Sokoland

Sokobox + Island

Dev: leeinhwan0421, LHBM04, kimilljoo

포트폴리오 가이드

본 포트폴리오는 제가 직접 구현한 부분만을 기록하였습니다.

- Unity Engine을 활용한 게임 개발 과정에서 직접 구현한 부분을 기록하였습니다.
- 코드의 확장성과 유지보수를 고려하거나, 기능별 주요 구현 사항 및 기술적 해결 과정을 서술하여 제가 어떠한 사고를 가지고 논리적으로 문제를 해결하는지 확인하실 수 있게 구성하였습니다.
- 기여한 역할과 담당 기능을 명확히 구분하여 서술하였으며, 기획 및 아트 자산 등 다른 팀원의 작업은 포함하지 않았습니다.
- 질문이 있으신 경우, lsnan421@naver.com 또는 lsinhwan0421@gmail.com 으로 문의 메일 남겨주시면 최대한 빠른 시일 내에 답변하겠습니다.

오늘도, 행복하고 사랑스러운 하루 보내시길 바랍니다.

사용한 기술 스택

엔진 및 언어

Unity 6 (6000.0.31f1)

C# .NET Standard 2.1

환경

Visual Studio 2022

JetBrains Rider

버전 관리

Github

플러그인

- Dotween
- NaughtyAttributes
- UniRx
- UniTask

참여 인원

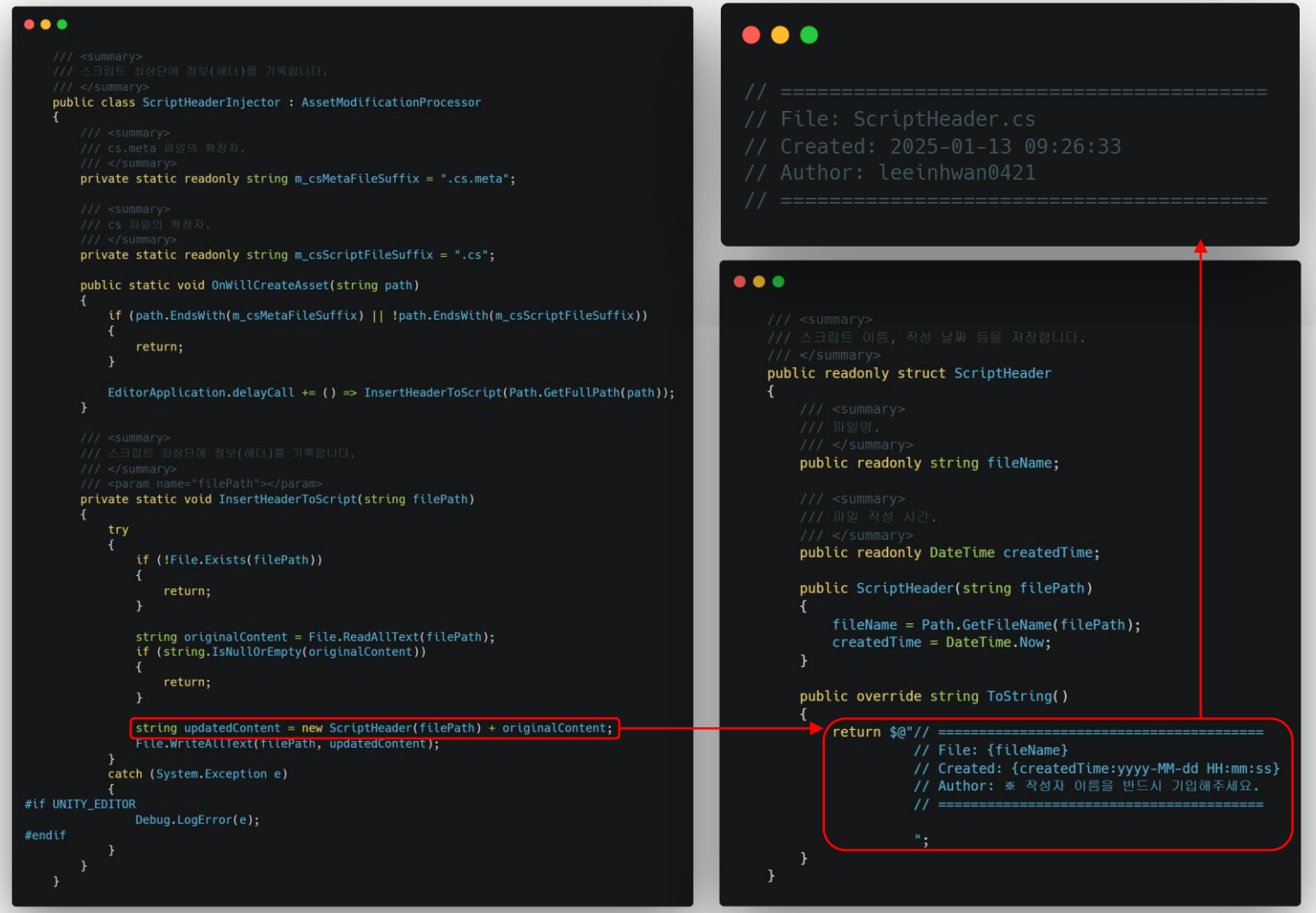
1. 개요

요약 각 인원의 **역할**과 **기여 내용**을 확인할 수 있습니다.

이름	역할	기여내용
이인환	- 프로젝트 리더 - 클라이언트 개발 - 레벨 디자인 - 기획	- 인게임 로직 작성 (플레이어, 기믹 등) - 각종 에디터 스크립트 작성 (스크린샷, 스크립트 헤더 등) - 유틸리티 스크립트 작성 - 레벨 디자인
이현범	- 클라이언트 개발 - 기획	- 오디오 스크립트 작성 - UI 스크립트 작성 - 씬 매니지먼트 작성 - 유틸리티 스크립트 작성
김일주	- UI 디자인	- Figma를 활용한 UI 디자인

유틸리티 코드

1. 자동으로 스크립트 헤더를 삽입해주는 클래스



```
/// <summary>
/// 스크립트 최상단에 정보(헤더)를 기록합니다.
/// </summary>
public class ScriptHeaderInjector : AssetModificationProcessor
{
    /// <summary>
    /// cs.meta 파일의 확장자.
    /// </summary>
    private static readonly string m_csMetaFileSuffix = ".cs.meta";

    /// <summary>
    /// cs 파일의 확장자.
    /// </summary>
    private static readonly string m_csScriptFileSuffix = ".cs";

    public static void OnWillCreateAsset(string path)
    {
        if (path.EndsWith(m_csMetaFileSuffix) || !path.EndsWith(m_csScriptFileSuffix))
        {
            return;
        }

        EditorApplication.delayCall += () => InsertHeaderToScript(Path.GetFullPath(path));
    }

    /// <summary>
    /// 스크립트 최상단에 정보(헤더)를 기록합니다.
    /// </summary>
    /// <param name="filePath"></param>
    private static void InsertHeaderToScript(string filePath)
    {
        try
        {
            if (!File.Exists(filePath))
            {
                return;
            }

            string originalContent = File.ReadAllText(filePath);
            if (string.IsNullOrEmpty(originalContent))
            {
                return;
            }

            string updatedContent = new ScriptHeader(filePath) + originalContent;
            File.WriteAllText(filePath, updatedContent);
        }
        catch (System.Exception e)
        {
#if UNITY_EDITOR
            Debug.LogError(e);
#endif
        }
    }
}
```

```
// ============================================
// File: ScriptHeader.cs
// Created: 2025-01-13 09:26:33
// Author: leeinhwan0421
// ============================================
```

```
/// <summary>
/// 스크립트 이름, 작성 날짜 등을 저장합니다.
/// </summary>
public readonly struct ScriptHeader
{
    /// <summary>
    /// 파일명.
    /// </summary>
    public readonly string fileName;

    /// <summary>
    /// 파일 작성 시간.
    /// </summary>
    public readonly DateTime createdTime;

    public ScriptHeader(string filePath)
    {
        fileName = Path.GetFileName(filePath);
        createdTime = DateTime.Now;
    }

    public override string ToString()
    {
        return $@"// ========================================
                // File: {fileName}
                // Created: {createdTime:yyyy-MM-dd HH:mm:ss}
                // Author: ※ 작성자 이름을 반드시 기입해주세요.
                // ========================================

                ";
    }
}
```

개발 의도

- 특정 파일에서 문제가 발생하면, 코드를 작성한 사람에게 빠르게 문의할 수 있는 효과를 기대하였습니다.
- 시간이 지나도, 누가 이 코드를 작성하였는지를 쉽게 추적할 수 있는 효과를 기대하였습니다.

작동 방식

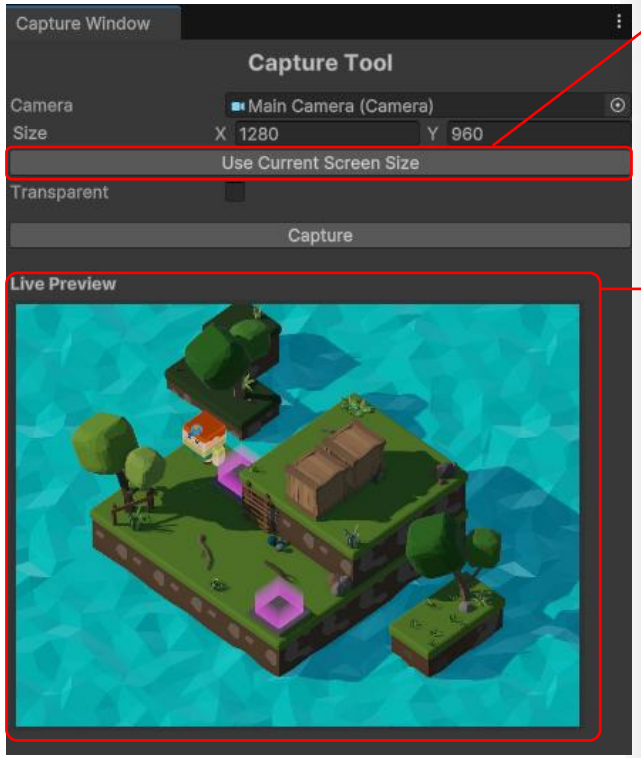
.meta 파일이 생성될 때 호출되는 OnWillCreateAsset 메서드를 활용하여

저장하려는 파일이 *.cs 확장자를 가지고 있을 경우

새로운 ScriptHeader 구조체를 생성하여 생성한 구조체의 ToString 메서드 반환 값을 새로 생성하려는 cs 파일 위에 붙여넣습니다.

유틸리티 코드

2-1. 에디터에서 편하게 게임 화면을 캡쳐할 수 있는 클래스 인스펙터




```
if (GUILayout.Button("Use Current Screen Size"))
{
    string[] screenres = UnityEditor.UnityStats.screenRes.Split('x');
    captureSize.x = int.Parse(screenres[0]);
    captureSize.y = int.Parse(screenres[1]);
}
```

```
/// <summary>
/// 실시간 미리보기 업데이트 (성능에 문제 있을 수 있습니다.)
/// </summary>
private void UpdatePreview()
{
    if (camera == null || captureSize.x <= 0 || captureSize.y <= 0)
    {
        return;
    }

    int captureWidth = (int)Mathf.Round(captureSize.x);
    int captureHeight = (int)Mathf.Round(captureSize.y);

    if (previewRenderTexture != null)
    {
        previewRenderTexture.Release();
        previewRenderTexture = null;
    }

    previewRenderTexture = new RenderTexture(captureWidth, captureHeight, 24);
    camera.targetTexture = previewRenderTexture;

    camera.Render();

    if (previewTexture == null ||
        previewTexture.width != captureWidth ||
        previewTexture.height != captureHeight)
    {
        previewTexture = new Texture2D(captureWidth, captureHeight, TextureFormat.RGB24, false);
    }

    RenderTexture.active = previewRenderTexture;
    previewTexture.ReadPixels(new Rect(0, 0, captureWidth, captureHeight), 0, 0);
    previewTexture.Apply();
    RenderTexture.active = null;

    camera.targetTexture = null;
}
```

원본 코드

- https://github.com/Bonnate/Unity-Editor-Screenshot-Capture

개발 의도

- 카메라가 여러 개 존재할 경우 결과를 확인하는 과정이 복잡해 단순하게 하기 위해 수정하였습니다.
- Size를 매번 해상도에 맞게 수정하는 것이 힘들어, 현재 Game View의 해상도를 사용하도록 수정하였습니다.

유틸리티 코드

2-2. 에디터에서 편하게 게임 화면을 캡쳐할 수 있는 클래스의 작동 방식

```
if (GUILayout.Button("Use Current Screen Size"))
{
    string[] screenres = UnityEditor.UnityStats.screenRes.Split('x');
    captureSize.x = int.Parse(screenres[0]);
    captureSize.y = int.Parse(screenres[1]);
}
```

```
/// <summary>
/// 실시간 미리보기 업데이트 (성능에 문제 있을 수 있습니다.)
/// </summary>
private void UpdatePreview()
{
    if (camera == null || captureSize.x <= 0 || captureSize.y <= 0)
    {
        return;
    }

    int captureWidth = (int)Mathf.Round(captureSize.x);
    int captureHeight = (int)Mathf.Round(captureSize.y);

    if (previewRenderTexture != null)
    {
        previewRenderTexture.Release();
        previewRenderTexture = null;
    }

    previewRenderTexture = new RenderTexture(captureWidth, captureHeight, 24);
    camera.targetTexture = previewRenderTexture;

    camera.Render();

    if (previewTexture == null ||
        previewTexture.width != captureWidth ||
        previewTexture.height != captureHeight)
    {
        previewTexture = new Texture2D(captureWidth, captureHeight, TextureFormat.RGB24, false);
    }

    RenderTexture.active = previewRenderTexture;
    previewTexture.ReadPixels(new Rect(0, 0, captureWidth, captureHeight), 0, 0);
    previewTexture.Apply();
    RenderTexture.active = null;

    camera.targetTexture = null;
}
```

원본 코드

- https://github.com/Bonnate/Unity-Editor-Screenshot-Capture

작동 방식

- Use Current Screen Size 버튼을 클릭한 경우 UnityEditor.UnityStats.screenRes의 값(1920 x 1080)을 불러와 'x' 문자를 기준으로 분리 한 뒤 Vector2 captureSize 변수에 넣어줍니다.

- OnGUI 메서드에서 previewTexture가 null이라면 텍스트를, previewTexture가 null이 아니라면 GUILayout.Box 메서드를 통해 Texture를 GUI에 출력하게 됩니다.

- OnGUI 메서드에서 UpdatePreview 메서드를 호출합니다. UpdatePreview 메서드에서 프리뷰를 출력할 수 있는 조건을 검사해보고 프리뷰를 출력할 수 있다면 Texture2D previewTexture 멤버 변수에 Texture를 저장합니다.

유틸리티 코드

3. 레이어 시스템

```cs
public static class LayerUtility
{
    /// <summary>
    /// 레이어 마스크 딕셔너리
    /// </summary>
    private static readonly Dictionary<ELayer, LayerMask> m_layers;

    /// <summary>
    /// 생성자
    /// </summary>
    static LayerUtility()
    {
        m_layers = Enum.GetValues(typeof(ELayer))
             .Cast<ELayer>()
             .ToDictionary(
             layer => layer,
             layerMask => (LayerMask)LayerMask.GetMask(layerMask.ToString())
             );
    }

    public static LayerMask GetLayerMask(ELayer layer)
    {
        if (m_layers.TryGetValue(layer, out LayerMask layerMask))
        {
            return layerMask;
        }

        Debug.LogWarning($"레이어 '{layer}'이(가) 존재하지 않습니다.");
        return default;
    }

    public static LayerMask GetCombinedLayerMask(ELayer[] layers)
    {
        LayerMask combinedMask = 0;

        foreach (ELayer layer in layers)
        {
            if (m_layers.TryGetValue(layer, out LayerMask layerMask))
            {
                combinedMask |= layerMask; // OR 연산
            }
            else
            {
                Debug.LogWarning($"레이어 '{layer}'이(가) 존재하지 않습니다.");
            }
        }

        return combinedMask;
    }


    public static bool IsEqualLayerMask(ELayer l1, int l2)
    {
        return (GetLayerMask(l1) & (1 << l2)) != 0;
    }
}
```

```cs
public enum ELayer
{
    Player,
    Ground,
    Iced,
    Box,
    BoxEjector,
    BoxPoint,
    Ladder,
    Portal,
    Spike,
    Obstacle,
    Interact
}
```

개발 의도

- String을 사용하여 태그를 사용하는 것보다. Layer로 처리하는 게 정말 괜찮다고 생각해서 레이어를 관리하는 LayerUtility 클래스를 작성하였습니다.

- 기존 사용하던 CompareTag("tag") 형태보다, 성능 부담이 적으면서도 코드가 간결한 클래스를 원했습니다.

작동 방식

- LayerMask.GetMask("LayerName")을 매번 호출하지 않고 딕셔너리에 캐싱하여 성능 부담을 줄이고, 레이어 연산을 편히 할 수 있도록 여러 메서드를 추가하였습니다.

- 왼쪽 스크립트에서는 나오지 않지만, 전체 코드에서 메서드 오버로딩을 통해 아래 메서드를 구현해두었습니다.

    - IsEqualLayerMask(ELayer, int);
    - IsEqualLayerMask(ELayer, LayerMask);

    - IsEqualLayerMask(LayerMask, int);

플레이어 코드

1. 플레이어 컨트롤러 클래스

```
/// <summary>
/// 플레이어 컨트롤러 클래스
/// </summary>
public class PlayerController : MonoBehaviour
{
    #region Fields
    /// <summary>
    /// 플레이어가 죽었을 때 Dispose 되는 컨테이너
    /// </summary>
    private CompositeDisposable m_Disposables;

    /// <summary>
    /// 죽는데 기준이 되는 Y 값, 플레이어의 y 값이 이 값 아래로 떨어지면 게임 오버된다.
    /// </summary>
    private readonly float m_FallDeathThresholdY = -5f;
    #endregion

    #region Properties
    public PlayerMovement Moves
    {
        get;
        private set;
    }

    private PlayerAnimator m_Animator;
    #endregion

    private void Reset()
    {
        Moves = GetComponent<PlayerMovement>();
        m_Animator = GetComponent<PlayerAnimator>();
    }

    private void Awake()
    {
        Moves = Moves != null ? Moves : GetComponent<PlayerMovement>();
        m_Animator = m_Animator != null ? m_Animator : GetComponent<PlayerAnimator>();

        m_Disposables = new CompositeDisposable();
    }

    private void Start()
    {
        Observable.EveryFixedUpdate().Subscribe(_ =>
        {
            if (transform.position.y <= m_FallDeathThresholdY)
            {
                OnDie();
            }
        }).AddTo(m_Disposables);
    }

    private void OnDestroy()
    {
        m_Disposables?.Dispose();
    }

    public void OnDie()
    {
        Moves.StopMoveToPosition();
        m_Disposables?.Dispose();

        m_Animator.Play(AnimatorState.Death, AnimatorLayer.BaseLayer);

        if (TryGetComponent(out Rigidbody rigid))
        {
            rigid.useGravity = false;
            rigid.isKinematic = true;
        }
    }
}
```

개발 의도

- 플레이어의 행동과 상태를 한 곳에서 관리하는 컴포넌트를 만들어 유지보수성을 높이려고 하였습니다.

작동 방식

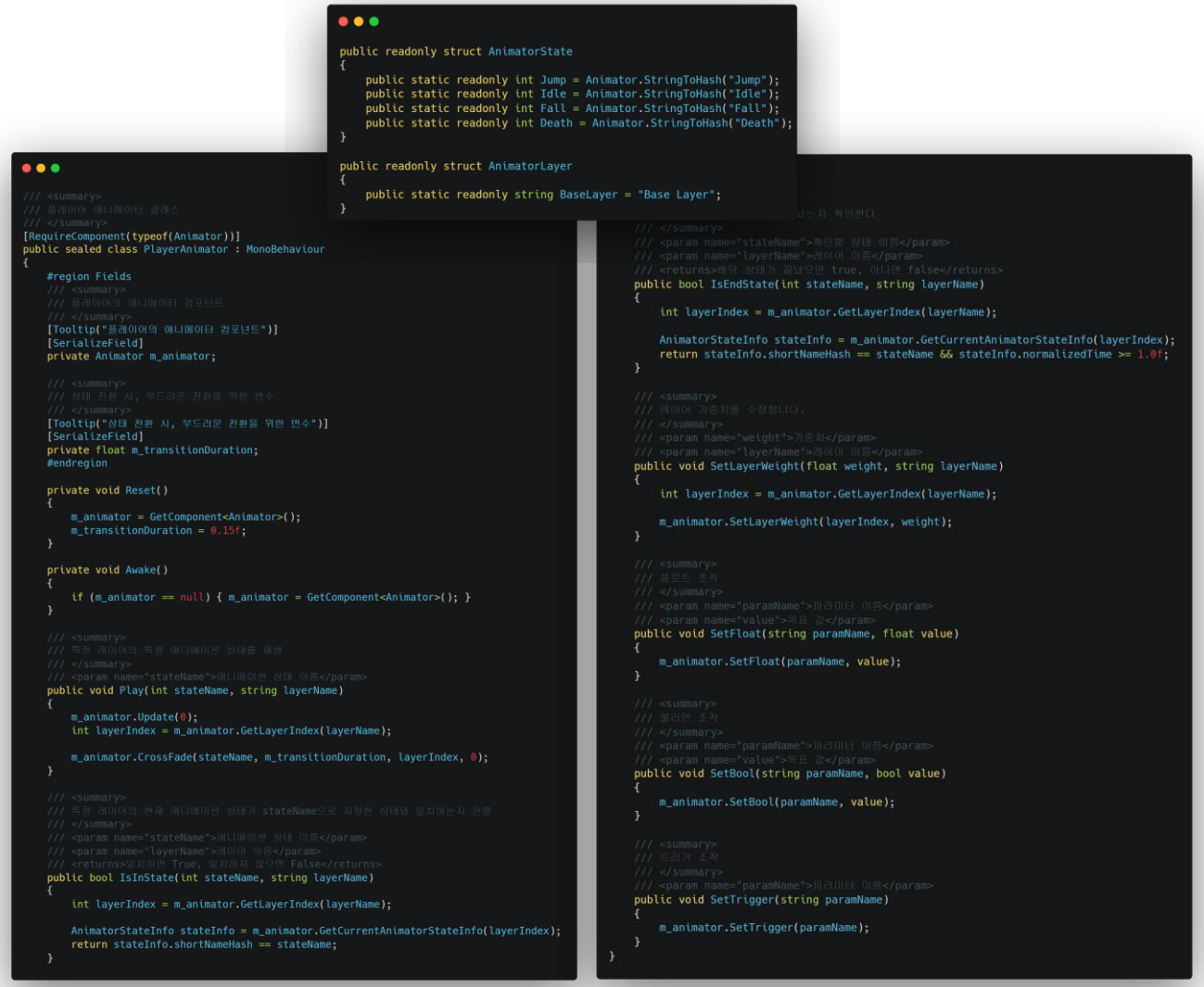
- UniRX의 Observable.EveryFixedUpdate() 이벤트를 구독하여 매 FixedUpdate 마다 현재 플레이어의 위치를 체크합니다. 만약 플레이어의 높이가 특정 값보다 낮아지면 OnDie 메서드를 호출합니다.

- OnDie 메서드에서는 플레이어의 움직임을 중단하고 플레이어의 사망 애니메이션을 재생합니다. 또한 구독 된 모든 UniRX 이벤트를 해제합니다.

- 플레이어 오브젝트가 OnDestroy 될 때, m_Disposables에 구독된 모든 이벤트를 해제하여 메모리 누수를 예방합니다

플레이어 코드

2. 플레이어 애니메이터 클래스



```
public readonly struct AnimatorState
{
    public static readonly int Jump = Animator.StringToHash("Jump");
    public static readonly int Idle = Animator.StringToHash("Idle");
    public static readonly int Fall = Animator.StringToHash("Fall");
    public static readonly int Death = Animator.StringToHash("Death");
}

public readonly struct AnimatorLayer
{
    public static readonly string BaseLayer = "Base Layer";
}
```

```
/// <summary>
/// 플레이어 애니메이터 클래스
/// </summary>
[RequireComponent(typeof(Animator))]
public sealed class PlayerAnimator : MonoBehaviour
{
    #region Fields
    /// <summary>
    /// 플레이어의 애니메이터 컴포넌트
    /// </summary>
    [Tooltip("플레이어의 애니메이터 컴포넌트")]
    [SerializeField]
    private Animator m_animator;

    /// <summary>
    /// 상태 전환 시, 부드러운 전환을 위한 변수
    /// </summary>
    [Tooltip("상태 전환 시, 부드러운 전환을 위한 변수")]
    [SerializeField]
    private float m_transitionDuration;
    #endregion

    private void Reset()
    {
        m_animator = GetComponent<Animator>();
        m_transitionDuration = 0.15f;
    }

    private void Awake()
    {
        if (m_animator == null) { m_animator = GetComponent<Animator>(); }
    }

    /// <summary>
    /// 특정 레이어의 특정 애니메이션 상태를 재생
    /// </summary>
    /// <param name="stateName">애니메이션 상태 이름</param>
    public void Play(int stateName, string layerName)
    {
        m_animator.Update(0);
        int layerIndex = m_animator.GetLayerIndex(layerName);

        m_animator.CrossFade(stateName, m_transitionDuration, layerIndex, 0);
    }

    /// <summary>
    /// 특정 레이어의 현재 애니메이션 상태가 stateName으로 지정한 상태와 일치하는지 판별
    /// </summary>
    /// <param name="stateName">애니메이션 상태 이름</param>
    /// <param name="layerName">레이어 이름</param>
    /// <returns>일치하면 True, 일치하지 않으면 False</returns>
    public bool IsInState(int stateName, string layerName)
    {
        int layerIndex = m_animator.GetLayerIndex(layerName);

        AnimatorStateInfo stateInfo = m_animator.GetCurrentAnimatorStateInfo(layerIndex);
        return stateInfo.shortNameHash == stateName;
    }
```

```
    /// </summary>
    /// <param name="stateName">확인할 상태 이름</param>
    /// <param name="layerName">레이어 이름</param>
    /// <returns>해당 상태가 끝났으면 true, 아니면 false</returns>
    public bool IsEndState(int stateName, string layerName)
    {
        int layerIndex = m_animator.GetLayerIndex(layerName);

        AnimatorStateInfo stateInfo = m_animator.GetCurrentAnimatorStateInfo(layerIndex);
        return stateInfo.shortNameHash == stateName && stateInfo.normalizedTime >= 1.0f;
    }

    /// <summary>
    /// 레이어 가중치를 수정합니다.
    /// </summary>
    /// <param name="weight">가중치</param>
    /// <param name="layerName">레이어 이름</param>
    public void SetLayerWeight(float weight, string layerName)
    {
        int layerIndex = m_animator.GetLayerIndex(layerName);

        m_animator.SetLayerWeight(layerIndex, weight);
    }

    /// <summary>
    /// 플로트 조작
    /// </summary>
    /// <param name="paramName">파라미터 이름</param>
    /// <param name="value">목표 값</param>
    public void SetFloat(string paramName, float value)
    {
        m_animator.SetFloat(paramName, value);
    }

    /// <summary>
    /// 불리언 조작
    /// </summary>
    /// <param name="paramName">파라미터 이름</param>
    /// <param name="value">목표 값</param>
    public void SetBool(string paramName, bool value)
    {
        m_animator.SetBool(paramName, value);
    }

    /// <summary>
    /// 트리거 조작
    /// </summary>
    /// <param name="paramName">파라미터 이름</param>
    public void SetTrigger(string paramName)
    {
        m_animator.SetTrigger(paramName);
    }
}
```

개발 의도

- 직접 Animator를 수정하지 않고 로직을 깔끔하게 하는 효과를 기대하였습니다.
- AnimatorState, AnimatorLayer 구조체를 통해 매직스트링 사용을 최소화하였습니다.

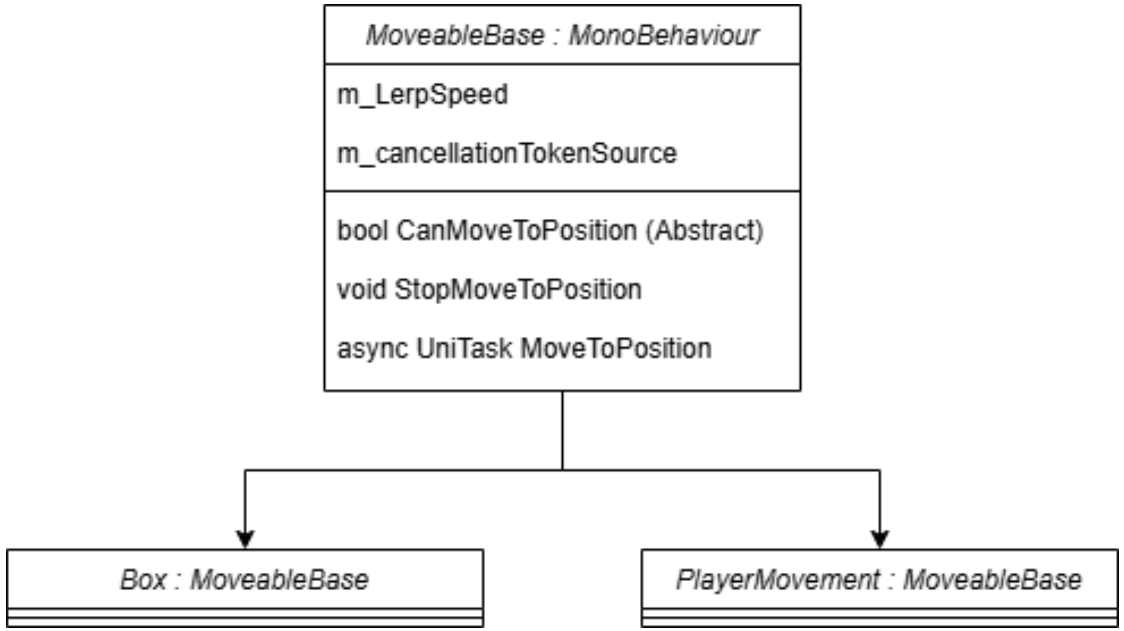
작동 방식

- PlayerAnimator.Play(AnimatorState.Jump)와 같은 형태로 작동할 수 있습니다.
- 매직스트링 대신 Animator.StringToHash 메서드를 사용해 성능적으로 개선하려고 하였습니다.
- IsEndState, IsInState 메서드에서는 현재 애니메이터 상태를 가져와 애니메이션이 얼마나 진행되었는지에 따라서 bool 값을 반환하도록 하였습니다.

플레이어 코드

3-1. 플레이어 무브먼트 클래스의 구조

요약 박스 오브젝트와 플레이어 오브젝트에 공통된 속성이 있기 때문에 MoveableBase 클래스에서 상속하는 식으로 구현

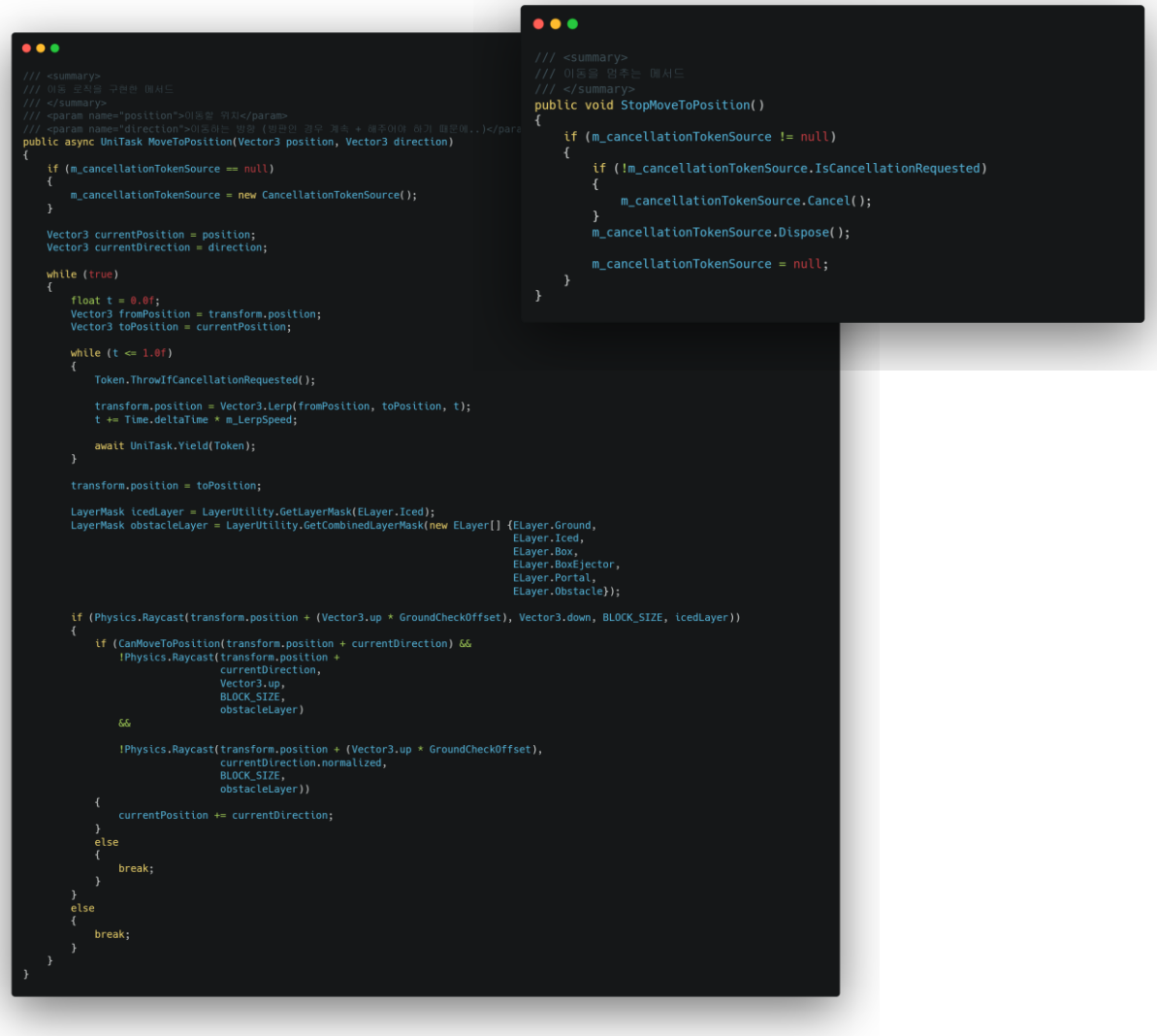


개발 의도

- 박스 오브젝트와 플레이어 오브젝트는 움직일 수 있는 객체라는 공통점을 가지기 때문에, MoveableBase 클래스를 상속하게 하여 확장성 있는 구조로 구현하였습니다.
- 추상 클래스, 추상 메서드를 사용하여 실수 방지 및 일관성 있는 설계를 유도하였습니다.
- CanMoveToPosition 추상 메서드를 박스 오브젝트 및 플레이어 오브젝트에 맞게 구현합니다.

플레이어 코드

3-2. 무브먼트 베이스 클래스



```
/// <summary>
/// 이동 로직을 구현한 메서드
/// </summary>
/// <param name="position">이동할 위치</param>
/// <param name="direction">이동하는 방향 (빙판인 경우 계속 + 해주어야 하기 때문에..)</para
public async UniTask MoveToPosition(Vector3 position, Vector3 direction)
{
    if (m_cancellationTokenSource == null)
    {
        m_cancellationTokenSource = new CancellationTokenSource();
    }

    Vector3 currentPosition = position;
    Vector3 currentDirection = direction;

    while (true)
    {
        float t = 0.0f;
        Vector3 fromPosition = transform.position;
        Vector3 toPosition = currentPosition;

        while (t <= 1.0f)
        {
            Token.ThrowIfCancellationRequested();

            transform.position = Vector3.Lerp(fromPosition, toPosition, t);
            t += Time.deltaTime * m_LerpSpeed;

            await UniTask.Yield(Token);
        }

        transform.position = toPosition;

        LayerMask icedLayer = LayerUtility.GetLayerMask(ELayer.Iced);
        LayerMask obstacleLayer = LayerUtility.GetCombinedLayerMask(new ELayer[] {ELayer.Ground,
                                                                                  ELayer.Iced,
                                                                                  ELayer.Box,
                                                                                  ELayer.BoxEjector,
                                                                                  ELayer.Portal,
                                                                                  ELayer.Obstacle});

        if (Physics.Raycast(transform.position + (Vector3.up * GroundCheckOffset), Vector3.down, BLOCK_SIZE, icedLayer))
        {
            if (CanMoveToPosition(transform.position + currentDirection) &&
                !Physics.Raycast(transform.position +
                                 currentDirection,
                                 Vector3.up,
                                 BLOCK_SIZE,
                                 obstacleLayer)
                &&
                !Physics.Raycast(transform.position + (Vector3.up * GroundCheckOffset),
                                 currentDirection.normalized,
                                 BLOCK_SIZE,
                                 obstacleLayer))
            {
                currentPosition += currentDirection;
            }
            else
            {
                break;
            }
        }
        else
        {
            break;
        }
    }
}
```

```
/// <summary>
/// 이동을 멈추는 메서드
/// </summary>
public void StopMoveToPosition()
{
    if (m_cancellationTokenSource != null)
    {
        if (!m_cancellationTokenSource.IsCancellationRequested)
        {
            m_cancellationTokenSource.Cancel();
        }
        m_cancellationTokenSource.Dispose();

        m_cancellationTokenSource = null;
    }
}
```

작동 방식

- StopMoveToPoisition 메서드를 호출하면, 현재 실행중인 비동기 이동 MoveToPosition 메서드를 중단하고 관련된 CancellationTokenSource를 정리하는 역할을 합니다. (새로운 이동 시, Token이 다시 생성됩니다.)

- MoveToPosition 비동기 메서드는, 이동 로직을 구현한 메서드입니다. 과정은 아래와 같습니다.

  - Lerp를 사용해 부드럽게 목표 위치로 이동합니다.

  - 목표 위치에 도달한 뒤 현재 바닥이 빙판인지 확인합니다. (Raycast를 사용해 확인합니다.)

  - 빙판이라면 추가 이동, 아니라면 이동을 종료합니다.

플레이어 코드

3-3-1. 플레이어 무브먼트 클래스

```
        public override bool CanMoveToPosition(Vector3 position)
        {
#if UNITY_EDITOR
            Debug.DrawLine(transform.position + m_RAY_OFFSET, position + m_RAY_OFFSET, Color.yellow, 1.0f);
            Debug.DrawLine(position, position + (Vector3.up * BLOCK_SIZE), Color.blue, 1.0f);
#endif

            bool inupper = !Physics.Raycast(position,
                                            Vector3.up,
                                            BLOCK_SIZE,
                                            LayerUtility.GetCombinedLayerMask(new ELayer[] { ELayer.Ground, ELayer.Iced, ELayer.Box,
                                                                                             ELayer.BoxEjector, ELayer.Portal,
                                                                                             ELayer.Obstacle }));

            bool infront = !Physics.Linecast(transform.position + m_RAY_OFFSET,
                                             position + m_RAY_OFFSET,
                                             LayerUtility.GetCombinedLayerMask(new ELayer[] { ELayer.Box, ELayer.BoxEjector,
                                                                                              ELayer.Portal, ELayer.Obstacle,
                                                                                              ELayer.Iced }));

            return inupper && infront;
        }
```

```
public void Move(Vector3 direction)
{
    if (IsMoving)
    {
        return;
    }

    Vector3 moveDirection = m_CameraSwapManager.GetCameraRelativeDirection(direction) * BLOCK_SIZE;
    Vector3 targetPosition = transform.position + moveDirection;

    RotateTowards(targetPosition);

    if (TryHandleGimmickInteraction(targetPosition, moveDirection))
    {
        ++MoveCounter.Value;

        GameObject dust = Instantiate(m_MoveParticle, transform.position, Quaternion.identity);
        Destroy(dust, dust.GetComponent<ParticleSystem>().main.duration);

        m_Animator.Play(AnimatorState.Jump, AnimatorLayer.BaseLayer);
    }
    else if (CanMoveToPosition(targetPosition))
    {
        ++MoveCounter.Value;

        GameObject dust = Instantiate(m_MoveParticle, transform.position, Quaternion.identity);
        Destroy(dust, dust.GetComponent<ParticleSystem>().main.duration);

        m_Animator.Play(AnimatorState.Jump, AnimatorLayer.BaseLayer);
        MoveToPosition(targetPosition, moveDirection).Forget();
    }
}
```

작동 방식

- ReactiveProperty 형태로 움직인 횟수, 박스를 민 횟수를 저장합니다. (구독 시 편하게 사용할 수 있습니다.)
- CanMoveToPosition 메서드는 이동할 위치에 레이캐스트를 활용하여 상호작용 할 수 없는 오브젝트, 이동하려는 위치에 타일이 배치되었는지 등을 확인해 bool 값을 반환합니다.
- Move 메서드에서는 우선, 이동할 위치에 상호작용을 시도한 뒤, 해당 위치에 상호작용을 시도할 수 있는 오브젝트가 없다면 일반 이동 로직을 수행합니다.
- 이동 시, 파티클을 재생합니다.

플레이어 코드

3-3-2. 플레이어 무브먼트 클래스

```
private new async UniTask MoveToPosition(Vector3 position, Vector3 direction)
{
    IsMoving = true;
    m_Animator.SetBool("IsMoving", IsMoving);

    OnBeforeMove.Invoke();

    await base.MoveToPosition(position, direction);

    OnAfterMove.Invoke();

    // Player Pivot이 왜인덕에 붙어있기 때문에 임의로 0.1f 만큼 올려서 측정
    await UniTask.WaitUntil(() => Physics.Raycast(transform.position + Vector3.up * GroundCheckOffset,
                                Vector3.down, GroundCheckDistance,
                                LayerUtility.GetCombinedLayerMask(new ELayer[] { ELayer.Ground, ELayer.Iced, ELayer.Box,
ELayer.BoxEjector })),
                                cancellationToken: Token);

    // 특정 시간을 기다립니다.
    await UniTask.Delay(m_MOVE_DELAY, cancellationToken: Token);

    IsMoving = false;
    m_Animator.SetBool("IsMoving", IsMoving);
}
```

```
private bool TryHandleGimmickInteraction(Vector3 nextPosition, Vector3 moveDirection)
{
    Vector3 from = transform.position + m_RAY_OFFSET;
    Vector3 to = nextPosition + m_RAY_OFFSET;

#if UNITY_EDITOR
    Debug.DrawLine(from, to, Color.cyan, 1.0f);
#endif

    if (Physics.Linecast(from, to, out RaycastHit hit))
    {
        int layer = hit.collider.gameObject.layer;

        if (LayerUtility.IsEqualLayerMask(ELayer.Box, layer))
        {
            Box box = hit.collider.GetComponent<Box>();

            if (box.CanMoveToPosition(box.transform.position + moveDirection))
            {
                box.MoveToPosition(box.transform.position + moveDirection, moveDirection).Forget();
                ++PushCounter.Value;

                MoveToPosition(nextPosition, moveDirection).Forget();

                return true;
            }
        }
        else if (LayerUtility.IsEqualLayerMask(ELayer.Ladder, layer))
        {
            nextPosition.y += BLOCK_SIZE;
            MoveToPosition(nextPosition, moveDirection).Forget();

            return true;
        }
    }

    return false;
}
```

작동 방식

- MoveToPosition 메서드를 호출 시, 부모 클래스의 MoveToPosition 비동기 메서드가 완료될 때 까지 기다립니다. 그 다음 플레이어가 바닥에 닿기까지를 기다립니다. 그 후 특정 시간을 기다립니다. (너무 빠르게 움직여 컨트롤이 잘 안되어, 조작감을 위해 딜레이를 추가하였습니다.)

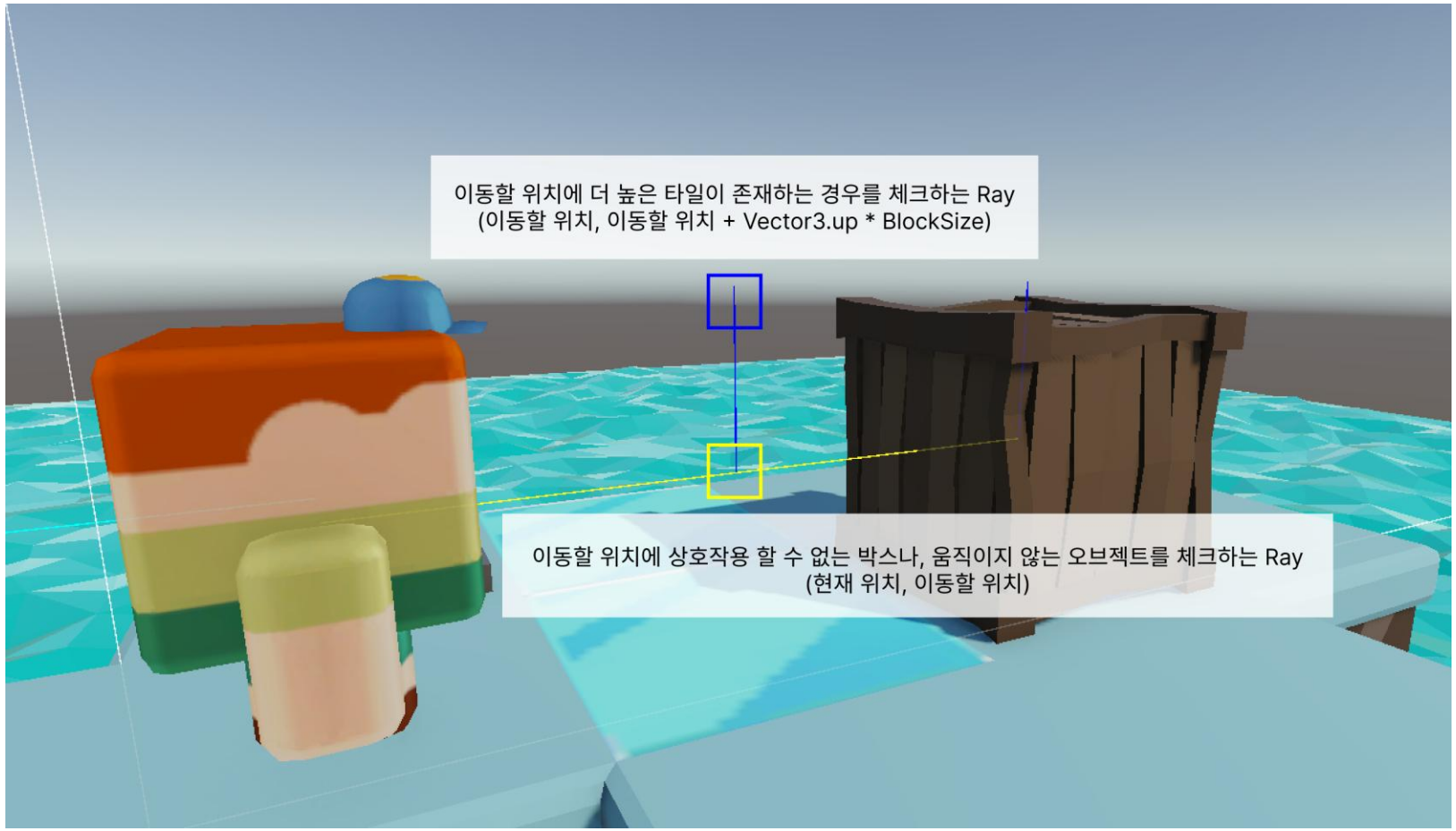
- TryHandleGimmickInteraction 메서드를 호출 시, 플레이어 정면에 있는 상호작용할 수 있는 오브젝트를 레이캐스트로 확인하여 가능하다면 상호작용 행위를 수행합니다. 만약 수행할 행위가 없다면 false를 반환합니다.

플레이어 코드

3-3-3. 플레이어 무브먼트 클래스의 실제 작동 사진

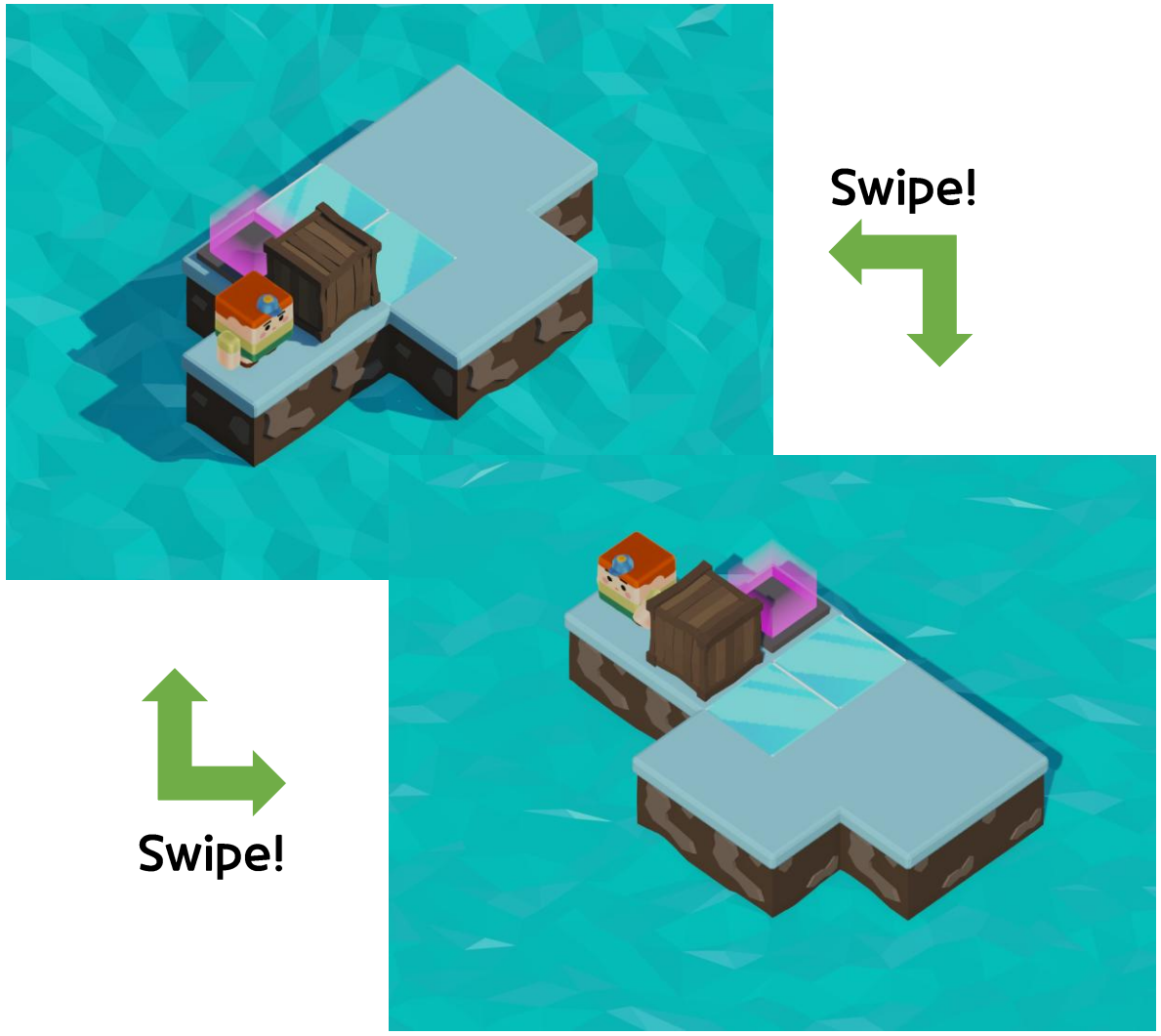
개요

실제 **유니티 씬에서 작동**하는 모습을 **캡쳐하여 설명과 함께 첨부**하였습니다.



카메라 회전 코드

1. 카메라 회전 코드의 실제 작동 사진

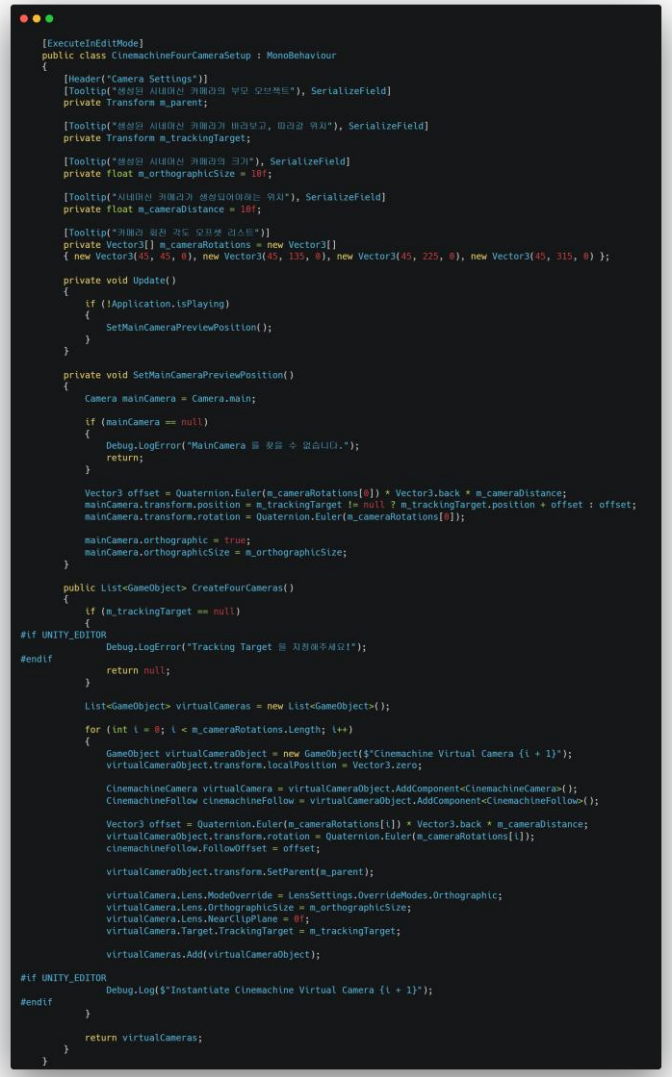


개발 의도

- 카메라를 늘 직접 배치할 필요 없이, 지정된 위치와 각도로 4개의 카메라를 자동으로 생성되면 효율적이라고 판단하여 제작하였습니다.
- 레벨 디자인 시, 에디터에서 미리 볼 수 있도록 하였습니다.
- 조작성을 위해 스와이프 조작을 통해 Camera 위치를 변경할 수 있도록 하였습니다.

카메라 회전 코드

2. 프리셋에 따라 시네머신 카메라 4개를 생성하는 클래스



```
[ExecuteInEditMode]
public class CinemachineFourCameraSetup : MonoBehaviour
{
    [Header("Camera Settings")]
    [Tooltip("생성된 시네머신 카메라의 부모 오브젝트"), SerializeField]
    private Transform m_parent;

    [Tooltip("생성된 시네머신 카메라가 바라보고, 따라갈 위치"), SerializeField]
    private Transform m_trackingTarget;

    [Tooltip("생성된 시네머신 카메라의 크기"), SerializeField]
    private float m_orthographicSize = 10f;

    [Tooltip("시네머신 카메라가 생성되어야하는 위치"), SerializeField]
    private float m_cameraDistance = 10f;

    [Tooltip("카메라 회전 각도 오프셋 리스트")]
    private Vector3[] m_cameraRotations = new Vector3[]
    { new Vector3(45, 45, 0), new Vector3(45, 135, 0), new Vector3(45, 225, 0), new Vector3(45, 315, 0) };

    private void Update()
    {
        if (!Application.isPlaying)
        {
            SetMainCameraPreviewPosition();
        }
    }

    private void SetMainCameraPreviewPosition()
    {
        Camera mainCamera = Camera.main;

        if (mainCamera == null)
        {
            Debug.LogError("MainCamera 를 찾을 수 없습니다.");
            return;
        }

        Vector3 offset = Quaternion.Euler(m_cameraRotations[0]) * Vector3.back * m_cameraDistance;
        mainCamera.transform.position = m_trackingTarget != null ? m_trackingTarget.position + offset : offset;
        mainCamera.transform.rotation = Quaternion.Euler(m_cameraRotations[0]);

        mainCamera.orthographic = true;
        mainCamera.orthographicSize = m_orthographicSize;
    }

    public List<GameObject> CreateFourCameras()
    {
        if (m_trackingTarget == null)
        {
#if UNITY_EDITOR
            Debug.LogError("Tracking Target 을 지정해주세요!");
#endif
            return null;
        }

        List<GameObject> virtualCameras = new List<GameObject>();

        for (int i = 0; i < m_cameraRotations.Length; i++)
        {
            GameObject virtualCameraObject = new GameObject($"Cinemachine Virtual Camera {i + 1}");
            virtualCameraObject.transform.localPosition = Vector3.zero;

            CinemachineCamera virtualCamera = virtualCameraObject.AddComponent<CinemachineCamera>();
            CinemachineFollow cinemachineFollow = virtualCameraObject.AddComponent<CinemachineFollow>();

            Vector3 offset = Quaternion.Euler(m_cameraRotations[i]) * Vector3.back * m_cameraDistance;
            virtualCameraObject.transform.rotation = Quaternion.Euler(m_cameraRotations[i]);
            cinemachineFollow.FollowOffset = offset;

            virtualCameraObject.transform.SetParent(m_parent);

            virtualCamera.Lens.ModeOverride = LensSettings.OverrideModes.Orthographic;
            virtualCamera.Lens.OrthographicSize = m_orthographicSize;
            virtualCamera.Lens.NearClipPlane = 0f;
            virtualCamera.Target.TrackingTarget = m_trackingTarget;

            virtualCameras.Add(virtualCameraObject);

#if UNITY_EDITOR
            Debug.Log($"Instantiate Cinemachine Virtual Camera {i + 1}");
#endif
        }

        return virtualCameras;
    }
}
```

유의 사항

- Cinemachine 3.1.3 버전을 사용하기에, 기존 버전과는 코드가 다를 수 있습니다.

작동 방식

- [ExecuteInEditMode] 를 사용하여 에디트 모드에서도 스크립트가 작동하도록 하였습니다. 이를 통해 업데이트 문에서 에디트 모드일 때, 카메라 미리보기를 지원할 수 있습니다.

- CreateFourCameras 메서드를 호출하면 4개의 Cinemachine Camera 컴포넌트를 가지고 있는 게임 오브젝트를 생성하며, 그 리스트를 반환합니다.

카메라 회전 코드

3. 카메라 회전 및 현재 시점을 기준으로 방향을 재설정하는 카메라 매니저 클래스

```
public class CameraManager : MonoBehaviour
{
    [Tooltip("시네머신 카메라 리스트")]
    [SerializeField]
    private List<GameObject> m_CinemachineCameraList = new List<GameObject>();

    [Tooltip("메인 카메라에서 사용하고 있는 시네머신 브레인")]
    private CinemachineBrain m_CinemachineBrain;

    [Tooltip("사용하고 있는 카메라 인덱스")]
    private int m_CurrentCameraIndex = 3;

    private void Awake()
    {
        m_CinemachineBrain = Camera.main.GetComponent<CinemachineBrain>();
        m_CinemachineCameraList = GetComponent<CinemachineFourCameraSetup>().CreateFourCameras();
    }

    private void Start()
    {
        for (int i = 0; i < m_CinemachineCameraList.Count; i++)
        {
            m_CinemachineCameraList[i].SetActive(i == m_CurrentCameraIndex);
        }
    }

    /// <summary>
    /// 현재 카메라를 기준으로 벡터 값을 재설정하는 함수
    /// </summary>
    /// <param name="vec">측정 벡터 값</param>
    /// <returns>재설정한 벡터 값</returns>
    public Vector3 GetCameraRelativeDirection(Vector3 vec)
    {
        Quaternion cameraRotation = Quaternion.Euler(0,
m_CinemachineCameraList[m_CurrentCameraIndex].transform.eulerAngles.y, 0);
        Vector3 adjustedDirection = cameraRotation * vec;

        if (Mathf.Approximately(m_CinemachineCameraList[m_CurrentCameraIndex].transform.eulerAngles.y, 45.0f))
        {
            adjustedDirection = cameraRotation * Quaternion.Euler(0, -90, 0) * vec;
        }

        if (Mathf.Abs(adjustedDirection.x) >= Mathf.Abs(adjustedDirection.z))
        {
            adjustedDirection.z = 0;
        }
        else
        {
            adjustedDirection.x = 0;
        }

        return adjustedDirection.normalized;
    }

    /// <summary>
    /// 다음 카메라로 전환
    /// </summary>
    public void swapToNextCamera()
    {
        if (m_CinemachineBrain.IsBlending)
        {
            return;
        }

        m_CinemachineCameraList[m_CurrentCameraIndex].SetActive(false);
        m_CurrentCameraIndex = (m_CurrentCameraIndex + 1) % m_CinemachineCameraList.Count;
        m_CinemachineCameraList[m_CurrentCameraIndex].SetActive(true);
    }

    /// <summary>
    /// 이전 카메라로 전환
    /// </summary>
    public void swapToPreviousCamera()
    {
        if (m_CinemachineBrain.IsBlending)
        {
            return;
        }

        m_CinemachineCameraList[m_CurrentCameraIndex].SetActive(false);
        m_CurrentCameraIndex = (m_CurrentCameraIndex - 1 + m_CinemachineCameraList.Count) %
m_CinemachineCameraList.Count;
        m_CinemachineCameraList[m_CurrentCameraIndex].SetActive(true);
    }
}
```

유의 사항

- Cinemachine 3.1.3 버전을 사용하기에, 기존 버전과는 코드가 다를 수 있습니다.

작동 방식

- swapTo~Camera 메서드를 호출하면 이전 또는 다음 카메라로 전환됩니다.
- 카메라가 전환될 시 조작 키 또한 회전하게 되는 문제가 생겨 GetCameraRelativeDirection 메서드를 구현하였습니다. 이 메서드에 Vector3 인자를 넣으면, 카메라 프리셋에 맞게 좌표를 변환한 뒤 반환합니다.

카메라 회전 코드

4. 스와이프 조작을 감지하는 ScrollRect 클래스

```
public class CameraScrollRect : MonoBehaviour, IBeginDragHandler, IDragHandler, IEndDragHandler
{
    #region Properties
    private ScrollRect m_scrollRect;

    public UnityEvent onLeftScrollThresholdReached;
    public UnityEvent onRightScrollThresholdReached;

    private Vector2 m_startDragPosition;

    private bool m_eventTriggered = false;
    #endregion

    private void Awake()
    {
        m_scrollRect = GetComponent<ScrollRect>();

        CameraManager cameraManager = FindFirstObjectByType<CameraManager>();

        if (cameraManager != null)
        {
            onLeftScrollThresholdReached.AddListener(() => cameraManager.swapToNextCamera());
            onRightScrollThresholdReached.AddListener(() => cameraManager.swapToPreviousCamera());
        }
    }

    public void OnBeginDrag(PointerEventData eventData)
    {
        m_startDragPosition = eventData.position;
        m_eventTriggered = false;
    }

    public void OnDrag(PointerEventData eventData)
    {
        if (m_eventTriggered) return;

        float dragDistance = Mathf.Abs(eventData.position.x - m_startDragPosition.x);
        float screenThreshold = Screen.width * 0.25f;

        if (dragDistance >= screenThreshold)
        {
            m_eventTriggered = true;

            if (eventData.position.x > m_startDragPosition.x)
            {
                onRightScrollThresholdReached?.Invoke();
            }
            else
            {
                onLeftScrollThresholdReached?.Invoke();
            }
        }
    }

    public void OnEndDrag(PointerEventData eventData)
    {
        m_scrollRect.content.localPosition = Vector3.zero;
        m_scrollRect.viewport.localPosition = Vector3.zero;
        m_eventTriggered = false;
    }
}
```

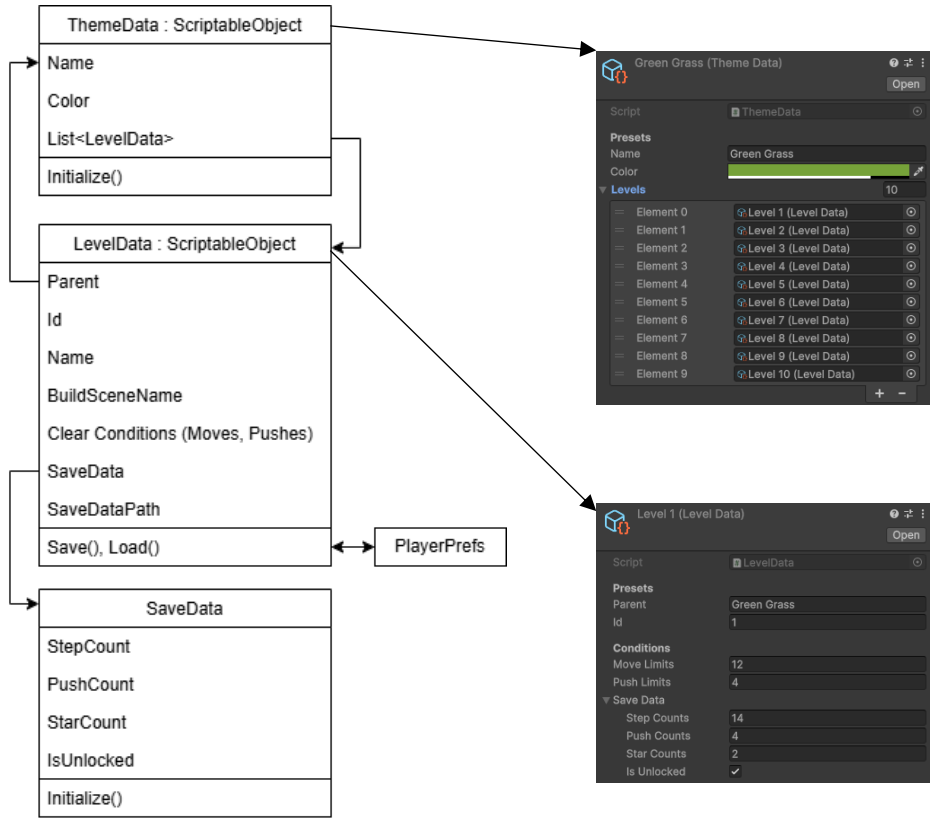
작동 방식

- 카메라 매니저 클래스를 찾아서 카메라 전환 이벤트를 바인딩합니다.
- IBeginDragHandler, IDragHandler, IEndDragHandler 인터페이스를 구현하도록 하여 드래그를 추적할 수 있게 하였습니다.
  - OnBeginDrag 메서드에서 초기 위치를 저장합니다.
  - OnDrag 메서드에서 처음 위치에서 얼마나 이동했는지 계산합니다. (화면의 25% 이상 이동 시 위치 계산 후 Camera 전환을 수행)
  - OnEndDrag 메서드에서 ScrollRect의 위치를 (0, 0)으로 초기화 하여 다음 드래그를 할 수 있도록 처리했습니다.

저장 및 레벨 데이터 코드

1. 레벨 데이터의 저장 구조

요약 **레벨 저장 구조**를 이미지로 쉽게 정리하여 표현하였습니다.



개발 의도

- 테마 데이터와 레벨 데이터 모두 ScriptableObject로 관리하여 레벨 디자인 시, 조금 더 편하게 작업할 수 있도록 하였습니다.
- 테마 단위로 레벨을 그룹화하여 전체적인 구조를 구상하였습니다.
- 레벨 데이터에서는 자동으로 레벨 이름과 저장 경로를 조합하는 방식으로 제작하여 수동으로 작업하지 않아도 되도록 하였습니다.
- PlayerPrefs를 이용해 JSON 형태로 세이브 데이터를 저장/로드 하도록 구현하였습니다.

저장 및 레벨 데이터 코드

2. 테마 데이터 클래스

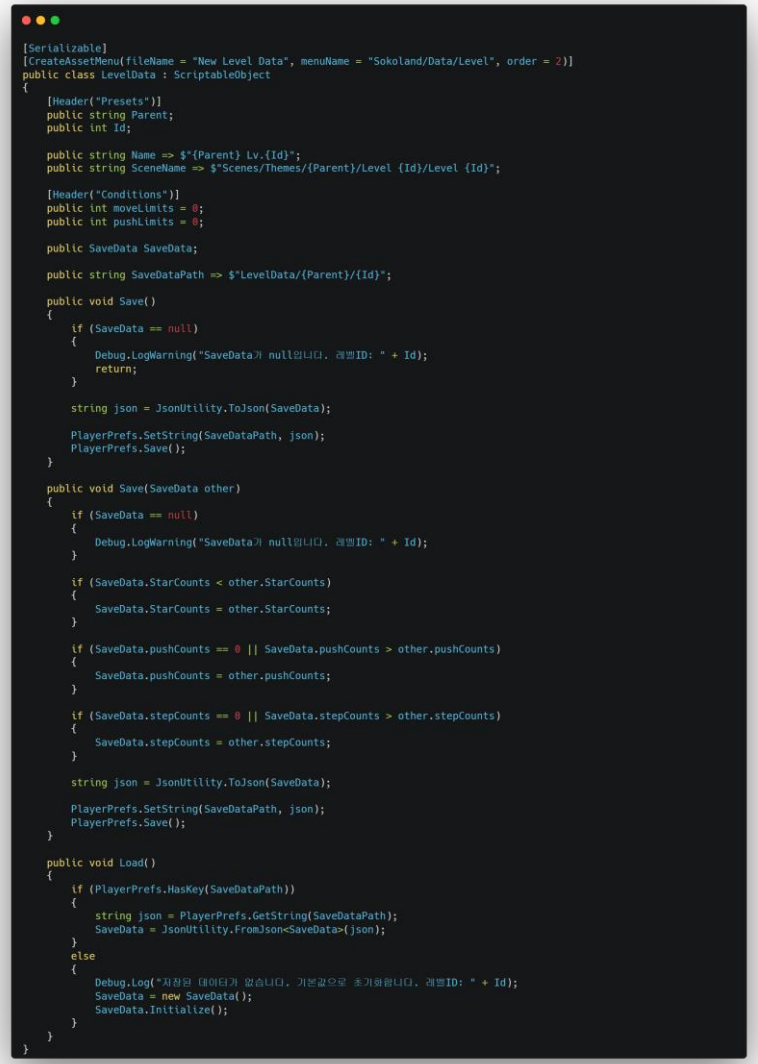
```
/// <summary>
/// 테마 데이터 클래스
/// </summary>
[Serializable]
[CreateAssetMenu(fileName = "New Theme Data", menuName = "Sokoland/Data/Theme", order = 1)]
public class ThemeData : ScriptableObject
{
    [Header("Presets")]
    /// <summary>
    /// 해당 테마의 이름.
    /// </summary>
    public string Name;

    /// <summary>
    /// 해당 테마의 컬러.
    /// </summary>
    public Color Color;

    /// <summary>
    /// 해당 테마의 레벨 리스트.
    /// </summary>
    public List<LevelData> Levels;

    /// <summary>
    /// 테마의 초기화 함수
    /// (레벨의 부모 테마 이름을 설정합니다)
    /// </summary>
    public void Initialize()
    {
        if (Levels != null)
        {
            for (int i = 0; i < Levels.Count; i++)
            {
                if (Levels[i] != null)
                {
                    Levels[i].Parent = this.Name;
                    Levels[i].Load();

                    if (i == 0)
                    {
                        Levels[i].SaveData.IsUnlocked = true;
                    }

                    Levels[i].Save();
                }
            }
        }
    }
}
```

작동 방식

- 테마의 이름, 테마의 대표 색상, 테마 하위 레벨들을 리스트로 가지고 있습니다.
- Initialize 메서드를 호출하면, 테마 하위 레벨 리스트를 순회하며 레벨의 데이터를 초기화하게 됩니다.
- 게임을 실행할 때 게임 매니저에서 Initialize를 호출합니다.

저장 및 레벨 데이터 코드

3. 레벨 데이터 클래스



```
[Serializable]
[CreateAssetMenu(fileName = "New Level Data", menuName = "Sokoland/Data/Level", order = 2)]
public class LevelData : ScriptableObject
{
    [Header("Presets")]
    public string Parent;
    public int Id;

    public string Name => $"{Parent} Lv.{Id}";
    public string SceneName => $"Scenes/Themes/{Parent}/Level {Id}/Level {Id}";

    [Header("Conditions")]
    public int moveLimits = 0;
    public int pushLimits = 0;

    public SaveData SaveData;

    public string SaveDataPath => $"LevelData/{Parent}/{Id}";

    public void Save()
    {
        if (SaveData == null)
        {
            Debug.LogWarning("SaveData가 null입니다. 레벨ID: " + Id);
            return;
        }

        string json = JsonUtility.ToJson(SaveData);

        PlayerPrefs.SetString(SaveDataPath, json);
        PlayerPrefs.Save();
    }

    public void Save(SaveData other)
    {
        if (SaveData == null)
        {
            Debug.LogWarning("SaveData가 null입니다. 레벨ID: " + Id);
        }

        if (SaveData.StarCounts < other.StarCounts)
        {
            SaveData.StarCounts = other.StarCounts;
        }

        if (SaveData.pushCounts == 0 || SaveData.pushCounts > other.pushCounts)
        {
            SaveData.pushCounts = other.pushCounts;
        }

        if (SaveData.stepCounts == 0 || SaveData.stepCounts > other.stepCounts)
        {
            SaveData.stepCounts = other.stepCounts;
        }

        string json = JsonUtility.ToJson(SaveData);

        PlayerPrefs.SetString(SaveDataPath, json);
        PlayerPrefs.Save();
    }

    public void Load()
    {
        if (PlayerPrefs.HasKey(SaveDataPath))
        {
            string json = PlayerPrefs.GetString(SaveDataPath);
            SaveData = JsonUtility.FromJson<SaveData>(json);
        }
        else
        {
            Debug.Log("저장된 데이터가 없습니다. 기본값으로 초기화합니다. 레벨ID: " + Id);
            SaveData = new SaveData();
            SaveData.Initialize();
        }
    }
}
```

작동 방식

- 스테이지의 특정 정보만 넣어주면, Scene 파일 위치, 스테이지 이름, 스테이지 데이터 저장 키 값까지 전부 자동으로 매치되도록 프로퍼티를 사용했습니다.
- PlayerPrefs를 이용하여 JSON 형태로 세이브 데이터를 세이브/로드하도록 구현하였습니다.
- 플레이 조건에 맞도록 새로운 데이터와 기존 데이터를 비교하여 더 좋은 성과만 갱신하도록 했습니다.
- 스테이지 클리어 시 Save(SaveData) 메서드가 호출됩니다.

저장 및 레벨 데이터 코드

4. 세이브 데이터 클래스

```csharp
/// <summary>
/// 직렬화 가능한 세이브 데이터 클래스
/// </summary>
[Serializable]
public class SaveData
{
    /// <summary>
    /// 스테이지에서 걸음 수를 기록하는 변수
    /// </summary>
    public int stepCounts;

    /// <summary>
    /// 스테이지에서 박스를 민 횟수를 기록하는 변수
    /// </summary>
    public int pushCounts;

    /// <summary>
    /// 스테이지에서 별을 획득한 횟수를 기록하는 변수
    /// </summary>
    public int StarCounts;

    /// <summary>
    /// 스테이지의 잠금 해제 여부를 기록하는 변수
    /// </summary>
    public bool IsUnlocked;

    /// <summary>
    /// 초기화 함수
    /// </summary>
    public void Initialize()
    {
        stepCounts = 0;
        pushCounts = 0;
        StarCounts = 0;
    }
}
```

작동 방식

- 걸음 수, 박스를 민 횟수, 획득한 별 횟수, 언락 여부를 저장하는 간단한 클래스 입니다.
- [Serializable]을 사용해서 직렬화가 가능하게 하였고, 그로 인해 JSON 형태로 저장 및 로드가 가능합니다.

기믹 오브젝트 코드

1. 박스 클래스

```
    public class Box : MoveableBase
    {
        /// <summary>
        /// 이동 가능 여부를 판단하는 메서드
        /// </summary>
        /// <param name="position">이동할 위치</param>
        /// <returns></returns>
        public override bool CanMoveToPosition(Vector3 position)
        {
#if UNITY_EDITOR
            Debug.DrawLine(transform.position, position, Color.yellow, 1.0f);
            Debug.DrawLine(position, position + Vector3.up, Color.blue, 1.0f);
#endif

            if (Physics.Linecast(transform.position,
                                 position,
                                 out RaycastHit hit,
                                 LayerUtility.GetLayerMask(ELayer.Portal)))
            {
                if (hit.transform.TryGetComponent<Portal>(out Portal portal))
                {
                    return portal.CanMoveToTargetPortal();
                }

                return false;
            }

            bool inupper = !Physics.Raycast(position,
                                            Vector3.up,
                                            BLOCK_SIZE,
                                            LayerUtility.GetCombinedLayerMask(new ELayer[] {
                                                ELayer.Ground,
                                                ELayer.Iced,
                                                ELayer.Obstacle
                                            }));

            bool infront = !Physics.Linecast(transform.position,
                                             position,
                                             LayerUtility.GetCombinedLayerMask(new ELayer[] {
                                                ELayer.Box,
                                                ELayer.BoxEjector,
                                                ELayer.Obstacle,
                                                ELayer.Ground,
                                                ELayer.Iced
                                            }));

            return inupper && infront;
        }
    }
```

작동 방식

- 앞서 살펴보았던, 플레이어 코드 3-2 (무브먼트 클래스)의 자식 클래스로써, CanMoveToPosition을 오버라이딩한다.

기믹 오브젝트 코드

2. 클리어 가능 오브젝트 (박스 포인트 클래스, 별 클래스)

```
public class ClearableBase : MonoBehaviour
{
    public bool IsCleared { get; protected set; } = false;
}
```

```
public class Star : ClearableBase
{
    #region Fields
    [Header("Presets")]
    [Tooltip("베이스 메쉬")]
    [SerializeField]
    private GameObject m_base;

    [Tooltip("파티클 오브젝트")]
    [SerializeField]
    private ParticleSystem m_particle;
    #endregion

    private void OnTriggerEnter(Collider other)
    {
        if (LayerUtility.IsEqualLayerMask(ELayer.Player,
                                          other.gameObject.layer))
        {
            IsCleared = true;

            m_base.SetActive(false);
            m_particle.Play();
        }
    }
}
```

```
public class BoxPoint : ClearableBase
{
    /// <summary>
    /// 이펙트 오브젝트.
    /// </summary>
    [Tooltip("이펙트 오브젝트.")]
    [SerializeField]
    private GameObject m_Effect;

    private void Start()
    {
        this.OnTriggerEnterAsObservable()
            .Where(other => LayerUtility.IsEqualLayerMask(ELayer.Box, other.gameObject.layer))
            .Subscribe(_ =>
            {
                IsCleared = true;
                m_Effect.SetActive(false);
            })
            .AddTo(this);

        this.OnTriggerExitAsObservable()
            .Where(other => LayerUtility.IsEqualLayerMask(ELayer.Box, other.gameObject.layer))
            .Subscribe(_ =>
            {
                IsCleared = false;
                m_Effect.SetActive(true);
            })
            .AddTo(this);
    }
}
```

개발 의도

- 게임 내에서 레벨 클리어 조건을 담당하는 오브젝트들은 일관성을 위해 ClearableBase 클래스를 상속받도록 하였다.

작동 방식

- UniRX에서 지원하는 기능인 OnTriggerEnterAsObservable, OnTriggerExitAsObservable을 사용하는데, 각각 OnTriggerEnter, OnTriggerExit라고 생각하면 됩니다.

- 박스 포인트 클래스에서 박스가 들어오면 IsCleared가 true가 됩니다.

- 플레이어가 별에 닿으면, 별이 사라지고 효과가 나타납니다.

기믹 오브젝트 코드

3-1. 움직일 때마다 업데이트 되어야 하는 클래스

```
/// <summary>
/// 플레이어가 매번 걸을 때, 호출 되어야만 하는 이 게임 특유의 라이프사이클을 구현하는 베이스 클래스
/// </summary>
public abstract class StepBehavior : MonoBehaviour
{
    /// <summary>
    /// 매 걸음을 시작할 때, 호출되는 메서드
    /// </summary>
    public abstract void EveryBeforeStep();

    /// <summary>
    /// 매 걸음이 끝날 때, 실행되는 메서드
    /// </summary>
    public abstract void EveryAfterStep();
}
```

개발 의도

- 플레이어가 매번 움직일 때마다 작동해야하는 클래스가 필요했기에 이 코드를 작성하였습니다.

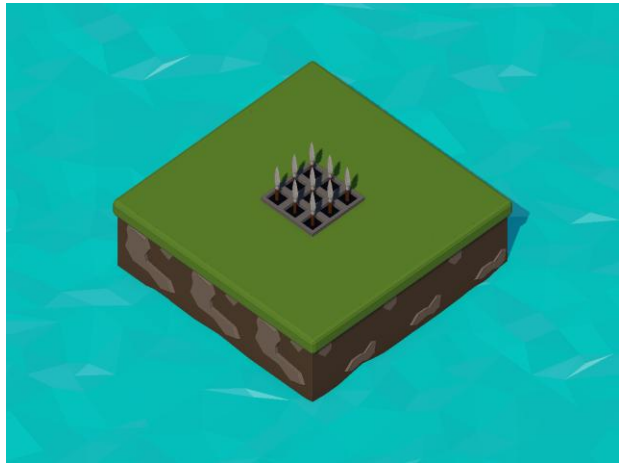
작동 방식

- 플레이어가 움직일 때마다, LevelHandler에 이벤트가 전송되고, LevelHandler에서는 List<StepBehaviour> 자료를 순회하며 각 요소에 대해 EveryBeforeStep, EveryAfterStep을 호출합니다.

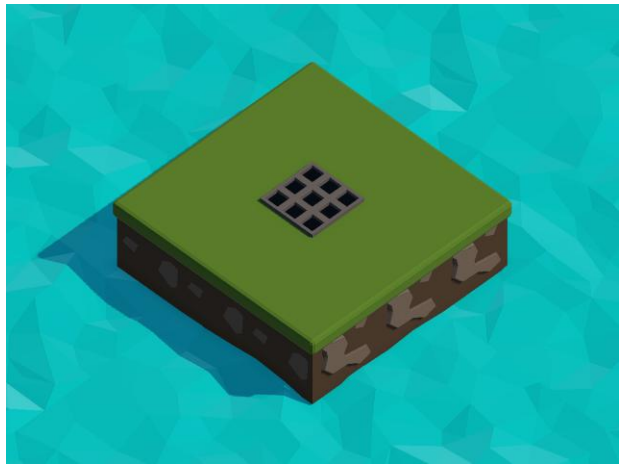
기믹 오브젝트 코드

3-2. 가시 클래스

```
public class Spike : StepBehavior
{
    [Serializable]
    public enum SpikeState
    {
        Open,
        Close
    }

    [Header("Presets")]
    [Tooltip("스파이크의 상태 변수")]
    [SerializeField]
    private SpikeState m_spikeType = SpikeState.Close;

    [Tooltip("스파이크가 몇 걸음마다 상태가 바뀌게 되는지 (0만 경우 메서드를 통해 수동으로 조정)")]
    [SerializeField]
    private int m_repeatChangeStateStepCount = 1;
    private int m_currentChangeStateStepCount = 0;

    private Animator m_animator;
    private Collider m_collider;

    private void Start()
    {
        m_animator = GetComponent<Animator>();
        m_collider = GetComponent<Collider>();

        if (m_spikeType == SpikeState.Open)
            Open();

        else if (m_spikeType == SpikeState.Close)
            Close();
    }

    private void OnTriggerEnter(Collider other)
    {
        if (LayerUtility.IsEqualLayerMask(ELayer.Player, other.gameObject.layer))
            other.GetComponent<PlayerController>().OnDie();
    }

    public void Open()
    {
        m_spikeType = SpikeState.Open; m_currentChangeStateStepCount = 0;
        m_collider.enabled = true; m_animator.Play("Open");
    }

    public void Close()
    {
        m_spikeType = SpikeState.Close; m_currentChangeStateStepCount = 0;
        m_collider.enabled = false; m_animator.Play("Close");
    }

    public override void EveryBeforeStep()
    {
        if (m_repeatChangeStateStepCount == 0)
            return;

        m_currentChangeStateStepCount++;

        if (m_currentChangeStateStepCount >= m_repeatChangeStateStepCount)
        {
            switch (m_spikeType)
            {
                case SpikeState.Open:
                    Close();
                    break;
            }
        }
    }

    public override void EveryAfterStep()
    {
        if (m_repeatChangeStateStepCount == 0)
            return;

        if (m_currentChangeStateStepCount >= m_repeatChangeStateStepCount)
        {
            switch (m_spikeType)
            {
                case SpikeState.Close:
                    Open();
                    break;
            }
        }
    }
}
```



Enable



Disable

작동 방식

- 가시가 활성화된 상태에서 플레이어가 가시에 충돌하면 PlayerController.OnDie를 호출합니다.
- 걸음 수 기반으로 EveryBeforeStep(), EveryAfterStep() 메서드에서 상태를 변경합니다.

기믹 오브젝트 코드

3-3. 박스 방출기 클래스

```
public class BoxEjector : StepBehavior
{
    private readonly Vector3 m_SPAWN_OFFSET = Vector3.up;

    [Header("Presets")]
    [SerializeField]
    private Transform m_EjectTransform;

    [SerializeField]
    private int m_RepeatEjectStepCount;
    private IntReactiveProperty m_CurrentEjectStepCount = new IntReactiveProperty();

    [SerializeField]
    private int m_MaxGenerateCount = 1;
    private int m_CurrentGenerateCount = 0;

    [SerializeField]
    private GameObject m_BoxPrefab;

    private Animator m_Animator;
    private EjectorView m_EjectorView;

    private int LeftEjectStepCount
    {
        get { return m_RepeatEjectStepCount - m_CurrentEjectStepCount.Value; }
    }

    private bool PossibleGenerate
    {
        get { return m_MaxGenerateCount > m_CurrentGenerateCount; }
    }

    private void OnDrawGizmos()
    {
        Gizmos.color = Color.magenta;
        Gizmos.DrawLine(transform.position + m_SPAWN_OFFSET, m_EjectTransform.position
    }

    private void Start()
    {
        m_Animator = GetComponent<Animator>();
        m_EjectorView = GetComponentInChildren<TextEjectorView>();

        m_CurrentEjectStepCount = new IntReactiveProperty(PossibleGenerate ? LeftEjectStepCount : -1);
        m_CurrentEjectStepCount.Subscribe(leftCounts => m_EjectorView.Display(leftCounts));
        UpdateLeftStepProperty();
    }

    public override void EveryBeforeStep()
    {
        m_CurrentEjectStepCount.Value++;
    }

    public override void EveryAfterStep()
    {
        if (m_CurrentEjectStepCount.Value >= m_RepeatEjectStepCount && PossibleGenerate)
        {
            if (TryEjectBox())
            {
                m_CurrentGenerateCount++;
                m_CurrentEjectStepCount.Value = 0;
            }
        }

        UpdateLeftStepProperty();
    }

    private void UpdateLeftStepProperty()
    {
        m_CurrentEjectStepCount.Value = (PossibleGenerate ? LeftEjectStepCount : -1);
    }

    public bool TryEjectBox()
    {
        if (Physics.Linecast(transform.position + m_SPAWN_OFFSET, m_EjectTransform.position))
        {
            return false;
        }

        m_Animator.Play("Open");
        Vector3 direction = (m_EjectTransform.position - (transform.position + m_SPAWN_OFFSET)).normalized *
MoveableBase.BLOCK_SIZE;
        Vector3 position = transform.position + m_SPAWN_OFFSET + direction;

        GameObject box = Instantiate(m_BoxPrefab, transform.position + m_SPAWN_OFFSET + direction / 2f,
Quaternion.identity);
        box.GetComponent<Box>().MoveToPosition(position, direction).Forget();

        return true;
    }
}
```

작동 방식

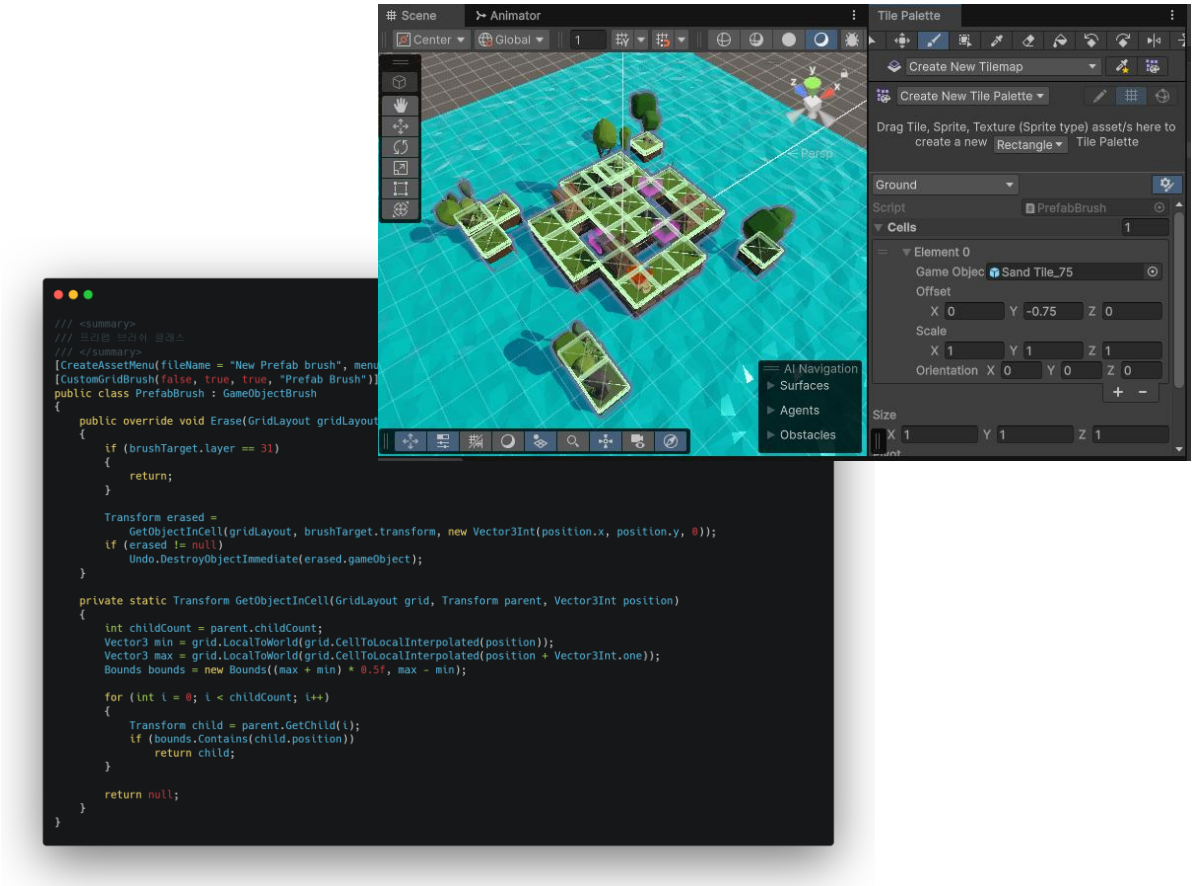
- 특정 움직임 횟수보다 박스를 최대치까지 방출하는 박스 방출기 클래스입니다.
- 매 EveryBeforeStep 때마다 걸음 수가 증가합니다.
- 매 EveryAfterStep 때마다 배출 가능 여부를 확인하고, 배출할 수 있으면 배출합니다.
- 남은 걸음수는 EjectorView 컴포넌트에서 UI화 되어 출력됩니다.

맵 제작 과정

1. 개요

요약

커스텀 브러시를 통한 맵 배치를 한 뒤, 기믹 오브젝트를 배치하여 레벨 디자인을 합니다.



제작 방식

- PrefabBrush 클래스를 통해 타일맵 프리셋을 저장할 수 있게 합니다. 이를 통해 스테이지마다 일관성 있는 디자인을 유지할 수 있도록 합니다.
- (-1, 1, -1) 월드 좌표를 센터로 가정하고 맵을 제작합니다.
- 클리어 가능 여부는 브루트 포스로 구현된 웹 사이트에서 확인해보며 작업했습니다.
- Static Batching을 활용하여 배치 수가 200을 넘기지 않도록 최대한 최적화에 공을 많이 들였습니다.

후기

협업 프로젝트를 통한 커뮤니케이션 능력 향상, 일정 관리 및 기술 개발에 대한 아쉬움이 많은 프로젝트였습니다.

- 매일 10분씩 회의를 통해 단기간 목표를 설정하고 다음날 목표 달성 여부를 통해 진척도를 파악했습니다.
- 기획 및 디자인을 현업자 및 전공자한테 많이 여쭈어보고 인사이트를 얻는 과정에서 스스로 부족한 점을 되돌아보는 계기가 되었습니다.
- 협업 과정에서 팀원들과의 원활한 커뮤니케이션이 프로젝트의 효율성을 크게 좌지우지함을 많이 실감하고 몸소 느꼈습니다.
- 더 큰 프로젝트라면, 일정 조율 및 개발 목표 설정에서 좀 더 체계적인 관리 방법이 필요할 것 같다고 많이 느꼈습니다.

Sokoland

제 포트폴리오를 읽어주셔서 감사합니다!

이메일: lsnan421@naver.com
깃허브: leeinhwan0421
유튜브: https://www.youtube.com/@GameDev-LeeInHwan
노션: https://leeinhwan-portfolio.notion.site/

작성 일지: 2025. 03. 05