

# 2024-2 게임 결과 보고서 양식

## 게임 결과보고서

게임 제목 : Game Fixer

과목명 : 게임프로그래밍

교수명 : 조경은

학과명 : 멀티미디어 소프트웨어공학전공, 데이터 사이언스전공

조이름 : 면공면

이름 : 이강민, 전지훈, 강동현

제출일 : 2024. 12. 14

## 목 차

• 게임 시놉시스 .....	3
1.1 게임 이름 .....	3
1.2 게임 장르 .....	3
1.3 게임 주제 .....	3
1.4 게임 배경 .....	3
1.5 게임의 목적과 목표 .....	3
1.6 게임의 특징과 비전 .....	3
1.7 게임 창작 의도 .....	3
• 게임 스토리 .....	3
2.1 게임 줄거리 .....	3
• 화면흐름도 .....	4
3.1 화면 흐름도 .....	4
3.2 화면 구성의 상세설명 .....	4
• 게임 규칙 .....	5
4.1 게임 규칙 .....	5
• 맵 설계 .....	5
5.1 맵 구성 .....	5

5.2 맵 상세 설계 .....	5
• 기능 설계 .....	9
6.1 핵심 기능 .....	9
6.2 보조 기능 .....	37
• 캐릭터 설계 .....	46
7.1 주인공 캐릭터 .....	46
7.2 Error Object .....	46
• 화면 구성 .....	47
8.1 타이틀 화면 .....	47
8.2 프롤로그 화면 .....	50
8.3 스테이지 클리어 화면 .....	52
8.4 ESC 화면 .....	52
8.5 엔딩 화면 .....	52
8.6 게임오버 화면.....	55
• 인공지능 기법 .....	55
• 인터페이스 .....	55
10.1 조작 인터페이스 설명 .....	55
• 스테이지 구성 .....	57
• 그래픽 데이터 리스트 .....	57
12.1 그래픽 데이터 .....	57
12.2 사용한 애셋 .....	57
• 사운드 데이터 리스트 .....	59
13.1 사운드 데이터 .....	59
13.2 사용한 애셋 .....	59
• 제작진 역할 분담 .....	59
14.1 역할 분담 .....	59
• 개발 시 애로사항 및 느낀 점 .....	60
15.1 개발 시 애로사항 .....	60

## 게임 시놉시스

**1-1. 게임 제목 :** Game Fixer

**1-2. 게임 장르 :** 오프라인 퍼즐 기반 스테이지형 탈출 게임

**1-3. 게임 주제 :** 물체 및 시간 조작을 통한 게임 스테이지 픽스

**1-4. 게임 배경 :** **The Exit**이라는 게임의 관리자인 플레이어, 어느 날 원인 불명의 **버그**가 발생하면서 스테이지들의 탈출구를 향한 길이 엉망이 되어있거나, 사용해야할 오브젝트들이 크기가 막무가내로 뒤바뀌어 있는 등 게임 내의 여러 스테이지가 엉망이 되어버린다. 과연 플레이어는 스테이지들을 모두 무사히 고쳐서 게임을 복구할 수 있을 것인가!

**1-5. 게임의 목적과 목표 :** 맵을 탈출하기 위해 버그에 걸려 여기저기 흩뿌려진 물건들을 옮기고, 사이즈를 조정해서 길을 만들어 탈출구까지 다가갈 수 있게 하여 맵을 통과하는게 목표

**1-6. 게임의 특징과 비전 :** 시간과 공간을 조종한다는게 가장 큰 특징이 된다. 시간을 조종하여 움직였던 물체를 되돌릴 수 있으며, 공간을 조종해 원하는 물체를 위치와 크기를 맞추어 출구로 향하는 길로 재탄생시킬 수 있다.

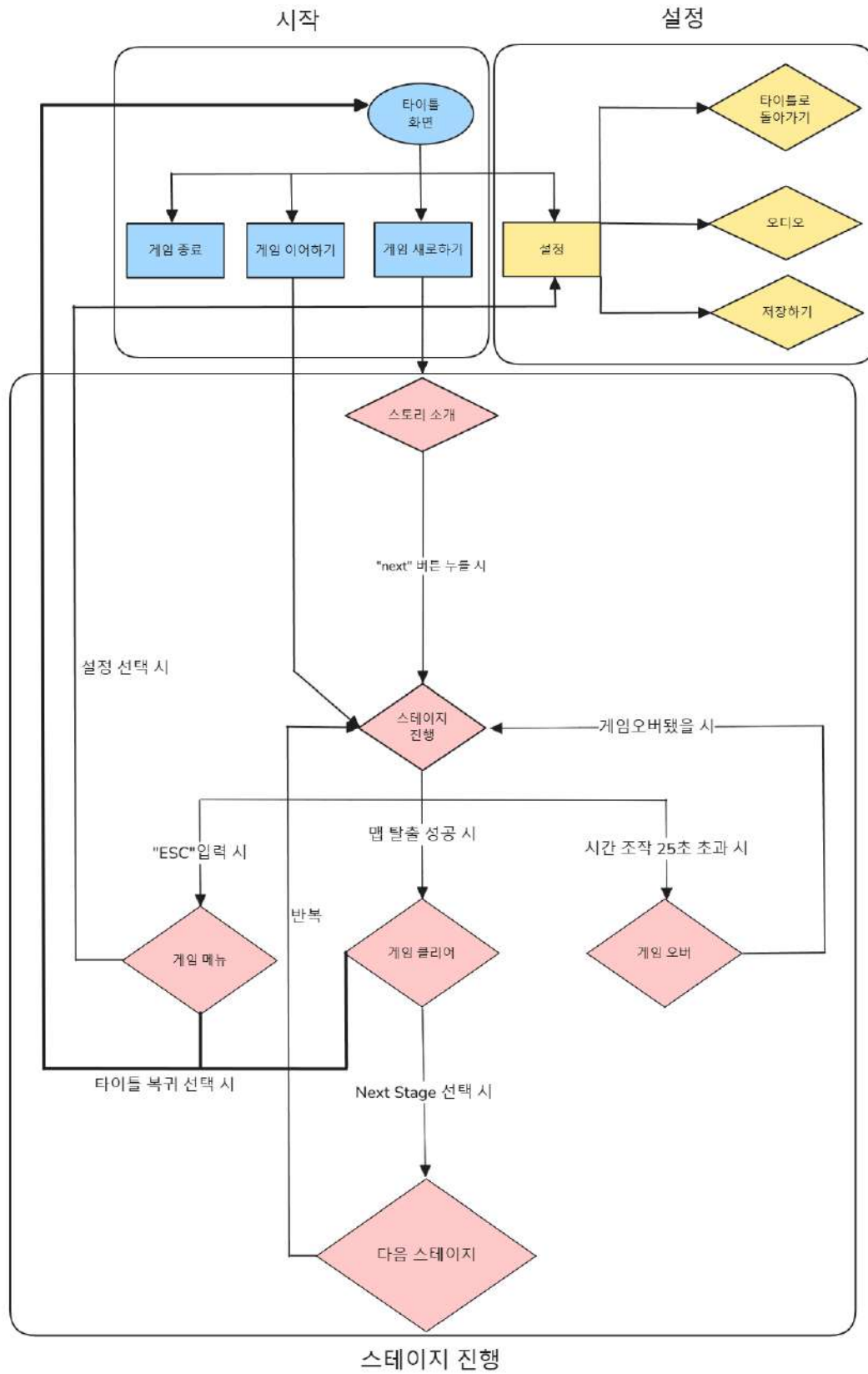
**1-7. 게임 창작 의도 :** 조원들끼리 어떤 게임을 만들까 상의를 해본 결과 퍼즐 게임이 가장 잘 표현할 수 있을 것이라 판단해 퍼즐로 장르를 정하였고, 개발 과정에서 기술적 측면의 난이도 및 신박함을 중점으로 개발하기 위해 생각하던 중 최근에 재밌게 플레이했던 게임 “SuperLiminal”에서 영감을 따와 그에 나왔던 기능들을 한 번 구현해보고자 생각하였다. 또한 이것만으로는 게임에 재미를 불어넣기 부족하다 싶어서, 탈출을 시도하는데 제한적 요소들을 두고 이를 재활용할 수 있게 해주는 시간 되돌리기도 추가해주었다. 그리고 조금 더 시각적인 자극을 투여하기 위해 게임 “Return of Obra Dinn”에 사용된 기법인 화면 셰이더를 1비트 프로세싱으로 바꿔주는 포스트 프로세싱 기법(1bit Dithering) 또한 사용함으로써 조금 더 보는 재미를 이끌 수 있도록 의도하였다.

## 2. 게임 스토리

**2-1. 게임 줄거리 :** 탈출맵 게임인 **The Exit**을 관리하는 주인공은 어느 날 의미불명의 이유로 게임이 망가져있는 것을 확인하고 탈출하는 길이 붕괴된 스테이지들을 찾아다닌다. 길이 붕괴된 스테이지의 물체들을 직접 옮겨 원래 있어야했던 길을 만들고 잘못 만들었을 경우 관리자의 눈을 통해 시간을 되돌려 다시 만드는 과정을 통해 스테이지를 통과함으로써 스테이지를 정상화 시킨다.

### 3. 화면구성도

#### 3-1. 화면 흐름도



### 3-2. 화면의 구성의 상세설명

게임을 시작할 시 타이틀 화면에 진입을 하게 된다. 타이틀 화면에는 총 3개의 버튼이 있으며, 각각 게임을 나가는 quit, 세이브 파일을 불러와 하던 게임을 계속 하는 Load Game, 아예 게임을 새로 시작하는 Start가 있다.

설정에는 게임 세이브 기능과, 오디오 조절 기능이 포함되어있다.

우선 게임 새로하기로 진입하면, 게임의 스토리를 보여주는 간단한 컷씬이 나온다.

게임에 진입해서는 크게 세 갈래로 나뉘는데, 첫번째로 게임 도중 ESC키를 누를 시 설정 진입 키와 타이틀로 돌아가기 키가 나온다. 설정창을 통해 게임 도중에 세이브를 할 수 있도록 구현하였다.

두번째로는 게임을 클리어 했을 때로 스테이지의 출구를 통해 클리어하면 별다른 창 없이 바로 다음 스테이지로 이어지게 된다.

마지막으로 게임 오버는 Time Rewind기능을 5번 초과하게 사용할 경우 발생하며, 클리어와 마찬가지로 사망 시 별다른 창 없이 다시 스테이지를 재시작하게 된다.

게임을 완전히 클리어할 경우 엔딩 컷씬이 나오고, 타이틀로 돌아갈 수 있게 된다.

## 4 게임 규칙

### 4-1. 게임 규칙

이 게임은 탈출맵 형식의 게임으로 시작 지점에서부터 Space Control과 Time Rewind를 적절히 이용하여 물체들을 이동시켜 길을 만들어 탈출구까지 도달하는 것이 목표인 게임이다.

사용된 바인딩 키들로는 플레이어를 움직이는 WASD, Time Rewind 키인 T, Space Control 키인 마우스 Lclick, 메뉴창을 불러오는 ESC가 있다.

	조작키	특징
이동	WASD	...
점프	Space	...
메뉴	ESC	...
TimeRewind	T,Q,E	총 25초 사용 가능, QE를 통해 회전 가능
Space Control	Lclick	물체 당 1번

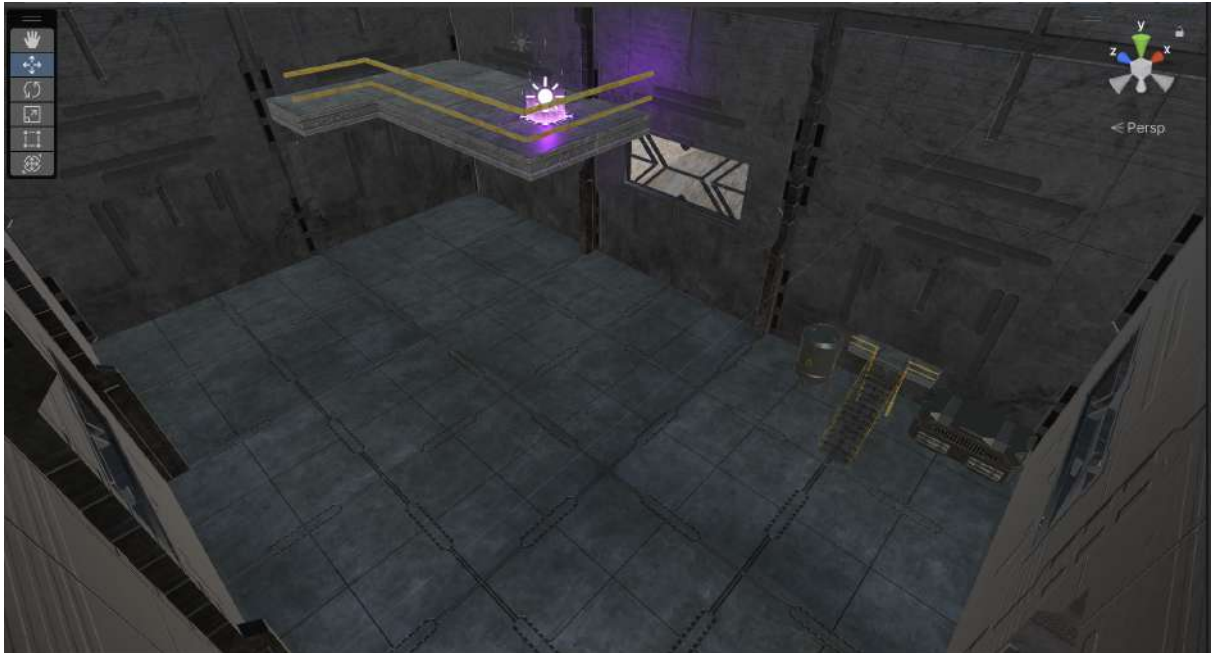
## 5 맵 설계

### 5-1 맵 구성

맵은 총 stage1과 stage2로 이루어져있다. stage1에는 간단히 Space Control기능을 이용해서 통과할 수 있도록 튜토리얼에 가깝도록 만들었고, stage2는 본격적으로 Time Rewind까지 이용하여 물체들을 옮겨 출구에 도달할 수 있게 하도록 설계하였다.

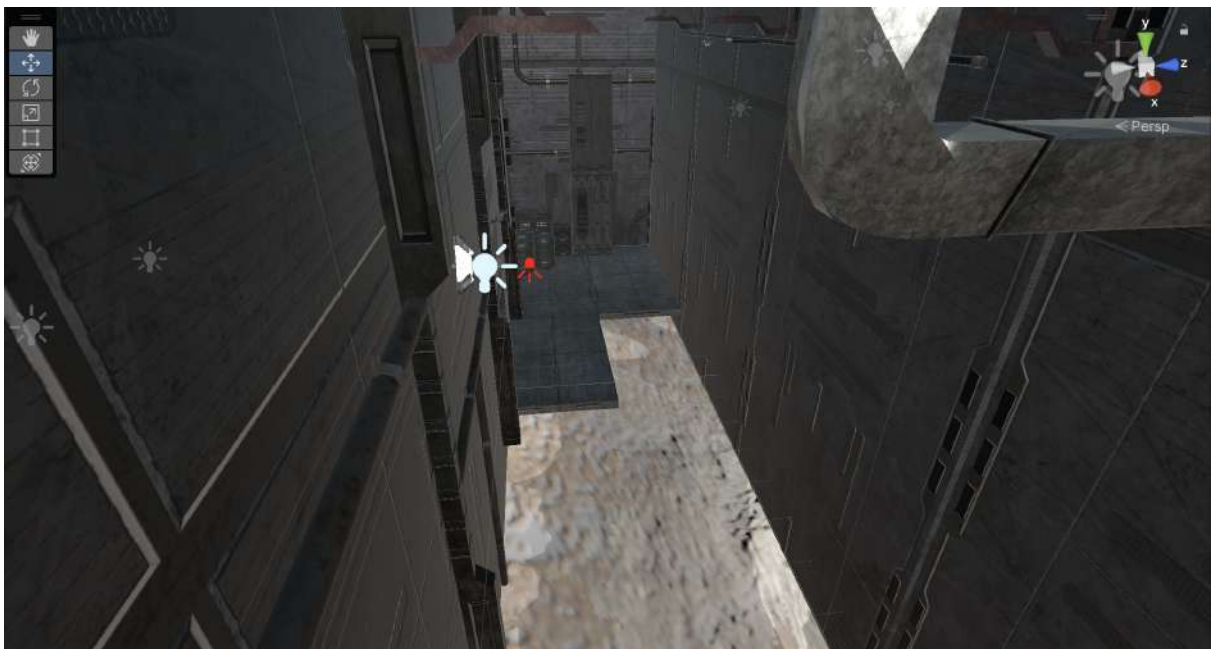
## 5-2 맵 상세 설계

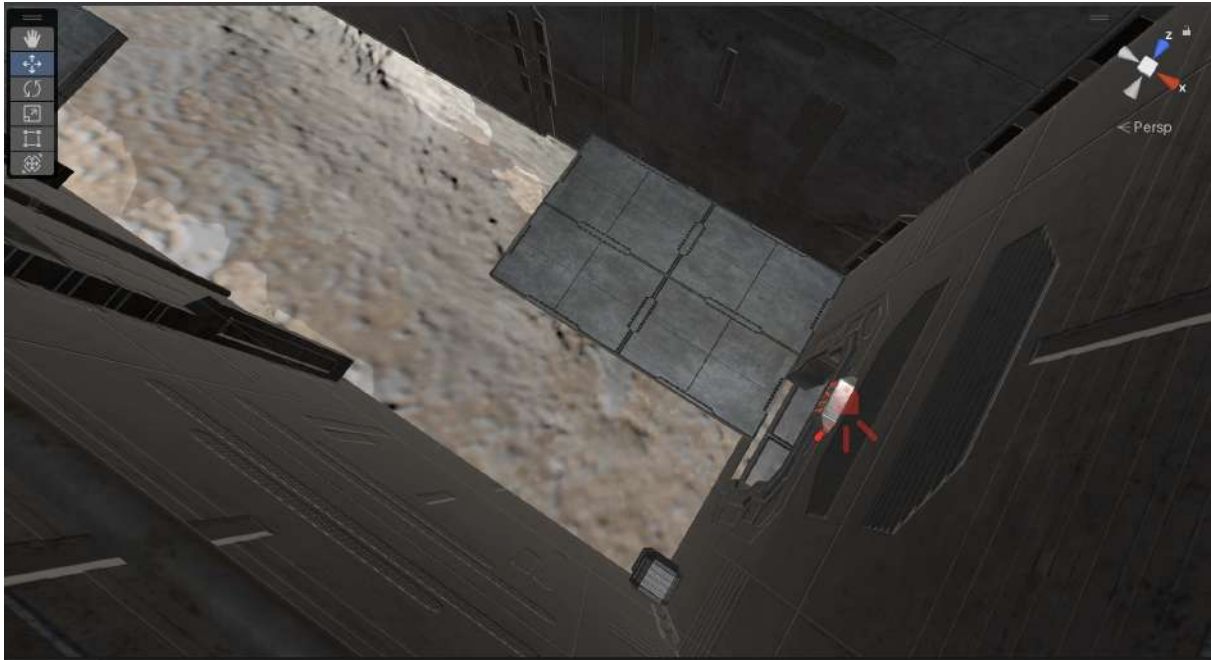
### stage1



위 스테이지에는 움직일 수 있는 물건이 3개 있다. 오른쪽 아래에 보이는 barrel, stair, crate이다. 기본적으로 stair을 이용하여 2층에 있는 portal에 도달하는게 기본 루트이다.

### stage2



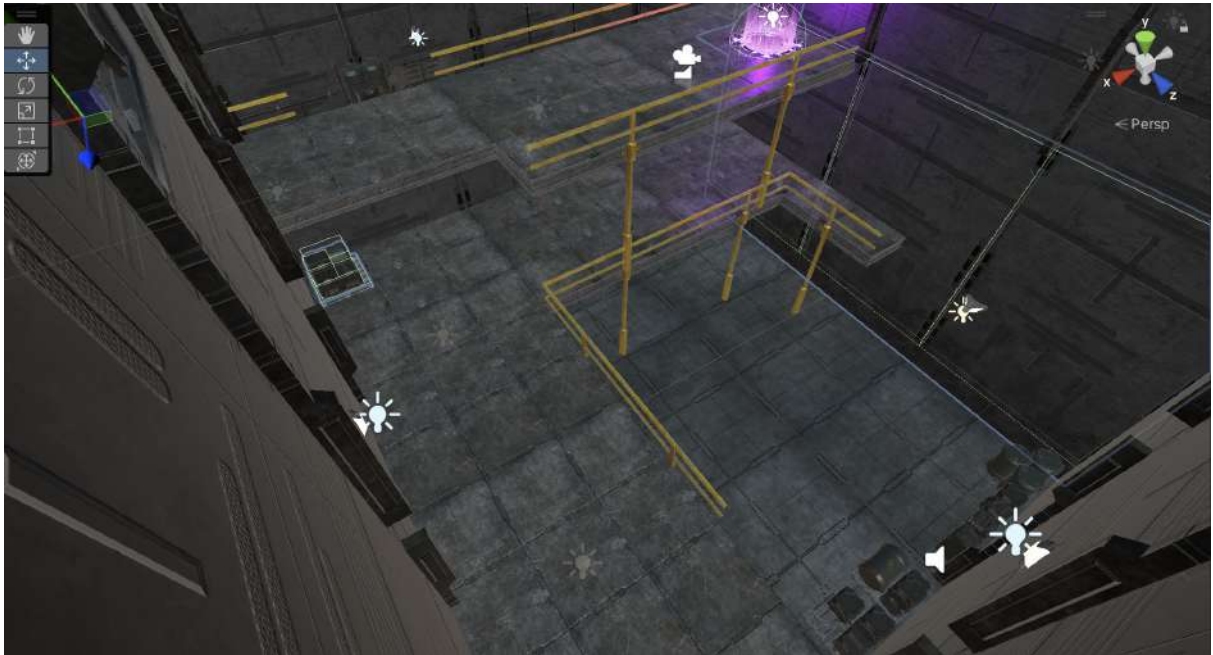


stage1의 포탈을 타면 나오는 stage2의 첫번째 구역이다. 맵에는 직사각형의 cube만 배치하여 다리처럼 쓸수 있게 했다. 또한 밑으로 떨어지는 구간이 있기 때문에 Player의 높이가 일정 높이 이하라면 스테이지를 처음부터 시작할수 있게 했다.



다리 부분을 통과하면 stage2의 마지막 부분이 나온다. stage1과 마찬가지로 포탈에 닿으면 game clear 창으로 넘어가게 된다. 이 맵에서는 2가지의 루트가 있다. 첫 번째 루트는 넘어올때 썼던 다리를 다시 갖고온후 오른쪽 상단에 뚫려있는 곳으로 한번에 올라가는 방법이다.





두 번째 루트는 정규 루트이다. 다리를 통과하고 왼쪽의 문을 지나면 3층 구조의 맵이 나온다. 1층에서는 첫 번째 사진처럼 비스듬히 놓여있는 pallet을 사용하여 2층으로 올라간다. 2층에서는 계단처럼 쓸수있는 pallet(1층에서 썼던 pallet과 다른 버전이다.)을 이용하여 포탈이 있는 3층으로 올라가면 된다.

## 6 기능의 설계

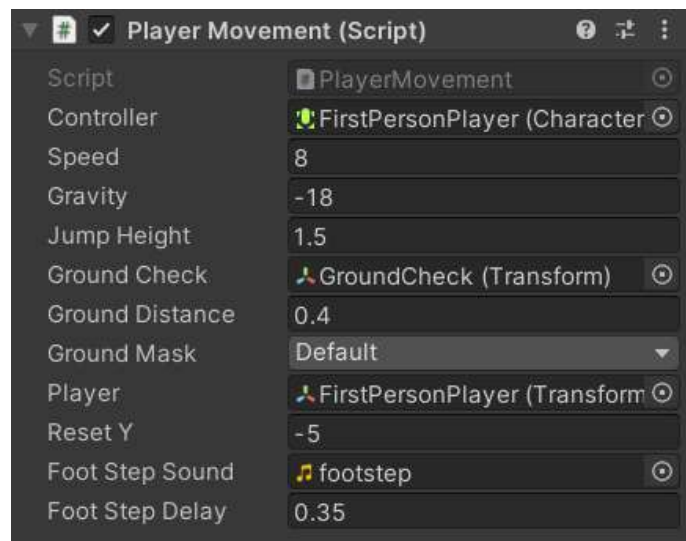


기능	효과	사용법	사용횟수
Space Control	선택한 오브젝트를 옮길 수 있으며, 원근감을 유지하며 이동. 이를 통해 물체의 실제 크기 조정 가능	오브젝트를 왼쪽 클릭으로 발동, QE를 통해 회전	각 오브젝트 당 1번
Time Rewind	움직였던 물체의 시간을 되돌려 이전에 있던 위치로 돌아간다. 시간을 되돌림으로써 이미 물체에 사용된 Space Control횟수를 다시 1회 채워준다.	T를 눌러 발동	최대 25초 사용 가능
1bit Dithering	플레이어가 보는 화면이 비트로 흐려지며, 총 다섯 단계에 걸쳐 흰 노 초 파 빨 순서대로 색이 바뀌며 시간이 다 지날 경우 화면이 까매지며 게임 오버가 됨	Time Rewind의 사용에 따른 시간경과를 따라간다.	Time Rewind 5초당 1번 바뀌며 총 5번 바뀜

## 6-1 핵심 기능

### 플레이어 움직임

구현 의도 - 플레이어는 통상적으로 가장 많이 사용하는 WASD를 이용하여 움직이도록 하였으며, space를 통해 점프를 하게 구현하였다.



이에 관한 스크립트인 Player Movement의 인스펙터창은 이렇게 구성되어있으며, 하나하나 설명해보고자 한다.

우선 이는 플레이어의 움직임에 필요한 변수들을 선언한 필드로,

```

public CharacterController controller;
public float speed = 8f;
public float gravity = -9.81f;
public float jumpHeight = 3f;
public Transform groundCheck;
public float groundDistance = 0.4f;
public LayerMask groundMask;
Vector3 velocity;
bool isGrounded;

public Transform player; // 플레이어의 Transform 참조
public float resetY = -100f; // 초기화할 Y 좌표 기준값

public AudioClip footStepSound;
public float footStepDelay;

private float nextFootstep = 0;

```

처음 업데이트 메소드에서는 Physics.CheckSphere(groundCheck.position, groundDistance, groundMask)를 이용하여 플레이어가 땅에 닿아있는지 확인을 해준다. groundCheck.position의 위치를 기준으로 groundDistance의 반경만큼 충돌체와의 감지를 하며 GroundMask로 지정된 레이어만을 땅으로 인식하게 해주, 플레이어가 붕 뜨는 것을 방지하기 위해, 땅에 붙어있을 때 y축을 -2로 고정해서 안정적으로 땅에 붙어있도록 해주었다.

```

isGrounded = Physics.CheckSphere(groundCheck.position, groundDistance, groundMask);

if (!isGrounded && velocity.y < 0)
{
    velocity.y = -2f;
}

```

다음으로 움직임을 표현하기 위해 Horizontal(A, D 키 또는 화살표)와 Vertical(W, S 키 또는 화살표)로 입력을 받는다.

입력 값을 기준으로 이동 방향인 motion을 계산합니다.CharacterController.move를 사용해 프레임 독립적인 속도로 이동한다.

```

float x = Input.GetAxis("Horizontal");
float z = Input.GetAxis("Vertical");

Vector3 motion = transform.right * x + transform.forward * z;
controller.Move(motion * speed * Time.deltaTime);

```

그 다음으로는 점프로, 점프 키(space)를 누른 상태이고, 앞서 설명했던 isGrounded, 즉 땅에 붙어있다는 상태가 확인 된다면 필드에서 지정해준 JumpHeight만큼에 중력값에 -2를 곱한 값을 곱해 제곱근을 씌워주워 점프 높이값을 계산해준다.

이제 점프를 했으므로, 떨어져야하기때 매 프레임마다 y값에 중력값을 더해주어 착지하게 된다.

```

if (Input.GetButtonDown("Jump") && isGrounded)
{
    velocity.y = Mathf.Sqrt(jumpHeight * -2f * gravity);
}

velocity.y += gravity * Time.deltaTime;
controller.Move(velocity * Time.deltaTime);

```

다음은 발소리를 재생하는 코드로, 이동키인 WASD중 하나 이상이 눌리고, isGrounded 상태의 전제에서 발소리가 겹치지 않게 하기 위해 다음 발소리가 재생되기까지 남은 시간인 nextfootstep에 딜레이 시간인 footstepDelay시간을 더해준다. 이때 nextfootstep이 0 아래로 떨어지면 소리가 나게 되는 구조이다. Delay값이 더해진 상태에서 발소리가 재생되면 deltatime만큼 nextfootstep 프레임마다 빼주면서 딜레이 값이 다 줄고 0 아래로 내려가면 다시 발소리가 재생되도록 구현하였다.

```

if (Input.GetKey(KeyCode.A) || Input.GetKey(KeyCode.S) || Input.GetKey(KeyCode.D) || Input.GetKey(KeyCode.W) && isGrounded)
{
    nextFootstep -= Time.deltaTime;
    if (nextFootstep <= 0)
    {
        GetComponent().PlayOneShot(footStepSound, 0.7f);
        nextFootstep += footStepDelay;
    }
}

```

마지막으로, 낙사에 관한 코드로 stage2 초반부에는 낙사를 할 수 있는 구간이 있기 때문에, 플레이어의 y좌표가 임의로 지정한 위치 초기화y좌표를 넘어서 떨어지게 되면 현재씬을 재로드해서 스테이지를 다시 시작하도록 하게 하였다.

```

if (player.position.y <= resetY)
{
    string currentSceneName = SceneManager.GetActiveScene().name;
    SceneManager.LoadScene(currentSceneName); // 현재 씬 로드
}

```

## Space Control

SpaceControl 기능은 **Superliminal.cs** 스크립트에서 작동하며, *Superliminal*의 게임 플레이 메커니즘에서 영감을 받아 만든 기능이다. 플레이어가 Raycasting과 객체를 선택, 크기 조정 및 회전할 수 있도록 한다.

유니티에디터 내에서 layer를 targetable로 설정한 오브젝트만 잡고 옮길 수 있다.

## 구성 요소

```

[Header("Components")]
public Transform target;

```

```
[Header("Parameters")]
public LayerMask targetMask;
public LayerMask ignoreTargetMask;
public float offsetFactor;

public List<Transform> usedObjects = new List<Transform>();
```

- **Components 헤더:**

- `Transform target` : 현재 선택된 객체를 나타낸다.

- **Parameters 헤더:**

- `LayerMask targetMask` : 레이캐스팅 시 대상이 될 레이어를 지정한다.
- `LayerMask ignoreTargetMask` : 레이캐스팅 시 무시할 레이어를 지정한다(주로 플레이어 및 타겟 레이어).
- `float offsetFactor` : 객체가 벽에 끼이지 않도록 하는 오프셋 거리이다.

- **Used Objects:**

- `List<Transform> usedObjects` : 조작된 객체를 추적하여 반복적인 상호작용을 방지한다.

```
float originalDistance;
float originalScale;
Vector3 targetScale;
```

- **originalDistance**: 플레이어의 카메라와 타겟 객체 간의 초기 거리이다.
- **originalScale**: 조작 전 타겟 객체의 원래 크기이다.
- **targetScale**: 매 프레임마다 타겟 객체에 적용할 크기이다.

## 초기 설정 ( `Start` 메서드)

```
void Start()
{
    Cursor.visible = false;
```

```

    Cursor.lockState = CursorLockMode.Locked;
}

```

- **커서 설정:** 게임 시작 시 커서를 숨기고 잠금 상태로 설정하여 플레이어의 시야를 방해하지 않도록 한다.

## 업데이트 루프 ( **Update** 메서드)

```

void Update()
{
    HandleInput();
    ResizeTarget();
    RotateTarget();
}

```

- **입력 처리:** 사용자 입력을 처리하여 객체를 선택하거나 해제한다.
- **타겟 크기 조정:** 선택된 객체의 크기를 실시간으로 조정한다.
- **타겟 회전:** 선택된 객체를 회전시킨다.

다음 3가지를 매 프레임마다 시행하면서 Space Control을 시행했다.

## 입력 처리 ( **HandleInput** 메서드)

```

void HandleInput()
{
    if (Input.GetMouseButtonDown(0))
    {
        if (target == null)
        {
            RaycastHit hit;
            if (Physics.Raycast(transform.position, transform.forward, out hit, Mathf.Infinity, targetMask))
            {
                if (usedObjects.Contains(hit.transform))
                {
                    Debug.Log("이미 사용된 오브젝트입니다.");
                    return;
                }
            }
        }
    }
}

```

```

        }

        target = hit.transform;

        Rigidbody rb = target.GetComponent<Rigidbody>
y>();

        if (rb != null)
        {
            rb.isKinematic = true;
        }

        originalDistance = Vector3.Distance(transform.position, target.position);
        originalScale = target.localScale.x;
        targetScale = target.localScale;
    }
}
else
{
    Rigidbody rb = target.GetComponent<Rigidbody>
();

    if (rb != null)
    {
        rb.isKinematic = false;
    }

    usedObjects.Add(target);
    target = null;
}
}
}

```

- **마우스 클릭 감지:** 왼쪽 마우스 버튼 클릭 시 객체를 선택하거나 해제한다.
- **레이캐스트:** `targetMask`에 해당하는 레이어에서 객체를 탐지한다.
- **사용 내역 확인:** 이미 사용된 객체는 선택되지 않도록 한다.
- **객체 설정:** 선택된 객체의 `Rigidbody`를 키네마틱으로 설정하여 물리적 영향을 받지 않도록 한다.



- **초기 거리 및 스케일 저장:** 객체의 초기 거리와 스케일을 저장하여 이후 조작에 활용한다.
- **객체 해제:** 다시 클릭 시 객체의 물리적 영향을 복원하고 사용 내역에 추가한다.

`HandleInput` 메서드는 마우스 클릭 입력을 감지하고 객체의 선택 또는 해제를 처리한다.

### 1. 왼쪽 마우스 버튼 클릭 감지: `if (Input.GetMouseButtonDown(0))`

사용자가 왼쪽 마우스 버튼을 클릭했는지 확인한다.

### 2. 타겟이 없는 경우 객체 선택 시도:

- `if (target == null)`

현재 선택된 타겟이 없을 때 실행된다.

#### • 레이캐스트 실행:

`Physics.Raycast` 를 사용하여 플레이어의 시선 방향으로 레이캐스트를 발사하고, `targetMask` 에 해당하는 레이어의 객체를 감지한다.

#### • 이미 사용된 객체인지 확인:

감지된 객체가 `usedObjects` 리스트에 포함되어 있는지 확인한다. 포함되어 있으면 "이미 사용된 오브젝트입니다."라는 메시지를 출력하고 선택을 취소한다.

#### • 타겟 설정:

감지된 객체를 `target` 으로 설정한다.

#### • Rigidbody 비활성화:

선택된 객체에 `Rigidbody` 컴포넌트가 있으면 `isKinematic` 을 `true` 로 설정하여 물리적 영향을 받지 않도록 한다.

#### • 초기 거리 계산:

플레이어의 위치와 선택된 객체 간의 거리를 `originalDistance` 에 저장한다.

#### • 초기 스케일 저장:

객체의 원래 스케일을 `originalScale` 에 저장한다.

#### • 타겟 스케일 설정:

현재 객체의 스케일을 `targetScale` 에 저장한다.

### 3. 타겟이 있는 경우 객체 해제 처리:

- `else`

이미 선택된 타겟이 있을 때 실행된다.

- **Rigidbody 활성화:**

선택된 객체에 `Rigidbody` 컴포넌트가 있으면 `isKinematic` 을 `false` 로 설정하여 물리적 영향을 받도록 한다.

- **사용된 객체 리스트에 추가:**

선택된 객체를 `usedObjects` 리스트에 추가한다.

- **타겟 해제:**

`target` 을 `null` 로 설정하여 객체 선택을 해제한다.

## 타겟 크기 조정 ( `ResizeTarget` 메서드)

```
void ResizeTarget()
{
    if (target == null)
    {
        return;
    }

    RaycastHit hit;
    if (Physics.Raycast(transform.position, transform.forward, out hit, Mathf.Infinity, ignoreTargetMask))
    {
        target.position = hit.point - transform.forward * offsetFactor * targetScale.x;

        float currentDistance = Vector3.Distance(transform.position, target.position);
        float s = currentDistance / originalDistance;

        targetScale.x = targetScale.y = targetScale.z = s;
        target.localScale = targetScale * originalScale;
    }
}
```

- **레이캐스트:** `ignoreTargetMask` 을 사용하여 환경과의 충돌 지점을 탐지한다.

- **위치 조정:** 객체가 벽에 끼이지 않도록 위치를 조정한다.
- **크기 조정:** 카메라와 객체 간의 현재 거리를 기반으로 객체의 크기를 조정한다.

**ResizeTarget** 메서드는 선택된 객체의 크기를 조정하고, 벽과의 충돌을 방지하기 위해 위치를 조정한다.

#### 1. 타겟 유무 확인: `if (target == null)`

선택된 타겟이 없으면 메서드를 종료한다.

#### 2. 레이캐스트 실행:

`Physics.Raycast` 를 사용하여 `ignoreTargetMask` 레이어를 무시하고, 플레이어의 시선 방향으로 레이캐스트를 발사하여 벽과의 충돌 지점을 감지한다.

#### 3. 타겟 위치 조정:

감지된 충돌 지점(`hit.point`)에서 플레이어의 시선 방향(`transform.forward`)에 `offsetFactor` 와 `targetScale.x` 를 곱한 값을 빼서 타겟의 새로운 위치를 설정한다.

```
target.position = hit.point - transform.forward * offsetFactor * targetScale.x;
```

#### 4. 현재 거리 계산:

플레이어의 위치와 조정된 타겟의 위치 간의 거리를 `currentDistance` 에 저장한다.

```
float currentDistance = Vector3.Distance(transform.position, target.position);
```

#### 5. 거리 비율 계산:

현재 거리(`currentDistance`)를 초기 거리(`originalDistance`)로 나누어 비율 `s` 를 계산한다.

```
float s = currentDistance / originalDistance;
```

#### 6. 스케일 비율 설정:

비율 `s` 를 객체의 x, y, z 스케일에 모두 적용하여 `targetScale` 을 업데이트한다.

```
targetScale.x = targetScale.y = targetScale.z = s;
```

#### 7. 타겟 스케일 적용:

업데이트된 `targetScale` 에 `originalScale` 을 곱하여 객체의 최종 스케일을 설정한다.

```
target.localScale = targetScale * originalScale;
```

## 타겟 회전 ( **RotateTarget** 메서드)

```

void RotateTarget()
{
    if (target == null)
    {
        return;
    }

    float rotationSpeed = 50f;

    if (Input.GetKey(KeyCode.Q))
    {
        target.Rotate(Vector3.up, rotationSpeed * Time.deltaTime, Space.World);
    }

    if (Input.GetKey(KeyCode.E))
    {
        target.Rotate(Vector3.up, -rotationSpeed * Time.deltaTime, Space.World);
    }
}

```

- **회전 속도 설정:** 객체의 회전 속도를 설정한다.
- **회전 입력 처리:** 'Q' 키를 누르면 시계방향, 'E' 키를 누르면 반시계방향으로 객체를 회전시킨다.

`RotateTarget` 메서드는 선택된 객체를 사용자 입력에 따라 회전시킨다.

1. **타겟 유무 확인:** `if (target == null)`  
선택된 타겟이 없으면 메서드를 종료한다.

2. **회전 속도 설정:**  
회전 속도를 `50f`로 설정한다.

```
float rotationSpeed = 50f;
```

3. **시계방향 회전 처리:**

- `if (Input.GetKey(KeyCode.Q))`

사용자가 'Q' 키를 누르고 있는지 확인한다.

- **타겟 회전:**

'Q' 키가 눌러 있으면, `rotationSpeed` 에 `Time.deltaTime` 을 곱한 값을 사용하여 객체를 월드 좌표계 기준으로 시계방향으로 회전시킨다.

```
target.Rotate(Vector3.up, rotationSpeed * Time.deltaTime, Space.World);
```

#### 4. 반시계방향 회전 처리:

- `if (Input.GetKey(KeyCode.E))`

사용자가 'E' 키를 누르고 있는지 확인한다.

- **타겟 회전:**

'E' 키가 눌러 있으면, `rotationSpeed` 에 `Time.deltaTime` 을 곱한 값을 사용하여 객체를 월드 좌표계 기준으로 반시계방향으로 회전시킨다.

```
target.Rotate(Vector3.up, -rotationSpeed * Time.deltaTime, Space.World);
```

따라서 위의 3개의 메서드를 매 프레임마다 실행하면서 Space Control의 기능을 사용할 수 있다.

## Time Rewind

Unity 게임 엔진에서 시간 되돌리기(Rewind) 기능을 구현하기 위한 시스템을 구성한다. 이 시스템은 객체의 상태(위치, 회전, 속도, 애니메이션 등)를 추적하고, 특정 시점으로 되돌리는 기능을 제공한다. 이를 통해 플레이어는 게임 내의 오브젝트를 원래 상태로 되돌릴 수 있다.

본 보고서에서는 세 가지 주요 파일인 **RewindAbstract.cs**, **RewindByKeyPress.cs**, **RewindManager.cs**를 상세히 분석하여, 이들이 어떻게 상호 작용하며 시간 되돌리기 기능을 구현하는지 설명하겠다.

### i) RewindAbstract.cs 분석

#### 클래스 개요

```
public abstract class RewindAbstract : MonoBehaviour
{
    // ...
    public abstract void Track();
}
```

```
public abstract void Rewind(float seconds);
}
```

**RewindAbstract** 클래스는 **MonoBehaviour** 를 상속받는 추상 클래스이다. 이 클래스는 시간 되돌리기 기능을 구현하기 위한 기본 구조를 제공하며, 다양한 컴포넌트(Rigidbody, Animator 등)의 상태를 추적하고 되돌리는 기능을 포함하고 있다. 구체적인 추적 및 되돌리기 로직은 이 클래스를 상속받는 하위 클래스에서 구현해 두었다.

## 주요 변수 및 구조체

### 컴포넌트 변수

```
Rigidbody body;
Rigidbody2D body2;
Animator animator;
AudioSource audioSource;
```

- **Rigidbody & Rigidbody2D:** 객체의 물리적 속도 및 회전을 추적한다.
- **Animator:** 객체의 애니메이션 상태를 추적한다.
- **AudioSource:** 객체의 오디오 상태를 추적한다.

### 추적 데이터 버퍼

```
CircularBuffer<bool> trackedActiveStates;
CircularBuffer<TransformValues> trackedTransformValues;
List<CircularBuffer<AnimatorCustomParameter>> trackedAnimationParameters;
CircularBuffer<VelocityValues> trackedVelocities;
List<CircularBuffer<AnimationValues>> trackedAnimationTimes;
CircularBuffer<AudioTrackedData> trackedAudioTimes;
List<CircularBuffer<ParticleTrackedData>> trackedParticleTimes;
```



- **CircularBuffer:** 시간에 따른 상태 변화를 저장하기 위한 순환 버퍼이다. 일정한 크기의 버퍼에 최신 데이터를 저장하고, 오래된 데이터는 덮어쓴다.
- **trackedActiveStates:** 객체의 활성 상태(true/false)를 추적한다.
- **trackedTransformValues:** 객체의 위치, 회전, 스케일을 추적한다.
- **trackedAnimationParameters:** Animator의 파라미터를 추적한다.
- **trackedVelocities:** Rigidbody의 속도와 각속도를 추적한다.
- **trackedAnimationTimes:** Animator의 애니메이션 시간을 추적한다.
- **trackedAudioTimes:** AudioSource의 오디오 상태를 추적한다.
- **trackedParticleTimes:** 파티클 시스템의 상태를 추적한다.

## 구조체 정의

- **TransformValues**

```
public struct TransformValues
{
    public Vector3 position;
    public Quaternion rotation;
    public Vector3 scale;
```

객체의 위치, 회전, 스케일 정보를 저장한다.

- **VelocityValues**

```
public struct VelocityValues
{
    public Vector3 velocity;
    public Vector3 angularVelocity;
    public float angularVelocity2D;
}
```

Rigidbody의 속도, 각속도 정보를 저장한다.

- **AnimatorCustomParameter**

```
public struct AnimatorCustomParameter
{
    public int parameterHash;
    public AnimatorControllerParameterType type;
    public float value;
}
```

Animator 파라미터의 해시값, 타입, 값을 저장한다.

- **AnimationValues**

```
public struct AnimationValues
{
    public float animationStateTime;
    public float clipLength;
    public int animationHash;
    public AnimatorTransitionData transitionsInfo;
}
```

애니메이션 상태 시간, 클립 길이, 애니메이션 해시값, 전환 정보를 저장한다.

- **AnimatorTransitionData**

```
public struct AnimatorTransitionData
{
    public bool isInTransition;
    public int targetStateName;
    public float normalizedTimeInTransition;
    public float transitionLength;
    public float clipLength;
    public DurationUnit durationUnit;
}
```

Animator 전환 상태 정보를 저장한다.

\*하단에 오디오와 파티클과 관련한 코드는 사용하지 않았다.

## 초기화 메서드 ( **MainInit** )

```
public void MainInit()
{
    body = GetComponent<Rigidbody>();
    body2 = GetComponent<Rigidbody2D>();
    animator = GetComponent<Animator>();
    audioSource = GetComponent<AudioSource>();

    trackedActiveStates = new CircularBuffer<bool>();
    trackedTransformValues = new CircularBuffer<TransformValues>();
    trackedAnimationParameters = new List<CircularBuffer<AnimatorCustomParameter>>();

    if(body!=null||body2!=null)
        trackedVelocities = new CircularBuffer<VelocityValues>();

    if (animator != null)
    {
        trackedAnimationTimes = new List<CircularBuffer<AnimationValues>>();
        for (int i = 0; i < animator.layerCount; i++)
            trackedAnimationTimes.Add(new CircularBuffer<AnimationValues>());

        animatorParameters = animator.parameters;
        for (int i = 0; i < animator.parameterCount; i++)
            trackedAnimationParameters.Add(new CircularBuffer<AnimatorCustomParameter>());
    }

    if(audioSource!=null)
        trackedAudioTimes = new CircularBuffer<AudioTrackedData>();

    Application.logMessageReceivedThreaded += LogMessageCat
```

```
cher;
}
```

**MainInit** 메서드는 객체가 생성될 때 호출되며, 필요한 컴포넌트를 초기화하고, 추적을 위한 버퍼를 설정한다.

### 1. 컴포넌트 초기화:

- `Rigidbody`, `Rigidbody2D`, `Animator`, `AudioSource` 컴포넌트를 가져온다.

### 2. 버퍼 초기화:

- 활성 상태, Transform 값, Animator 파라미터 등을 추적하기 위한 순환 버퍼를 생성한다.
- `Rigidbody` 또는 `Rigidbody2D`가 존재하면 속도 추적을 위한 버퍼를 생성한다.
- `Animator`가 존재하면 각 레이어별로 애니메이션 시간 추적 버퍼를 생성하고, 파라미터별로 파라미터 추적 버퍼를 생성한다.
- `AudioSource`가 존재하면 오디오 상태 추적 버퍼를 생성한다.

### 3. 로그 메시지 수신 설정:

- `Application.logMessageReceivedThreaded` 이벤트에 `LogMessageCatcher` 메서드를 등록하여 특정 로그 메시지를 처리한다.

## 활성 상태 추적 ( `ActiveState` )

### 변수 및 속성

```
public bool IsManagerTrackingActiveState { get; set; } = false;
public bool IsActiveStateTracked { get; set; } = false;
CircularBuffer<bool> trackedActiveStates;
```

- **IsManagerTrackingActiveState:** 매니저가 활성 상태 추적을 활성화했는지 여부를 나타낸다.
- **IsActiveStateTracked:** 객체의 활성 상태가 추적되었는지 여부를 나타낸다.
- **trackedActiveStates:** 객체의 활성 상태를 시간에 따라 추적하는 버퍼다.

## 메서드

### TrackObjectActiveState

```
public void TrackObjectActiveState()
{
    IsActiveStateTracked = true;
    trackedActiveStates.WriteLastValue(gameObject.activeSel
f);
}
```

- 객체의 현재 활성 상태(`activeSelf`)를 `trackedActiveStates` 버퍼에 기록한다.
- `IsActiveStateTracked` 를 `true` 로 설정하여 활성 상태가 추적되었음을 표시한다.

### RestoreObjectActiveState

```
public void RestoreObjectActiveState(float seconds)
{
    gameObject.SetActive(trackedActiveStates.ReadFromBuffer
(seconds));
}
```

- 지정된 `seconds` 만큼 과거의 활성 상태를 `trackedActiveStates` 버퍼에서 읽어와 객체의 활성 상태를 복원한다.

## Transform 추적 ( `Transform` )

### 변수 및 구조체

```
CircularBuffer<TransformValues> trackedTransformValues;

public struct TransformValues
{
    public Vector3 position;
    public Quaternion rotation;
```

```
public Vector3 scale;
}
```

- **trackedTransformValues:** 객체의 위치, 회전, 스케일을 시간에 따라 추적하는 버퍼이다.
- **TransformValues:** 객체의 Transform 상태를 저장하는 구조체이다.

## 메서드

### TrackTransform

```
protected void TrackTransform()
{
    TransformValues valuesToWrite;
    valuesToWrite.position = transform.position;
    valuesToWrite.rotation = transform.rotation;
    valuesToWrite.scale = transform.localScale;
    trackedTransformValues.WriteLastValue(valuesToWrite);
}
```

- 현재 객체의 위치, 회전, 스케일을 **TransformValues** 구조체에 저장하고, **trackedTransformValues** 버퍼에 기록한다.

### RestoreTransform

```
protected void RestoreTransform(float seconds)
{
    TransformValues valuesToRead = trackedTransformValues.ReadFromBuffer(seconds);
    transform.SetPositionAndRotation(valuesToRead.position, valuesToRead.rotation);
    transform.localScale = valuesToRead.scale;
}
```



- 지정된 `seconds` 만큼 과거의 Transform 상태를 `trackedTransformValues` 버퍼에서 읽어와 객체의 위치, 회전, 스케일을 복원한다.

## 속도 추적 ( `Velocity` )

### 변수 및 구조체

```
csharp
코드 복사
public struct VelocityValues
{
    public Vector3 velocity;
    public Vector3 angularVelocity;
    public float angularVelocity2D;
}
CircularBuffer<VelocityValues> trackedVelocities;
```

- VelocityValues:** 객체의 속도 및 각속도를 저장하는 구조체이다.
- trackedVelocities:** 객체의 속도 변화를 시간에 따라 추적하는 버퍼이다.

## 메서드

### TrackVelocity

```
protected void TrackVelocity()
{
    if (body != null)
    {
        VelocityValues valuesToWrite;
        valuesToWrite.velocity = body.velocity;
        valuesToWrite.angularVelocity = body.angularVelocity;
        valuesToWrite.angularVelocity2D = 0;
        trackedVelocities.WriteLastValue(valuesToWrite);
    }
}
```

```

else if (body2 != null)
{
    VelocityValues valuesToWrite;
    valuesToWrite.velocity = body2.velocity;
    valuesToWrite.angularVelocity = Vector3.zero;
    valuesToWrite.angularVelocity2D = body2.angularVelocity;
    trackedVelocities.WriteLastValue(valuesToWrite);
}
else
{
    Debug.LogError("Cannot find Rigidbody on the object, while TrackVelocity() is being called!!!");
}
}

```

- **Rigidbody가 있는 경우:**

- 현재 속도( `velocity` )와 각속도( `angularVelocity` )를 `VelocityValues` 에 저장하고, `trackedVelocities` 버퍼에 기록한다.

- **Rigidbody2D가 있는 경우:**

- 현재 속도( `velocity` )와 2D 각속도( `angularVelocity2D` )를 `VelocityValues` 에 저장하고, `trackedVelocities` 버퍼에 기록한다.

- **Rigidbody와 Rigidbody2D가 모두 없는 경우:**

- 에러 메시지를 출력한다.

## RestoreVelocity

```

protected void RestoreVelocity(float seconds)
{
    if(body != null)
    {
        VelocityValues valuesToRead = trackedVelocities.ReadFromBuffer(seconds);
        body.velocity = valuesToRead.velocity;
        body.angularVelocity = valuesToRead.angularVelocity;
    }
}

```

```

y;
    }
    else
    {
        VelocityValues valuesToRead = trackedVelocities.ReadFromBuffer(seconds);
        body2.velocity = valuesToRead.velocity;
        body2.angularVelocity = valuesToRead.angularVelocity;
    }
}

```

- **Rigidbody가 있는 경우:**

- `trackedVelocities` 버퍼에서 지정된 `seconds` 만큼 과거의 속도 데이터를 읽어와 `Rigidbody`의 속도와 각속도를 복원한다.

- **Rigidbody2D가 있는 경우:**

- `trackedVelocities` 버퍼에서 지정된 `seconds` 만큼 과거의 속도 데이터를 읽어와 `Rigidbody2D`의 속도와 2D 각속도를 복원한다.

## Animator 추적 ( `Animator` )

### 변수 및 구조체

```

List<CircularBuffer<AnimationValues>> trackedAnimationTimes;
List<CircularBuffer<AnimatorCustomParameter>> trackedAnimationParameters;
AnimatorControllerParameter[] animatorParameters;

public struct AnimatorCustomParameter
{
    public int parameterHash;
    public AnimatorControllerParameterType type;
    public float value;
}

```

```

public struct AnimationValues
{
    public float animationStateTime;
    public float clipLength;
    public int animationHash;
    public AnimatorTransitionData transitionsInfo;
}

public struct AnimatorTransitionData
{
    public bool isInTransition;
    public int targetStateName;
    public float normalizedTimeInTransition;
    public float transitionLength;
    public float clipLength;
    public DurationUnit durationUnit;
}

```

- **trackedAnimationTimes:** 각 애니메이션 레이어별로 애니메이션 시간을 추적하는 버퍼 리스트이다.
- **trackedAnimationParameters:** Animator 파라미터를 추적하는 버퍼 리스트이다.
- **animatorParameters:** Animator의 파라미터 배열이다.
- **AnimatorCustomParameter:** Animator 파라미터의 해시값, 타입, 값을 저장하는 구조체이다.
- **AnimationValues:** 애니메이션 상태 시간, 클립 길이, 애니메이션 해시값, 전환 정보를 저장하는 구조체이다.
- **AnimatorTransitionData:** Animator 전환 상태 정보를 저장하는 구조체이다.

## 메서드

### TrackAnimator

```

protected void TrackAnimator()
{
    if(animator == null)

```

```

    {
        Debug.LogError("Cannot find Animator on the object,
while TrackAnimator() is being called!!!");
        return;
    }

    animator.speed = 1;

    for (int i = 0; i < animator.layerCount; i++)
    {
        AnimatorStateInfo animatorInfo = animator.GetCurrent
AnimatorStateInfo(i);
        AnimatorClipInfo[] transitionClip = animator.GetNex
tAnimatorClipInfo(i);

        AnimatorTransitionData transitionData;
        bool hasWrittenToBuffer = trackedAnimationTimes[i].
TryReadLastValue(out AnimationValues lastReadValue);

        if (transitionClip.Length > 0)
        {
            AnimatorTransitionInfo unreliableTran = animato
r.GetAnimatorTransitionInfo(0);

            transitionData.isInTransition = true;
            transitionData.targetStateName = Animator.Strin
gToHash(transitionClip[0].clip.name);
            transitionData.transitionLength = unreliableTra
n.duration;
            transitionData.normalizedTimeInTransition = unr
eliableTran.normalizedTime;
            transitionData.durationUnit = unreliableTran.du
rationUnit;
            transitionData.clipLength = transitionClip[0].c
lip.length;
        }
        else
        {

```

```

        transitionData.isInTransition = false;
        transitionData.targetStateName = 0;
        transitionData.normalizedTimeInTransition = 0;
        transitionData.transitionLength = 0;
        transitionData.durationUnit = DurationUnit.Fixed;

        transitionData.clipLength = 0;
    }

    AnimationValues valuesToWrite;
    valuesToWrite.transitionsInfo = transitionData;
    valuesToWrite.animationHash = animatorInfo.shortNameHash;

    if (hasWrittenToBuffer && valuesToWrite.animationHash == lastReadValue.animationHash)
    {
        valuesToWrite.clipLength = lastReadValue.clipLength;

        valuesToWrite.animationStateTime = lastReadValue.animationStateTime + Time.fixedDeltaTime;
    }
    else
    {
        valuesToWrite.clipLength = animatorInfo.length;
        valuesToWrite.animationStateTime = 0;
    }

    trackedAnimationTimes[i].WriteLastValue(valuesToWrite);
}

for (int i = 0; i < animatorParameters.Length; i++)
{
    var val = animatorParameters[i];
    AnimatorCustomParameter par;
    par.parameterHash = val.nameHash;
    par.type = val.type;
}

```



```

        par.value = GetValueFromAnimatorParameter(val);
        trackedAnimationParameters[i].WriteLastValue(par);
    }
}

```

- **Animator 초기 확인:**

- Animator가 존재하지 않으면 에러 메시지를 출력하고 메서드를 종료한다.

- **Animator 속도 설정:**

- Animator의 속도를 1로 설정하여 애니메이션을 정상 속도로 진행시킨다.

- **애니메이션 레이어별 추적:**

- 각 애니메이션 레이어에 대해 현재 상태 정보( `AnimatorStateInfo` )와 다음 클립 정보( `AnimatorClipInfo[]` )를 가져온다.
- 전환 상태( `AnimatorTransitionData` )를 설정한다.
- 이전에 기록된 애니메이션 상태와 현재 상태를 비교하여 애니메이션 시간을 업데이트한다.
- 업데이트된 애니메이션 값을 `trackedAnimationTimes` 버퍼에 기록한다.

- **Animator 파라미터 추적:**

- 모든 Animator 파라미터에 대해 현재 값을 `AnimatorCustomParameter` 구조체에 저장하고, `trackedAnimationParameters` 버퍼에 기록한다.

## RestoreAnimator

```

protected void RestoreAnimator(float seconds)
{
    animator.speed = 0;

    for (int i = 0; i < animator.layerCount; i++)
    {
        AnimationValues readValues = trackedAnimationTimes
[i].ReadFromBuffer(seconds, out bool wasLastAccessedIndexSame);

        if (!readValues.transitionsInfo.isInTransition && w

```

```

asLastAccessedIndexSame)
    return;

    animator.Play(readValues.animationHash, i, readValues.animationStateTime / readValues.clipLength);


    if (readValues.transitionsInfo.isInTransition)
    {
        animator.Update(0);

        if(readValues.transitionsInfo.durationUnit == DurationUnit.Fixed)
            animator.CrossFadeInFixedTime(readValues.transitionsInfo.targetStateName, readValues.transitionsInfo.transitionLength, i, 0, readValues.transitionsInfo.normalizedTimeInTransition);
        else
            animator.CrossFade(readValues.transitionsInfo.targetStateName, readValues.transitionsInfo.transitionLength, i, 0, readValues.transitionsInfo.normalizedTimeInTransition);
    }
}

for (int i = 0; i < trackedAnimationParameters.Count; i++)
{
    AnimatorCustomParameter par = trackedAnimationParameters[i].ReadFromBuffer(seconds);
    ApplyParametersToAnimator(par);
}
}

```

- **Animator 속도 설정:**

- Animator의 속도를  으로 설정하여 애니메이션을 일시 정지한다.

- **애니메이션 레이어별 복원:**

- 각 애니메이션 레이어에 대해 `trackedAnimationTimes` 버퍼에서 지정된 `seconds` 만큼의 과거 애니메이션 상태를 읽어온다.
- 애니메이션 상태를 복원하고, 전환 상태가 있다면 해당 전환을 재생한다.
- **Animator 파라미터 복원:**
  - 모든 Animator 파라미터에 대해 `trackedAnimationParameters` 버퍼에서 지정된 `seconds` 만큼의 과거 파라미터 값을 읽어와 Animator에 적용한다.

## GetValueFromAnimatorParameter

```
float GetValueFromAnimatorParameter(AnimatorControllerParameter par)
{
    switch (par.type)
    {
        case AnimatorControllerParameterType.Trigger:
        case AnimatorControllerParameterType.Bool:
            return animator.GetBool(par.name) ? 1 : 0;
        case AnimatorControllerParameterType.Float:
            return animator.GetFloat(par.name);
        case AnimatorControllerParameterType.Int:
            return animator.GetInteger(par.name);
    }
    return 0;
}
```

- **Animator 파라미터 값 가져오기:**
  - 파라미터 타입에 따라 현재 값을 가져와 `float` 로 반환한다.
  - `Trigger` 와 `Bool` 타입은 `bool` 값을 `1` 또는 `0` 으로 변환하여 반환한다.
  - `Float` 과 `Int` 타입은 해당 값을 직접 반환한다.

## ApplyParametersToAnimator

```

public void ApplyParametersToAnimator(AnimatorCustomParameter parameter)
{
    switch (parameter.type)
    {
        case AnimatorControllerParameterType.Bool:
            animator.SetBool(parameter.parameterHash, (parameter.value == 0) ? false : true);
            break;

        case AnimatorControllerParameterType.Float:
            animator.SetFloat(parameter.parameterHash, (float)parameter.value);
            break;

        case AnimatorControllerParameterType.Int:
            animator.SetInteger(parameter.parameterHash, (int)parameter.value);
            break;
        case AnimatorControllerParameterType.Trigger:
            if (parameter.value != 0)
                animator.SetTrigger(parameter.parameterHash);
            else
                animator.ResetTrigger(parameter.parameterHash);
            break;
    }
}

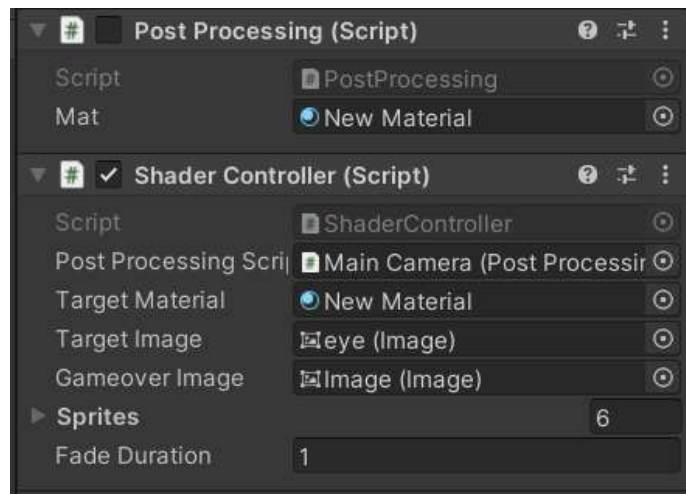
```

#### • Animator 파라미터 적용:

- `AnimatorCustomParameter` 구조체에 저장된 파라미터 타입과 값을 기반으로 Animator에 파라미터를 설정한다.
- `Bool`, `Float`, `Int` 타입은 해당 값을 직접 설정한다.
- `Trigger` 타입은 `value` 가 `0` 이 아니면 `SetTrigger` 를 호출하고, `0` 이면 `ResetTrigger` 를 호출한다.

이후에 있는 코드는 오디오와 파티클에 대한 코드로 실제 게임에서는 사용하지 않아 보고서에는 작성하지 않았다.

## 6-2 보조 기능 : 1bit dithering



**Return of the Obra Dinn**이라는 게임에서 영감을 얻어 단색 컬러의 그래픽을 구현해 보았다. 이 렌더링을 구현하기 위해 세 가지 주요 특징을 정리했다:

1. 외각선이 보인다.
2. 그 외각선이 어두운 곳에서는 하얗게 보인다.
3. 음영이 진 부분에 스크린톤 느낌의 패턴이 보여야 한다.

이 보고서에서는 이 세 가지 특징을 구현하는 과정을 간단하게 설명하고, 전체 코드를 상세히 설명하겠다.

### 1. 외각선 구현

외각선 구현은 **덱스 텍스처(Depth Texture)**를 사용하여 이루어진다. 외각선을 효과적으로 표현하기 위해 다음과 같은 단계를 거쳤다:

1. **덱스 텍스처 준비:**
  - 현재 화면의 덱스 정보를 담은 텍스처를 하나 준비한다.
  - 이 덱스 텍스처를 1px만큼 옆으로 이동시켜 같은 덱스 텍스처를 겹쳐놓는다.
2. **차이 계산:**

- 두 개의 텍스처 간의 차이를 계산한다.
- 깊이 차이가 동일하면 외각선을 검은색으로, 깊이 차이가 클수록 선명한 하얀 외각선이 나타나도록 한다.

### 3. 외각선 합산:

- 상하좌우의 차이값을 모두 더해주면, 깊이값을 기반으로 한 외각선이 완성된다.

## 외각선 셰이더 코드

외각선을 구현하기 위한 셰이더의 주요 부분은 다음과 같다:

```
Shader "Custom/OutlineShader"
{
    Properties
    {
        _MainTex ("Base (RGB)", 2D) = "white" {}
        _DepthMul ("Depth Multiplier", Float) = 1.0
        _DepthPow ("Depth Power", Float) = 1.0
        _NormalMul ("Normal Multiplier", Float) = 1.0
        _NormalPow ("Normal Power", Float) = 1.0
        _OutlineLimit ("Outline Threshold", Float) = 0.1
        _ToneTex ("Tone Texture", 2D) = "white" {}
        _ToneLimit ("Tone Threshold", Float) = 0.5
        _ForeCol ("Foreground Color", Color) = (1,1,1,1)
        _BackCol ("Background Color", Color) = (0,0,0,1)
    }
    SubShader
    {
        // 셰이더 구현 내용
    }
}
```

### Properties 설명:

- **\_MainTex**: 기본 텍스처.
- **\_DepthMul**, **\_DepthPow**: 깊이를 기반으로 외각선을 계산할 때 사용하는 곱셈 및 지수 값.
- **\_NormalMul**, **\_NormalPow**: 노멀 벡터 차이를 기반으로 외각선을 계산할 때 사용하는 곱셈 및 지수 값.

- `_OutlineLimit` : 외각선 색상 변경에 사용하는 임계값.
- `_ToneTex` : 명암 단계별 색상을 결정하는 텍스처.
- `_ToneLimit` : 톤의 분할 기준값. 이를 통해 스크린톤 포스트 프로세싱 수치를 조절할 수 있다.
- `_ForeCol`, `_BackCol` : 전경과 배경색. 본 게임에서는 `_BackCol` 은 검정색으로 고정하고 `_ForeCol` 만 조정했다.

## 외각선 구현 과정

### 1. 뎀스 텍스처 준비 및 이동:

```
float4 depth1 = tex2D(_DepthTex, i.uv);
float4 depth2 = tex2D(_DepthTex, i.uv + float2(1.0/_ScreenParams.x, 0))
```

- 현재 픽셀의 깊이값(`depth1`)과 1px 옆 픽셀의 깊이값(`depth2`)을 샘플링한다.

### 2. 깊이 차이 계산:

```
float depthDifference = abs(depth1 - depth2);
float outline = depthDifference > _OutlineLimit ? 1.0 : 0.0;
```

- 두 깊이값의 절대 차이를 계산하고, 임계값(`_OutlineLimit`)을 초과하면 외각선을 하얀색(1.0)으로, 그렇지 않으면 검은색(0.0)으로 설정한다.

### 3. 외각선 합산:

```
float finalOutline = outline + outlineUp + outlineDown +
outlineLeft + outlineRight;
finalOutline = saturate(finalOutline);
```

- 상하좌우의 외각선 값을 모두 더한 후, `saturate` 함수를 사용하여 값을 [0,1] 범위로 제한한다.

### 4. 외각선 적용:

```
fixed4 color = lerp(_BackCol, _ForeCol, finalOutline);
return color;
```

- 최종 외각선 값을 기반으로 전경색과 배경색을 보간하여 최종 색상을 결정한다.

## 2. 스크린톤 음영 구현

스크린톤 음영은 **Return of the Obra Dinn** 제작자의 블로그에서 영감을 받아 구현하였다. 이 기능은 오브젝트에 직접적으로 셰이더를 적용하는 것이 아니라, 화면 전체에 포스트 프로세싱으로 적용해야 한다. 이를 위해 다음과 같은 단계를 거쳤다:

### 1. 스크립트 설정:

- 셰이더를 적용하기 위한 스크립트를 **Main Camera**에 추가한다.

### 2. OnRenderImage 함수 구현:

- **OnRenderImage** 함수는 렌더링된 모든 화면을 특정 매터리얼(**mat**)로 덮어준다.
- 이를 통해 전체 화면에 스크린톤 효과를 적용할 수 있다.

## 스크린톤 셰이더 스크립트

```
using UnityEngine;

[ExecuteInEditMode]
public class ScreenTonePostProcessing : MonoBehaviour
{
    public Material mat;

    void OnRenderImage(RenderTexture src, RenderTexture dest)
    {
        {
            if(mat != null)
            {
                Graphics.Blit(src, dest, mat);
            }
            else
            {
                Graphics.Blit(src, dest);
            }
        }
    }
}
```



```

    }
}
}

```

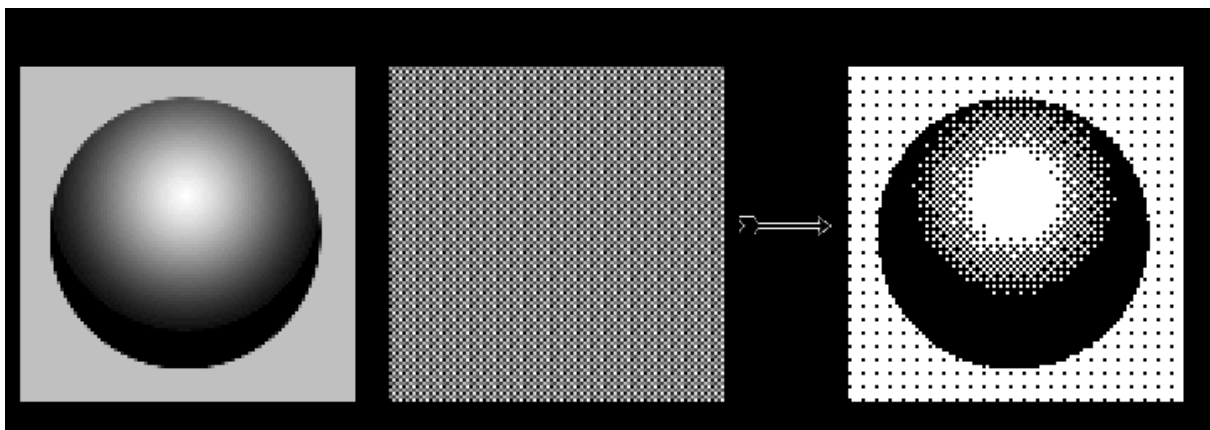
- **mat**: 스크린톤 효과를 적용하기 위한 매터리얼.
- **OnRenderImage**: 모든 화면이 렌더링된 후, **mat** 을 사용하여 화면 전체를 덮어씌운다.

## 스크린톤 텍스처 및 픽셀 크기 조정

스크린톤 이미지는 게임 제작자의 블로그에서 제공한 이미지를 사용하였다. 픽셀의 크기를 정사각형으로 만들기 위해 다음과 같은 코드를 추가했다:

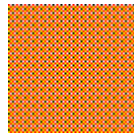
```
float4 tone = tex2D(_ToneTex, (i.uv * _ScreenParams.xy * _ToneTex_TexelSize.xy));
```

- **tex2D**: 텍스처 샘플링 함수.
- **\_ToneTex**: 스크린톤 텍스처.
- **\_ScreenParams**: 화면 해상도 정보.
- **\_ToneTex\_TexelSize**: 텍스처의 텍셀 크기.
- 이를 통해 픽셀의 크기를 정사각형으로 맞추어 스크린톤 패턴이 일관되게 보이도록 한다.



(출처: <https://forums.tigsource.com/index.php?topic=40832.780> )

마지막으로 스크린톤 음영을 설명하자면, 위 그림은 게임 "Return of the Obra Dinn"의 제작자의 블로그에서 나온 구현 과정이다. 셰이더를 오브젝트에 적용하는게 아닌 화면 전체에 적용시켜야 하므로 다음 스크립트를 main camera에 넣어주었다.



(스크린톤 텍스처)

스크린톤 이미지는 다음 이미지를 사용하였다. 이때 픽셀의 크기를 정사각형으로 만들기 위해 `float4 tone = tex2D(_ToneTex, (i.uv*_ScreenParams.xy*_ToneTex_TexelSize.xy));` 라는 코드를 추가하였다.

이제 전체 코드에 대해 설명하겠다. 먼저 셰이더의 Properties는 다음과 같다.

```
Shader "Hidden/Subshader2"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        _DepthMul("Depth Outline Multiplier", Range(0,4)) = 1
        _DepthPow("Depth Outline Bias", Range(1,4)) = 1
        _NormalMul("Normal Outline Multiplier", Range(0,4)) = 1
        _NormalPow("Normal Outline Bias", Range(1,4)) = 1
        _OutlineLimt("OutlineColorLimit",range(0,1)) = 0.5
        _GrayScalePow("GrayScalePow",float) = 1
        _ToneTex("ToneTex",2D) = "white"{}
        _ToneLimit("ToneLimit",Vector) = (0,0,0,0)
        _ForeCol("ForegroundColor",Color) = (1,1,1,1)
        _BackCol("BackgroundColor",Color) = (0,0,0,1)
    }
}
```

\_MainTex : 기본 텍스처

\_DepthMul, DepthPow: 깊이를 기반으로 외각선을 계산할 때 사용하는 곱셈 및 지수 값

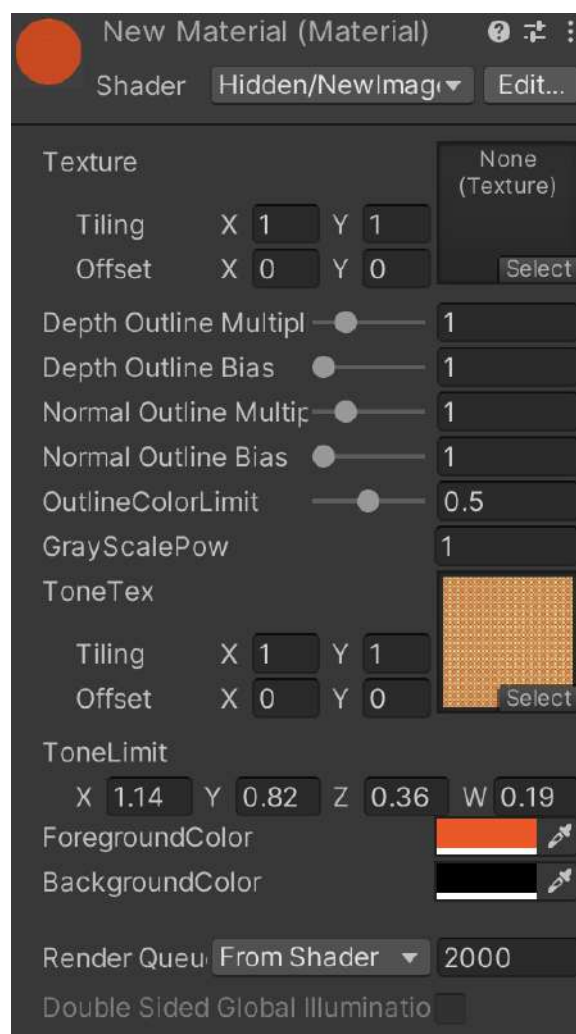
\_NormalMul, \_NormalPow, 노멀 벡터 차이를 기반으로 외각선을 계산할 때 사용하는 곱셈 및 지수 값

\_OutlineLimit: 외각선 색상 변경에 사용하는 임계값

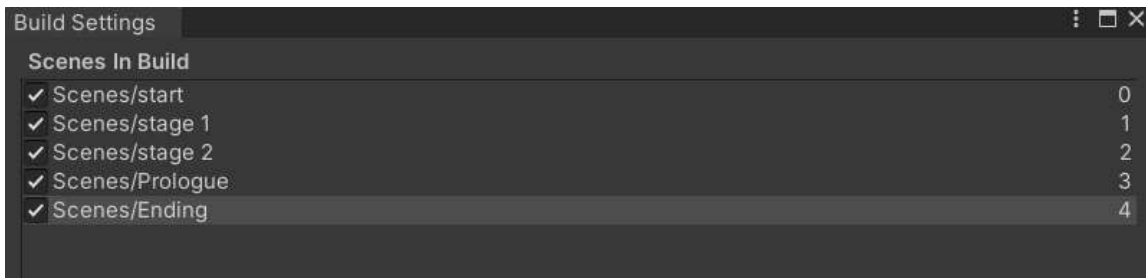
\_ToneTex: 명암 단계별 색상을 결정,

\_ToneLimit: 톤의 분할 기준값, \_Tonelimit을 통해 스크린톤 포스트 프로세싱 수치를 조절할 수 있게 하였다.

\_ForeCol, \_BackCol: 전경과 배경색이다. 본 게임에서는 \_BackCol은 검정색으로 고정하고 \_Forecol만 조정하면서 개발하였다.



## 세이브&로드



혹시 빌드파일에 빌드세팅 순서 망가져있다면 패키지파일에선 이렇게 세팅해주고 돌려주시기 바랍니다.

세이브와 로드는 스테이지가 몇 개 없기에 json보다는 간편한 방식인 txt파일을 이용한 방법으로 구성하였다. 총 두 개의 스크립트를 사용하며, saveload와 Scene Data Saver를 사용한다.

후에 설명할 Scene Data Saver 스크립트를 통해 txt파일에 스테이지 1이면 1, 스테이지 2면 2가 적히도록 하였으며, 이를 통해 로드를 하도록 구성하였다. 일단 적힌 숫자의 공백 및 개행문자를 제거해주고, int.TryParse를 통해 stageData를 정수로 변환해준다. 그 다음에 저장된 정수에 맞는 스테이지를 불러오는 방식으로 로드를 한다.

이는 스테이지에 들어갈때 생성된 player에게 스크립트를 넣어 Start 메시드가 시작할때 작동하도록 하였다.

```
if (dataFile != null)
{
    string stageData = dataFile.text.Trim();
    int stageNumber;
    if (int.TryParse(stageData, out stageNumber))
    {
        string sceneName = "stage " + stageNumber;
        Debug.Log("Loading Scene: " + sceneName);

        SceneManager.LoadScene(sceneName);
    }
    else
    {
        Debug.LogError("Invalid stage data in file.");
    }
}
else
{
    Debug.LogError("Data file is not assigned in the inspector.");
}
```

scenedatasaver 스크립트에서 씬들을 빌드세팅을 해준 순서대로 인덱스에 저장을 해준 후 data.txt가 들어있는 파일의 경로를 추적하여 data.txt를 찾은 후 여기에 씬의 인덱스 번호를 입력해주는 방식으로 세이브를 진행한다.

```

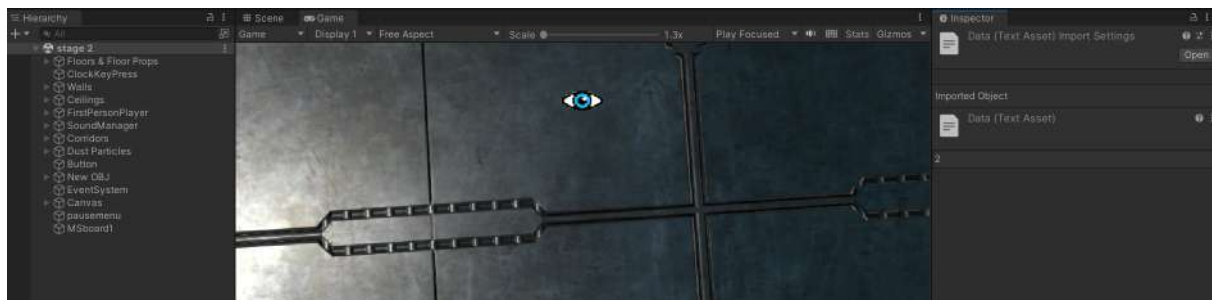
public TextAsset dataFile;

private void Start()
{
    if (dataFile != null)
    {
        int sceneNumber = SceneManager.GetActiveScene().buildIndex;

        string filePath = Path.Combine(Application.persistentDataPath, dataFile.name);
        Debug.Log(filePath);

        File.WriteAllText(filePath, sceneNumber.ToString());
        Debug.Log("Scene " + sceneNumber + " saved to " + filePath);
    }
    else
    {
        Debug.LogError("Data file not assigned in the inspector.");
    }
}

```



다음과 같이 스테이지에 진입하면 그 스테이지의 값으로 data.txt가 변경된다.

## 7 캐릭터 설계

	상세 설명	특징
관리자	플레이어가 조작해야하는 캐릭터. 관리자 권한을 통해 에러 오브젝트들의 크기 및 위치를 조정하여 출구로 향하는 길을 개척함	관리자 본인임으로 본인이 사용하는 Time Rewind와 Space Control의 영향을 받지 않음.
Error Object	에러로 인해 침식된 오브젝트들. 플레이어가 옮겨야하는 물체들이다. 인게임에서는 Layer를 Targetable로 설정해줌으로써 관리자권한의 타겟임을 나타내주며 구분하였다.	Time Rewind와 Space Control의 영향으로 크기가 변하고 위치가 이동될 수 있음.

### 7-1 주인공 캐릭터

주인공(관리자)는 자신의 관리자 권한을 통해 망가진 게임 레벨을 고쳐나간다.

사용가능한 권한으로는 마우스 왼쪽 클릭을 통해 물체를 들어서 원근법에 따라 크기를 조정하는 Space Control, T키를 눌러 시간을 되돌리는 Time Rewind가 있다.

### 7-2 Error Object

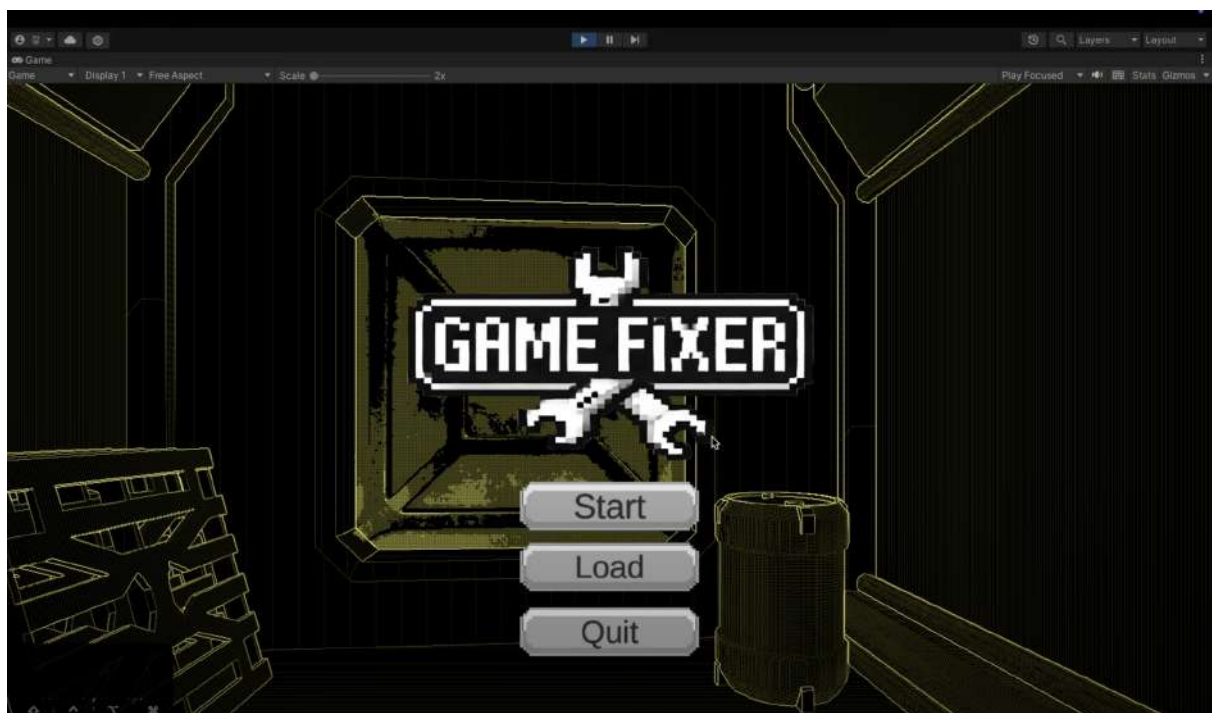
Error Object들은 플레이어가 옮겨서 길을 이어야할 물체로, targetable이라는 Layer를 만들어서 구분시켜두었다.

## 8.화면구성

### 8-1 타이틀화면

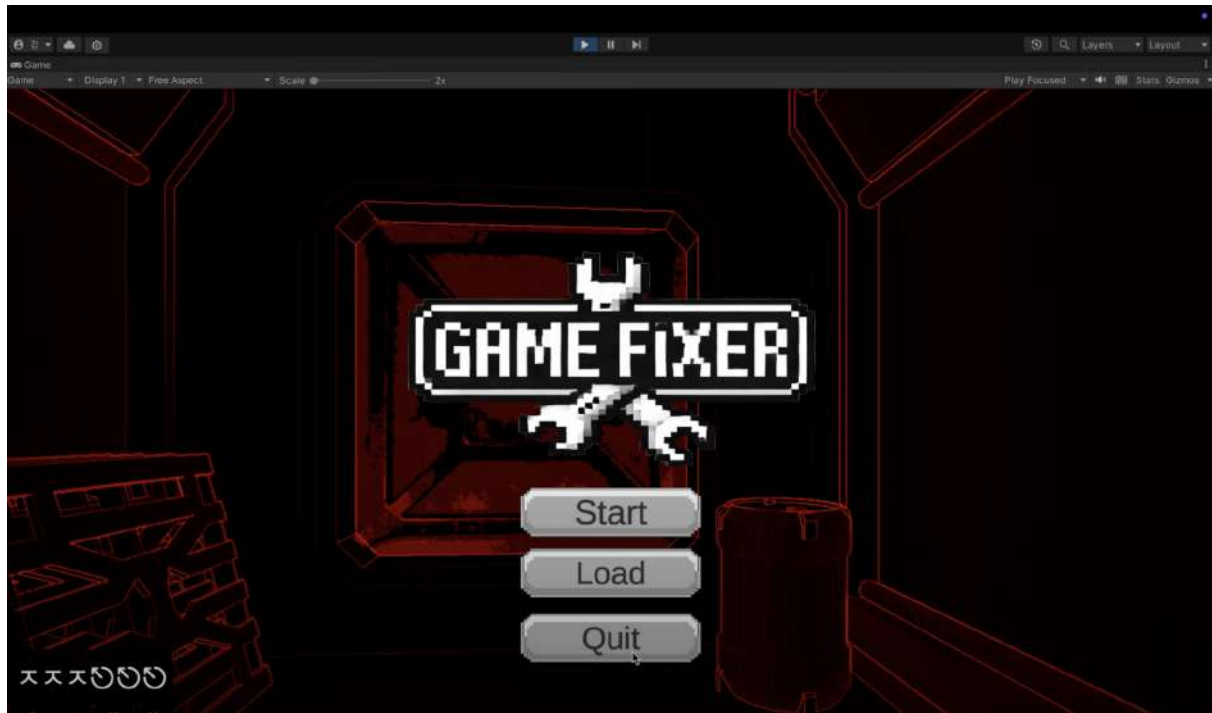


GPT로 만든 타이틀 화면에 쓸 게임 로고이다.



타이틀 화면에는 게임을 시작하는 Start버튼,세이브했던 곳으로 돌아가는 Load버튼,게임을 종료하는 Quit버튼이 있다. 또한 타이틀 배경에 1 bit 디더링을 적용하고 post

processing에 쓰인 material의 rgb값을 변경하여 색깔이 계속 변하도록 하였다. rgb값 변경 방식은 r,g,b가 순서대로 0에서부터 255까지 증가하고 다시 순서대로 255에서 0으로 감소하게 했다.





```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ColorChaging : MonoBehaviour
{
    public Material postProcessingMaterial; // 렌더링 매터리얼
    public float colorChangeSpeed = 1.0f; // 색상 변경 속도

    private float r = 0, g = 0, b = 0; // RGB 값
    private int phase = 0; // 현재 단계 (0-5)
    private bool isIncreasing = true; // 값이 증가하는 동안지 여부

    void Update()
    {
        if (postProcessingMaterial == null) return;

        float changeAmount = Time.deltaTime * colorChangeSpeed * 255;

        // 단계에 따라 RGB 값을 변경
        switch (phase)
        {
            case 0: // R 증가
                r += changeAmount;
                if (r >= 255)
                {
                    r = 255;
                    phase++;
                }
                break;

            case 1: // G 증가
                g += changeAmount;
                if (g >= 255)
                {
                    g = 255;
                    phase++;
                }
                break;

            case 2: // B 증가
                b += changeAmount;
                if (b >= 255)
                {
                    b = 255;
                    phase++;
                    isIncreasing = false; // 감소로 전환
                }
                break;

            case 3: // R 감소
                r -= changeAmount;
                if (r <= 0)
                {
                    r = 0;
                    phase++;
                }
                break;

            case 4: // G 감소
                g -= changeAmount;
                if (g <= 0)
                {
                    g = 0;
                    phase++;
                }
                break;

            case 5: // B 감소
                b -= changeAmount;
                if (b <= 0)
                {
                    b = 0;
                    phase = 0; // 다시 R 증가로 전환
                    isIncreasing = true;
                }
                break;
        }

        // 매 프레임마다 색상 변경
        postProcessingMaterial.SetColor("_ForeColor", new Color(r / 255f, g / 255f, b / 255f, 1));
    }
}

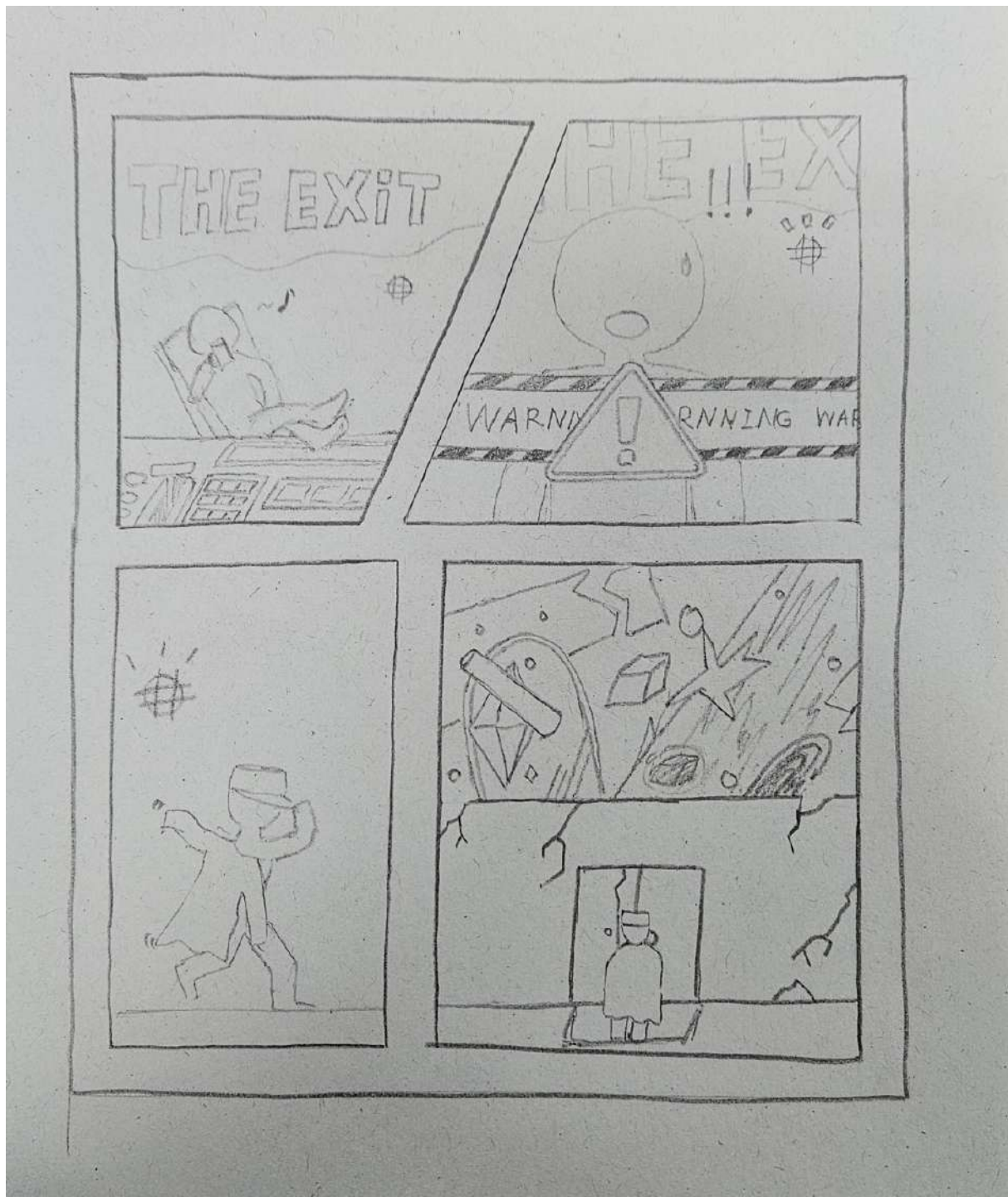
```

(ColorChange.cs)

## 8-2 프롤로그화면

생성형 AI를 통해 만들었으며, 게임의 관리자인 주인공이 평화롭게 쉬고있다가, 갑자기 게임에 생긴 오류로 인해 급하게 게임을 고치러 뛰어나가는 스토리를 표현하였다. 아래는 제작 단계에서 기획했던 콘티이다.



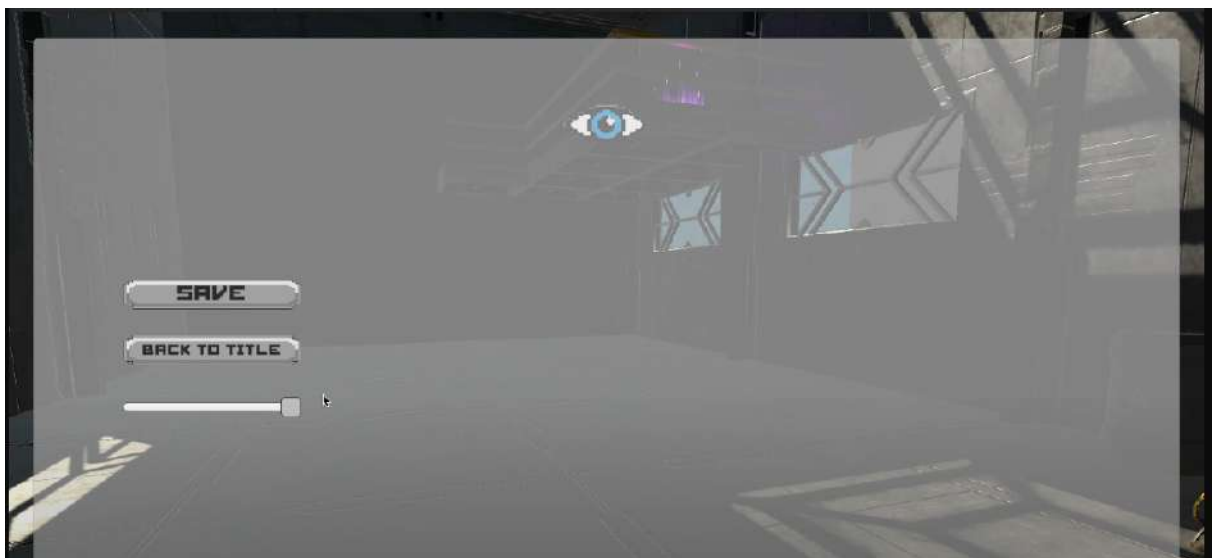




### 8-3 스테이지 클리어



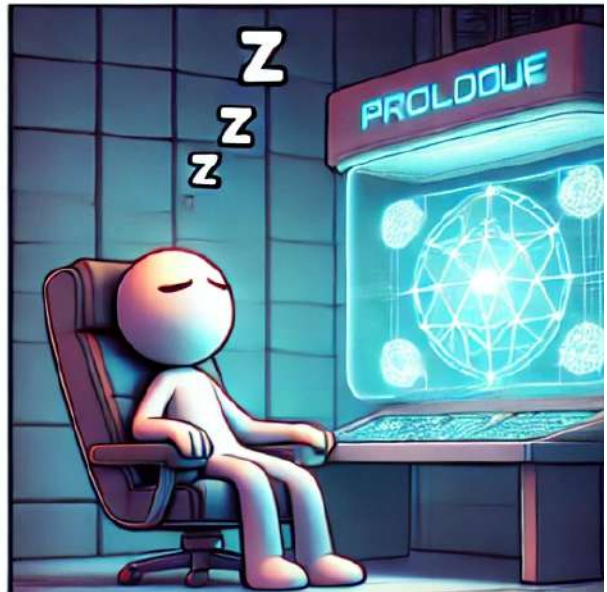
### 8-4 ESC화면

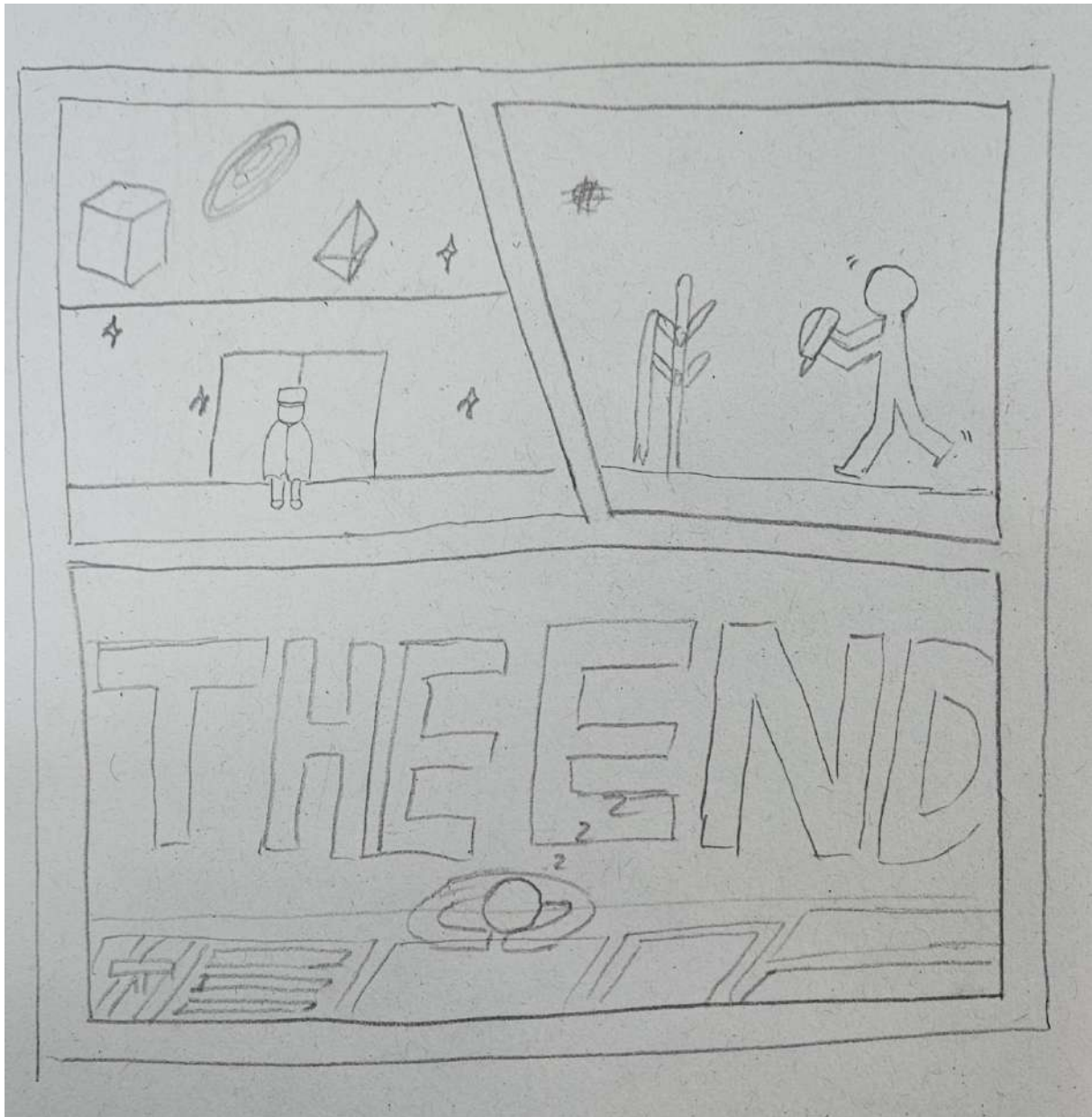


esc화면에는 save 버튼,back to title 버튼, 음량 조절을 위한 sidebar가 들어가있다.

### 8-5 엔딩화면

생성형 AI를 통해 만들었으며, 스테이지를 다 고친 관리자가 다시 자신의 자리로 돌아와 커피 한 잔의 여유를 가지며 다시 평화로운 일상으로 돌아가는 것을 표현하였다. 아래는 제작 단계에서 기획한 콘티이다.





## 8-6 게임오버화면



## 9 인공지능

-사용되지 않음

## 10 인터페이스

### 10-1. 조작 인터페이스 설명

이 게임에서 인터페이스는 현재 Time Rewind를 얼마나 썼는지 확인할 수 있는 눈 모양 인터페이스 하나 뿐이다. 모양이 눈인 이유는, 스토리상 관리자는 본인의 눈에 깃들어있는 Time Rewind힘을 사용하는 것이기에 힘을 쓰면 쓸 수록 눈에 과부화가 온다는 것을 표현하기 위해 눈 모양으로 나타내었다.

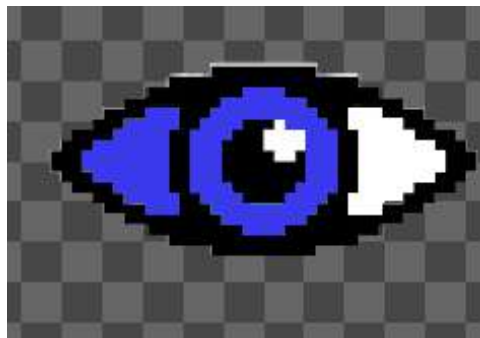


이는 힘을 사용하기 전의 기본 상태 및 phase1에서의 인터페이스이고, phase2, 즉 화면이 노랗게 되면

그만큼 시간이 지났음을 나타내기 위해 눈에 색을 채워 게이지 효과를 내줌과 동시에 화면과 색을 맞추어주었다.

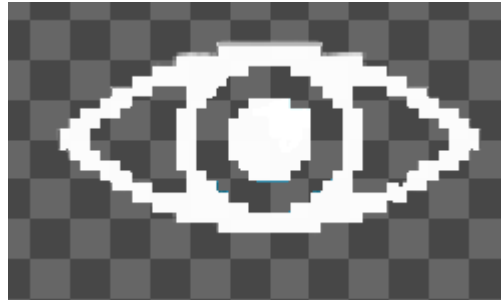


이후에도 차례대로 색이 변하며 게이지가 차오르는 것을 표현해주었다.





그리고 마지막으로 힘을 다 써 화면이 까만색이 되면 눈 게이지가 까만색으로다 차게되며, 이를 표시하기 위해 안쪽은 까만색 테두리는 흰색을 통해 힘을 다 써버린 눈을 표현하였다. 배경이 까맣기에 눈의 안쪽은 자동으로 까맣게 차게된다.



## 11 스테이지 구성

스테이지 설계는 5. 맵 설계에서 모두 함께 설명.

## 12 그래픽 데이터 리스트

### 12-1 그래픽 데이터

맵 제작에 사용된 에셋과 도착지점 포탈을 사용하였다.

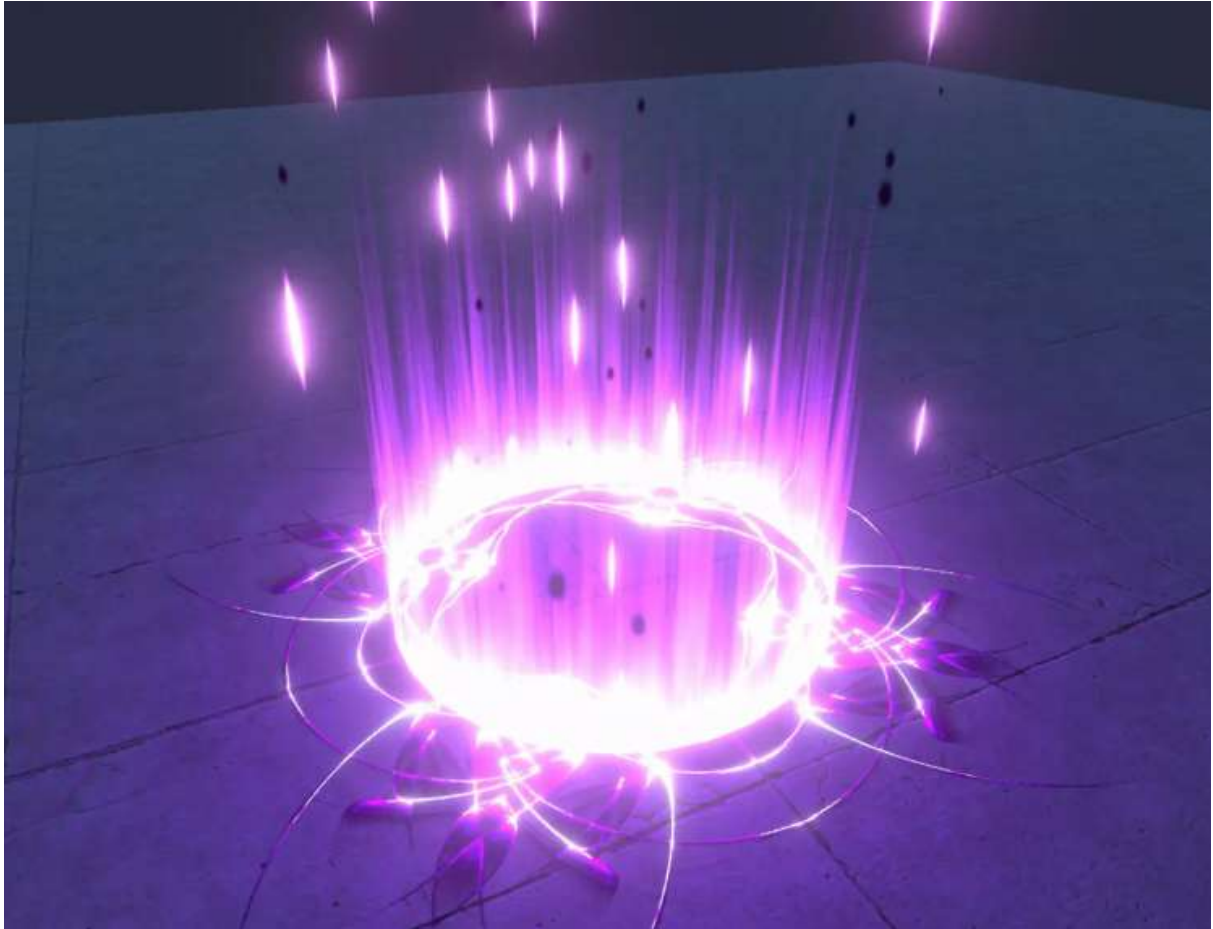
### 12-2 사용한 에셋

맵 <https://assetstore.unity.com/packages/3d/environments/sci-fi/sci-fi-construction-kit-modular-159280>



포탈

<https://assetstore.unity.com/packages/vfx/particles/spells/magic-effects-free-247933>



버튼 ui (직접 제작)



### 13 사운드 데이터 리스트

#### 13-1 사운드 데이터

기본적으로 플레이어가 움직일 때 나오는 발걸음 소리, 타이틀 음악, 배경음악, Time Rewind사운드가 있다.

#### 13-2 사용한 에셋

타이틀 음악, 배경음악 및 발소리 효과음 : 맵 에셋에 들어있는 자체 음향

<https://assetstore.unity.com/packages/3d/environments/sci-fi/sci-fi-construction-kit-modular-159280>

Time Rewind사운드 : <https://mixkit.co/free-sound-effects/rewind/> - tape rewind cinematic transition

### 14 제작진 역할 분담

#### 14-1 역할 분담

이강민 : SpaceControl기능 개발 및 stage1, 세이브 기능 개발, 최종 보고서 작성

전지훈 : 기획 총괄, TimeRewind기능 개발 및 stage2제작, 최종 보고서 작성

강동현 : 1bit dithering 개발 및 UI개발, 최종 보고서 작성

### 15 개발 시 애로사항 및 느낀 점

## 15-1 개발 시 애로사항

개발 시 느꼈던 애로사항으로 가장 먼저 들었던건 구현 가능성에 관한 의문이었다. 기획은 당차게 시작하여 그럴듯하게 진행되었지만, 막상 개발에 돌입해보니, 이걸 어떻게 구현하지? 라는 생각이 들었던 것 같다. 하지만 유니티의 장점이 바로 이미 많이 개발된 라이브러리들이 인터넷이 많다는 점이었기에, 이를 이용해 깃허브에 올라가있는 수많은 관련 코드 자료들을 보며 개조한 끝에 기능 구현에 성공하였다. UI에 관한 고민도 많이 존재했었다. 팀원들 중에 UI개발을 할 줄 아는 사람이 없었기에, 막막했지만, unity에서 제공해주는 Textmeshpro를 이용해서 어떻게든 버튼과같은 UI를 만들고, 생성형 AI 또한 이용해서 장면들을 제작했지만, 확실히 아쉬운 부분이 많이 존재한다는 것을 느낄 수 있었다. 가장 큰 문제점은 아마 시간..이었을 것이다. 한 달 정도라는 그렇게 짧지는 않은 기간이었지만 조원들이 모두 너무 바쁜 스케줄을 지냈던 상황이라 개발 시간이 턱없이 부족해지는 상황에 처하기도 하였다. 또한 Unity에서 데이터를 저장해야하는데 이를 할 수가 없어서 데이터 베이스를 사용하는 대신 텍스트파일에 읽고 쓰는 형식을 사용했다. 배우지는 않은 내용이라 최선의 선택을 한것이라고 생각하나 기회가 된다면 보통 게임에서 어떤 데이터를 어떻게 DB에 저장하는지도 알고 이를 적용시키고 싶다.

## 15-2 느낀 점

전지훈 - 이번 프로젝트에 대해서 느꼈던 점은 너무 학점에 몰두해 안정적인 게임을 만든다는 방향보다는 만들 때 흥미를 느끼고 재밌을 새로운 "도전"을 해보는 방향으로 게임을 제작해보고 싶었다. 개발보다는 기획 직군을 진로로 생각하고 있는 나에게 내 기획력을 시험해볼 차례였다고 생각하기도 했다. 다행이 이런 도전적 기획에 팀원들도 흔쾌히 동의를 하여 독창성에 집중된 기획에 적극적으로 나서게 되었고, 그 결과적으로 원근법을 통한 물체를 옮기고, 잘못 옮겼을 경우 시간을 되돌린다는 나름 괜찮은 주제로 결정하게 되었다. 이에 눈을 즐겁게 해주는 1bit Dithering까지 들어갔으니, 만족스러운 기획을 했다고는 생각했다. 하지만 문제는 개발이었다. 기본적으로 Space Control과 Time Rewind의 구현. 이는 생각보다 머리를 아프게 만들었고, 내가 맡았던 부분인 Time Rewind는 이를 어떻게 구현해야할지만 으로일주일 가까이 생각했을 정도로 애를 먹었던 기억이 있다. 인터넷에서 여러 소스를 얻어 개발을 본격적으로 시작했을 때도 이 소스들을 우리의 입맛에 맞게 개조해야했기에 머리가 아팠지만 막상 완성하고 우리의 게임에서 기능이 제대로 돌아가는 것을 확인했을 때 오히려 들어 가장 큰 희열을 느낄 수 있었다. 기획부터 시작해서 정말 많은 고민이 들어가서 특히 완성에서 더욱 기쁨을 느낀 것 같다. Critical Error시스템, Time Rewind기능을 이용한 더 많은 트랩들, TimeRewind Abstract를 보면 물체의 위치 말고도 애니메이션이나 파티클 등등 무궁무진하게 되돌릴 수 있게 구현을 해놓았지만, 개발 시간이 부족하기도 하였고 짧은 발표 시간 안에 가장 효율적으로 메인 기능들을 보여주기 위해 전부 다 폐기한게 정말 아쉬웠다. 다음에 이 게임을 develop 할 기회가 생기면 꼭 개발 중 폐기한 기능들을 모두 넣어보고싶다는 생각을 했다.

이강민 - 개인적으로 게임개발을 할정도로 게임개발쪽에 관심이 많고 게임개발자를 꿈으로 가지고 있다. 이번에 게임프로그래밍 강의를 수강하면서 평소 같이 의견을 나누던 친구들과 함께 한학기 동안 게임을 만들 수 있어서 정말 좋은 경험이 되었던것 같다. 이번에는 늘 있는 식상한 양산형 게임의 장르에서 벗어나 새로운 게임의 형식에 도전해보고 싶었다. 오브라딘 호의 1bit dithering 그리고 superliminal의 착시 효과가 이야기에 나왔고, 이를 결합한 퍼즐 게임을 만들게 되었다. 평소 게임을 만들때와는 다른 새로운 개념들을 많이 알게 되었고, 이번에 셰이더 프로그래밍과 각종 복잡한 기믹들을 어떻게 구현하는지 많이 배웠다. 나를 포함한 팀원들이 많이 바빠 힘을 많이 못썼던것은 사실이지만 그럼에도 좋은 결과물이 나온것 같아서 좋다. 몇가지 아쉬운 부분이 많지만, 기회가 된다면 나중에 이 게임을 좀더 발전시켜서 좋은 게임으로 만들어 보고싶다. 이번 수업을 들으면서 평소 독학만 했던 유니티를 체계적으로 배우고 이를 토대로 직접 게임을 만들어보는 좋은 경험을 한것 같다. 다른 팀들도 엄청난 퀄리티의 게임들을 많이 만들어 주어서 많이 배운듯 하다.

강동현 - 언리언 엔진5와 같은 게임 엔진들이 발전하면서 최근에 나온 게임들의 모델링 퀄리티가 날이 갈수록 올라가고 있다. 하지만 예전부터 AAA게임들 보단 인디게임에도 많은 관심을 가지게 되었다. 그 중에서도 "Return of the obradinn" 을 접하면서 어떻게 이런 그래픽을 혼자서 구현할 수 있었을까 엄청 놀랐던것 같다. 게임프로그래밍 수업을 듣기 전에도 1bit dithering을 구현해보려고 실패하기는 했지만 "Return of the obradinn"의 제작자인 루카스 포프의 블로그(<https://forums.tigsource.com/index.php?topic=40832.60>)도 찾아보기도 했다.이번 학기에 게임프로그래밍 수업을 듣게된 이유도 1 bit dithering을 제대로 구현해보기 위함이었다. 이번 프로젝트에 1 bit dithering을 약간 억지로 밀어붙인거 같기도 한데 생각보다 결과물이 잘 나와줘서 엄청 기뻐던 것 같다. 몇년전과는 다르게 포스트 프로세싱 관련 자료들이 많이 있어서 구현하는데도 크게 어렵진 않았다. 이번 학기에 팀원들이 전부 바쁜 나머지 게임을 제대로 만든 날이 2일정도 밖에 없었는데 만약 시간이 더 있었다면 좀 더 완성도 있게 만들 수 있지 않았을까 하는 아쉬움도 있다. 하지만 전체적으로 모두 고생해서 만족스러운 결과물이 나온것 같아서 좋았다.