종합설계 프로젝트 수행 보고서

프로젝트명	“3D 슈팅 멀티 PC 게임 구현”
팀번호	S4-4
문서제목	수행계획서(O) 2차발표 중간보고서(O) 3차발표 중간보고서(O) 4차발표 중간보고서(O) 최종결과보고서( )

2021.06.03

팀원 : 오동호 (팀장)
강성준 (팀원)
이원찬 (팀원)

지도교수 : 이 보 경 교수

문서 수정 내역

작성일	대표작성자	버전(Revision)	수정내용	
2021.12.30	오동호	1.0	요약계획서 작성	최초 작성
2022.03.01	이원찬	2.0	2차 보고서 작성	수행보고서 작성
2022.03.05.	오동호	2.01		내용 최신화
2022.04.20	강성준	2.02	3차 보고서 작성	프로토타입 내용 추가
2022.06.03.	오동호	2.03	4차 보고서 작성	시험/테스트, 코딩, 데모 작성

문서 구성

진행단계	프로젝트 계획서 발표	중간발표1 (3월)	중간발표2 (5월)	학기말발표 (6월)	최종발표 (10월)
기본양식	계획서 양식	계획서 양식	계획서 양식	계획서 양식	계획서 양식
포함되는 내용	Ⅰ. 서론 (1~6) Ⅱ. 본론 (1~3) 참고자료	Ⅰ. 서론 (1~6) Ⅱ. 본론 (1~4) 참고자료	Ⅰ. 서론 (1~6) Ⅱ. 본론 (1~5) 참고자료	Ⅰ. 서론 (1~6) Ⅱ. 본론 (1~7) 참고자료	Ⅰ Ⅱ Ⅲ

이 문서는 한국산업기술대학교 컴퓨터공학부의 “종합설계”교과목에서 프로젝트“3D 슈팅 멀티 PC 게임 구현”을 수행하는 (S4-4, 오동호, 강성준, 이원찬)들이 작성한 것으로 사용하기 위해서는 팀원들의 허락이 필요합니다.

목 차

Ⅰ. 서론

1. 작품선정 배경 및 필요성

"트리플 A 타이틀"이라고 부르는 게임은 대작이라 불릴 만큼 수많은 컨텐츠와 즐거움을 제공해준다. 하지만 대작을 만들기 위해선 개발 비용이 천문학적인 제작비가 들어간다. 성공이 보장되지 않는 게임에 개발 비용을 투자하는 것은 수지타산에 맞지 않게 보일 수 있다. 계속 이어지는 흐름으로 인해 출시되는 대작들은 성공을 위해 너무나 많은 콘텐츠를 제공하기 때문에 간단히 즐길 라이트 유저들이 대상으로 잡지 않고 숙달된 헤비 유저들을 대상으로 개발한다.
라이트 유저들이 즐기기엔 적합하지 않다고 판단하여, 남녀노소 간편하게 즐길 수 있는 게임을 개발하고자 한다. 본 프로젝트는 라이트 유저들을 대상으로 잡은 슈팅 게임을 제공한다.

2. 기존 연구/기술동향 분석

< 표 1 > 라이트 유저를 대상으로 하는 비교군 게임을 보면 1시간이란 짧은 플레이 타임과 1인 단독 플레이를 중심으로 개발된 속도감 있고 화려한 전투를 즐길 수 있는 게임이다.
본 프로젝트에서 개발하고자 하는 '3D 슈팅 멀티 PC 게임 구현'은 다른 게임들과 비교하였을 때, 카툰 렌더링과 실사 그래픽을 융합된 그래픽을 제공하는 점, 1인~4인 멀티 플레이가 제공되는 점, 물리 효과를 부여하여 스테이지가 동적으로 변하여 상황에 따라 전략 전술을 바꿔서 사용할 수 있는 차이가 있었다.

< 표 1 > 기존 연구 차이점 분석

항목	기존 사례 특징	차별 및 개선점
그래픽	카툰 렌더링의 그래픽 스타일을 이용	카툰 렌더링 및 실사 그래픽을 **혼합하여 사용**
플레이어 수	1인 싱글 플레이	**1~4인 멀티 환경을 제공** 협동 및 재미 제공
물리 효과	(관련 내용 없음)	**물리적인 파괴 기능 추가** 시각적인 재미 추가 전략 및 전술 활용

3. 개발 목표

현 게임시장의 수요를 만족시키지 못하는 3D 슈팅 멀티 PC게임 소프트웨어의 테스트 버전 개발을 목표로 한다.

물리 효과 구현을 통해 화려하고 전략적인 플레이를 제공한다. 라이트 유저들이 쉽게 접근할 수 있게 간단한 조작을 통해 즐길 수 있다. 카툰 및 실사 그래픽을 혼합하여 더 풍부하고 깊은 미적 요소를 보여준다. 멀티 플레이 구현을 통해 협동 플레이를 할 수 있다.

4. 팀 역할 분담

팀 역할 분담은 아래 < 표 2 >을 참조한다.

< 표 2 > 팀원별 역할 분담

팀장 오동호	● 플레이어블 캐릭터 및 시스템제작 ● 적 AI 시스템 제작 ● 그래픽 및 스테이지 제작 ● 3D 리소스 및 UI 제작	각 기능별 테스트 통합 테스트 유지보수
팀원 강성준	● 네트워크 및 서버 기획 및 구축 ● 각 기능의 네트워크 연결 및 구축	
팀원 이원찬	● 사례 조사 ● 문서 및 프로젝트 관리 ● 멀티 플레이 구현	

5. 개발 일정

< 그림 1 >의 개발 일정을 준수하여 11월부터 1월까지 컨텐츠 및 시스템 기획한다. 기획 내용을 기반으로 1월부터 3월까지 시스템 설계 및 프로토타입을 개발한다.

<그림 1> 개발 일정

6. 개발 환경

개발 환경은 < 표 3 > 의 내용을 바탕으로 유니티를 이용하여 개발한다.

< 표 3 > 개발 환경

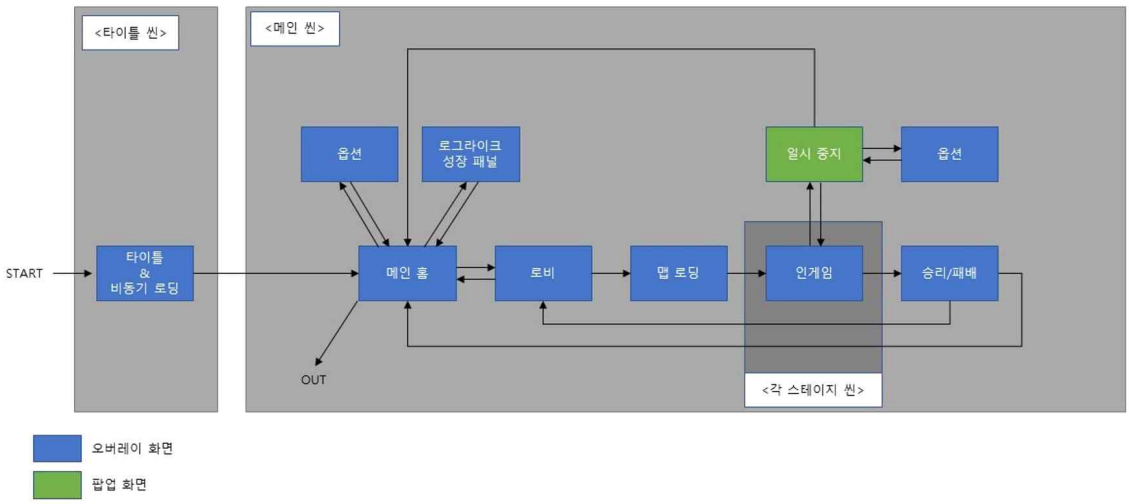
게임 엔진	Unity 2021(LTS)
3D 리소스 개발	3Ds Max2021 Blender Substance Painter
2D 리소스 개발	ClipStudio

Ⅱ. 본론

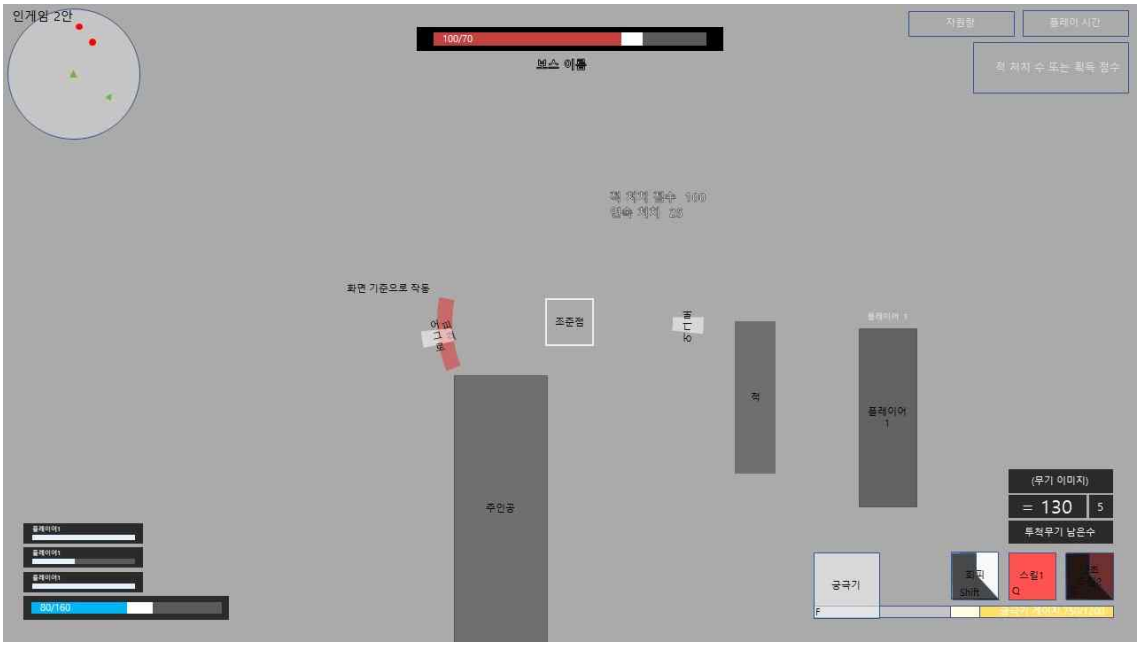
1. 개발 내용

< 그림 2 >와 같이 클라이언트를 실행하면 메인 씬이 불려오며 메인 홈에서 로비를 생성하고 서버를 통해 멀티 플레이어를 불러온다. 인게임에선 < 그림 3 >과 같이 화면을 구성한다.
서버는 포톤(Photon)을 이용하여 클라우드 서버를 생성하고 Pun2을 이용하여 로비의 구성 및 멀티플레이어 매칭, 인게임에서의 동기화를 진행한다.
3Ds Max2021으로 스테이지, 플레이어블 캐릭터의 모델, 적 캐릭터의 모델, 무기 모델 및 애니메이션을 제작하고 유니티를 이용해 인게임에서의 플레이어, 적, 스테이지, 이동 및 상호작용을 제어할 수 있도록 한다.
스테이지는 총 3개로 구성된다. 스테이지마다 물리 효과를 구현하여 전술적인 게임이 가능하게 한다.

< 그림 2 > UI Flow 기획



< 그림 3 > 인게임 UI 기획



2. 문제 및 해결방안

포톤(Photon)은 클라우드 서버를 사용하므로 해외에 서버가 위치하고 있어 핑(Ping)문제가 발생할 수 있다.

멀티 플레이가 불가능할 정도로 네트워크 작업에 차질이 발생하는 경우 닷넷(.Net) 서버를 사용하여 개발해야한다.

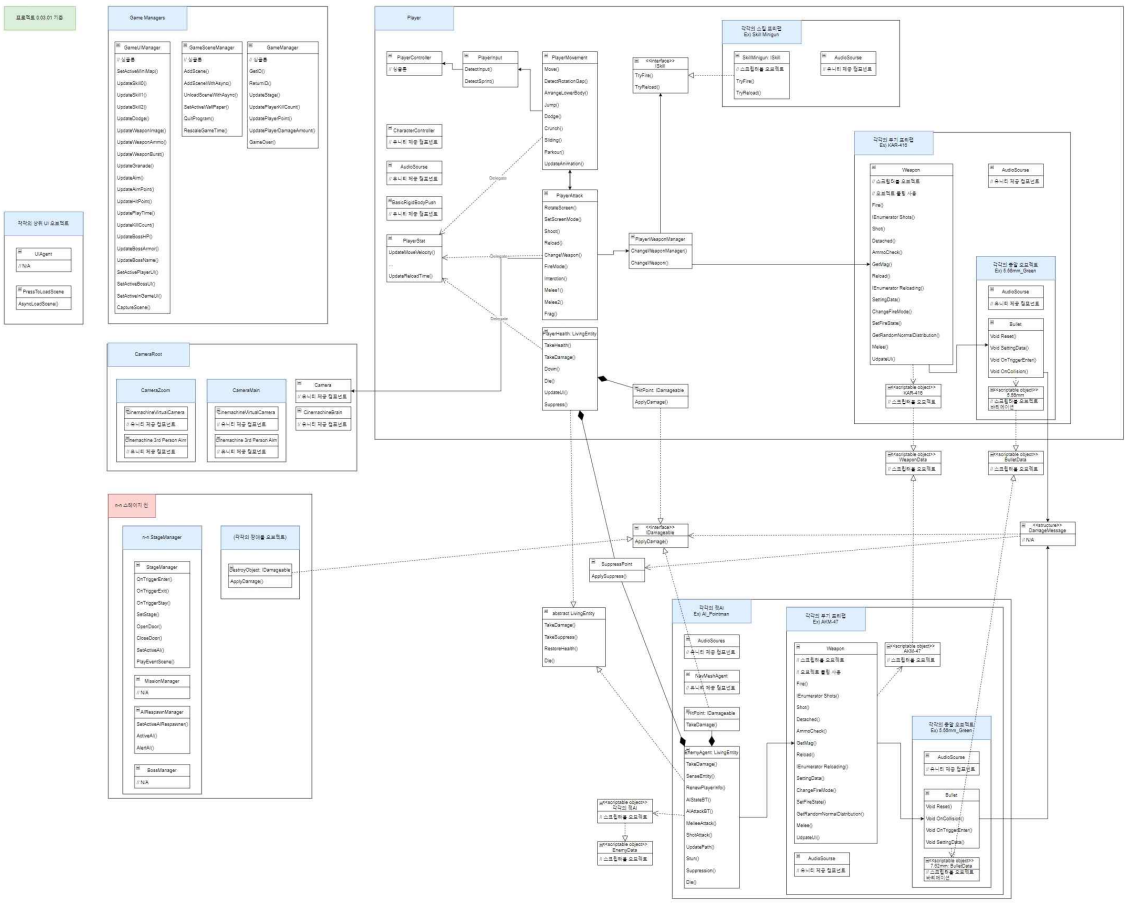
3. 시험시나리오

사용자 설정 시나리오는 다음과 같다. 사용자는 로비에서 플레이어들을 모집한 후 인게임으로 들어간다. 인게임에선 지하 주차장 스테이지에서 시작하여 사무 빌딩 스테이지, 옥상 스테이지로 총 3가지 스테이지를 주파한다. 유저는 적 AI를 상대한다. 적은 일반 병사, 엘리트 병사, 보스로 구분된다. 이때 플레이어들은 포톤(Photon) 서버를 통해 정보를 주고받으며 실시간으로 상호작용 및 협력할 수 있다.

플레이어는 스테이지를 공략할 때 목표 임무와 챕터별 보스를 처리한다. 스테이지에선 엄폐물과 파괴 효과가 있는 사물이 있다. 이를 이용하여 지형지물을 이용한 공략적인 게임이 가능하다. 마지막 스테이지까지 보스 공략을 한 경우 클리어와 함께 정산 화면이 출력되며 게임은 타이틀 화면으로 돌아간다.

4. 상세 설계

< 그림 4 > ComponentView



4.1. GameManager 오브젝트 설계

4.1.1. GameSceneManager.cs

씬 전환, UI 페이지 전환, 시간 조절, 프로그램 종료 등 프로그램의 전반적인 관리를 한다.
싱글톤으로 접근할 수 있다.

[자료구조]
[SerializeField] private PostProcessVolume m_postProcesser; // 배경에 필터를 줄 포스트프로세서

[메소드]

Void AddScene(string _sceneName)	씬을 추가 모드로 로딩한다. 씬의 이름을 입력을 받으면 해당 씬을 추가로 불러온다.
IEnumerator UnloadSceneWithAsync(string _sceneName)	씬을 비동기 추가 모드로 로딩한다. 씬의 이름을 입력 받으면 해당 씬을 비동기, 추가 모드로 불러온다.
IEnumerator UnloadSceneWithAsync(string _sceneName)	씬을 비동기 모드로 언로드한다. 씬의 이름을 입력받으면, 씬이 로드되어 있는지 확인 후 비동기 모드로 언로드한다.
Void SetActiveWallPaper(bool _active)	현재 씬을 블러 및 컬러그레이딩을 주어 배경화면으로 (비)활성화한다. 포스트프로세싱 컴포넌트를 저장하고 있다가 활성/비활성화 한다.
void QiutProgram()	프로그램을 종료한다.
Void RescaleGameTime(float _time)	시간 배속을 입력받아 게임 시간을 조절한다.

4.1.2. GameUIManager.cs

인게임에 관련된 모든UI를 직접 접근하여 관리한다.
싱글톤으로 접근할 수 있다.

[자료구조]
[SerializeField] private GameObject m_miniMap; // 플레이어_미니맵
[SerializeField] private TextMeshProUGUI playerHealth; // 플레이어_체력
[SerializeField] private TextMeshProUGUI remainAmmo; // 플레이어_남은 탄환
[SerializeField] private TextMeshProUGUI remainMag; // 플레이어_남은 탄창
...

[메소드]

Void SetActiveMiniMap(bool _active)	미니맵(비)활성한다.
Void Update~(int _text)	UI 내용을 업데이트한다. 스킬, 궁극기, 회피, 무기, 조준점, 점수(추후 스크롤방식으로) 구현, 피격, 자원량, 플레이 시간, 적 처치 수, 보스 체력, 방어막, 이름UI
Void SetActiveUI(bool _active)	보스 체력, 방어막, 이름UI 활성/비활성
Void SetActiveInGameUI(bool _active)	'적 방어막, 멀티 플레이어 이름, 적 레이저, 적 투척물 예상 경로' 등UI의 (비)활성화 정책을 변경하고 전파한다. On/Off정책을 가지고 있다가, 변경을 하게 되면, 액션(델리게이트)을 통해 구독자들에게 변경된 값을 반환한다. 각 객체들은 생성과 파괴시 델리게이트 등록/해제 처리를 해야 하며, UI를 띄울 수 있는 상황이더라도 정책이 false이면 UI를 띄우지 않는다.
Void CaptureScene()	씬 화면을 캡처 및 내부 변수에 저장한다.

4.1.3. GameManager.cs

게임 상황에 대한 기록을 하고, 게임오버를 감지한다.
플레이어의 점수, 킬 수, 데미지량, 상태 등을 기록한다.
싱글톤으로 존재한다.

[자료구조]
public int curLivingPlayer; // 현재 게임의 살아있는 플레이어 수
private int[] curPlayerCount = new int[4]; // 플레이어 ID 풀: 0-없음, 1-있음, 0번 ID-ID발급 거부

[메소드]

int GetID()	플레이어 캐릭터의 ID발급을 처리한다. 플레이어 캐릭터가 생성되면, 해당 메소드를 통해 ID를 발급받아야 한다. curPlayerCount 배열의 빈자리를 확인하고, 빈자리가 있다면 1~4를 리턴하고 해당 배열을 1로 채워준다. 자리가 없다면 0을 발급해준다.
void ReturnID(int _ID)	플레이어 캐릭터의 ID반환을 처리한다. 플레이어 캐릭터가 소멸되면, 해당 메소드를 통해 ID를 반환해야 한다.
Void UpdateStage(int _curStage)	진행중인 스테이지를 기록하고 다음 스테이지의 로딩을 시작한다.
Void UpdatePlayerKillCount(int _playerNum, int _killCount)	플레이어의 번호, 킬 수를 받아 저장한다.
Void UpdatePlayerPoint(int _playerNum, int _point)	플레이어의 번호, 획득 점수를 받아 저장한다.
Void UpdatePlayerDamageAmount(int _playerNum, int _damageAmount)	플레이어의 번호, 데미지양을 받아 저장한다.
Void GameOver(int _playerNum, bool _active)	각 플레이어들의 상태를 받아 저장하고, 특정 조건이 되면 게임 패배를 발동한다.

4.2. Player 오브젝트 설계

4.2.1. PlayerController.cs

캐릭터의 생성과 죽음을 처리하며,
싱글톤을 이용하여, 일부 입력처리(PlayerInput) 가능여부를 조작한다.

이를 통해 특수한 '체력, 공격, 이동' 중에 '이동, 화면, 공격, 스킬, 무기변경'등의 조작이 되지 않도록 한다.

입력 제한 구현 방법은 싱글톤으로, 제한/제한해제 여부와 ID(디버그용 임의의 문자열)를 받아 string list에 저장하고, 해제시에는 ID를 찾아 리스트에서 제거한다.

리스트에 원소가 1개라도 있다면 입력이 제한된다.

[자료구조]
public int ID;　　// 플레이어 ID

[메소드]

Void SetMoveControl(bool _active, string _ID)	움직임 조작 가능여부를 통제한다.
Void SetAttackControl(bool _active, string _ID)	공격 조작 가능여부를 통제한다.
Void SetScreenControl(bool _active, string _ID)	화면 조작 가능여부를 통제한다.
Void SetDead()	캐릭터가 죽으면 발동되는 메소드로, 관련 조작들을 통제한다.
Void SetAlive()	캐릭터가 스폰되면 발동되는 메소드로, 관련 조작들의 통제를 해제한다.

4.2.2. PlayerInput.cs

유니티의 인풋매니저에게서 플레이어의 입력을 받아와서
캐릭터 조작에 필요한 변수로 저장한다.

[자료구조]
#각 변수는 유니티의 입력시스템에서 입력관련 변수를 받아 새로 저장한다.
public Vector2 move
public Vector2 look
public bool jump
public bool sprint
public bool dodge
public bool crouch
public bool fire
public bool zoom
public float mouseWheel
public bool reload
public bool interaction
public bool melee
public bool frag
public bool fireMode
public bool weapon1
public bool weapon2
public bool weapon3
public bool weapon4
[SerializeField] private float m_sprintInvokeTime = 0.25f; // 질주 감지 시간
private float m_lastSprintInputTime = 0f; // 마지막 질주 키 입력 시간
private float m_curSprintInputTime = 0f; // 현재 질주 키 입력 시간: 나중에 '마지막 질주 키 입력시간'으로 복사됨

[메소드]

Void DetectInput()	메인 기능 메소드
bool DetectSprint()	질주 입력을 감지하는 메소드 W를 일정 시간 내에 연속 2번 누르거나, 달리기 입력 상태라면 True를 반환한다.

4.2.3. PlayerMovement.cs

캐릭터의 이동시스템을 책임진다.

[자료구조]

```
[SerializeField] float m_walkFrontVelo = 1f;      // 앞 이동속도
[SerializeField] float m_runVelo = 2.5f;      // 질주 이동속도
[SerializeField] float m_walkSideRat = 0.7f;      // 옆뒤 이동속도 감소비율
[SerializeField] float m_speedSmoothTime = 0.05f;    // 이동속도 스무스 시간
[SerializeField] Transform m_camCradle; // 카메라 거치대 자리
[SerializeField] float m_minMovementLowerBodyArrange = 0.1f;      //
상체하체정렬을 위해 필요한 최소한의 움직임
[SerializeField] float m_jumpPower = 5; // 점프 속도
[SerializeField] float m_gravityMultiple = 1;    // 점프시 중력 계수
[SerializeField] float m_maxDodgeCount = 3; // 최대 회피 개수
[SerializeField] float m_dodgeCount = 3;      // 현재 남은 회피 개수
[SerializeField] float m_dodgeRecoverPerSecond = 0.66f; // 초당 회피 개수 회복량
[SerializeField] float m_dodgeSpeed = 10;    // 회피 중 이동 속도
[SerializeField] float m_dodgeTime = 0.1f;   // 회피 지속시간
[SerializeField] float m_CrouchSpeed = 4;    // 앉기 중 이동 속도
[SerializeField] float m_CrouchHeightRatio = 0.5f;   // 앉기 중 키 비율
private Vector3 tarDirect;   // 목표이동방향
private float tarVelo;   // 목표이동속도
private bool isRun; // 질주 여부
private float currentVelocityY; // 목표 y축 속도
private float speedSmoothVelocity;   // 이동속도 변환의 부드러운 정도
private float curSpeed => new Vector2(charController.velocity.x,
charController.velocity.z).magnitude;      // 현재 캐릭터 속도
private bool isDodge = false;    // 회피 상태 여부
private float lastDodgeTime = 0;      //마지막 회피 입력시간
private bool isCrouch = false;   // 앉기 상태 여부
```

[메소드]

Void Move()	캐릭터의 이동을 처리한다. 목표 이동 속도와 방향을 정하고, CharacterController에 해당 정보를 전달하여 캐릭터에 움직임을 준다.
Bool DetectRotationGap()	하체와 상체 각도 차이 감지를 감지하고, 90도 이상이라면 true, 그렇지 않으면 false를 반환한다.
Void ArrangeLowerBody()	하체를 재정렬한다. 하체를 카메라의 y축 방향과 일치시킨 후, 카메라의 y축 방향을 보간한다.
Void Jump()	점프 처리 캐릭터가 공중에 있는 상태가 아니라면, 캐릭터의 y축 속도에 값을 주어 점프를 실행한다.
Void Dodge()	회피 처리 회피 개수가 남아있고 정지 및 점프상태가 아니라면, 짧은 시간동안, 현재 이동속도를 회피 속도로 설정한다. 이후 시간에 따라 회피 개수를 충전한다.
Void Crouch()	앉기 처리 이동속도가 앉기 속도로 설정되고, 이동 콜라이더가 변한다. 앉기 상태에서는, 서있을 때 캐릭터와 충돌이 예상된다면 일어서지 못한다.
Void Sliding()	슬라이딩 처리
Void UpdateAnimation()	캐릭터의 애니메이션 파라미터를 유니티의 애니메이터로 전달한다.

4.2.4. PlayerAttack.cs

캐릭터의 공격, 투척무기, 상호작용 및 화면조작을 책임진다.

[자료구조]
[SerializeField] Transform m_cameraHolder; // 카메라 거치대 자리
[SerializeField] float m_screenXSpeed = 1f; // X축 화면 이동속도
[SerializeField] float m_screenYSpeed = 1f; // Y축 화면 이동속도
[SerializeField] float m_screenNormalSpeed = 1f; // 평상시 화면 이동속도 계수
[SerializeField] float m_screenZoomSpeed = 0.5f; // 줌화면 이동속도 계수
[SerializeField] float m_minScreenAngle = 80f; // 화면 최대 아래 각도
[SerializeField] float m_maxScreenAngle = 80f; //화면 최개 위 각도
[SerializeField] GameObject m_virtualMainCamera; // 메인 카메라 자리
[SerializeField] CinemachineVirtualCamera m_mainCam; // 메인 카메라의 가상카메라
[SerializeField] CinemachineVirtualCamera m_zoomCam; // 줌 카메라의 가상카메라
[SerializeField] private float m_idleCamHolderHeight = 1.8f; // 평소 상태 카메라 높이
[SerializeField] private float m_zoomCamHolderHeught = 1.55f; // 줌 상태 카메라 높이
private float curScreenSpeed; // 현재 화면 회전 속도
private Vector3 curCamHolderLocalPosition = Vector3.zero; // 현재 카메라 홀더의 로컬 위치
private bool isZoomMode = false; // 줌 상태 여부

[메소드]

Void RotateScreen()	화면 회전을 처리한다. 동시에 상체를 화면 방향에 맞게 동기화한다.
Void SetScreenMode()	화면 모드의 변경을 처리한다.
Void ChangeWeapon()	무기1, 2 변경을 처리한다.
Void Shoot()	무기 사격을 명령한다.
Void Reload()	재장전을 명령한다.
Void FireMode()	사격모드 변환을 명령한다.
Void Interction()	상호작용을 처리한다.
Void Melee1()	근접공격을 처리한다.
Void Melee2()	근접공격을 처리한다.
Void Frag()	투척무기을 처리한다.

4.2.5. Health.cs: LivingEntity

캐릭터의 피격, 생명, 다운, 죽음, 제압시스템을 책임진다.
ILivingEntity을 상속받아 구현한다.

[메소드]

void RestoreHealth(float _restoreAmount)	체력 회복을 처리한다. 입력받은 양만큼 캐릭터의 체력을 회복한다.
void TakeDamage(DamageMessage _damageMessage, HitParts _hitPart)	데미지를 처리한다. 데미지 메시지와 피격부위를 입력받아, 피격부위에 알맞은 계수의 데미지를 처리한다.
TakeHealth(float _restoreAmount)	회복양을 입력받아 캐릭터의 체력을 회복시킨다.
Void Down()	캐릭터의 상태를 다운으로 바꾸고, 움직임 조작을 통제한다.
Void Die()	캐릭터의 상태를 죽음으로 바꾸고, 관련 조작을 통제한다.
Void Suppress(float _suppressionAmount)	입력받은 제압수치만큼, 캐릭터의 제압수치를 올리고 현재 제압수치에 알맞은 효과를 낸다. 제압수치는 점차 하락한다.
void UpdateUI()	캐릭터의 체력UI를 업데이트 한다.

4.2.6. PlayerStat.cs

캐릭터의 성장정보를 관리한다.
외부 시스템으로부터 성장정보를 받으면 변경된 정보를 액션(델리게이트)를 이용하여 전파한다.
Movement, attack, health에 영향을 끼친다.

4.2.7. PlayerWeaponManager.cs

캐릭터가 들고 있는 무기 관리(획득, 변경), 운용명령(사격, 재장전, 발사모드)을 한다.

[자료구조]
[SerializeField] Weapon[] weaponArray = new Weapon[2]; // 1~4번 슬릇에 사용할 무기 배열
private int curWeapon = 0; // 현재 들고 있는 무기의 배열 번호

[메소드]

void ChangeWeaponCommand()	무기변경 상위 메소드 외부에서 무기 변경 명령을 입력 받아, 알맞은 알고리즘을 실행한다.
Void ChangeWeapon(int _newWeaponIndex)	무기변경을 처리한다. 캐릭터는 입력받은 번호의 배열에 있는 무기로 변경한다.

4.3. 적 AI 오브젝트 설계

4.3.1. EnemyAgent.cs: LivingEntity

적 AI의 구현한다.
행동트리, 체력, 스턴, 제압, 이동, 사격, 근접공격, 감각, 타깃결정

[자료구조]
public enum AIState{ wait, move, engage} // AI가 가질 수 있는 경계상태
[SerializeField] private Transform eyeTransform; // 눈의 위치 정보
[SerializeField] private float eyeDistance; // 시야 거리
[SerializeField] private LayerMask attackTarget; // 공격 및 타겟 대상 레이어
[SerializeField] private float fieldOfView; // 시야각
[SerializeField] private float engageDistance; // 교전을 위한 이동 정지거리
[SerializeField] private float melleeDistance; // 근접공격을 시작하는 거리
[SerializeField] private float distanceTargetWeight; // 공격 타겟 선정을 위한 거리별 가중치
[SerializeField] private float attackerTargetWeight; // 공격 타겟 선정을 위한 마지막 공격자 가중치
private AIState curState; // AI의 경계상태
private int[] isPlayerOnSight = new int[4]; // 플레이어가 시야 내에 보이는지 여부
private GameObject[] players = new GameObject[4]; // 플레이어들의 게임 오브젝트 정보
private float[] playerDistance = {9999,9999,9999,9999}; // 각 플레이어와의 거리
private int[] targetWeight = new int[4]; // 각 플레이어별 타겟팅(어그로) 가중치
private int lastAttacker; // 마지막으로 AI를 공격한 플레이어: 0은 판단불가 의미
private Ray ray; // 플레이어 탐색용 레이
private int targetPlayer; // 공격 대상: 0은 없음

[메소드]

Void TakeDamage(DamageMessage _damageMessage, HitParts _hitPart)	데미지를 처리한다. 연결된 HitPoint 클래스한테 DamageMessage를 받으면 피격부위에 따라 다른 배율의 데미지, 스턴 및 제압량을 받고, 애니메이션을 재생한다.
Void SenseEntity()	AI 시야 내에 플레이어가 있는지 검사한다. 시야 내에 있거나 데미지를 받으면 해당 플레이어를 식별한다.
Void OnDrawGizmoSelected()	디버그 목적의 GUI를 그린다. AI의 시야를 에디터에서 볼 수 있다.
Void RenewPlayerInfo()	각 플레이어의와의 거리, 타겟 가중치를 계산한다.
void AIStateBT()	AI의 행동트리를 처리한다. 거리와 시야 내 플레이어 감지 여부에 따라 이동 또는 전투의 상태를 가진다.
void AIAttackBT()	공격 타겟을 타겟 가중치가 높은 순서대로 선정한 후, 타겟과의 거리에 따라 공격방법을 선택하고 공격을 한다.
Void MelleeAttack()	타깃의 방향으로 근접공격을 한다.
Void ShotAttack()	랜덤으로 사격주기, 정확도, 발사량을 정해서 타깃에 발사한다.
Void UpdatePath()	이동경로 갱신&이동 최우선 타깃을 목표로 이동한다.
Void Stun()	피격 부위에 따라 시간이 다른 애니메이션을 재생한다. 피격동안은 행동트리에 따른 행동을 하지 못한다.
Void Suppression(float _suppressAmount)	플레이어와 마찬가지로 피격수치가 존재하고 파라미터에 따라 이동속도, 사격 관련 파라미터가 떨어지고 제압 애니메이션을 재생할 수 있다.
Void Die()	AI의 죽음을 처리한다.

4.4. 무기 오브젝트 설계

4.4.1. Weapon.cs

스크립터블 오브젝트 방식으로 구현한다.
총알 프리팹을 필요로 한다.
총알 프리팹을 오브젝트 풀링 방식으로 사용한다.

[자료구조]
[SerializeField] private WeaponData weaponData; // 총기 SO
[SerializeField]private Bullet bulletPrefab; // 총알 프리팹
[SerializeField]private Transform muzzlePosition; // 총구 위치
[SerializeField] private bool autoReload = true; // 자동 재장전 정책 허용여부
[SerializeField] private float moaMultiple = 4f; // 실제 총기 명중률의 계수
[SerializeField] private FireMode[] havingFireMode; // 해당 무기가 가지는
[SerializeField] private int curFireMode = 0; // 현재 발사모드
private float fireRPM; // 발사 속도(1rpm = 1round per 1minute)(실제스펙)
private float MOA; // 총기 명중률 (1moa = 1inch per 100yard)(실제스펙)
[SerializeField] private float maxSpreadStandard; // 최대 스프레드
[SerializeField] private float recoilPerShot; // 발사 당 증가 스프레드
[SerializeField] private float recoilRecoverTime; // 반동 회복 속도
[SerializeField]private int magCappacity = 5; // 탄창 용량
[SerializeField]private int curRemainAmmo; // 현재 탄창에 남은 총알 수
[SerializeField]private int curRemainMag = 5; // 현재 남은 탄창 수
private float reloadTime = 3; // 재장전 시간
private ObjectPool<Bullet> bulletPool; // 총알 오브젝트 풀
private GameObject weaponUser; // 무기 사용자 오브젝트
private float fireInterval => (60 / fireRPM); // 발사 속도에 따른 발사 간격
private enum State{ready, empty, reloading, shooting}
private State state; // 현재 상태
private bool isTriggered = false; // 트리거가 눌려저 있는지
private bool isHipFire = true; // 기본 사격 상태인지
private float lastFireTime; // 마지막 발사 시간
private Vector3 fireDirection; // 의도하는 사격 방향
[SerializeField] private float curSpread; // 현재 스프레드
private float curSpreadVelocity;
private float xVar; // x스프레드 값
private float yVar; // y스프레드 값

[메소드]

Void Fire()	1티어 공격 명령 발사가 가능한지 확인한 후, 현재의 발사모드에 따라 발사 횟수와 발사 명령을 2티어 사격 메소드에 전달한다.
IEnumerator Shots(int _remainShotCount)	2티어 사격 메소드: 사격 통제 장치 명령받은 사격이 끝났거나 총알이 없으면 중지한다. 그렇지 않다면 발사간격을 확인한 후 최종 발사를 명령한다.
Void Shot()	3티어 사격 메소드: 최종 1회 사격 총알 수을 1 감소시키고 현재 총기의 상태에 따라 발사 방향을 정하고 1개의 총알 프리팹을 오브젝트 풀에서 방출한다.
Void Detached()	사용자가 트리거에서 손을 떼면 상태변수를false로 변경한다.
Void AmmoCheck()	총알 수 확인 및 총기 상태 변경한다.
Void GetMag(int_ count)	입력만큼 탄창 수를 증가시킨다.
Void Reload()	남은 탄창 수를 확인한 후 재장전 처리를 명령한다.
IEnumerator Reloading()	실제 재장전처리를 한고 UI를 업데이트 한다.
Void SettingData()	스크립터블 오브젝트에서 총기정보를 가져와 초기화한다.
Void ChangeFireMode()	무기의 발사모드를 변경한다.
Void SetFireState(bool _hipFire)	입력에 따라 기본사격 상태인지, 조준사격 상태인지 상태변수를 변경한다.
float GetRandomNormalDistribution(float mean, float standard)	평균 값과 정규분포 값을 입력하여, 정규분포 난수를 리턴한다. 출처: https://github.com/IJEMIN/Unity-TPS-Sample
Bool Melee()	근접공격이 있다면 근접공격 명령한다.
Void UpdateUI()	캐릭터가 들고 있는 무기의 잔탄과 남은 탄창의 수를 UI에 업데이트한다.

4.5. 총알 오브젝트 설계

4.5.1. Bullet.cs

[자료구조]
[SerializeField] private string bulletName; // 총알 이름
[SerializeField] private float bulletSpeed; // 총알 속도
[SerializeField] private float bulletDamage; // 총알 데미지
[SerializeField] private float bulletSuppress; // 제압량
[SerializeField] private float lifeTime; // 생명주기
private float enabledTime; // 발사된 시간
private GameObject topLevelParent; // 총알 주인
public IObjectPool<Bullet> poolToReturn; // 자신을 관리하는 오브젝트 풀 변수

[메소드]

Void Reset(GameObject _topLevelParent, Vector3 _firePosition, Vector3 _direction)	오브젝트 풀에서 생성되었을 때, 공격자, 위치, 회전, 속도를 리셋한다.
void OnTriggerEnter(Collider other)	총알이 트리거에 들어갔을 때를 처리한다. 만약 트리거의 오브젝트가 SuppressPoint 컴포넌트를 가지고 있다면 제압 동작을 하기 위한 DamageMessage를 작성하여 전달한다.
void OnCollisionEnter(Collision other)	총알이 다른 물체와 충돌했을 때를 처리한다. 만약 트리거의 오브젝트가 IDamageable 컴포넌트를 가지고 있다면 공격 처리를 하기 위한 DamageMessage를 작성하여 전달한다.
Void SettingData()	Scriptable Object에서 총알의 정보를 가져와 총알 정보를 초기화한다.

4.6. 인터페이스, 추상클래스, 스크립터블 오브젝트, 공통 클래스 설계

4.6.1. interface IDamageable.cs

[메소드]

void ApplyDamage(DamageMessage _damageMessage)	데미지 메시지를 받아, 체력에 데미지를 적용한다.

4.6.2. Interface ISkill.cs

4.6.3. Class WeaponData.cs : ScriptableObject

총기의 정보를 저장한다.

[자료구조]
public enum FireMode { automatic, burst, twice, single } // 발사모드
[SerializeField]private string weaponName; // 무기 이름
[SerializeField]private FireMode[] havingFireMode; // 해당 무기가 가지는 발사모드
[SerializeField]private float fireRPM; // 사격 RPM
[SerializeField] private float moa; // 명중률 MOA
[SerializeField] private float recoilPerShot; // 1회 사격 당 반동
[SerializeField] private float recoilRecoverTime; // 반동 회복 속도
[SerializeField]private int magCappacity; // 탄창 용량
[SerializeField]private int initMagCount; // 초기 지급 탄창 수
[SerializeField]private float reloadTime; // 재장전 시간

4.6.4. Class BulletData.cs : ScriptableObject

총알의 정보를 저장한다.

[자료구조]
[SerializeField]private string bulletName; // 총알이름
[SerializeField]private float damage; // 총알 데미지
[SerializeField] private float suppress; // 총알 제압량
[SerializeField]private float speed; // 총알 속도
[SerializeField]private float lifeTime; // 총알 생명시간

4.6.5. Struct DamageMessage.cs

데미지와 제압 정보를 가진 공격관련 자료구조를 가진다.

[자료구조]
public enum DamageKind{ bullet, explosion, fire } // 데미지의 종류
public GameObject attacker; // 공격자
public int ID; // 공격체 식별을 위한 난수 저장공간
public float damageAmount; // 데미지 양
public float suppressAmount; // 제압량
public DamageKind damageKind; // 데미지 종류
public Vector3 hitPoint; // 충돌 위치
public Vector3 hitNormal; // 충돌 노멀

4.6.6. EnemyData.cs : ScriptableObject

적 AI의 정보를 가진다.

[자료구조]
[SerializeField]private string enemyName; // 적 이름

4.6.7. LivingEntity.cs : IDamageable

체력을 가지는 생물체를 나타내는 추상클래스이다.

[자료구조]
[SerializeField] protected float MaxHealth = 100; // 시작 및 최대 체력
[SerializeField] protected float MaxSuppress = 100; // 최대 제압수치
[SerializeField] protected float UnsuppressAmount = 7; // 초당 제압해제수치
[SerializeField] protected float[] HitMultiple = {2f, 1f, 0.8f, 0.8f}; // 부위별 데미지 계수 / 머리,몸통,팔,다리
protected float curHealth; // 현재 체력
protected float curSuppress = 0; // 현재 제압량
public bool isDead; // 사망여부

[메소드]

void TakeDamage(DamageMessage _damageMessage, HitParts _hitPart)	하위 피격부위의 오브젝트들에게 DamageMessage와 피격부위를 입력받아 데미지 처리를 한다.
void TakeSuppress(DamageMessage _damageMessage)	하위 제압부위의 오브젝트에게 DamageMessage를 입력받아 제압 처리를 한다.
void RestoreHealth(float _restoreAmount)	입력받은 양만큼 체력을 회복한다.
void Die()	죽음을 처리한다.

4.6.8. HitPoint.cs: IDamaeable

각 피격용 콜라이더가 있는 오브젝트에서 사용되며, 피격 정보를 Health: LivingEntity로 보내 처리한다.

[자료구조]
public enum HitParts{ head, upBody, arm, leg} // 피격 부위 종류
private LivingEntity EntityHealth; // 데미지를 전달할 상위 LivingEntity
[SerializeField] HitParts hitPart; // 피격 부위

[메소드]

Void ApplyDamage(DamageMessage _damageMessage)	공격체로부터 받은 데미지와 피격 부위를 상위 LivingEntity에게 전달한다.

4.6.9. SuppressPoint

각 제압용 콜라이더가 있는 오브젝트에서 사용되며, 제압 정보를 Health: LivingEntity로 보내 처리한다.

[메소드]

void ApplySuppress(DamageMessage _damageMessage)	공격체로부터 받은 제압량을 상위 LivingEntity에게 전달한다.

4.7. 스테이지 오브젝트

한 구역의 '세이프룸~스테이지~목표임무'의 레벨 디자인과 스트리밍 레벨, 이동 경로 통제, 이벤트 발생, 목표임무를 관리한다.

4.7.1. StageManager.cs

스테이지의 스트리밍 레벨과 이동 경로 통제 및 이벤트 발생을 관리한다.

[자료구조]
[SerializeField] private string nextSceneName; // 다음 스테이지 씬 이름
[SerializeField] private string prevSceneName; // 이전 스테이지 씬 이름
private int curPlayerCountInRoom; // 현재 세이프룸 인원
private bool startedNextScene = false; // 다음 씬의 로드가 시작되었는지 여부

[메소드]

void OnTriggerEnter(Collider other)	세이프룸의 트리거에 물체의 진입이 감지되면, 태그를 확인하여 Player라면 curPlayerCountInRoom를 1 증가 시킨다.
void OnTriggerExit(Collider other)	세이프룸의 트리거에 물체의 진출이 감지되면, 태그를 확인하여 Player라면 curPlayerCountInRoom를 1 감소 시킨다.
void OnTriggerStay(Collider other)	세이프룸의 트리거에 물체가 감지되면, 태그를 확인하여 Player라면 SetStage()를 실행한다.
void SetStage()	세이프룸 안의 플레이어 수가 살아있는 플레이어 수와 같은지 검사하고, 다음 스테이지를 비동기 로딩 및 이전 스테이지를 비동기 언로딩한다.
Void OpenDoor(int _doorNum)	특정 조건이 되면 입력 받은 번호의 문을 연다.
Void CloseDoor(int _doorNum)	특정 조건이 되면 입력 받은 번호의 문을 닫는다.
Void SetActiveAI(bool _active)	해당 스테이지 내의 AI를 활성화/비활성화 한다.
Void PlayEventScene()	특정 조건 시, 이벤트 씬을 플레이한다.

4.7.2. MissionManager.cs

목표임무의 시작부터 종료까지 관리한다.
각 목표임무에 따라 상이하다.

4.7.3. AIManager.cs

AI 리스폰 지점의 생성과 폐쇄 및 AI 유닛을 관리한다.

[메소드]

Void SetActiveAIRespawner(bool _active)	리스폰 오브젝트를 활성화/비활성화한다.
Void ActiveAI()	관할 지역 내 AI 오브젝트를 활성화한다.
Void AlertAI()	플레이어가 특정 지역 내에서 첫 사격을 하면 매니저에게 해당 위치를 주고, 매니저는 활성화된 AI들에게 해당 위치 이동 명령을 한다.

4.7.4. BossManager.cs

보스 스테이지의 시작부터 종료까지 관리한다.
각 보스에 따라 상이하다.

4.8. UI 패널 오브젝트 설계

4.8.1. UIAgent.cs

마우스 클릭을 통한 버튼 컴포넌트의 역할을 키보드의 입력을 통해 대신 수행하는 역할을 한다.

[자료구조]
[SerializeField] private Button ESC; // ESC 버튼을 ESC키로 대체한다.
[SerializeField] private Button AnyKey; // 아무 버튼을 아무 키로 대체한다.

4.8.2 PressToLoadScene.cs

타이틀 씬에서 메인 씬으로 전환하기 위해 사용된다.
시작과 동시에 메인 씬을 비동기 모드로 로딩하며, 로딩바를 조작한다.

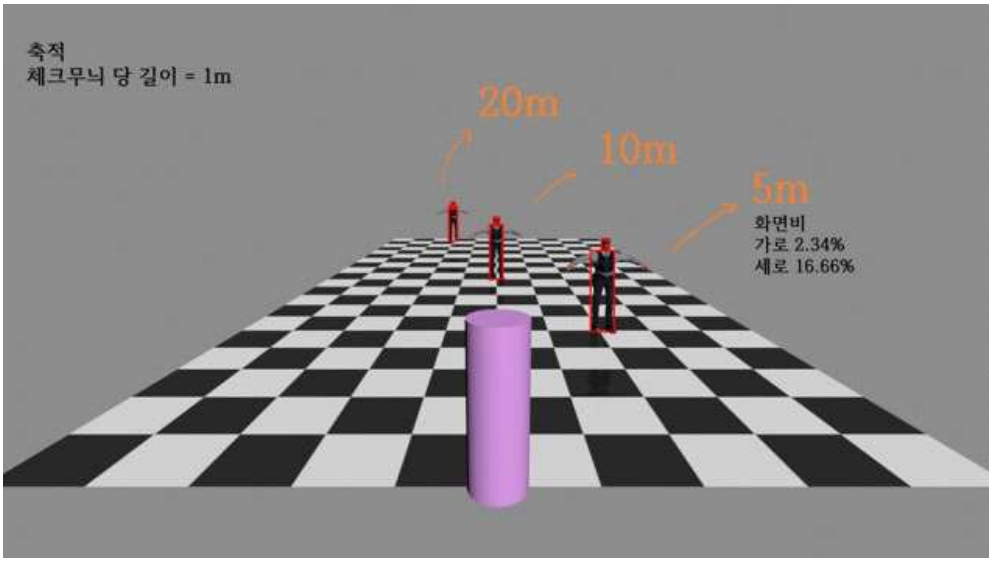
[자료구조]
[SerializeField] Image progressBar; // 로딩 바 이미지
[SerializeField] private string nextSceneName; // 다음 씬 이름

[메소드]

IEnumerator AsyncLoadScene()	씬을 비동기 모드로 로드한다. 동시에 로딩바를 조작한다. 로딩이 완료된 후, 아무 키를 누르면 메인 씬으로 전환된다.

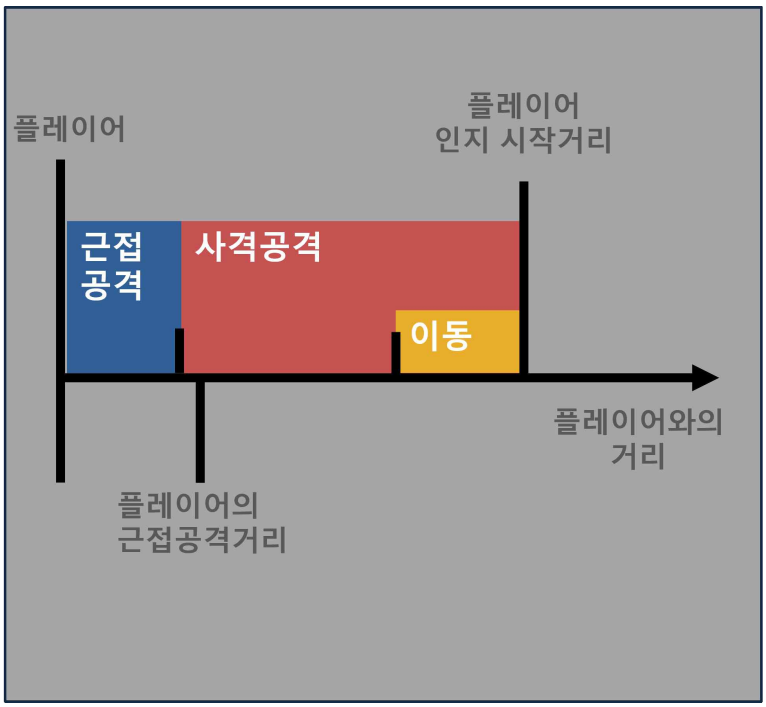
4.9. < 그림 5 >은 캐릭터와의 거리에 따라 외형의 크기를 예상하는 이미지이다.

< 그림 5 > 외형 설계



4.10. < 그림 6 >은 플레이어의 공격 매커니즘을 설계한 이미지이다.

< 그림 6 >



4.11. < 그림 7 >은 설계한 적 Ai의 특징과 명칭을 설계한 자료이다.

< 그림 7 >

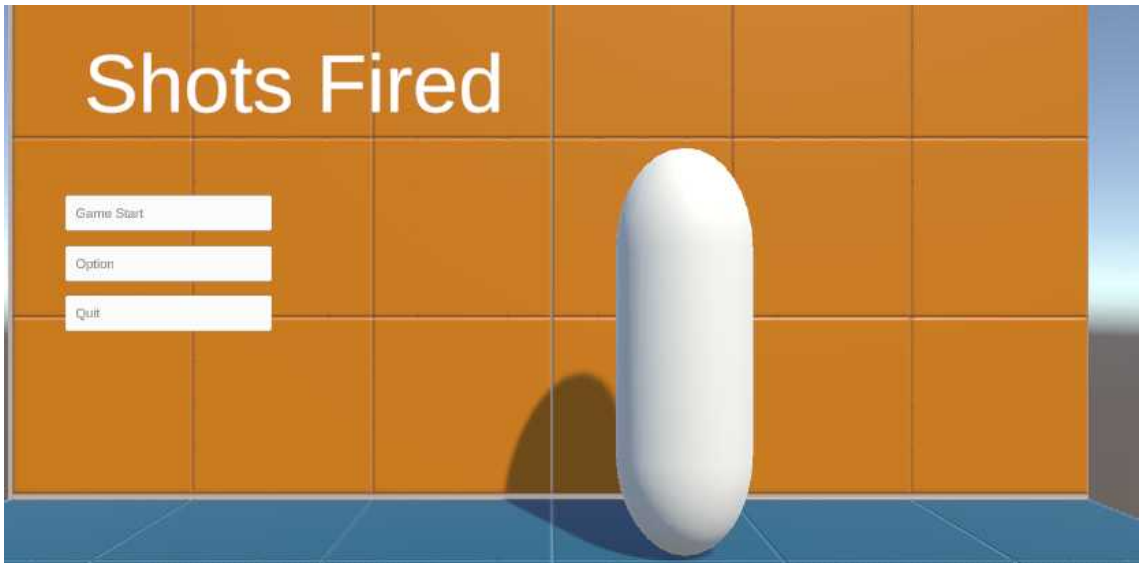
클래스	포인트맨	라이플맨	엘리트
특징	SMG, 샷건을 위주로 사용한다. 상대적으로 라이플맨보다 짧은 거리에서 교전을 한다. 상대적으로 이동속도가 빠르다.	AR을 사용한다. 상대적으로 포인트맨보다 긴거리에서 교전한다. 보통의 이동속도를 가진다.	강화된 포인트맨과 라이플맨 개념이다. **방탄헬멧과 방탄복**을 사용한다. **매우 근접한 거리에서만 사격데미지가 적용**된다. 그렇지 않다면 피격애니메이션만 재생되고 대미지는 입지 않는다. 참고: https://www.youtube.com/watch?v=C_caZ0ck3TM
의도	N/A	N/A	플레이어가 **다가오도록 유도**한다. **폭발물 또는 CC기의 소비**를 유도한다.
공략방법	N/A	N/A	가까운 거리에서의 사격하거나 폭발물로 공략한다. 스킬(철갑탄, 아머피어싱 탄, 기타 화력 스킬)로 공략한다.
외형특징	N/A	AR무기	검은색의 방탄헬멧과 방탄복

5. Prototype 구현

프로토타입에서는 기초, 기반 기술 제작을 목표로 정하였다. 그리하여 본 프로토타입에서 네트워크를 통한 플레이어 간의 상호작용 , 플레이어의 사격 및 조종 컨트롤 , 플레이어의 피격 판정 , 다양한 총기 구현등 여러 기능에 대해서 아래에 서술할 것 이다.

5.1 네트워크 환경 구현

< 그림 8 > 초기 화면



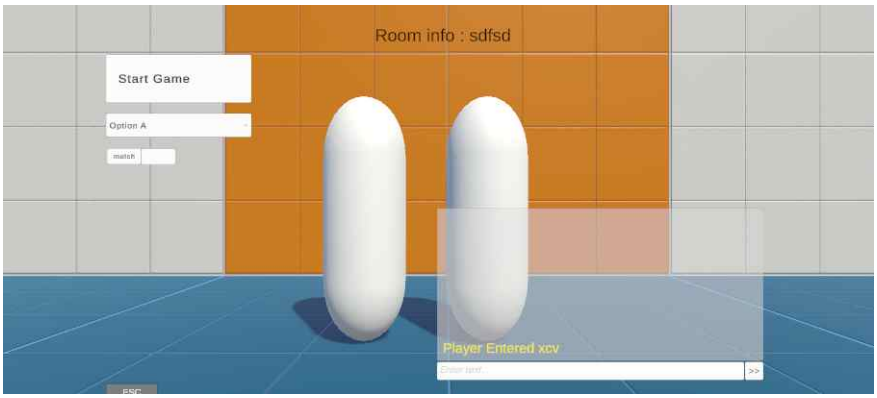
위 < 그림 8 > 은 게임의 초기 화면이다. 플레이어는 게임을 시작하면 위와 같은 인터페이스가 나오는데 여기서 좌측 에 있는 게임 스타트 버튼을 클릭하면 플레이어는 사전에 구성된 서버와 연결되며 통신 가능 상태가 된다.
이후 플레이어가 게임에 접속을 성공하면 플레이어의 ID 입력 버튼이 나오게 된다. 플레이어는 게임에서 사용할 자신의 ID을 입력하고 게임에 접속하게 된다. 게임에 접속하면 플레이어는 자신이 게임을 플레이를 할 방을 선택하고 이후 방에 들어가

면 게임을 시작할 수 있다.

< 그림 9 > 게임 ID 입력 및 게임 룸 접속

< 그림 10 > 게임 로비

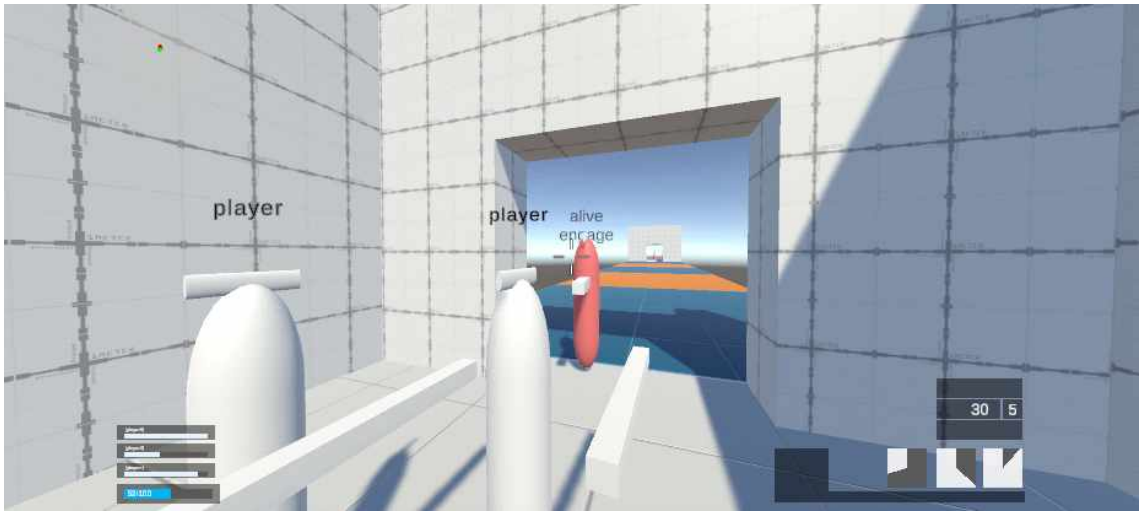
<그림 11> 게임 룸 접속시



플레이어가 < 그림 11 >처럼 게임 룸에 들어가면 방에 존재하는 플레이어 수만큼 프로토 타입 캐릭터들이 나타난다. 현재는 2명의 플레이어가 방에 들어와서 2개의 원기둥이 있는 것을 알 수 있다. 그리고 방 내에서 플레이어는 다른 이와 게임을 소통하고 게임을 플래이 하기 쉽도록 채팅 기능이 구현되어 있다. 그리고 방의 주인인 방장이 게임 시작 버튼을 클릭하면 모든 플레이어는 게임 화면으로 씬이 전환된다.

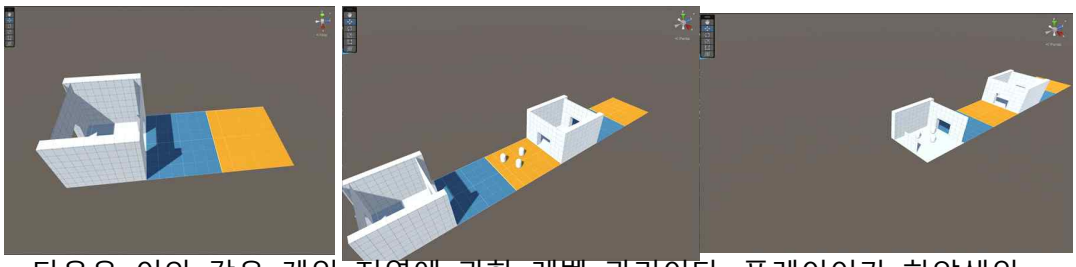
5.2 게임 플레이

< 그림 12 > 게임 접속시



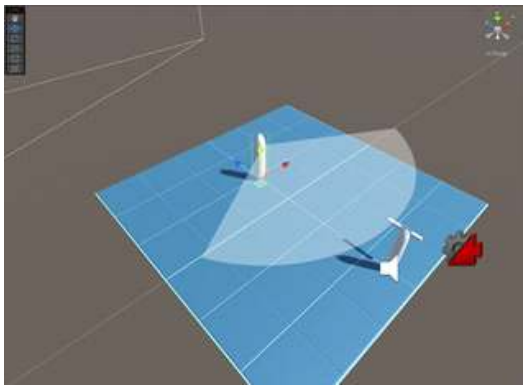
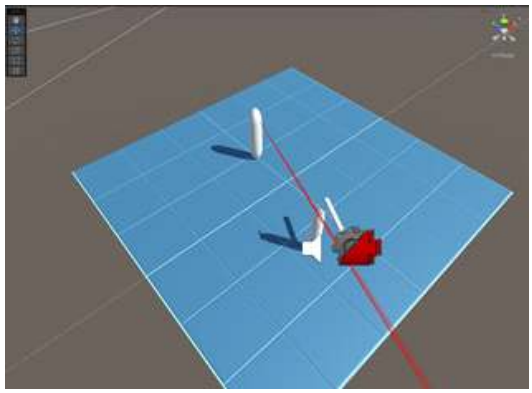
플레이어는 게임에 접속을 하면 모든 플레이어는 각자의 캐릭터를 가지고 게임을 진행 할 수 있게 된다. 게임에는 안전 지역과 게임 지역이 존재한다. 게임 시작 시 모든 플레이어는 안전 지역에 머물게 된다. < 그림 12 >처럼 AI는 안전 지역에 있는 플레이어에게 접근하지 못 한다.

< 그림 13 > 스트리밍 레벨



다음은 이와 같은 게임 지역에 관한 레벨 관리이다. 플레이어가 하얀색인 안전지역에서 위험 지역을 지나서 다음 안전 지역에 도달하면 게임 시스템에서 이전에 있던 지역을 삭제 함으로 메모리 디자인을 많이 사용하는 레벨 디자인을 플레이어에 위치에 따라 유동적으로 로드 / 언로드를 함으로 메모리를 절약 할 수 있다.

< 그림 14 > 플레이어 감지

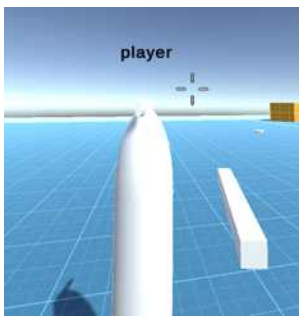
다음은 플레이어 감지 부분이다. < 그림 14 >에서 보면 AI 감지 범위를 가지고 이러한 감지 범위 안에서 플레이어를 발견하게 되면 이를 감지하고 공격을 하게 된다. 이러한 경우 AI 범위안에 들어온 적을 감지하여 공격하는데 이에 우선순위가 있다. 플레이어 2명을 감지 했으면 AI 플레이어는 다음과 같은 방식으로 적을 식별하고 공격한다.

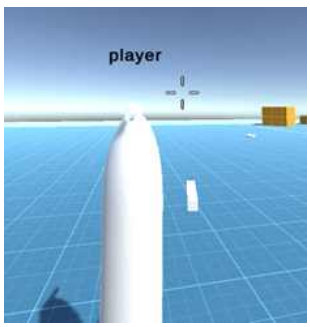
<그림 15 > 타깃 결정 가중치

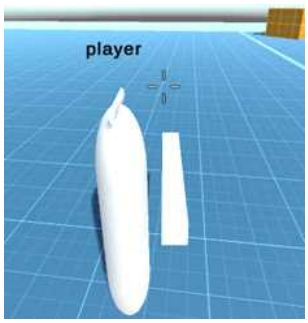
[가까운 거리] × [거리 가중치] + [마지막 피격 여부] × [피격 가중치]

이러한 가중치는 모든 플레이어 마다 따로 계산되고 이를 이용하여 공격을 할 플레이어를 AI는 정하게 된다.

< 그림 16 > 플레이어 총기 시스템



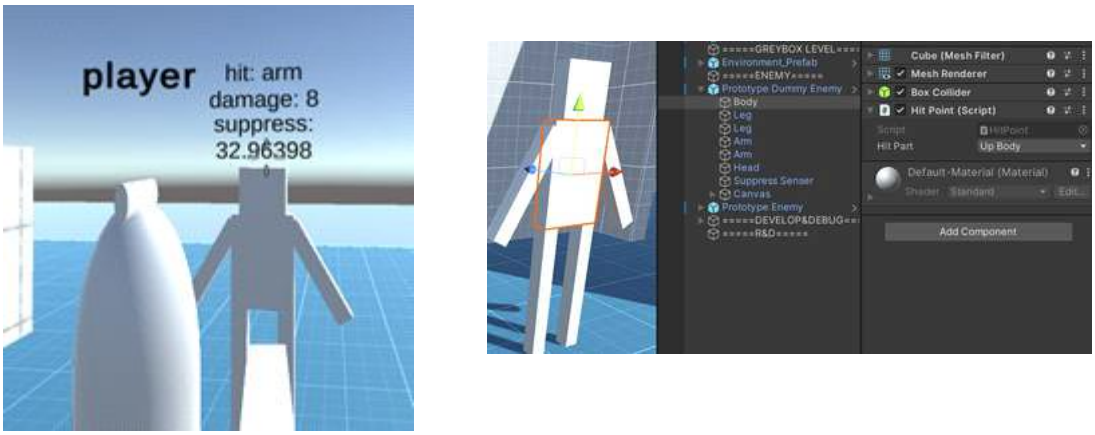




< 그림 16 >을 보면 알 수 있듯이 다음은 각각의 플레이어들의 총기이다. 플레이어는 총알을 발사하여 AI를 쓰러트리게 된다. 이때 플레이어는 3가지의 총기를 다룰 수 있게 된다. 첫 번째는 소총으로 마우스를 누루고 있으면 계속해서 나가게 된다. 또한 실제 소총처럼 단발, 연사, 점사모드를 사용 할 수 있다. 또한 2번째는 권총으로 플레이어가 첫 번째 총의 총알이 부족할 시 위급하게 사용할 수 있다. 이는 소총 보다는 약하고 단발 사격만 가능하다. 마지막은 샷건으로 마우스 클릭 시 산탄을 발사한다. 샷건의 장전방식은 볼트 액션식으로 1발씩 장전한다. 각

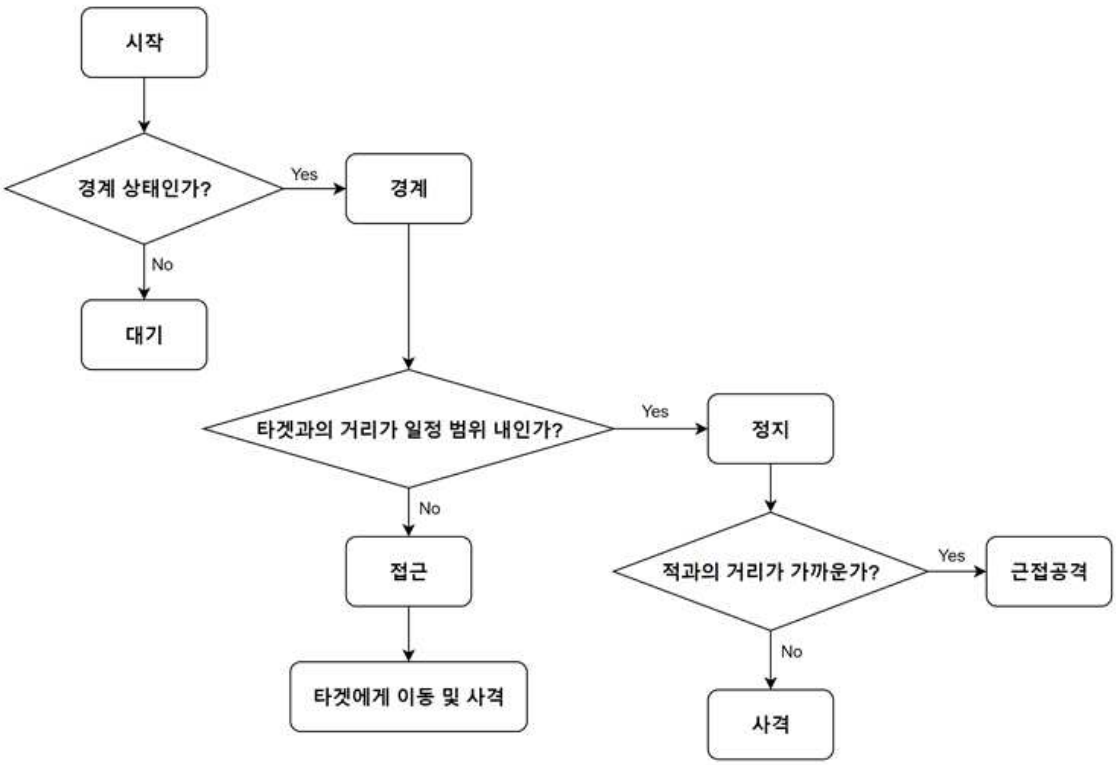
총에 대한 총알은 오브젝트 폴링 방식으로 관리하여 메모리 관리를 편하게 할 수 있다.

< 그림 17 > 부위 피격 시스템



플레이어는 총으로 어디를 공격하느냐에 따라서 다른 데미지를 AI에게 줄 수 있다. AI의 피격 부위에 따라서 서로 다른 데미지를 입히며 머리, 몸통, 팔, 다리의 피격 부위가 존재한다.

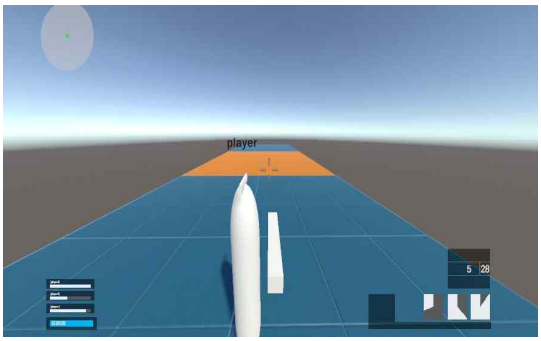
< 그림 18 > AI 행동 방향



AI는 위처럼 적 플레이어를 발견하고 다음과 같은 알고리즘으로 작동한다. 경계 상태인지, 플레이어가 공격 범위 인지, 플레이어가 근접해 있는지에 따라서 위와

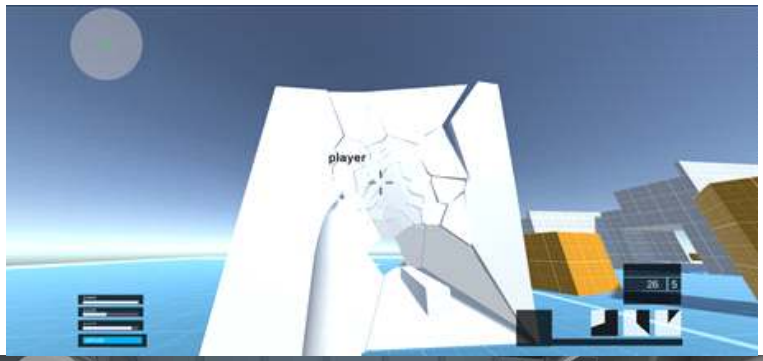
같은 방식으로 작동하게 되어 있다.

< 그림 19 > 플레이어 승리



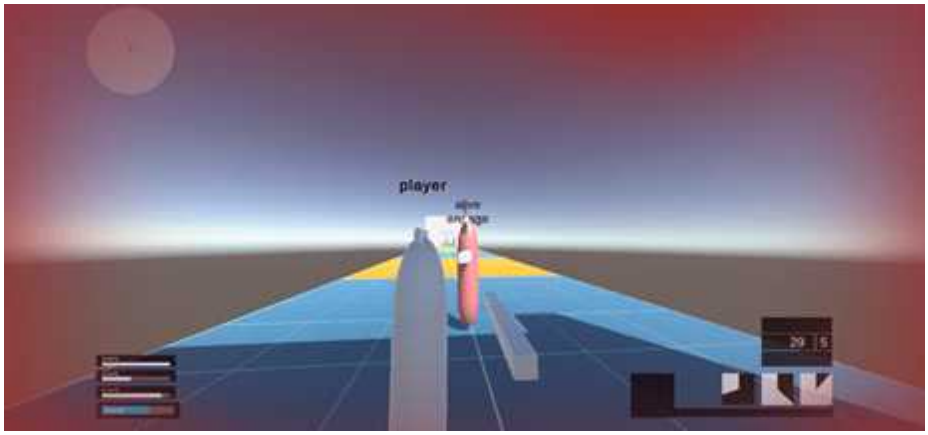
위에서 보여준 것처럼 플레이어는 이러한 방식으로 게임을 진행하며 게임에 마지막에 가면 미션의 성공에 따라 승리 또는 패배씬이 나오면서 게임이 끝나게 된다.

< 그림 21 > 지형 파괴 및 맵 디자인



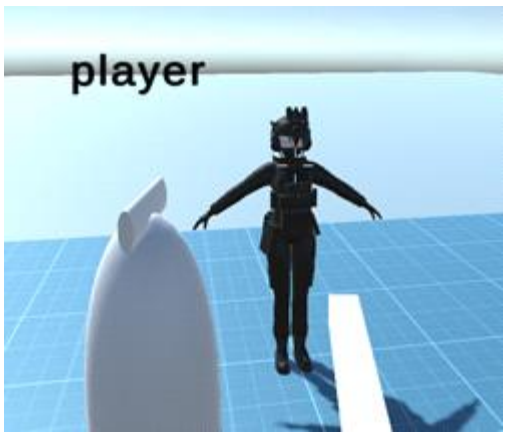
위 그림처럼 게임 내에서 구현할 내용이다. 현제 프로토 타입 버전으로 구현되어 있으며 게임 씬에 적용해야 된다. 플레이어는 실제 총으로 벽을 부수거나 오브젝트를 변경하면서 자신에게 유리한 지형을 만들 수 있고 이를 이용하여 적을 공격 할 수 있다. 또한 아래와 같이 실제 건물 주차장과 같은 배경을 이용함으로 플레이어에게 실제 총기 총기 싸움을 하는 느낌을 줄 예정이다.

< 그림 22 > 게임내 피격 이미지



게임 내 피격 이미지이다. 플레이어가 AI에게 공격을 받으면 플레이어는 위처럼 화면이 변하면서 하단의 체력 바의 체력이 줄어들게 된다.

< 그림 23 > 플레이볼 캐릭터



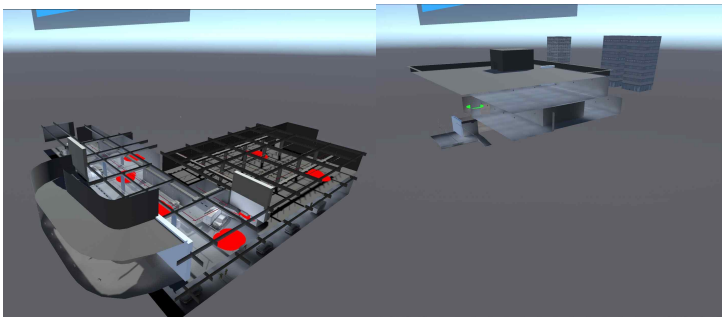
다음은 플레이볼 캐릭터이다. 실제 군인 캐릭처럼 디자인을 하는 것이 목표이며 현제 이미지는 캐릭터에 대한 프로토타입이다.

6. 시험/테스트 결과

6.1. 스테이지 테스트

기능 분류	검증방법	검증결과
최적화	스트리밍 레벨디자인이 적용되는가	O
	배칭 수를 줄여 60fps 이상이 성능을 내는가	O

< 그림 24 > 스트리밍 레벨 디자인



플레이어들의 위치에 따라 레벨 디자인의 일부만 로드된다.
보이지 않는 레벨 디자인을 로드하지 않아 컴퓨터의 성능을 낭비하지 않는다.

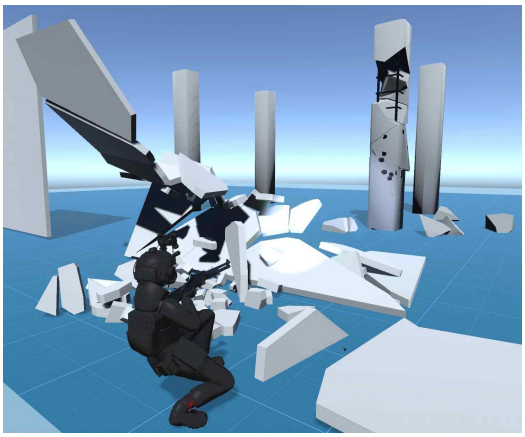
6.2. 무기 테스트

기능 분류	검증방법	검증결과
스프레드 (명중률)	idle 상태에서 사격 시 스프레드 방식이 적용되는가	O
	총기별 서로다른 스프레드 속성을 가지는가	O
화면 반동	zoom 상태에서 사격 시 화면 반동이 적용되는가	O
	사격 후 조준점이 원래 위치로 복귀되는가	O
사격 모드	사격 모드를 변경할 수 있는가	O
	사격 모드에 따라 발사 횟수가 적절히 결정되는가	O
재장전	재장전을 통해 총알과 탄창의 개수가 적절히 변하는가	O
	재장전을 하는 도중 취소 및 재개가 가능한가	O
	약실 구조를 가진 총을 재장전할 때, 약실 탄약 1발이 남는 디테일이 묘사되었는가	O
	탄창식과 샷건식 재장전 방법이 분리되어 구현되었는가	O
데미지 전달	생명체에게 데미지를 전달할 수 있는가	O
	파괴가능한 물체에게 데미지를 전달할 수 있는가	O

< 그림 25 > 스프레드 방식으로 퍼지는 총알

플레이어가 조준한 곳을 중심으로 총알이 랜덤하게 퍼져 나가는 모습이다.
조준하는 곳의 방향에 영향없이, 총기의 서로 다른 성능에 따라 랜덤하게 퍼져 나간다.

< 그림 26 > 파괴 가능한 물체에 데미지 전달

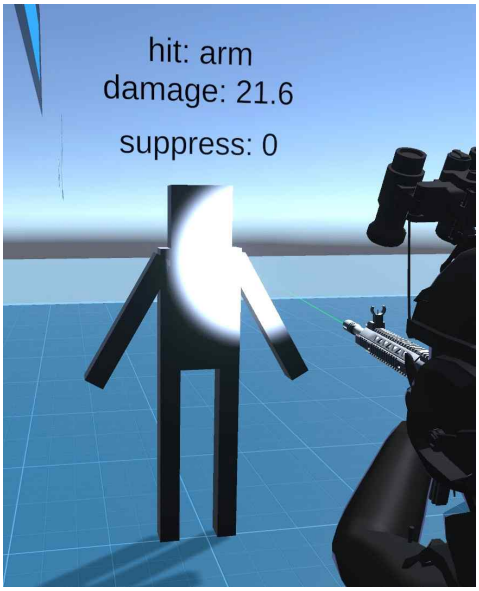


총알이 파괴 가능한 오브젝트에 영향을 주어 실시간으로 물리시뮬레이션을 하는 모습이다.
사전에 오브젝트는 나뉘어져 있으며, 영향을 받은 후에 시뮬레이션이 시작된다.

6.3. 적 AI 테스트

기능 분류	검증방법	검증결과
플레이어 감지	플레이어를 감지할 수 있는가	O
	플레이어 감지에 장애물이 영향을 주는가	O
체력	공격을 통해 체력을 줄일 수 있는가	O
	부위별 데미지를 전달할 수 있는가	O
	죽음의 상태를 가질 수 있는가	O
	죽을 시 래그돌 상태를 가지는가	O
리스폰	적이 리스폰 될 수 있는가	O
	리스폰 기능 작동 시, AI 개체 수가 제한되는가	O
	리스폰 기능이 On/Off 될 수 있는가	O

< 그림 27 > 부위별 데미지



왼쪽 이미지는 부위별 데미지 디버그 물체이며, 오른쪽 이미지는 실제 적 AI에게 적용된 부위별 콜라이더의 모습이다.
이를 통해 움직이는 적에게 부위별로 다른 데미지를 전달할 수 있다.

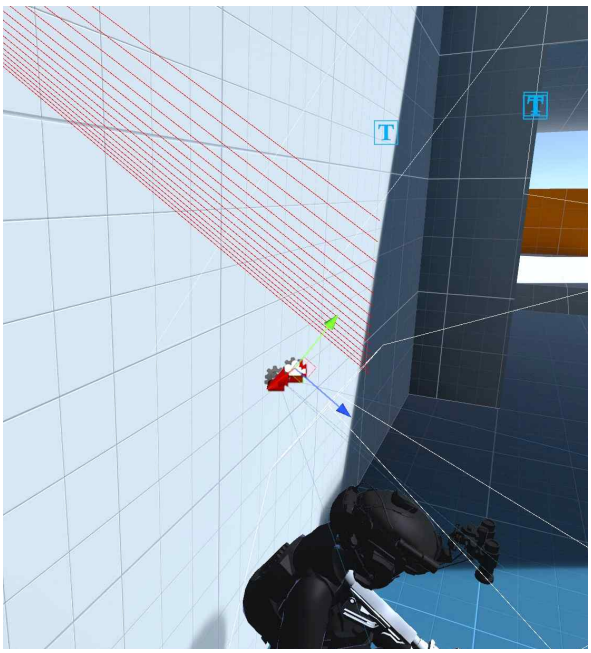
< 그림 28 > 래그돌 상태

적AI가 죽음 상태가 되면 위의 이미지와 같이 레벨 디자인 및 다른 적 AI끼리 실시간 물리 시뮬레이션이 계산된다.

6.4. 플레이어블 캐릭터 테스트

기능 분류	검증방법	검증결과
기본 이동	8방향 이동이 가능한가	O
	이동시 하체가 상체 방향으로 정렬되는가	O
	앞옆뒤의 이동속도가 다르게 구현되었는가	O
	경사로에서도 문제가 생기지 않는가	O
	질주입력이 적절히 감지되는가	O
	W를 떼면 비활성화 되는가	O
	질주상태의 변화에 따라 목표이동속도가 증감하는가	O
카메라	idle, zoom 상태에 따라 카메라 속성이 변경되는가	O
	카메라가 벽을 뚫지 않는가	O

< 그림 29 > 카메라의 벽 감지 및 회피



위의 빨간 선은 카메라가 벽을 감지하는데 사용하는 광선의 경로이다.
경로상에 벽이 감지되면 벽에 영향을 받지 않도록 앞으로 전진한다.

7. 코딩 & 데모

7.1. 적 AI FSM

```
129  #region FSM
130      // 상태를 변경한다.
         5 references
131      private void ChangeState(AIState _state)
132      {
133          if(AICroutine != null) StopCoroutine(AICroutine);//aiState.ToString());
134          aiState = _state;
135          AICroutine = StartCoroutine(aiState.ToString());
136      }
137
```

```
    // 상태변경타이밍을 감지한다
    4 references
    private AIState Transition()
    {
        // 죽음
        if (entityState == EntityState.dead) return AIState.dead;
        // 대기 -> 이동, 전투
        else if (aiState == AIState.wait)
        {
            // 적 감지: 이동
            for (int i = 0; i < players.Length; i++)
            {
                if (players[i] != null)
                {
                    if(animator != null) animator.SetBool("Engage", true);
                    if(lineRenderer != null) lineRenderer.enabled = true;
                    return AIState.move;
                }
            }
            // 그렇지 않다면
            return AIState.wait;
        }
        // 이동 <-> 전투
        else
        {
            // 시야 내에 있고 교전거리 내에 있을 때: 전투
            for (int i = 0; i < 4; i++)
            {
                if (isPlayerOnSight[i] == 1 && playerDistance[i] < engageDistance)
                {
                    return AIState.engage;
                }
            }
            // 그렇지 않다면
            return AIState.move;
        }
    }
```

```
    // 아래의 FSM_행동은 입력-판단-출력의 시퀀스를 가진다.
    0 references
    IEnumerator dead()
    {
        // 시작시 코드
        agent.enabled = false;
        if(stageManager != null && stageManager.isActiveAndEnabled) stageManager.GetAliveEnemyCount();

        if(animator == null) yield break;
        animator.SetBool("DeathBack", true);

        if(lineRenderer != null) lineRenderer.enabled = false;
        if(minimapHolder != null) minimapHolder.SetActive(false);

        if(!useRagDoll) yield break;    // 이하 레그돌 영역
        var ragdollTime = UnityEngine.Random.Range(0f,.75f);
        yield return new WaitForSeconds(ragdollTime);
        CopyCharacterTransformToRagdoll(animModel.transform, ragdollModel.transform);
        Destroy(animMeshRoot);
        Destroy(animModel);
        ragdollModel.SetActive(true);
        ragdollMeshRoot.SetActive(true);

        // 수행중 코드
        yield break;
    }
```

```
0 references
IEnumerator wait()
{
    // 시작시 코드
    // 수행중 코드
    while (true)
    {
        SenseEntity();
        RenewPlayerInfo();

        AIState newState = Transition();
        if(aiState != newState) ChangeState(newState);

        yield return new WaitForSeconds(0.1f);
    }
}
```

```
0 references
IEnumerator move()
{
    // 시작시 코드
    agent.speed = enemyData.MoveSpeed;
    // 수행중 코드
    while (true)
    {
        SenseEntity();
        RenewPlayerInfo();

        AIState newState = Transition();
        if(aiState != newState) ChangeState(newState);

        if(agent.enabled == true)
        {
            agent.SetDestination(players[target].transform.position);
        }
        Attack();
        yield return new WaitForSeconds(0.1f);
    }
}

0 references
IEnumerator engage()
{
    // 시작시 코드
    agent.speed = 0;
    // 수행중 코드
    while (true)
    {
        SenseEntity();
        RenewPlayerInfo();

        AIState newState = Transition();
        if(aiState != newState) ChangeState(newState);

        Attack();
        yield return new WaitForSeconds(0.01f);
    }
}
#endregion
```

7.2. 적 AI 플레이어 감지

```
// 시야내 타겟 감지: 플레이어별로 시야 확인 유무를 저장하고, 1명이라도 시야 내에 있다면 true를 반환한다.
3 references
private bool SenseEntity()
{
    // 시야유무 정보 리셋
    for (int i = 0; i < 4; i++)
    {
        isPlayerOnSight[i] = 0;
    }

```

```
        // 시야 확인
        var colliders = Physics.OverlapSphere(eyeTransform.position, enemyData.EyeDistance, attackTarget);
        foreach (var collider in colliders)
        {
            // 시야 거리 내에서 존재한 상황
            var livingEntity = collider.GetComponent<LivingEntity>();
            if (livingEntity != null && livingEntity.entityState != EntityState.dead)
            {
                var direction = collider.transform.position - eyeTransform.position;
                direction.y = eyeTransform.forward.y;

                // 대기상태라면 시야각 내에 있는지 파악한다.
                if ((aiState == AIState.wait && Vector3.Angle(direction, eyeTransform.forward) < enemyData.FieldOfView * 0.5f)
                    || (aiState != AIState.wait))
                {
                    ShotRay(direction);
                    Debug.DrawRay(ray.origin, ray.direction * 10f, Color.red, 5f);
                    if (hit.transform != null)
                    {
                        if (hit.transform.root.gameObject == collider.gameObject)
                        {
                            int ID = hit.transform.root.gameObject.GetComponent<PlayerController>().ID;
                            if (ID != 0) isPlayerOnSight[ID - 1] = 1;

                            // 플레이어 정보 등록
                            if (players[ID - 1] == null)
                            {
                                players[ID - 1] = hit.transform.root.gameObject;
                                playerState[ID - 1] = players[ID - 1].GetComponent<LivingEntity>();
                            }
                        }
                    }
                }
            }
        }

        // 플레이어 확인시 true 리턴
        for (int i = 0; i < 4; i++)
        {
            if (isPlayerOnSight[i] == 1) return true;
        }
        return false;
    }
```

7.3. 적 AI 최우선 공격 목표 선정

```
    // 각 플레이어와의 거리, 타겟 가중치 계산 및 최우선 공격 대상 선정
    3 references
    private void RenewPlayerInfo()
    {
        // 거리와 가중치 계산
        for (int i = 0; i < 4; i++)
        {
            if (players[i] == null) continue;
            if( playerState[i] == null) playerState[i] = players[i].GetComponent<LivingEntity>();

            playerDistance[i] = (players[i].transform.position - transform.position).magnitude; // 거리 계산

            // 타겟 가중치 계산
            targetWeight[i] = playerState[i].entityState == EntityState.dead ? 0 :  // 죽음상태 가중치
                              (int)((Mathf.Clamp((-1 * playerDistance[i] + 25), 0, 25) * enemyData.DistanceTargetWeight  // 거리 가중치
                              + ((lastAttacker == i + 1) ? enemyData.AttackerTargetWeight : 0))   // 마지막 공격자 가중치
                              * (playerState[i].entityState == EntityState.alive ? 1 : enemyData.DownTargetWeight));   // 다운 상태 가중치
        }

        // 최우선 타겟 선정
        int maxValue = targetWeight.Max();
        target = targetWeight.ToList().IndexOf(maxValue);
    }
```

7.4. 무기 발사

```
    #region 사격
    // 1티어 사격 메소드
    // 발사횟수를 입력받는다: 0_플레이어용, 총기세팅값을 따른다. , 0 이상: AI용
    3 references
    public void Fire(int _fireCount)
    {
```

```
136         // 발사가 가능한지 검사
137         if (weaponData.UseMag && state != State.ready) return;  // 탄창식 장전방식을 사용하고, 발사준비가 안되었을 때 -> 리턴
138         else if (!weaponData.UseMag)  // 샷건식 장전방식을 사용하고, 발사준비가 안되었을 때 -> 리턴
139         {   // 잔탄이 없거나, 사격중이거나, 재장전중이면서 잔탄이 없는 경우
140             if ((state == State.empty || state == State.shooting) || (state == State.reloading && curRemainAmmo == 0))
141                 return;
142         }
143 
144         // 샷건식 장전방식에, 재장전 중이라면, 재장전 코루틴 중지
145         if (!weaponData.UseMag && state == State.reloading)
146         {
147             if (reloadCoroutine != null) StopCoroutine(reloadCoroutine);
148             isRunningReloadCoroutine = false;
149         }
150 
151         if (_fireCount == 0)
152         {
153             switch (weaponData.HavingFireMode[curFireMode])
154             {
155                 case FireMode.automatic:
156                     state = State.shooting;
157                     shotCoroutine = StartCoroutine(Shots(1));
158                     break;
159 
160                 case FireMode.burst:
161                     if (isTriggered == true) return;
162                     state = State.shooting;
163                     shotCoroutine = StartCoroutine(Shots(3));
164                     break;
165 
166                 case FireMode.twice:
167                     if (isTriggered == true) return;
168                     state = State.shooting;
169                     shotCoroutine = StartCoroutine(Shots(2));
170                     break;
171 
172                 case FireMode.single:
173                     if (isTriggered == true) return;
174                     state = State.shooting;
175                     shotCoroutine = StartCoroutine(Shots(1));
176                     break;
177             }
178             isTriggered = true;
179         }
180         else
181         {
182             state = State.shooting;
183             shotCoroutine = StartCoroutine(Shots(_fireCount));
184         }
185     }
```

```
187     // 2티어 사격 메소드: 사격 통제 장치
        6 references
188     IEnumerator Shots(int _remainShotCount)
189     {
190         // 재장전 중일 경우
191         if (isRunningReloadCoroutine == true)
192         {
193             yield break;
194         }
195 
196         // 자동사격이 끝났거나, 총알이 없는 경우
197         if (_remainShotCount <= 0 || curRemainAmmo <= 0)
198         {
199             AmmoCheck();
200             yield break;
201         }
202 
```

```
        // 사격
        if (Time.time >= lastFireTime + fireInterval)
        {
            Shot();

            yield return new WaitForSeconds(fireInterval);
            shotCoroutine = StartCoroutine(Shots(--_remainShotCount));
        }
    }
```

```
// 3티어 사격 메소드: 최종 1회 사격
1 reference
private void Shot()
{
    curRemainAmmo--;
    UpdateUI();

    for (int i = 0; i < weaponData.BallPerOneShot; i++)    // 산탄 방식을 위해, 여러 발 처리
    {
        // 1. 투사체 방식 (폐기)
        // fireDirection = muzzlePosition.eulerAngles;
        // 2. 레이캐스트 방식
        if(playerAttack != null ) fireDirection = (playerAttack.aimTarget.position - muzzlePosition.position).normalized;
        else fireDirection = transform.forward * fittingForward;

        curRecoilX = Random.Range(weaponData.RecoilHorizontal.x, weaponData.RecoilHorizontal.y);
        curRecoilX2 = Random.Range(weaponData.RecoilHorizontal.x, weaponData.RecoilHorizontal.y);
        curRecoilY = Random.Range(weaponData.RecoilVertical.x, weaponData.RecoilVertical.y);
        curRecoilZ = Random.Range(weaponData.RecoilZ.x, weaponData.RecoilZ.y);

        // idle 조준상태 또는 샷건일 경우 -> 화면 반동과 랜더스프레드 적용
        if (isHipFire || weaponData.BallPerOneShot != 1)
        {
            // 화면 반동
            if (playerAttack != null && useRecoilInIdle)
            {
                playerAttack.FireRecoil(new Vector3(-1 * Mathf.Abs(curRecoilY), curRecoilX, curRecoilZ) * weaponData.RecoilMultipleInIdle);
            }

            // 랜덤 스프레드
            fireDirection += new Vector3(curRecoilX, curRecoilY, curRecoilX2) * curSpread * weaponData.RecoilMultiple;
            fireDirection = fireDirection.normalized;
        }
        // zoom 조준 상태일 경우 -> 화면 반동 적용 (플레이어만 가능한 사격 방법)
        else
        {
            playerAttack.FireRecoil(new Vector3(-1 * Mathf.Abs(curRecoilY), curRecoilX, curRecoilZ) * weaponData.RecoilMultipleInZoom);
        }

        curSpread += weaponData.RecoilPerShot;
        if (curSpread > weaponData.MaxSpread) curSpread = weaponData.MaxSpread;

        // 방식1 (폐기)_투사체 발사 처리
        //var bullet = bulletPool.Get();
```

```
        // 방식2_레이캐스트_총구에서 fireDirection방향으로
        // 2.1. 공격처리
        ray.origin = muzzlePosition.position;
        ray.direction = fireDirection;
        if(Physics.Raycast(ray, out hit, bulletPrefab.bulletData.MaxDistance, gunLayerMask)){
            // (1) Damageable 물체 또는 미확인물체
            var target = hit.transform.GetComponent<IDamageable>();
            if( target != null)
            {
                hitPoint = hit.point;

                if(hit.transform.root.tag != weaponUser.tag)
                {
                    DamageMessage damageMessage;

                    damageMessage.attacker = weaponUser;
                    damageMessage.ID = Random.Range(0, 2147483647); // 사용안함_원래는 관통시스템 사용시 중복 공격 방지용이였던 것
                    damageMessage.damageKind = DamageKind.bullet;
                    damageMessage.damageAmount = bulletPrefab.bulletData.Damage;
                    damageMessage.suppressAmount = bulletPrefab.bulletData.Suppress;
                    damageMessage.hitPoint = hit.point;
                    damageMessage.hitNormal =  hit.normal;

                    target.ApplyDamage(damageMessage);
                }
            }
            else hitPoint = hit.point;
        }
```

```
284             }
285             // (2) 제한된 거리에 도달할 경우
286             else{
287                 hitPoint = muzzlePosition.position + fireDirection * bulletPrefab.bulletData.MaxDistance;
288             }
289             #if UNITY_EDITOR
290             Debug.DrawRay(muzzlePosition.position, hitPoint - muzzlePosition.position, Color.green, 1f);
291             #endif
292             // (3) 장애물
293             rayfireTarget.position = hitPoint;
294             rayfireGun.Shoot();
295 
296             // 2.2. 제압처리_끝점까지 제압콜라이더 감지 및 처리
297             var hits = Physics.RaycastAll(muzzlePosition.position, fireDirection, hit.distance, suppressLayerMask);
298             foreach( var j in hits){
299                 var target1 = j.transform.GetComponent<SuppressPoint>();
300                 if( target1 != null)
301                 {
302                     if(j.transform.root.tag != weaponUser.tag) target1.ApplySuppress(bulletPrefab.bulletData.Suppress);
303                 }
304             }
305 
306             // 2.3. 예광탄
307             Bullet2 bullet = Instantiate(raytracerPrefab);
308             bullet.Completed += OnCompleted;
309             bullet.DrawLine(muzzlePosition.position, hitPoint, bulletPrefab.bulletData.Speed, 0);
310         }
311 
312         foreach( var i in shotParticle){ i.Emit(1); }
313         foreach( var i in shotParticle2){ i.Play(); }
314         lastFireTime = Time.time;
```

참고자료

(1) 방승언, "'이게 트리플A?' 2021년, 실망 혹은 만족 안긴 '대작' 타이틀은?", THIS IS GAME, 2021.12.09. https://www.thisisgame.com/webzine/news/nboard/4/?n=138887

(2) 로딩 화면 구현하기(로딩 씬 방식) | 유니티 . (2020). https://www.youtube.com/watch?v=xRiqSmUggpg.

(3) 스트리밍 레벨 기능 구현하기 | 유니티 . (2020). https://www.youtube.com/watch?v=MeNOdL2mg18.

(4) 타임라인 기초 | 유니티 . (2020). https://www.youtube.com/watch?v=jwXW7Q-Bbic.

(5) 액션게임에서 근접무기로 때리는 방법 . (2020). https://openplay.tistory.com/12.

(6) [유니티 3D게임] FPS Prototype #16 무기 교체 시스템 . (2021). https://www.youtube.com/watch?v=Petq5lS68gw.

(7) 무기 전환하기 . (2020). https://openplay.tistory.com/13.

(8) Unity-TPS-Sample . (2019). https://github.com/IJEMIN/Unity-TPS-Sample.

(9) 이런 캐릭터 물리 효과는 어떻게 만들까? 랙돌과 캐릭터 조인트 | 유니티 . (2020). https://www.youtube.com/watch?v=cTHceZpwGt4.

(10) Procedural Recoil System | Unity Tutorial . (2021). https://www.youtube.com/watch?v=geieixA4Mqc.

(11) [#10] Weapon Reloading using animation events in unity . (2020). https://www.youtube.com/watch?v=QCi2GOyHIIk.