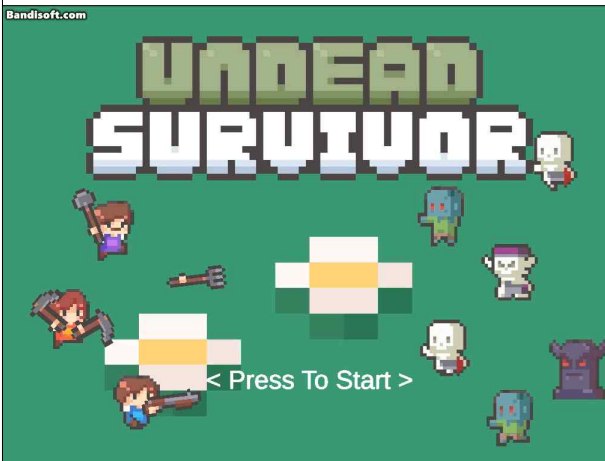





프로젝트명	Undead Survivor
개발 환경	Unity 2021.03.15.f1 / Github, 1인 개발 / 개발 기간 : 22/12:31 ~ 23/1/19(20日)
목표	다양한 연령층이 즐길 수 있는 사용자의 조작 난이도가 낮은 시스템의 뱀서라이크 게임을 개발
주요 기능	1. 싱글톤 패턴 기반 디자인 2. 오브젝트 풀링 3. 상속관계 4. 몬스터 5. 아이템 6. 플레이어 7. 무기
이미지	
	
	

기능 설명

1. 싱글톤 패턴

프로그램이 실행될 때 특정 클래스가 최초 단 한번만 메모리에 할당되고, 해당 메모리에서 인스턴스를 만들어서 사용하는 패턴이다. 이 프로젝트의 경우 Managers라는 클래스에서 자기 자신의 인스턴스를 static으로 선언하고, 각 기능을 담당하는 전용 Manager 클래스들의 인스턴스를 Manager에서 인스턴스화하여 게임 내 데이터를 공유한다.

● 싱글톤 패턴 사용 이유

1. **데이터의 공유** - static 선언된 Manager가 할당된 메모리에 데이터가 저장되기 때문에, 게임 내에서 자주 호출되는 데이터를 공유할 수 있다. UIManager를 통해 현재 생성된 UI의 종류, 현재 실행되는 Sound의 변경 혹은 추가, 현재 생성된 Player의 정보와 몬스터의 종류 등의 데이터들이 유지되기 때문에 필요할 때 데이터를 불러올 수 있다.
2. **메모리 낭비 방지** - 기본적으로 인스턴스화를 할 시 heap에 메모리가 쌓이게 된다. C#에서는 사용되지 않는 메모리를 알아서 해제해주는 가비지 컬렉터(Garbage Collection)이 존재하지만, 반복적인 생성과 해제로 인해 메모리 낭비가 발생한다. 싱글톤 패턴을 이용하면 static 영역에 메모리가 단 한번만 할당이 되고 재사용되기 때문에 new를 통해 반복되는 인스턴스화보다 메모리 낭비가 적다.

```
static Managers s_instance;
public static Managers Instance { get { Init(); return s_instance; } }

#region Contents
GameManagerEx _game = new GameManagerEx();

public static GameManagerEx Game { get { return s_instance._game; } }
#endregion

#region core
DataManager _data = new DataManager();
PoolManager _pool = new PoolManager();
ResourceManager _resource = new ResourceManager();
UIManager _ui = new UIManager();
SoundManager _sound = new SoundManager();
private EventManager _event = new EventManager();
private SceneManagerEx _scene = new SceneManagerEx();

public static DataManager Data { get { return Instance._data; } }
public static PoolManager Pool { get { return Instance._pool; } }
public static ResourceManager Resource { get { return Instance._resource; } }

public static UIManager UI { get { return Instance._ui; } }
public static SoundManager Sound { get { return Instance._sound; } }
public static EventManager Event { get { return Instance._event; } }
public static SceneManagerEx Scene { get { return Instance._scene; } }
static void Init()
```

Managers 컴포넌트에서 Manager를 static으로 선언하고, Manager에서 다른 Manager들을 인스턴스화하고 각 매니저들에 대한 정적 프로퍼티를 선언함으로써 각 매니저의 호출의 편의성을 증가시킨다. s_instance는 정적 할당하였기에 제거되지 않으므로 그 아래 선언된 다른 매니저들 또한 선언된 후 제거되지 않고 지속적인 재사용이 가능해진다.

<pre> { if (s_instance == null) { GameObject go = GameObject.Find("@Managers"); if (go == null) { go = new GameObject { name = "@Managers" }; go.AddComponent<Managers>(); } DontDestroyOnLoad(go); s_instance = go.GetComponent<Managers>(); s_instance._sound.Init(); s_instance._pool.Init(); s_instance._data.Init(); } } </pre>	<p>씬 내에 @Managers 게임오브젝트가 없다면(정적 선언을 하지 않았다면) 새로운 Managers 컴포넌트를 갖고 있는 오브젝트를 생성하고, s_instance에 Managers 컴포넌트를 저장함으로써 @Manager의 Managers 컴포넌트에 데이터를 저장하는 방식의 싱글톤 패턴이 완성된다. 아래는 각 매니저의 초기화를 실시하여 기본값을 저장한다.</p>
--	---

1-2 Manager의 종류

1. Game Manager : 게임 내에서 오브젝트 생성 및 소멸에 대한 이벤트를 관리한다.

```

{
    GameObject _player;

    // int <-> gameobject
    //Dictionary<int, GameObject> _player = new Dictionary<int, GameObject>();
    HashSet<GameObject> _monster = new HashSet<GameObject>();
    public Action<int> OnSpawnEvent;

    참조 5개
    public Vector3 MousePos { get; set; }
    참조 2개
    public Vector3 WorldMousePos { get; set; }
}

```

플레이어, 몬스터, 마우스 포지션 등의 데이터를 저장하고, 각 오브젝트 타입의 이벤트를 저장한다.

2. UIManager : UI의 생성 및 제거 등의 기능을 담당한다. UI에는 popupUI와 SceneUI, WorldObjectUI가 존재한다.

● PopupUI

필요에 따라 생성하고, 제거할 수 있는 UI.

나중에 생성된 UI는 최상단에 위치할 수 있도록 추가 팝업창이 생성될 때마다 sort Order값을 증가시키고, 닫히면 sort Order를 감소시킨다.

● SceneUI

씬이 실행될 때 기본적으로 생성되는 UI.

유저가 제거하거나 추가적으로 생성되지 않는다.

필요에 따라서 한 씬에 여러개의 SceneUI가 생성될 수도 있으나, 이 게임의 경우 각 씬 당 1개의 SceneUI가 배치되어 있다.

번호 순서대로 소유 무기 출력 이미지 패널, 게임시간 출력 Text, 경험치 슬라이더, Level 출력 텍스트이다.

● WorldSpaceUI

ScreenSpace가 아닌 WorldSpace에 존재하는 UI. 캐릭터별 HP바를 달아주거나 DamageText를 WorldSpace상에 생성할 수 있다.

```
private int _order = 10;

private Dictionary<Define.PopupUIGroup, Stack<UI_Popup>> _popupStackDict
= new Dictionary<Define.PopupUIGroup, Stack<UI_Popup>>();
private UI_Scene _sceneUI = null;
// UI 순서 배치
public void SetCanvas(GameObject go, bool sorting = false)
{
    Canvas canvas = go.GetOrAddComponent<Canvas>();
    canvas.renderMode = RenderMode.ScreenSpaceOverlay;
    canvas.overrideSorting = true;
    //배열이 필요한 UI는 order를 부여받고, 필요하지 않은 ui는 0으로 고정됨
    //기본적으로 popupUI는 order를 부여받고, 순서가 필요없는 sceneUI는 0이
    //부여된다.
    if (sorting)
        canvas.sortingOrder = _order++;
    else
        canvas.sortingOrder = 0;
}
// 팝업창 생성(입력받는 데이터 타입을 UI_Popup을 상속받는 클래스로 제한한다.
public T ShowPopupUI<T>(string name = null, bool forEffect = false) where
T : UI_Popup
{
    //UI가 생성되면 BGM을 잠시 줄인다.
    Managers.Sound.SetAudioVolumn(Define.Sound.Bgm,
    Managers.Sound.SoundVolumn/2);
    //popupUI 컴포넌트와 볼러울 프리팹의 이름이 같으므로 name에 컴포넌트명을
    //저장한다.
    if (string.IsNullOrEmpty(name))
        name = typeof(T).Name;
    //Prefab에서 name을 이름으로 갖는 UI Prefab 복사해서 생성
    //생성 UI의 집합을 확인하기 위해 Root 오브젝트의 자식으로 배치
    GameObject go = Managers.Resource.Instantiate($"UI/Popup/{name}");
    go.transform.SetParent(Root().transform);
    //GetOrAddComponent - 오브젝트에 관련 컴포넌트가 존재하면 그대로 값을 return,
    없으면 컴포넌트를 추가하고 return함
```

1. **_order**는 sorting order를 부여하기 위한 필드다.

2. 각 PopupUI는 그룹이 존재하며, 같은 그룹에 존재하는 팝업창의 경우 **같은 그룹의 팝업창을 한꺼번에 제거** 혹은 **순서대로 제거**할 필요가 있기 때문에 각 그룹별로 스택을 갖는 덱서너리 형태로 팝업창 데이터를 저장한다.

3. 각 UI를 프리팹으로 에셋에 저장하여 필요할 때 프리팹에서 복사하여 생성한다.

4. 제네릭을 통해 popupUP 형식의 클래스만 매개변수로 받을 수 있도록 제한함으로써 SceneUI를 입력받는 오류를 예방하였다.

<pre> T popup = go.GetOrAddComponent<T>(); Define.PopupUIGroup popupType = popup._popupID; //현재 Popup타입이 딕셔너리에 저장되어있지 않다면, 새로 키를 추가하고 저장, 아니면 그대로 스택에 추가한다. if (!_popupStackDict.ContainsKey(popupType)) _popupStackDict.Add(popupType, new Stack<UI_Popup>()); _popupStackDict[popupType].Push(popup); //UI중엔 스킬 이펙트를 위한 UI도 존재한다. Effect용 UI는 SceneUI보다 낮은 값이 필요하므로 sortOrder를 가장 낮은 값을 부여한다. if (forEffect) popup.GetComponent<Canvas>().sortingOrder = -1; return popup as T; } //팝업창 제거 public void ClosePopupUI(Define.PopupUIGroup popupType) { if (_popupStackDict.TryGetValue(popupType, out Stack<UI_Popup> popupStack) == false _popupStackDict[popupType].Count == 0) return; UI_Popup popup = _popupStackDict[popupType].Pop(); Managers.Resource.Destroy(popup.gameObject); _order--; popup = null; CheckPopupUICountAndRemove(); } public T ShowSceneUI<T>(string name = null) where T : UI_Scene { if (string.IsNullOrEmpty(name)) name = typeof(T).Name; GameObject go = Managers.Resource.Instantiate(\$"UI/Scene/{name}"); T sceneUI = Util.GetOrAddComponent<T>(go); _sceneUI = sceneUI; </pre>	<p>5. UI관련 컴포넌트들은 프리팹과 같은 이름을 가지고 있다. 그렇기 때문에 제네릭 매개변수만으로 UI 프리팹을 불러올 수 있고, 만약에 컴포넌트명과 프리팹명이 다를 경우 name에 프리팹명을 입력할 수 있다.</p> <p>6. 활성화/비활성화 방식으로 구현하지 않은 이유는 이후 추가적인 개발 도중 UI를 재배치하거나 기능의 수정이 필요할 때 스크립트와 유니티 둘 다 수정이 불가피해져 리팩토링 작업 도중 문제가 발생할 수도 있기 때문에 관리가 쉽게 생성/파괴 식으로 구현했다.</p> <p>7. popupUI를 제거할 땐 먼저 현재 popupUI가 존재하는지 확인하기 위해 _popupStackDict에 키가 존재하는지 확인한다. 나의 경우 현재 생성된 popupUI그룹의 개수를 확인하기 위해 popupUI 키에 저장된 stack의 count가 0일 경우(popupUI가 존재하지 않을 경우) 키를 제거함으로써 관리한다.</p> <p>8. 만약 관련 popupUI가 존재하지 않을 경우 그냥 return하고, 존재할 경우 해당popupUI의 최상단에 존재하는 Object를 파괴하고 order를 내림으로써 이후 생성될 popupUI의 sortOrder를 관리한다.</p>
---	---

<pre> go.transform.SetParent(Root().transform); return sceneUI; } public T MakeWorldSpaceUI<T>(Transform parent, string name = null) where T : UI_Base { if (string.IsNullOrEmpty(name)) name = typeof(T).Name; GameObject go = Managers.Resource.Instantiate(\$"UI/WorldSpace/{name}"); if (parent != null) go.transform.SetParent(parent); Canvas canvas = go.GetComponent<Canvas>(); canvas.renderMode = RenderMode.WorldSpace; canvas.worldCamera = Camera.main; //return go.GetOrAddComponent<T>(); return Util.GetOrAddComponent<T>(go); } void CheckPopupUICountInScene() { Debug.Log(\$"popupCount : {_popupStackDict.Count}"); foreach(Define.PopupUIGroup popupKey in _popupStackDict.Keys) { Debug.Log(\$"popupList : {popupKey}"); } if (_popupStackDict.Count == 0) { Managers.Sound.SetAudioVolumn(Define.Sound.Bgm, Managers.Sound.SoundVolumn); Managers.GamePlay(); } } } </pre>	<p>9. WorldSpaceUI의 경우 해당 UI가 사용되는 경우는 어떤 WorldObject의 위치에 종속적으로 생성되는 경우다. 예를 들어 플레이어의 HpBar, floated Damage Text등이 존재한다.</p> <p>10.RenderMode.WorldSpace를 부여하여 해당 UI를 ScreenSpace가 아닌 WorldSpace에 배치한다.</p> <p>11. CheckPopupUICountInScene()은 popupUI가 열려있지 않을 때 게임을 다시 실행하는 메소드다. 온라인 게임과 달리 싱글플레이 게임의 경우 popup창이 존재할 경우 게임을 일시정지하는 경우가 있다. 그렇기 때문에 popupUI가 하나라도 존재할 경우 게임은 일시정지해야 하므로 popupUI의 count를 popupUI를 닫을 때마다 확인해준다.</p>
3. SoundManager : 게임 내에서 실행되는 BGM과 Effect 등의 Sound 실행 및 변경을 담당한다.	
<pre> //실행되는 audioSource의 종류를 구분하기 위한 배열이다. (BGM, Effect) AudioSource[] _audioSources = new AudioSource[(int)Define.Sound.MaxCount]; //Effect Sound를 저장하는 덱서너리. 한번 저장되면 이후 다시 필요할 때 다시 //생성할 필요 없이 덱서너리에 저장된 값을 가져오면 된다. Dictionary<string, AudioClip> _audioClips = new Dictionary<string, AudioClip>(); </pre>	<p>1. audioSource의 종류는 BGM과 Effect로 구분되는데, BGM의 경우 audioSource의 Loop값이 True로 설정되어 음악이 반복되고, Effect의 경우 단 한번만 실행되는 차이점이 존재한다.</p>

<pre> public void Init() { GameObject root = GameObject.Find("@Sound"); if (root == null) { root = new GameObject { name = "@Sound" }; Object.DontDestroyOnLoad(root); string[] soundNames = System.Enum.GetNames(typeof(Define.Sound)); for (int i = 0; i < soundNames.Length - 1; i++) { GameObject go = new GameObject { name = soundNames[i] }; _audioSources[i] = go.AddComponent<audiosource>(); go.transform.parent = root.transform; } _audioSources[(int)Define.Sound.Bgm].loop = true; } } public void Play(AudioClip audioClip, Define.Sound type = Define.Sound.Effect, float pitch = 1.0f) { if (audioClip == null) return; if (type == Define.Sound.Bgm) { AudioSource audioSource = _audioSources[(int)Define.Sound.Bgm]; if (audioSource.isPlaying) audioSource.Stop(); BGM = (Define.BGMs)System.Enum.Parse(typeof(Define.BGMs), audioClip.name); audioSource.pitch = pitch; audioSource.clip = audioClip; audioSource.Play(); } else { AudioSource audioSource = _audioSources[(int)Define.Sound.Effect]; audioSource.pitch = pitch; audioSource.PlayOneShot(audioClip); } } </audiosource></pre>	<p>2. enum으로 저장된 Sound type명으로 각 Sound의 설정을 저장할 Component를 부착할 오브젝트를 생성하고 이름을 부여한다.</p> <p>3. BGM은 음악이 종료되면 반복되어야 하므로 loop를 true로 설정한다.</p> <p>4. BGM을 교체할 경우 현재 BGM audioSource의 재생을 멈추고, 불러올 AudioClip으로 변경하고 재생한다.</p> <p>5. Effect의 경우 각 Effect Sound 별로 audioSource를 부여하고 한번만 실행하고 종료되기 때문에 Effect에 해당하는 audioSource를 가져오고, 해당 audioSource에 AudioClip을 저장하지 않고, playOneShot(audioClip)을 통해 한번 실행하고 종료시킨다. 이를 이용하면 한번에 여러 효과음을 동시에 실행할 수 있다.</p>
--	--

4. **ResourceManager** : 자원의 Load, Instantiate, Destroy의 동작 과정을 관리한다.

- 이를 따로 관리하는 Manager를 만든 이유는 해당 자원을 불러오지 못하거나 처리하지 못했을 경우 **NullException Error 디버그 오류**가 발생할 수 있기 때문에 따로 예외를 추가하여 **디버그 오류를 회피**할 수 있고 **오브젝트 풀링**의 경우같은 예외적인 방식을 처리하기 위해서다.

EX)

```
public GameObject Instantiate(string path, Transform parent = null)
{
    GameObject original = Load<GameObject>($"Prefabs/{path}");
    //에셋에 저장된 프리팹을 불러오는데 실패할 경우, 로그를 출력함으로써 오류를 회피할 수 있다.
    if (original == null)
    {
        Debug.Log($"Failed to load prefab : {path}");
        return null;
    }
    //Poolable Object에는 Poolable이라는 컴포넌트를 부착하여 non-Poolable Object와 다른 생성방식으로 호출한다.
    if (original.GetComponent<Poolable>() != null)
        return Managers.Pool.Pop(original, parent).gameObject;

    GameObject go = Object.Instantiate(original, parent);
    go.name = original.name;
    return go;
}
```

5. **PoolManager** : 오브젝트 풀링에 관련된 기능을 관리한다.

2. 오브젝트 풀링

백서라이크 계열의 게임은 같은 형태의 오브젝트가 대량으로 생성되고 파괴되는 것이 반복되는 형태다. 이를 지속적인 생성 및 파괴로 관리하면 **생성될 때마다 힙의 새로운 메모리로 할당**되고 **파괴될 때 메모리 해제를 반복**하여 성능에 부하를 준다. 그렇기 때문에 지속적인 메모리 부하를 방지할 수 있는 **오브젝트 풀링 기법**이 필수적이다.

오브젝트 풀링 기법은 사용할 **오브젝트를 미리 생성하여 Pool에 저장**하고, 필요할 때마다 **Pool에서 빼내어 사용**하고 사용 종료 시 다시 **Pool에 반납**한다. 사용하는 Object 개수를 제한하거나, 혹은 추가적인 Object가 필요할 경우에만 추가 생성을 하기 때문에 **CPU의 성능 소모를 줄이고 게임 속도를 향상**시킬 수 있다.

```
public class PoolManager
{
    #region Pool
    class Pool
    {
        //원본 오브젝트
        public GameObject Original { get; private set; }
        //오브젝트 풀 역할의 Root Object
        public Transform Root { get; set; }
        //Poolable Object를 저장하는 poolStack. stack이 아니라 queue를 사용해도 된다.
        Stack<Poolable> _poolStack = new Stack<Poolable>();
    }
}
```



```

//original Object를 Pooling할 Pool이 존재하지 않을 경우 Pool을 생성한다.
public void Init(GameObject original, int count = 5)
{
    Original = original;
    Root = new GameObject().transform;
    Root.name = $"{Original.name}_Root";
    for (int i = 0; i < count; i++)
        Push(Create());
}

//Pool에 저장되는 Poolable Object를 생성한다.
Poolable Create()
{
    GameObject go = Object.Instantiate<GameObject>(Original);
    go.name = Original.name;
    return go.GetComponent<Poolable>();
}

//생성된 Poolable Object를 pool에 배치한다.
public void Push(Poolable poolable)
{
    //object가 존재하지 않을 시 배치하지 않는다.
    if (poolable == null)
        return;

    //pool역할을 하는 Object의 자식오브젝트로 배치하고, 비활성화하여 사용되지 않음을 확인시킨다.
    poolable.transform.parent = Root;
    poolable.gameObject.SetActive(false);
    poolable.isUsing = false;

    //poolStack에 저장
    _poolStack.Push(poolable);
}

//pool에 배치된 오브젝트를 꺼내온다.
public Poolable Pop(Transform parent)
{
    Poolable poolable = null;

    //pool에 남아있는 Poolable Object가 있는지 확인한다. 없을 경우 새로 생성.
    while (_poolStack.Count > 0)
    {
        poolable = _poolStack.Pop();
        if (poolable.gameObject.activeSelf == false)
            break;
    }

    //pool에 남아있는 Object가 없다면 원본을 통해 새로 생성하여 꺼내온다.
    if (poolable == null || poolable.gameObject.activeSelf == true)
        poolable = Create();
    poolable.gameObject.SetActive(true);

    //부모 오브젝트로 배치되지 않을 경우 Scene 위치에 배치된다.
    if (parent == null)
        poolable.transform.parent = Managers.Scene.CurrentScene.transform;

    poolable.transform.parent = parent;
    poolable.isUsing = true;
    return poolable;
}

```

```

    }
}
#endregion
//전체 Pool을 담당하는 딕셔너리. 각 Pool마다 배치되는 Poolable Object가 다르기 때문에 각각의 원본마다 Pool을
따로 생성하고 관리해줘야한다.
Dictionary<string, Pool> _pool = new Dictionary<string, Pool>();
Transform _root;
public void Init()
{
    //전체 Pool Object를 부모 오브젝트 "@Pool_Root"에 배치하여 관리한다.
    if (_root == null)
    {
        _root = new GameObject { name = "@Pool_Root" }.transform;
        Object.DontDestroyOnLoad(_root);
    }
}
//Pool사용을 다 한 Object를 해당 원본이 존재하는 Pool에 다시 배치한다.
public void Push(Poolable poolable, float time)
{
    //해당 원본을 담당하는 Pool이 존재하는지 확인. 존재하지 않을 경우 Poolable Object가 아니라는
    의미이므로 해당 오브젝트를 파괴한다.
    string name = poolable.gameObject.name;
    if (_pool.ContainsKey(name) == false)
    {
        GameObject.Destroy(poolable.gameObject, time);
        return;
    }
    //Pool이 존재할 경우 해당 pool에 저장한다.
    _pool[name].Push(poolable);
}
//해당 원본에 해당하는 오브젝트를 pool에서 꺼낸다.
public Poolable Pop(GameObject original, Transform parent = null)
{
    //원본에 해당되는 pool이 존재하지 않을 경우 pool을 생성
    if (_pool.ContainsKey(original.name) == false)
        CreatePool(original);
    //해당 pool의 Object를 꺼낸다.
    return _pool[original.name].Pop(parent);
}
//원본을 배치하는 pool을 생성한다.
public void CreatePool(GameObject original, int count = 5)
{
    //pool 객체를 생성하고 해당 원본과 지정 개수에 맞춰 pool을 초기화한다.
    Pool pool = new Pool();
    pool.Init(original, count);
    pool.Root.parent = _root;
    //해당 풀을 pool을 _pool 딕셔너리에 추가
    _pool.Add(original.name, pool);
}
//해당 풀의 원본을 가져온다.
public GameObject GetOriginal(string name)

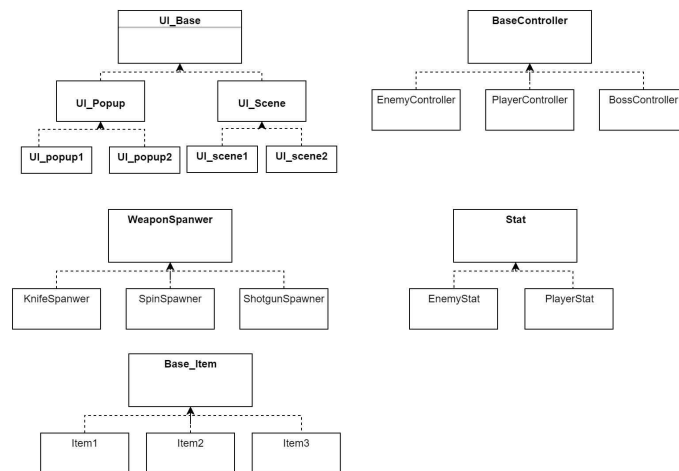
```

<pre> { if (_pool.ContainsKey(name) == false) return null; return _pool[name].Original; } //모든 pool을 제거하여 초기화한다. public void Clear() { foreach (Transform child in _root) { GameObject.Destroy(child.gameObject); } _pool.Clear(); } } </pre>	
<p>6. SceneManager : 현재 Scene에 대한 정보를 저장하고 다른 Scene을 불러오는 기능을 담당합니다.</p>	
<p>7. DataManager : 게임 내의 고정적인 데이터를 저장 및 불러온다. 데이터는 Json형식의 파일로 저장되어 있으며, Weapon, Player, Enemy에 대한 기본 데이터를 불러온다.</p> <p>● DataManager의 필요성</p> <p>게임 내에선 몬스터마다 능력치가 다르고, 또 레벨별, 직업별, 무기별로 각각 다른 수치와 다른 기능을 갖고 있다. 이를 효율적으로 관리하기 위해 각 형태마다 하나의 파일로 데이터를 저장하여 필요할 때마다 파일을 통해 해당하는 데이터를 가져올 필요가 있다.</p>	
<pre> //데이터를 불러와 저장하는 클래스 public interface ILoader<Key, Value> { Dictionary<Key, Value> MakeDict(); } public class DataManager { public Dictionary<int, Data.WeaponData> WeaponData { get; private set; } = new Dictionary<int, Data.WeaponData>(); public Dictionary<int, Data.Player> PlayerData { get; private set; } = new Dictionary<int, Data.Player>(); public Dictionary<int, Data.Monster> MonsterData { get; private set; } = new Dictionary<int, Data.Monster>(); public void Init() { PlayerData = LoadJson<Data.PlayerData, int, Data.Player>("PlayerData").MakeDict(); WeaponData = LoadJson<Data.WeaponDataLoader, int, Data.WeaponData>("WeaponData").MakeDict(); MonsterData = LoadJson<Data.MonsterData, int, Data.Monster>("MonsterData").MakeDict(); } } </pre>	<p>동작 과정</p> <ol style="list-style-type: none"> 1. DataManager에서 각 데이터를 저장하는 딕셔너리를 생성한다. 2. Init() - 각 데이터를 해당하는 딕셔너리에 저장하는 초기화 과정이다. 각 데이터에 해당되는 json파일을 불러온다. 3. LoadJson - 해당 Json파일에서 데이터를 불러온다. Json파일을 텍스트 형식으로 불러온 후, FromJson을 통해 텍스트 형식에서 클래스 형식으로 변환한다. 각 데이터마다 불러오는 클래스 형식이 다르기 때문에, Loader에 변환할 형태의 클래스를

<pre> Loader LoadJson<Loader, Key, Value>(string path) where Loader : ILoader<Key, Value> { TextAsset textAsset = Managers.Resource.Load<TextAsset>(\$"Data/{path}"); return JsonUtility.FromJson<Loader>(textAsset.text); } [Serializable] //Playable Character 데이터를 저장하는 클래스 public class Player { public int id; public string name; public int weaponID; public int maxHp; public int damage; public int defense; public float moveSpeed; public int coolDown; public int amount; } [Serializable] //Playable Character의 id를 key로 하고 데이터를 value로 갖는 딕셔너리를 생성 public class PlayerData : ILoader<int, Player> { public List<Player> players = new List<Player>(); public Dictionary<int, Player> MakeDict() { Dictionary<int, Player> dict = new Dictionary<int, Player>(); foreach (Player player in players) dict.Add(player.id, player); return dict; } } </pre>	<p>지정하여 각 클래스 타입으로 변환하는 방식으로 메소드를 재사용한다. Loader Interface를 상속받은 클래스이어야만 한다.</p> <p>4. PlayerData를 예시로 설명한다. Playable Character에 관련된 데이터는 Player라는 이름의 클래스와 같은 형식으로 저장되어 있다. Init()에서 LoadJson을 통해 데이터를 List<Player> 형태로 불러왔고, Loader에 해당되는 PlayerData에 List<Player> players로 저장되었다.</p> <p>5. MakeDict()를 실행한다. Player 데이터를 반환할 딕셔너리를 생성하고, List에 저장된 데이터를 순차적으로 불러와 Dictionary에 원하는 player를 특정할 수 있는 key와 player 객체를 추가하고 해당 딕셔너리를 반환한다.</p> <p>6. 이를 통해 PlayerData에 Player.id에 해당되는 Player 데이터가 저장되어 필요할 때마다 Managers를 통해 호출할 수 있다.</p>
<p>8. EventManager : 공용으로 사용되는 게임 전용 메소드를 저장하고 필요할 때에 호출하여 실행하기 위한 매니저. 각 스크립트별로도 전용 메소드를 저장하고 호출할 수 있지만, 전용 메소드들을 한번에 관리하기 위해 전용 매니저를 만들었다.</p> <p>LevelUpOverEvent ex)</p> <p>가. SetRandomWeaponFromItemBox - 아이템박스 내 랜덤 무기를 선택하는 메소드 나. LevelUpOverEvent - 레벨업이 종료됐을 때 플레이어의 능력치 및 무기를 갱신하는 메소드 다. etc...</p>	

3. 상속 관계 - 다형성

공통으로 사용되는 메소드가 존재하는 클래스들의 경우 부모 클래스에 메소드를 정의하여 자식 클래스에서 부모 클래스의 메소드를 호출하여 사용하는 방식으로 **코드 재사용성**을 높이고 **코드의 추가 및 변경**이 용이해진다. 또한, 클래스 간 **계층적 분류를 통한 관리**가 가능해지며, 다형성을 통해 여러 객체를 하나의 타입으로 관리할 수 있기 때문에 **유지보수성**이 증가한다.



```

public abstract class BaseController : MonoBehaviour
{
    protected Rigidbody2D _rigid;
    protected SpriteRenderer _sprite;
    public Animator _anime;
    public Define.WorldObject _type = Define.WorldObject.Unknown;

    private void Awake()
    {
        Init();
    }

    public abstract void OnDead();

    public abstract void OnDamaged(int damage, float force = 0);

    protected abstract void Init();
}

public class PlayerController : BaseController
{
    protected PlayerStat _stat;
    public RuntimeAnimatorController[] animeCon;
    [SerializeField] Vector2 _inputVec;
    [SerializeField] public Vector2 _lastDirVec = new Vector2(1, 0);
    bool _isDamaged = false;
    float _invincibility_time = 0.2f;
}
    
```

1. _Controller에 해당하는 클래스들의 경우 Rigidbody2D, SpriteRenderer, Animator, WorldObject 필드를 공통적으로 갖고 있다. 부모 클래스인 BaseController에서 해당 필드들을 선언하고 자식 클래스에서 해당 필드들을 사용한다.

2. OnDead(), OnDamaged(), Init()는 BaseController라는 추상 클래스에서 선언한 추상 메소드다. 이는 아래 자식 클래스에게 공통적으로 존재하는 클래스이나 각 자식클래스마다 해당 메소드의 동작과정이 다르기 때문에 메소드 기능의 변경을 강제하여 각 클래스별로 비슷하지만 다른 동작을 실시할 수 있게 해준다.

<pre> Slider _slider; protected override void Init() { _stat = GetComponent<PlayerStat>(); _rigid = GetComponent<Rigidbody2D>(); _sprite = GetComponent<SpriteRenderer>(); _anime = GetComponent<Animator>(); _type = Define.WorldObject.Player; if (gameObject.GetComponentInChildren<UI_HPBar>() == null) Managers.UI.MakeWorldSpaceUI<UI_HPBar>(transform, "UI_HPBar"); } ... } public class EnemyController : BaseController { public GameObject hudDamageText; protected EnemyStat _stat; public RuntimeAnimatorController[] animeCon; public Rigidbody2D _target; bool _isLive = true; bool _isRange = false; bool _isAttack = false; protected override void Init() { _stat = GetComponent<EnemyStat>(); _rigid = GetComponent<Rigidbody2D>(); _sprite = GetComponent<SpriteRenderer>(); _anime = GetComponent<Animator>(); _type = Define.WorldObject.Enemy; _target = Managers.Game.getPlayer().GetComponent<Rigidbody2D>(); } ... } </pre>	<p>3. PlayerController의 Init() 경우 해당 _type의 초기화 값이 Define.WorldObject.Player이고, UI_HPBar라는 WorldSpaceUI를 생성하여 배치하는 과정이 추가된다.</p> <p>4. EnemyController의 init()의 경우, _type의 초기화 값이 Define.WorldObject.Enemy이고, _target이라는 필드가 존재하는데, Init()에서 _target을 Player로 지정해주는 과정이 추가된다.</p> <p>5. 이처럼 상속을 통해 비슷한 형태의 메소드를 재사용하고 또는 기능을 확장할 수 있도록 상속관계를 설정해주었다.</p>
<p>다형성을 이용하지 않을 경우의 코드</p> <pre> switch(type){ case Define.Knife: weaponPool.GetComponentInChildren<KnifeController>().Level = weapon.Value; break; case Define.Fireball: weaponPool.GetComponentInChildren<FireballController>().Level = weapon.Value; break; ... } </pre>	<p>1. 각 Weapon은 같은 필드를 가지고 있지만 각자 다른 이름의 컴포넌트를 가지고 있다. 각 무기일 때의 조건을 설정하여 유지보수를 한다면 이후 새로운 weapon이 추가될 경우 조건 설정을 통한 유지보수를 한 코드를 다시 찾아내어 추가한 무기에 대한 조건을 지속적으로 추가해야 하기 때문에 코드 복잡도와 유지보수성이 떨어진다.</p>

<pre>} 다형성을 이용한 유지보수 적용 코드 weaponPool.GetComponentInChildren<WeaponController>().Level = weapon.Value;</pre>	<p>2. 각 weapon들의 Controller는 WeaoonController라는 부모 클래스에게 상속받고 있다. 각 클래스별로 관리하지 않고 부모클래스를 이용한 하나의 타입으로 관리할 수 있기 때문에 이후 코드의 확장이 용이해지고 유지보수성이 올라간다.</p>
--	--

4. 몬스터

▶ 3-1 몬스터 종류

몬스터는 여러 종류가 존재하며 각 종류에 따라 기본 능력치, 능력이 다르게 설정되어있다.



1. **Zombie** - 능력치가 가장 낮다. 속도가 느리고 낮은 데미지를 준다.
2. **EliteZombie** - Zombie의 강화형. 체력이 2배 높다.
3. **Skeleton** - 체력이 낮은 대신 공격력이 높으며 속도가 빠르다.
4. **EliteSkeleton** - Skeleton의 강화형. 체력이 1.5배 높으며 속도가 매우 빠르다.
5. **TombStone** - 원거리 유닛. 속도가 매우 느리지만 체력이 매우 높고 플레이어를 향해 총알을 발사한다.

▶ 3-2 몬스터 타입

몬스터는 크게 3가지로 분류되어 있으며, Enemy, MiddleBoss, Boss로 분류되어 있다.

Enemy : 일반형.

MiddleBoss : 중간보스. 1분마다 하나가 등장하며, 일반형보다 큰 능력치를 부여한다.

Boss : 보스. 독자적인 Controller를 가지고 있으며 스킬 시스템을 적용하여 스킬 재사용 대기 시간을 통해 스킬 사용 제한 조건을 만들고, 스킬 사용 조건이 만족될 경우 스킬을 시전한다.

몬스터 Spawn 과정

```
private void Update()
{
    //일정 시간이 지날 때마다 보스 몬스터 생성
    if ((timeLevel + 1) * 60 < Managers.GameTime)
    {
        timeLevel = (int)Managers.GameTime / 60;
        if (timeLevel <= 5)
        {
            Debug.Log($"{timeLevel}Boss Spawn!");
            SpawnBoss(timeLevel);
        }
    }
}
```

```

//생성 대기시간이 지나면 몬스터 생성
if (!_isSpawning)
    //코루틴을 사용하여 몬스터 생성 시간을 조절한다.
    StartCoroutine(SpawnMonster());
}
//몬스터 생성 코루틴
IEnumerator SpawnMonster()
{
    //몬스터 생성 도중에 추가적인 SpawnMonster 코루틴 실행을 중지
    _isSpawning = true;
    //생성 몬스터의 Count를 제한을 걸어놓는다. Count가 제한보다 높을 경우 생성되지 않음.
    if (enemyCount < _maxSpawnUnit)
    {
        //난수값을 이용하여 생성되는 몬스터 종류를 반환.
        //시간이 지날수록 고급 유닛이 생성될 가능성이 높도록 조정함.
        int monsterType = SetRandomMonster(timeLevel);
        //현재 플레이어의 레벨을 불러온다.
        int level = Managers.Game.getPlayer().GetComponent<PlayerStat>().Level;
        //기본 몬스터 프리팹을 통해 오브젝트 생성
        GameObject enemy = Managers.Game.Spawn(Define.WorldObject.Enemy, "Monster/Enemy");
        //플레이어 주변의 몬스터가 생성되는 장소인 SpawnPoint가 존재한다.
        //난수를 통해 랜덤으로 생성 장소를 지정하여 몬스터를 배치한다.
        enemy.transform.position = spawnPoint[Random.Range(1, spawnPoint.Length)].position;
        //플레이어의 레벨, 몬스터의 타입을 입력하여 몬스터의 능력치를 설정한다.
        enemy.GetOrAddComponent<EnemyController>().Init(monsterStat[monsterType], level, Define.MonsterType.Enemy);
    }
    //spawnTime만큼 코루틴을 중지시켜 몬스터 생성 시간에 제한을 둠
    yield return new WaitForSeconds(_spawnTime);
    _isSpawning = false;
}

```

5. 아이템

몬스터는 사망 시 Exp와 일정 확률로 Item을 생성한다. 캐릭터는 ItemGetterObject를 통해 특정 거리의 아이템의 존재를 확인할 수 있다. ItemGetterObject는 전용 Collider를 가지고 있으며, 아이템과의 충돌을 확인 시 아이템을 플레이어쪽으로 이동시킨다. 각 아이템마다 Update를 사용하는 것은 낭비가 심하다고 판단했기에, 충돌판정을 Item에서 하지 않고 ItemGetter에서 확인하고 Coroutine을 이용하여 아이템의 이동을 반복적인 형식으로 이동하도록 구현해냈다.

5-1 아이템을 끌어당기는 코루틴

```

IEnumerator GetItemInField(GameObject item)
{
    //아이템의 목적지를 설정. 만약 아이템이 이미 목적지가 설정되어있다면 끌어당겨지고 있다는 의미이기 때문에 해당 아이템에 대해 추가적인 coroutine 실행을 하지 않는다.
    if (item.GetOrAddComponent<Base_Item>().target != _player)
    {
        item.GetOrAddComponent<Base_Item>().target = _player;
        //아이템이 존재하지 않거나 비활성화될 경우 coroutine 실행을 종료한다. 아이템은 poolable아이템과 non-poolable 아이템이 존재하기 때문에 poolable 아이템은 비활성화하는 경우, non-poolable은 Destroy되는 경우를 확인한다.
    }
}

```



```

        while (item != null && item.activeSelf)
        {
            //FixedUpdate가 동작하는 형태처럼 아이템을 플레이어 방향으로 이동시킨다.
            item.transform.position =
                Vector3.MoveTowards(item.transform.position, _player.transform.position, _movSpeed *
Time.fixedDeltaTime);
            yield return new WaitForSeconds(Time.fixedDeltaTime);
        }
    }
}

```

5-2 아이템의 종류

각 아이템은 Base_Item에서 선언된 추상메소드인 OnEventItem()을 Override하여 Player와 충돌 시 OnEventItem() 메소드를 실행한다.

- **Exp** : 특정 값만큼 캐릭터의 Exp를 증가시킨다.
- **Health** : 특정 값만큼 캐릭터의 체력을 회복시킨다.
- **Magnet** : ItemGetter Object의 LocalScale을 100배 증가시켜 월드맵에 존재하는 모든 Item과 충돌시켜 아이템들을 플레이어에게 끌려오게 한다. 0.1초 후 자동으로 ItemGetter를 기본 사이즈로 변경한다.
- **ItemBox** : 캐릭터를 향해 움직이지 않는 Item이다. 획득 시 ItemBoxOpenUI를 생성하며, 이를 통해 랜덤으로 무기를 획득 및 강화할 수 있다. 만약 더이상 무기를 강화하거나 획득할 수 없다면 Health 아이템을 획득한다.

6. 플레이어

플레이어는 사용자가 움직일 수 있는 캐릭터다. 플레이어블 캐릭터는 2개가 존재하며, 각 캐릭터는 전용 무기와 스탯이 존재한다.

6-1 플레이어 스탯

각 캐릭터마다 능력치가 다르게 시작하며, 능력치에 따라 캐릭터의 독자적인 기능과 무기의 성능이 달라진다.

6-2 레벨업 시스템

플레이어가 Exp 프로퍼티에서 Exp에 값이 저장될 때, MaxExp보다 더 높은 Exp를 갖게 될 경우 LevelUpEvent를 실시한다. LevelUpEvent는 LevelUpUI를 생성하여 난수값을 통한 랜덤 선택을 통해 전용 무기, 스탯, 무기 중 3가지를 골라 리스트로 반환한다. 이후 UI의 GridPanel의 자식 오브젝트로 UpgdPanel SubItem을 3개 배치하고, UpgdPanel에 순차적으로 리스트의 정보를 저장한다. 선택지 중 하나를 선택하면 선택한 패널의 정보를 통해 스탯을 올리거나, 무기의 레벨을 올릴 수 있다.

6-3 소유 무기 스탯 설정

플레이어 능력치의 변화 혹은 새로운 무기를 획득했을 때마다 무기의 능력치를 다시 설정해준다.

<pre> Dictionary<Define.Weapons, int> _weaponDict = new Dictionary<Define.Weapons, int>(); void SetWeaponLevel() { //Set CommonWeaponSpawingPool foreach (KeyValuePair<Define.Weapons, int> weapon in GetWeaponDict()) { string weaponName = weapon.Key.ToString(); string weaponSpawningPool = weapon.Key + "SpawningPool"; GameObject weaponPool = Util.FindChild(gameObject, weaponSpawningPool, true); if (weaponPool == null) { weaponPool = Managers.Resource.Instantiate(\$"Weapon/SpawningPool/{weaponSpawningPool}", transform); } if (weapon.Value == 0) { GetWeaponDict().Remove(weapon.Key); Managers.Resource.Destroy(weaponPool); } weaponPool.GetComponentInChildren<WeaponController>().Level = weapon.Value; Managers.UI.getSceneUI().GetComponent<UI_Player>().SetWeaponImage(Managers.Game.getPlayer().GetComponent<PlayerStat>()); } } =====WeaponSpanwer===== public int Level { get { return _level; } set { _level = value; SetWeaponStat(); } } protected virtual void SetWeaponStat() { if (_level > 5) _level = 5; _damage = (int)(_weaponStat[_level].damage * ((float)(100+ _playerStat.Damage)/100f)); _movSpeed = _weaponStat[_level].movSpeed; _force = _weaponStat[_level].force; _cooldown = _weaponStat[_level].cooldown * (100f/(100f+_playerStat.Cooldown)); _size = _weaponStat[_level].size; _penetrate = _weaponStat[_level].penetrate; _countPerCreate = _weaponStat[_level].countPerCreate + _playerStat.Amount; } </pre>	<p>_weaponDict 현재 플레이어가 소유하고 있는 무기의 타입과 레벨을 저장하는 딕셔너리.</p> <p>SetWeaponLevel() _weaponDict에 저장된 데이터를 기준으로 무기의 stat을 갱신하거나 새로운 WeaponSpawner를 생성한다.</p> <ol style="list-style-type: none"> 1. _weaponDict를 반복문을 통해 각 데이터의 key와 value를 가져온다. 2. key를 통해 weaponName과 weaponStat을 가지고 있는 플레이어의 자식 오브젝트인 weaponSpawner를 검색한다. 3. 만약 해당 weaponSpawner가 존재하지 않는다면 weaponSpanwer를 새로 생성하고, Player의 자식 오브젝트로 배치한다. 4. weaponSpanwer의 weaponController 컴포넌트에 존재하는 level에 플레이어 정보에 저장된 해당 weapon의 level을 입력한다. weaponSpawner는 level의 set 프로퍼티를 통해 weaponStat을 재설정한다. 5. UI_Player의 소유 무기를 보여주는 ImagePanel에 현재 소유 무기 이미지를 갱신한다.
--	--

7. 무기

각 캐릭터는 전용 무기를 가지고 시작하며, 게임 진행에 따라 추가적인 무기 획득 및 무기의 능력치를 증가시킬 수 있다.

- 무기의 능력치는 무기의 레벨과 플레이어의 능력치에 따라 결정되며, DataManager를 통해 각 무기의 레벨별 능력치를 불러와 무기ID와 현재 무기 레벨에 맞는 데이터를 무기 능력치에 저장한다.

현재 구현 무기 - Knife, Fireball, SpinWeapon, Poison, Shotgun, Lightning



문제 정의

1. 싱글톤 패턴으로 인한 과도한 결합도 증가

싱글톤 패턴은 데이터의 재사용성과 메모리 낭비를 방지할 수 있지만 그만큼 같은 싱글톤 인스턴스를 여러 스크립트에서 데이터를 공유하여 사용 하게되면서 결합도가 매우 높아지게 되었다. 각 매니저를 통해 각 기능마다 하나의 기능을 수행하는 SRP를 만족하는 듯 하나 결과적으로 각 매니저들 또한 Managers와 결합되어 호출되기 때문에 Managers가 여러 가지 기능을 하게 되는 결과를 불러왔고, 추가적인 수정 혹은 기능 추가 시의 위험성 또한 커졌기에 OCP를 위반하는 형태가 되었다. 작은 프로젝트라서 상관없다고 생각했으나, 각 오브젝트가 서로에 대해 큰 결합도를 갖게 되었기 때문에 각 오브젝트를 특정 순서대로 불러내지 않으면 오류가 발생했다.

예를 들자면, 플레이어가 생성되면 SpawninfPool이 활성화되면서 Monster가 Spawn이 되는데, Player의 데이터를 GameManager에 저장하기 전에 Monster에서 아직 null인 Player 객체를 불러와 nullPointerException이 발생하는 경우가 존재했다.

또한 하나의 기능을 구현하여 테스트하려고 해도 하나의 군집체처럼 뗄 수가 없는 상황이 되었기 때문에 테스트조차 힘든 형태가 되었다. 또한 코드의 수정도 함부로 할 수 없는 상태가 되었다. 소형 프로젝트였음에도 불구하고 싱글톤 패턴의 단점인 높은 결합도의 위험성이 얼마나 큰지 알게 된 계기였다.

2. 라이프사이클에 대한 이해도 부족

라이프사이클에 대한 이해도의 부족으로 인해 애를 먹었다. Managers의 인스턴스가 아직 초기화되지 않았는데 Managers의 다른 Manager 객체의 필드를 불러오려고 하여 null값을 호출하는 경우가 많이 발생했다. Object의 생성 시 Awake를 많이 사용했는데, 이 때문에 Managers 객체가 초기화되기 전 생성된 오브젝트에서 static Instance를 불러오려고 하는 문제가 발생했다.

문제 해결 : 라이프사이클에 대한 중요성을 되새기고, 라이프사이클의 흐름을 공부하여 적절한 라이프사이클 함수를 이용할 수 있게 되었다.

3. 오브젝트 풀링 오류

오브젝트 풀링을 통한 오브젝트 관리 중에 이미 풀에서 꺼내온 오브젝트가 다시 재배치되는 경

우를 확인했다. 이로 인해 weapon Projectile 오브젝트가 실행 도중 초기 발생 위치로 이동하거나, 몬스터가 spawnPoint로 다시 재배치되는 등의 오류가 발생했다.

문제 해결 : 원인은 유니티 멀티쓰레딩 방식으로 판단했다. 내 오브젝트 풀링 방식은 오브젝트를 poolStack에 저장하고 호출 시 pool에서 꺼내고 파괴 시 pool에 다시 저장하는 방식인데, 풀에서 꺼내고 반납하는 과정이 동시에 발생하는 비동기 처리로 인해 poolStack에 pool에서 꺼내온 오브젝트가 활성화된 채로 다시 pool에 반납되었기 때문에 발생하는 문제였다. 이를 해결하기 위해 pool에서 꺼낼 때 오브젝트가 활성화 상태인지(pool에서 빼온 상태인지) 확인하고, 활성화상태가 아닌 오브젝트를 꺼내거나 pool에 오브젝트를 전부 꺼낸 후 새 pool 오브젝트를 생성후 꺼내오는 방식을 통해 해결했다.

```
//오브젝트 풀링 오류 해결 소스코드
public Poolable Pop(Transform parent)
{
    Poolable poolable = null;
    //pool에 남아있는 Poolable Object가 있는지 확인한다. 없을 경우 새로 생성.
    while (_poolStack.Count > 0)
    {
        poolable = _poolStack.Pop();
        if (poolable.gameObject.activeSelf == false)
            break;
    }
    //pool에 남아있는 Object가 없다면 원본을 통해 새로 생성하여 꺼내온다.
    if (poolable == null || poolable.gameObject.activeSelf == true)
        poolable = Create();
    poolable.gameObject.SetActive(true);
    //부모 오브젝트로 배치되지 않을 경우 Scene 위치에 배치된다.
    if (parent == null)
        poolable.transform.parent = Managers.Scene.CurrentScene.transform;

    poolable.transform.parent = parent;
    poolable.isUsing = true;
    return poolable;
}
```

결론 : 아직 멀티쓰레드에 대한 이해가 부족하기에 나중에 멀티쓰레딩을 배워 동기화하는 방식으로 문제를 더 효율적으로 처리하는 방법 필요.

참고자료	언데드 서바이버 에셋 팩 : https://assetstore.unity.com/packages/2d/undead-survivor-assets-pack-238068 2d Flat Explosion : https://assetstore.unity.com/packages/2d/textures-materials/2d-flat-explosion-66932 8-Bit Sfx : https://assetstore.unity.com/packages/audio/sound-fx/8-bit-sfx-32831 Casual Game BGM #5 : https://assetstore.unity.com/packages/audio/music/casual-game-bgm-5-135943 2D Potions Pixel Art : https://assetstore.unity.com/packages/2d/gui/icons/2d-potions-pixel-art-196023 Buttons Set : https://assetstore.unity.com/packages/2d/gui/buttons-set-211824 Monsters_Creatures_Fantasy : https://assetstore.unity.com/packages/2d/characters/monsters-creatures-fantasy-167949
Github 주소	https://github.com/newtron-vania/Vampire-Survivor
소개영상	https://www.youtube.com/watch?v=Xxo9mynXYRA

--	--