자열은 제거된다. 매개변수 중 message는 기본 메세지 문자열이고 replacements는 메시지 문자열에 특정 값을 끼워서 표시하고 싶을 때 사용한다. params가 붙었으므로 문자열 개수는 자유롭다. AddDate 함수는 메시지 문자열에 게임 상의 날짜를 추가해주는 함수다.

나) 일정 시간 간격으로 문자열 표시

```
private void Update()
{
    if (_messageGap > _timer)
    {
        _timer += Time.deltaTime;
        return;
    }

    switch (_messageQueue.Count)
    {
        case 0:
            return;

        default:
            // 소리 재생
            AudioManager.Instance.PlayAuido(AudioType.Alert);

            // 메세지 표시
            _messageText.text = _messageQueue.Peek();
            _messageBoxObject.SetActive(true);

            // 로그 추가
            TMP_Text newLog = Instantiate(_logObject, _logArea).GetComponent<TMP_Text>();
```

```
                newLog.font = Language.Instance.GetFontAsset();
                newLog.text = _messageQueue.Dequeue();
                _timer = 0.0f;
                return;
            }
        }
```

_messageGap은 메시지 표시할 시간 간격 변수로, SerializeField를 통해 유니티 에디터에서 값을 변경할 수 있다. _timer가 _messageGap보다 작으면 시간 변화량을 더해주고 다음 코드가 동작하지 않도록 그 자리에서 함수를 반환한다. _timer가 _messageGap 이상이면 그 if문은 지나칠 것이고 그 아래 switch문이 실행될 것이다. switch문에서 확인하는 것은 _messageQueue의 길이가 0인지 아닌지 밖에 없는데, if문을 쓰지 않고 switch문을 쓴 이유는 if문 보다 switch문이 실행 속도가 더 빠르기 때문이다. 이는 Update 함수에서 유리하게 작용할 것이고 여기서는 코드의 가독성도 크게 해치지 않는다. _messageQueue의 길이가 0이 아닐 때는 사용자에세 메시지를 표시하기 위한 동작을 수행한다.

여기서 _timer는 _messageQueue의 길이와 상관없이 시간이 누적되도록 만들어져있는데, 만약 _messageQueue의 길이가 0이라고 해서 _timer의 시간 누적이 동작하지 않으면, 메시지 한 번 표시할 때마다 무조건 _messageGap만큼의 시간을 기다려야 한다. 처음 메시지는 즉시 표시되고 그 다음 연속으로 나오는 메시지만 _messageGap만큼의 시간을 기다리게 할 것이기 때문에 _timer의 시간 누적은 _messageQueue의 길이와 상관없이 동작한다.

3. Language

가) 인덱서를 통한 편리한 접근

```
    /// <summary>
    /// 이것이 주 목적이므로 편리한 접근을 위해 만들었다. 한국어 '키'를 입력하면 번역된 '값'을
반환한다.
    /// </summary>
    public string this[string koreanKey]
    {
        get
        {
#if UNITY_EDITOR
            if (_texts.ContainsKey(koreanKey))
            {
#endif
                return _jsonLanguage.Texts[_texts[koreanKey]];
#if UNITY_EDITOR
            }
            else
            {
                Debug.LogError($"없는 한국어 키 ₩"{koreanKey}₩"");
                return null;
            }
#endif
        }
    }
```

Language 클래스에 접근하는 경우의 대다수가 언어 배열에 접근하기 위함이다. 인덱서를 사용하면 이 인스턴스를 마치 배열인 것처럼 사용할 수 있다. 그러면 따로 함수를 사용할 때보다 코드 길이를 줄일 수 있어서 반복되는 타이핑을 줄일 수 있다. 이 인덱서를 참조한 회수는 99가 넘으므로 상당히 많은 타이핑 수를 절약했다고 볼 수 있다.

_texts는 한국어 키에 대한 배열 인덱스 값이 저장된 해쉬테이블이다. 원래 처음에는 이 해쉬테

이블에서 직접 번역된 값을 저장하려고 했으나, 언어 설정이 변경될 때 언어 파일을 읽어올 때마다 이 해쉬테이블을 갱신하려면, 한국어 키 정보를 따로 저장해두고 있어야 하고 해쉬테이블은 모두 비웠다가 다시 값을 등록하는 과정이 필요한데, 이는 상당히 비효율적으로 보인다. 언어 파일을 읽어오면 언어는 배열로 저장되기 때문에 언어는 배열 상태 그대로 두고 대신 헤쉬테이블은 한국어 키를 입력했을 때 배열 인덱스 값을 반환하도록 한다. 그러면 그 인덱스 값으로 원하는 배열 인덱스에 접근할 수 있다. _jsonLanguage는 json 파일 저장 및 불러오기를 위한 구조체고 이 구조체 내의 Texts 배열이 언어를 담을 배열이다.

언어 문자열을 가져올 때는 먼저 헤쉬테이블에 해당 한국어 키가 등록됐는지 확인하고 없는 키일 경우에 유니티 에디터 콘솔에 에러 로그를 띄운다. 이 기능은 작업 중에 큰 도움이 되는데, 코드 작성 중에는 추가되는 단어, 문장이 많아서 그때마다 언어 파일을 수정하는 것은 힘든 일이다. 그러므로 코드 작성 중에는 코드 작성에만 집중하고 새로 추가해야 될 단어, 문장은 유니티 에디터 콘솔창을 보면서 한꺼번에 추가하면 된다. 그리고 게임을 출시할 시점에는 이 에러 로그 출력 기능이 반드시 쓸모없는 상태여야 하기 때문에 #if UNITY_EDITOR로 쓸모없어야 하는 부분을 감싸서 최종 빌드 때는 컴파일되지 않도록 한다. 그러면 사용자 입장에서는 키의 존재 여부를 확인하는 동작을 하지 않아도 된다.

나) 언어 파일 읽어오기

```
    /// <summary>
    /// json 파일 불러온다.
    /// </summary>
    public void LoadLangeage(LanguageType language)
    {
        // json 파일 명을 담는다.
        string filename;
        switch (language)
        {
            case LanguageType.Korean:
                filename = "Korean";
                break;
            case LanguageType.English:
                filename = "English";
                break;
            case LanguageType.German:
                filename = "German";
                break;
            case LanguageType.French:
                filename = "French";
                break;
            case LanguageType.Taiwanese:
                filename = "Taiwanese";
                break;
            case LanguageType.Japanese:
                filename = "Japanese";
                break;
            default:
#if UNITY_EDITOR
                Debug.LogError("LoadLangeage - LanguageType 수정 요망.");
#endif
                return;
        }
```

```
        // json 파일을 불러온다.
        _jsonLanguage = JsonUtility.FromJson<JsonLanguage>(Resources.Load(filename).ToString());
        _fontAsset = (TMP_FontAsset)Resources.Load($"{filename}Font SDF");

        // 대리자
        OnLanguageChange?.Invoke();
        GameManager.Instance.CurrentLanguage = language;
    }
```

이 함수는 언어 설정 UI에서 호출할 것이고 호출하는 측 입장에서는 인자로 LanguageType 열거형 값만 전달하면 된다. 열거형 값에 따라 불러올 파일 이름을 결정하고 System.IO 네임스페이스에 포함된 JsonUtility로 json 파일을 읽어온다. 언어 파일은 Resources 폴더에 있으므로 Resources 클래스를 사용해서 경로를 설정하면 된다.

언어 파일을 읽어온 이후로는 게임 내의 텍스트 UI를 찾아 업데이트해야 되는데, 업데이트가 필요한 텍스트 UI는 모두 OnLanguageChange 대리자에 등록되어 있다. ? 기호로 null 확인 후 Invoke로 대리자에 등록된 모든 동작을 수행하면 모든 텍스트 UI가 업데이트 된다.

다) 구글 번역을 위한 텍스트 파일 출력

```
    /// <summary>
    /// 구글 번역을 위해 텍스트 파일로 저장
    /// </summary>
    private void TextFileForGoogleTranslate(JsonLanguage jsonLanguage)
    {
        // 문자열 생성
        StringBuilder text = new StringBuilder();
        for (ushort i = 0; i < jsonLanguage.Texts.Length; ++i)
        {
            text.Append($"{jsonLanguage.Texts[i]};₩n");
        }

        // 텍스트 파일로 저장
        File.WriteAllText($"{Application.dataPath}/TranslateThis.txt", text.ToString());
    }
```

언어 파일은 json 형식으로 저장되므로 번역가가 있으면 그 json 파일을 전달하면 되는데 지금은 번역가가 없으므로 구글 번역을 사용한다. 그런데 구글 번역에 번역하면 json 파일의 형태가 망가지는 일이 발생해서 json의 형태보다 더 간단한 형태가 필요하다. JasonLanguage는 json 파일 저장 및 불러오기를 위한 구조체고 그 안의 Texts가 언어 배열에 해당한다. 텍스트 파일로 저장할 문자열은 StringBuilder로 만든다. Texts의 한 단어가 끝날 때 세미콜론을 붙이고 줄 바꿈하는 것이 이 텍스트 파일의 형태다. 번역가가 있었으면 이 과정은 불필요했을 것이다.

라) 번역된 텍스트 파일 json으로 다시 출력

```
    /// <summary>
    /// 다른 언어 json 파일을 생성한다.
    /// </summary>
    public void SaveOtherLanguages()
    {
        // 준비 단계
        string[] path = Directory.GetFiles($"{Application.dataPath}/Translations/", "*.txt",
SearchOption.AllDirectories);
        string jsonForm = Resources.Load("Korean").ToString();
        List<StringBuilder> words = new List<StringBuilder>();
```

```
        StringBuilder result = new StringBuilder();

        // 존재하는 모든 번역본 생성
        for (byte i = 0; i < path.Length; ++i)
        {
            // 언어 하나 준비 단계
            string text = File.ReadAllText(path[i]);
            ushort index = 0;
            words.Clear();

            // 단어 추출
            for (int j = 0; j < text.Length; ++j)
            {
                if (';' == text[j])
                {
                    // 다음 단어
                    j += 2;
                    ++index;
                }
                else
                {
                    // 가변배열에 추가 안 됐으면 추가
                    if (index == words.Count)
                    {
                        words.Add(new StringBuilder());
                    }

                    // 단어 기록
                    words[index].Append(text[j]);
                }
            }

            // json화 시작 준비 단계
            int count = 0;
            bool proceed = true;
            index = 0;
            result.Clear();

            // json 형식 따라가기
            for (int j = 0; j < jsonForm.Length; ++j)
            {
                // 기록
                if (proceed)
                {
                    result.Append(jsonForm[j]);
                }

                // 큰따옴표 세기
                if ('₩"' == jsonForm[j])
                {
                    ++count;

                    // 큰따옴표 3개 이상일 때
                    if (3 <= count)
                    {
                        // 큰따옴표 열렸을 때
```

```
                    if (1 == count % 2)
                    {
                        // 새로운 단어 저장
                        if (index < words.Count)
                        {
                            result.Append(words[index]);
                        }

                        result.Append('₩"');

                        // 다음 단어
                        ++index;

                        // 기록 중지
                        proceed = false;
                    }
                    // 큰따옴표 닫혔을 때
                    else
                    {
                        // 기록 재개
                        proceed = true;
                    }
                }
            }
        }

        // json 파일로 저장

File.WriteAllText($"{Application.dataPath}/Resources/{Path.GetFileNameWithoutExtension(path[i])}.
Json", result.ToString());
        }
    }
```

구글 번역으로 번역된 텍스트 파일을 만들었으면 그 파일을 json 파일로 만든다. 위 함수를 한 번 호출하는 것으로 존재하는 번역 파일을 전부 읽어와서 json 파일을 만든다. json 파일의 형태를 직접 타이핑하면서 읽어온 단어를 추가하는 방식을 쓸 수 있는데, 여기서는 이미 존재하는 json 파일을 읽어와서 그 안의 단어만 치환하는 방식으로 했다. 한국어는 번역 없이 직접 json 파일로 저장되니 한국어 json 파일은 이미 존재한다. 따라서 한국어 json을 읽어와서 번역된 값으로 치환하는 것이다. 그러면 한국어 json의 단어 개수와 읽어온 번역 단어 개수가 일치하지 않을 때 오류가 발생하므로 즉각 오류를 해결할 수 있다. 언어 파일은 게임 실행 중에 생성되는 것이 아니므로, 위 코드에서는 보이지 않으나 이 함수 전체가 #if UNITY_EDITOR로 감싸져 있다.

구글 번역을 위한 텍스트 파일 형태를 만든 것은 구글 번역에 사용하기 위함이고, 번역가가 있었으면 json 파일 그대로 전달했을 것이다. 그러나 위 함수를 작성하면서 들었던 생각은 굳이 json 파일이 아니어도 된다는 것이다. 물론 여기서는 System.IO에서 제공하는 함수로 간단하게 json 파일을 저장하고 읽어올 수 있어서 json 파일을 사용했는데, 번역가 입장에서 보기 좋은 텍스트 형식을 만들 수 있으면 그렇게 하는 것도 무방해보인다.

4. AudioManager

가) 채널 생성

```
    private void Awake()
```

```
    {
        // 채널 배열 생성
        _channels = new AudioSource[_numberOfChannel];

        // 1개는 기본
        _themeChanel = GetComponent<AudioSource>();
        _channels[0] = _audioSource;

        // 나머지는 추가 생성
        for (byte i = 1; i < _numberOfChannel; ++i)
        {
            AudioSource aS = Instantiate(_audioSource.gameObject,
transform).GetComponent<AudioSource>();
            _channels[i] = aS;
        }
    }
```

생성할 채널 개수는 _numOfChannel이고 _channels는 AudioSource의 배열이다. _audioSource는 미리 생성돼있던 AudioSource의 참조고 채널 수가 _numOfChannel만큼 될 때까지 _audioSource를 복사해서 추가 생성한다.

나) 채널 할당

```
    /// <summary>
    /// 오디오 채널 할당 후 재생
    /// </summary>
    private void UseChannel(AudioClip clip, float volume)
    {
        // 오디오 채널에 소리 등록하고 재생
        _channels[_currentChannel].clip = clip;
        _channels[_currentChannel].volume = volume * GameManager.Instance.SoundVolume;
        _channels[_currentChannel].Play();

        // 사용할 채널 인덱스 증가
        ++_currentChannel;

        // 채널 개수 초과 시 되돌아온다.
        if (_currentChannel == _numberOfChannel)
        {
            _currentChannel = 0;
        }
    }
```

사용할 채널 인덱스는 _currentChannel이고 채널에 오디오 클립, 음량 설정 후 소리 재생하면 _currentChannel은 1 증가한다. 채널 개수는 _numOfChannel이므로 _currentChannel이 1 증가한 후 _numOfChannel과 같으면 다시 0으로 돌아간다.

다) 배경음악 작아지기

```
    private void Update()
    {
        if (_themeMusicFadeOut)
        {
            _themeChanel.volume -= Time.deltaTime * Constants.THEME_FADE_OUT_SPEEDMULT;
            if (0.0f >= _themeChanel.volume)
            {
```

```
                _themeChanel.Stop();
                _themeMusicFadeOut = false;
                if (ThemeType.None != _reserve)
                {
                    PlayThemeMusic(_reserve);
                }
            }
        }
    }
```

씬이 변경되는 경우 배경음악 음량은 서서히 줄어들고 음량이 0이면 재생을 멈춘다. 이때 배경음악이 완전히 멈추지 않은 상태에서 누군가 다른 배경음악 재생을 요청하면 그 배경음악정보는 _reserve에 저장되는데 _reserve는 열거형 변수다. 배경음악이 완전이 멈춘 시점에서 _reserve가 None이라는 값이 아니면 _reserve 값으로 다시 배경음악을 재생한다.

라) 불필요한 생성 파괴

```
    public static AudioManager Instance
    {
        get
        {
            return _instance;
        }
        set
        {
#if UNITY_EDITOR
            if (null == _instance)
            {
#endif
                _instance = value;
#if UNITY_EDITOR
            }
            else if (value != _instance)
            {
                // 이미 생성돼있는 경우 새로 생성한 것을 파괴한다.
                Destroy(value.gameObject);
                Debug.LogError("이미 생성된 AudioManager.");
            }
#endif
        }
    }
```

AudioManager는 게임 시작 시 동적으로 생성되고 DontDestroyOnLoad에 들어간다. 만약 유니테 에디터에서 시작 씬에 AudioManager를 미리 생성해두면 시작 씬으로 돌아갈 때마다 중복된 AudioManager를 생성하고 파괴하는 과정을 반복한다. 이를 막기 위해 AudioManager 참조를 저장하는 정적변수 _instance가 null일 때만 AudioManager를 동적으로 생성하도록 한다. _instance가 null이 아닌 상태에서 _instance에 새로운 인스턴스를 대입하려고 하면 그것은 문제가 있는 것이므로 그 새로운 인스턴스를 파괴하도록 했다. 이 문제있는 동작은 게임 출시 시점에는 이미 해결된 상태여야 되기 때문에 그 부분을 #if UNITY_EDITOR로 감싸서 출시 때는 컴파일되지 않도록 했다.

5. AnimationManager

가) 무한한 배경 움직임

```
        #region 노이즈 애니메이션
        vector3_0 = _backgroundTransform.localPosition + new Vector3(0.0f, _noiseSpeed, 0.0f);
        if (Constants.HALF_CANVAS_HEIGHT < vector3_0.y)
        {
            vector3_0 -= new Vector3(0.0f, Constants.CANVAS_HEIGHT, 0.0f);
        }
        _backgroundTransform.localPosition = vector3_0;
        _messageBoxBackgroundTransform.localPosition = vector3_0 *
Constants.MASSAGEBOX_HEIGHT_RATIO;
        #endregion
```

위는 AnimationManager의 Update 함수 코드 중 일부다. 배경 이미지는 y축 방향으로 끊임없이 움직이는데, y 좌표가 Canvas의 높이를 벗어나면 다시 원래 자리로 되돌아온다. 그래서 사용자가 볼 때는 배경 이미지가 무한히 이동하는 것처럼 보여야 되므로 무한 스크롤을 적용해도 될 것이다. 무한 스크롤이라면 두 개 이상의 오브젝트가 화면 위를 번갈아가면서 지나가게 되는데, 두 배경 이미지를 한 오브젝트로 합쳐도 무방할 것이다.

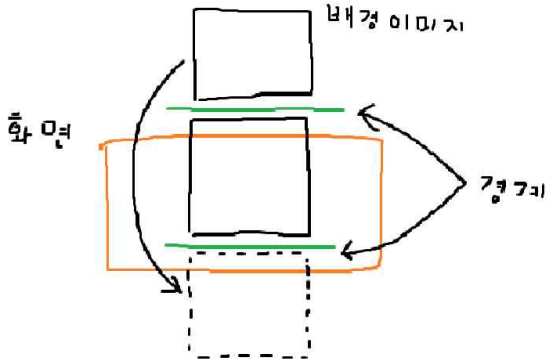

그림 라.4.0

일단 무한 스크롤에서는 위 그림 라.4.0처럼 한 배경 이미지가 화면 밖을 벗어났을 때 그 이미지는 다시 처음 자리로 돌아오는데, 사용자가 실제로 보게 될 화면은 똑같은 두 이미지의 경계가 화면 위를 왔다 갔다 하는 것이다. 그렇다면 이 두 이미지를 하나로 합칠 수 있는 것이다.

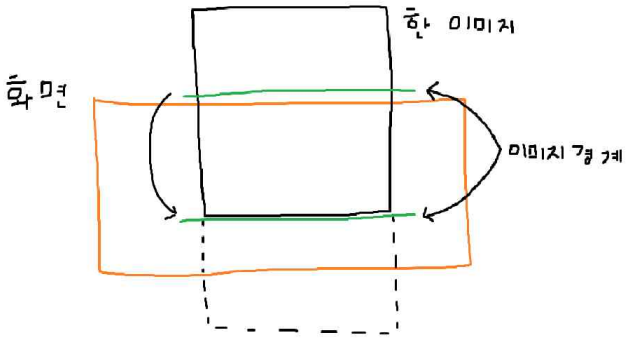

그림 라.4.1

어차피 화면에 보일 것은 두 이미지의 경계 부분이 화면 상에서 무한히 이동하는 모습이므로 두 이미지를 합친 한 이미지를 이동하는 방법으로 해도 결과는 동일하다. 단점은 사용할 이미지의 크기가 2배 늘어나는 것이고, 만약 y축 뿐만 아니라 x축까지 이동해야 되는 것이었으면 이미지 크기는 4배가 늘어났을 것이다. 장점은 정반대로 참조해야 될 Transform의 수가 절반 줄어드는 것이고, x축까지 이동해야 하는 경우 1/4까지 줄어드는 것이다. 여기서는 배경 이미지가 고화

질이 아니고, 이 게임에서 이미지 파일은 많지 않기 때문에 이처럼 한 이미지를 사용하는 방법을 써도 무방할 것이다.

6. UIString

가) 여러 화면에서 공통으로 사용할 문자열 반환

```
    /// <summary>
    /// 이것이 주 목적이므로 편리한 접근을 위해 만들었다. 문자열을 단위 포함해서 가져온다.
    /// </summary>
    public string this[VariableFloat variable]
    {
        get
        {
            // 값이 바뀌었거나 문자열을 생성한 적 없을 때
            if (_currentFloatValues[(int)variable] != PlayManager.Instance[variable] || null ==
_floatStrings[(int)variable])
            {
                // 현재 값 저장
                _currentFloatValues[(int)variable] = PlayManager.Instance[variable];

                // 표시할 단위가 없을 때
                if (null == _floatUnits[(int)variable])
                {
                    _floatStrings[(int)variable] =
_currentFloatValues[(int)variable].ToString("F2");
                }
                // 표시할 단위가 있을 때
                else
                {
                    _floatStrings[(int)variable] =
$"{_currentFloatValues[(int)variable].ToString("F2")}{_floatUnits[(int)variable]}";
                }
            }

            // 반환
            return _floatStrings[(int)variable];
        }
    }
```

어떤 정보를 사용자한테 문자열로 표시할 때 같은 정보를 여러 화면에 표시할 수 있다. 각 화면마다 문자열을 따로 생성하는 것보다는 같은 정보면 문자열을 하나만 생성하고 한 문자열을 공유하는 것이 더 효율적일 것이다. 따라서 UIString은 공통으로 사용할 문자열을 관리하고, 위 인덱서는 접근해온 쪽에서 원하는 문자열을 반환해준다.

위 인덱서에서 다루는 자료형은 float다. 다른 자료형을 다루는 인덱서는 오버로드해서 만들 수 있다. 누군가가 float형의 정보 문자열을 가져오고 싶을 때, UIString은 가장 처음에 해당 float 값이 변경되었는지 혹은 문자열을 생성한 적 없는지 확인한다. 문자열을 생성하는 시점에 해당 값은 _currentFloatValues에 저장하기 때문에 값이 변경됐는지 확인할 수 있다. 저장된 값과 실제 값이 같으면 변하지 않은 것이므로 이전에 생성한 문자열을 반환하고, 같지 않으면 값이 변한 것이므로 새 문자열을 생성한다. 물리량의 경우 단위 표시까지 하기 때문에 미리 설정된 단위를 붙여서 _floatStrings에 저장한 후 반환한다. _floatStrings에 저장된 문자열은 이후에 값 변경이 없는 경우에 그대로 반환될 수 있다.

이 방식의 장점은 불필요한 문자열 생성을 줄일 수 있다는 것이다. 단점은 값이 변경됐는지 확인하기 위해 기억하고 있어야 되는 변수만큼 메모리 공간을 더 차지한다는 것이다. 이 방식의 효

율성은 같은 정보 문자열을 얼마나 많은 화면이 가져오느냐에 따라 달라질 것이다.

7. PlayManager

가) 게임 데이터 json 파일로 저장

```
    /// <summary>
    /// 게임 저장
    /// </summary>
    public void SaveGame()
    {
        // 게임 데이타 저장
#if PLATFORM_STANDALONE_WIN
        File.WriteAllText($"{Application.dataPath}/Saves.Json", JsonUtility.ToJson(_data,
false));
#endif
#if PLATFORM_ANDROID
        File.WriteAllText($"{Application.persistentDataPath}/Saves.Json",
JsonUtility.ToJson(_data, false));
#endif

        // 게임이 저장됐다는 설정이 안 됐을 경우
        if (!GameManager.Instance.IsThereSavedGame)
        {
            GameManager.Instance.IsThereSavedGame = true;
            GameManager.Instance.SaveSettings();
        }
    }
```

게임 데이터 저장은 플레이 중이던 게임을 저장하는 것으로, 저장 파일은 빌드하는 시점에 존재하는 것이 아니라 사용자의 동작으로 생성되는 것이다. 이 파일은 유니티의 Resources 폴더에 저장되는 것이 아니니 저장 위치를 직접 지정해줬다. 여기서 Application.dataPath는 실행 파일의 위치를 반환하는데, 이것이 안드로이드에서는 작동하지 않는 것을 발견했다. 유니티 에디터는 아무래도 안드로이드가 아니니 오류가 없었던 모양이다. 안드로이드에서 실행 파일 경로는 Application,persistentDataPath가 반환해준다. 이는 Window에서는 실행 파일 경로가 아닌 다른 경로를 반환해준다. 그래서 #if를 사용해서 어느 플랫폼인지에 따라 다르게 컴파일 되도록 했다.

8. TechTrees

가) 테크트리 노드 정보를 저장하는 자료구조

```
    [SerializeField] private Node[] _facilityNodes = null;
    [SerializeField] private Node[] _techNodes = null;
    [SerializeField] private Node[] _thoughtNodes = null;
    private Dictionary<string, byte> _indexDictionary = new Dictionary<string, byte>();
    private List<SubNode>[][] _nextNodes = new List<SubNode>[(int)TechTreeType.TechTreeEnd][];
```

TechTrees 클래스는 ScriptableObject를 상속받으므로 유니티 에디터에서 파일로 생성할 수 있다. TechTrees 클래스는 테크트리 노드 정보를 담을 클래스인데, 테크트리 노드는 자주 수정할 가능성이 높기 때문에 ScriptableObject를 상속받고 파일로 다루면 유니티 에디터에서 편집하기 용이하다.

이 게임의 테크트리는 이름은 '테크트리'인데 실제로는 트리의 구조가 아니다. Node는 내부 클래스고 노드 정보를 저장할 클래스인데, ScriptableObject를 상속받고 유니티 에디터에서 편집할 때 Node는 생성된 다른 노드를 참조하지 못한다. 노드 간 링크로 연결되지 못하는 것이다. 그래서 링크를 대신할 것이 SubNode 구조체다. 노드는 노드 이름 문자열로 식별할 것이므로

SubNode는 연결할 노드의 이름을 저장한다. 게다가 테크트리의 구조 자체가 트리 모양이 아니다. 테크트리 종류는 시설, 기술, 사상 등이 있는데, 이 서로 다른 테크트리의 노드가 서로 얽혀있다. 한 테크트리 내에서만이 아니라 다른 테크트리로부터도 연결되어있어서 부모 노드가 2개 이상일 수 있고, 심지어는 한 테크트리 내에서도 부모 노드가 2개 이상일 수 있다. 여기서는 테크트리이므로 한 노드는 한 '테크'가 될 것이고, 한 테크의 부모 노드는 요구되는 테크가 될 것이다. 따라서 한 테크로 넘어가려면 해당 테크의 요구 테크가 모두 활성화 상태여야 하는 것이다.

결국 테크트리를 트리 구조로 사용할 이유가 없음이 설명됐다. 그렇다면 차라리 접근이 빠르도록 배열의 형태로 만드는 것이 낫다. 각 노드는 미름 문자열로 식별하기로 했으니, 헤쉬테이블을 통해 문자열을 입력하면 인덱스 값을 반환하는 것으로 한다. 그러면 그 인덱스 값으로 원하는 테크트리 배열의 인덱스로 접근할 수 있다. 이는 Language 클래스에서 언어 배열에 접근하고자 할 때 사용했던 방법과 동일하다. 다만, 테크트리의 경우 같은 테크트리 뿐만 아니라 다른 테크트리로도 연결되어 있기 때문에, 링크를 대신하려 했던 SubNode 구조체는 노드의 이름 뿐만 아니라 어느 종류의 테크트리인지도 저장한다.

Node 클래스는 링크를 대신해 SubNode 구조체를 저장하는데, 여기서 필요한 정보는 해당 노드의 요구 조건은 무엇이고 다음 노드는 무엇이냐다. 즉, 이전 노드와 다음 노드, 이 둘은 서로 일치해야 된다. 다시 말하면, 이전 노드만 정해줘도 다음 노드는 이미 정해진 것과 마찬가지인 것이다. 둘 다 사람이 수동으로 정하게 되면 실수 때문에 이전 노드, 다음 노드가 서로 일치하지 않을 수도 있기 때문에 둘 중 하나만 사람이 정하도록 한다. 여기서는 이전 노드를 정하는 것이 더 편할 것 같아서 이전 노드만 정하게 했다. 이는 물론 기획자의 요구에 따라 달라질 것이다. 이전 노드는 사람이 정했으니 다음 노드만 저장하게 하면 되는데, 다음 노드의 개수는 불규칙하므로 일단 가변배열에 저장한다. 그러면 이 가변배열이 어느 노드에 대응하는지 정해야 되는데, 이 가변배열을 배열로 만들어 해당 노드의 인덱스에 대응하도록 했다. 그 다음으로는 그 노드 인덱스가 어느 테크트리의 인덱스인지도 정해야 된다. 따라서 배열을 이중배열로 만들어 어느 종류 테크트리 배열에서 어느 인덱스인지에 대응하도록 했다. 결과적으로 다음 노드를 담는 자료구조는 가변배열의 이중배열이다. 예시로, _nextNodes[시설][5]면 시설 테크트리의 5번 인덱스에 해당하는 노드의 다음 노드를 담은 가변배열에 접근하는 것이다.

9. TechTreeViewBase

가) 다음 노드 사용 가능 여부 확인

```
/// <summary>
/// 다음 노드 활성화 가능 여부
/// </summary>
protected virtual bool EnableCheck(TechTrees.SubNode nextNode)
{
    // 이전 노드로 설정된 것
    TechTrees.SubNode[] requiredNodes = NodeData[NodeIndex[nextNode.NodeName]].Requirments;
    for (byte i = 0; i < requiredNodes.Length; ++i)
    {
        // 모두 승인된 것이 아니면 거짓 반환
        if (0.0f >= Adopted[(int)requiredNodes[i].Type][NodeIndex[requiredNodes[i].NodeName]])
        {
            return false;
        }
    }

    // 모두 승인됐으면 참 반환
    return true;
```

```
    }
```

앞서 보았듯 다음 노드를 활성화하기 전에 해당 노드의 요구 조건이 모두 활성화 상태인지 확인해야 된다. 여기서 NodeData는 해당 테크트리의 노드를 담은 배열이고 NodeIndex는 노드의 인덱스를 반환하는 헤쉬테이블이다. 먼저 NodeIndex에서 확인하고 싶은 노드의 인덱스를 찾으면 NodeData에서 그 인덱스로 접근할 수 있다. 그러면 확인하고 싶은 노드가 반환되는 것인데, 그 노드의 요구 조건을 가져오면 된다. requiredNodes 지역 변수가 그 요구 조건의 배열을 참조한다. 그 배열을 하나씩 확인하면서 활성화가 된 것인지 확인하면 된다. 여기서는 승인이라고 표현했는데, 하나라도 승인되지 않았으면 그 즉시 거짓을 반환한다. 그렇다면 이 함수의 마지막까지 갔을 때는 요구 조건이 모두 승인된 것이므로 참을 반환한다. 승인 여부를 확인할 때 Adopted의 값을 확인하는데, Adopted는 이중배열이다. TechTrees에서 다음 노드를 가변배열의 이중배열에 저장했던 것과 같이 Adopted의 첫 번째 인덱스는 테크트리의 종류, 두 번째 인덱스는 노드의 인덱스가 들어가는 자리다.

나) 화면에 노드 배치

```
        // 노드 배치
        float sizeX = 0.0f;
        float sizeY = 0.0f;
        for (byte i = 0; i < length; ++i)
        {
            // 위치 계산
            float posX = (NodeData[i].NodePosition.x + 0.5f) * _width;
            float posY = (NodeData[i].NodePosition.y + 0.5f) * _height;

            // 노드 생성 후 위치 조정
            NodeBtnObjects[i] = Instantiate(NodeObject, _techTreeContentArea);
            NodeBtnObjects[i].transform.localPosition = new Vector3(posX, posY, 0.0f);

            // 노드 참조
            NodeBtns[i] = NodeBtnObjects[i].GetComponent<TechTreeNode>();

            // 노드 초기화
            NodeBtns[i].SetTechTree(this, i);

            // x, y 최대 값
            if (posX > sizeX)
            {
                sizeX = posX;
            }
            if (posY > sizeY)
            {
                sizeY = posY;
            }
        }

        // 전체 크기
        float areaWidth = sizeX + _width * 0.5f;
        float areaHeight = sizeY + _height * 0.5f;
        RectTransform contentArea = _techTreeContentArea.GetComponent<RectTransform>();
        contentArea.SetSizeWithCurrentAnchors(RectTransform.Axis.Horizontal, areaWidth);
        contentArea.SetSizeWithCurrentAnchors(RectTransform.Axis.Vertical, areaHeight);
```

```
        // 가운데 정렬
        float pivotX = 0.0f;
        float pivotY = 0.0f;
        if (Constants.TECHTREE_AREA_WIDTH > areaWidth)
        {
            pivotX = (Constants.TECHTREE_AREA_WIDTH_CENTER * Constants.TECHTREE_AREA_WIDTH -
areaWidth * 0.5f) / (Constants.TECHTREE_AREA_WIDTH - areaWidth);
        }
        if (_yCenterize && Constants.TECHTREE_AREA_HEIGHT > areaHeight)
        {
            pivotY = (Constants.TECHTREE_AREA_HEIGHT_CENTER * Constants.TECHTREE_AREA_HEIGHT -
areaHeight * 0.5f) / (Constants.TECHTREE_AREA_HEIGHT - areaHeight);
        }
        contentArea.pivot = new Vector2(pivotX, pivotY);
```

위 코드는 노드 정보에 따라 화면에 노드를 배치한다. 노드의 가로, 세로 간격은 SerializeField를 통해 유니티 에디터에서 수정 가능하다. 지역변수 sizeX, sizeY는 노드 위치를 정할 때 노드의 x, y 좌표 최대 값을 저장해서, 배치된 노드의 전체 크기를 알 수 있다. 모든 노드는 유니티 ScrollView의 Contents에 해당하는 오브젝트의 자식으로 배치되는데, Contents 오브젝트는 pivot의 위치에 따라 정렬 위치가 달라진다.

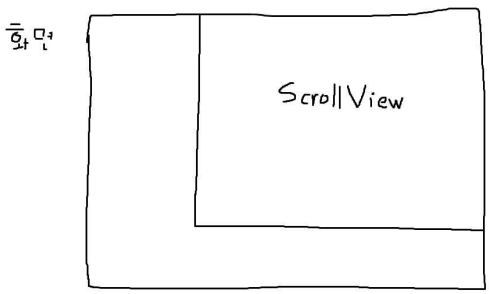

그림 라.9.0

화면 상에서 ScrollView는 위 그림 라.9.0처럼 한쪽으로 치우쳐져 있기 때문에 Contents 오브젝트가 화면 정 중앙에 위치하려면 pivot을 조정해야 된다. Contents 오브젝트의 크기는 sizeX, sizeY에 의해 결정되고, 그 크기에 따라 pivot도 달라진다.

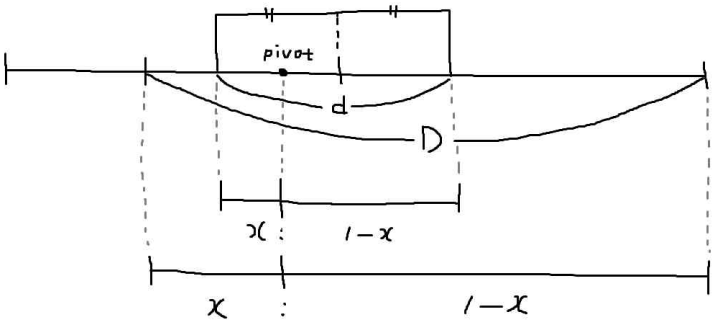

그림 라.9.1

pivot의 x값을 x라고 하자. 전체 화면 중 ScrollView의 가로 길이는 D고, Contents 오브직트의 가로 길이는 d다. pivot을 기준으로 각각의 좌측 길이와 우측 길이의 비율은 x:1-x가 된다. 여기서 전체 화면 크기(Cavas 크기)와 ScrollView 크기는 고정값이다. 그러므로 ScrollView의 좌측 끝에서 화면 중앙까지의 거리 또한 고정값이다. 전체 화면의 가로 길이는 1920 픽셀로 고정했고 ScrollView 죄측 끝의 위치는 전체 화면에서 1:4의 위치로 정했다. 그렇다면 ScrollView의 기준으로만 봤을 때의 화면 중앙의 위치는 아래 그림 라.9.2와 같이 나온다.

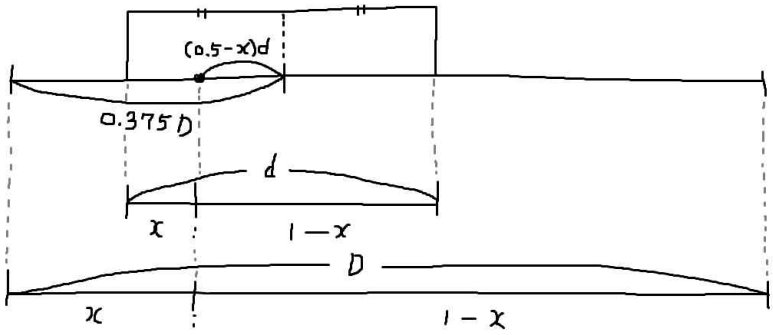

그림 라.9.2

위 그림에서 아래의 식이 성립한다.

$0.375D - (0.5 - x)d = xD$

이 식을 x에 대한 식으로 정리하면 다음과 같다.

$x = (0.375D - 0.5d) / (D - d)$

x가 pivot의 x값이다. 위 코드에서는 Constants.TECHTREE_AREA_WIDTH_CENTER가 0.375, Constantes.TECHTREE_AREA_WIDTH가 D, sizeX가 d에 해당한다. 그렇게 해서 나온 값을 pivotX에 저장한다. y축으로 가운데 정렬할 때로 이와 같은 식을 쓰면 된다. y축으로의 가운데 정렬은 필요한 경우가 있고 필요 없는 경우가 있기 때문에 bool 변수 _yCenterize로 bool 체크 후 계산하도록 한다. 최종 pivot 값은 Contents 오브젝트 pivot에 대입한다.

마. 주차별 진행 상황

1. 1주차 프로토타입

기본적인 화면 동작을 구현하고 물리 계산, 각 물리량이 평형을 유지하는지 확인했다. 그리고 물리량을 확인하기 위해 화면에 물리량 정보를 표시하는데, 이때 UIString 클래스를 만들어서 불필요한 문자열 생성을 줄였다. 또한 언어, 소리, 배경 애니메이션 등 프로그램의 기본적인 기능을 수행할 클래스를 작성했다.

2. 2주차 알파

테크트리 기능을 구현했다. 각 노드는 어떤 정보를 담을지, 그 노드를 어떤 자료구조에 담을지, 테크트리 화면과 노드의 배치는 어떻게 할지, 한 노드(테크)의 활성화 동작과 다음 노드로의 이동과 노드의 사용 가능 여부 확인 등 테크트리에 대한 모든 것에 대해 어떻게 하면 생산성 좋은 코드를 만들 수 있을지 1주일 내내 고민했다. 목표는 한 테크트리를 구현하면 나머지 테크트리를 쉽게 구현할 수 있게 하는 것이었는데, 한 테크트리를 구현한 후에도 다른 테크트리를 구현하면서 수정된 사항이 매우 많아 생산성이 높았다고 말할 수 없다.

3. 3주차 베타

기존 테크트리와 상이했던 사회 테크트리를 구현하고, 재사용 대기시간을 가지는 버튼을 구현했다. 구현된 것을 가지고 게임 컨셉 계획에 따라 미디어 문화, 외교, 무역 등의 기능을 구현했다.

PlayManager 클래스는 게임 진행 상황을 json 파일로 저장한다. 그리고 불러오기 기능으로 이전에 진행 중이었던 상황에 이어서 게임을 재개할 수 있다. 이 기능은 디버깅할 때 유용하게 사용할 수 있다.

4. 4주차 최종

GameManager가 게임 설정을 저장한다. 저장 기능은 PC, 모바일 모두 작동하도록 코드를 수정했다. 또한 시각적인 디자인이나 켄셉 계획에 따른 기능 구현을 완료했다. 프로젝트가 완성 단계에 이른 만큼 디버깅을 하는 데에 시간이 걸려서 개발자용 치트를 만들었다. 개발자용 치트를 이용해서 원하는 상황으로 빠르게 도달할 수 있다. 또한 윈도우, 안드로이드 플랫폼으로 빌드하고 테스트해서 추가적인 오류가 없는지 확인했다.

바. 개선할 점

1. 생산성 높은 코드를 구현하고 작업 시간을 단축하여는 목표가 있었으나 만족스럽지 않았다. 공통 기능과 개별 기능이 무엇인지 확실히 정하고 어떻게 추상화할 것인지 계획하는 것이 필요하다.
2. 코드 가독성 확인이 필요하다. 작성자 본인이 볼 때는 상관 없으나 다른 인원이 볼 때는 어떠할지 불확실하다.
3. 프로젝트 외적으로, GitHub에 commit할 때 commit 설명이 부실하다. 원래 GitHub는 백업 용도로만 사용해서 commit 설명은 대략적으로만 작성했는데, 구체적으로 작성해두면 작업 기록을 보기가 더 편할 것이다.