Geme Develpment Portfolio

박 완 우

목 차

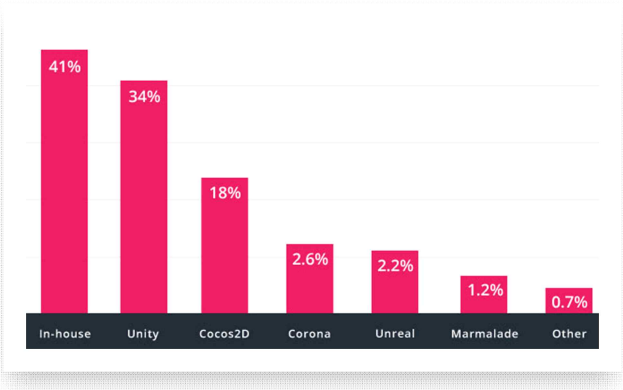
동영상 링크
Z-World : https://vimeo.com/286297650
Spell Selector : https://vimeo.com/286297760
Prize Pang : https://vimeo.com/286297826

1. 제작 동기

현재 대한민국을 이끄는 주요 문화 콘텐츠 산업에 대해 이야기한다면 단연 게임 산업을 빼놓을 수 없습니다. 하지만 게임 강국이라는 타이틀에 어울리지 않을 정도로 한국에서 개발하는 게임들 중 글로벌 시장에서 큰 영향력을 미친 게임은 존재하지 않습니다. 하지만 최근 '배틀그라운드'의 성공 신화를 통해 한국 게임 개발사에서도 글로벌 시장에서 통하는 게임을 개발하고 운영할 수 있다는 사실이 밝혀졌습니다. 대한민국 게임업계의 혁명이라고 할 수 있는 '배틀그라운드'의 게임 시스템과 구성 요소를 차용하여 일종의 모작을 만들어 봄으로써 앞으로 게임 개발자로서 수 많은 게임을 개발하는데 있어 기초적인 역량을 기르기 위해 본 프로젝트를 진행하게 되었습니다.

2. 개발 환경

가. Unity Engine (2017.2.0f3)



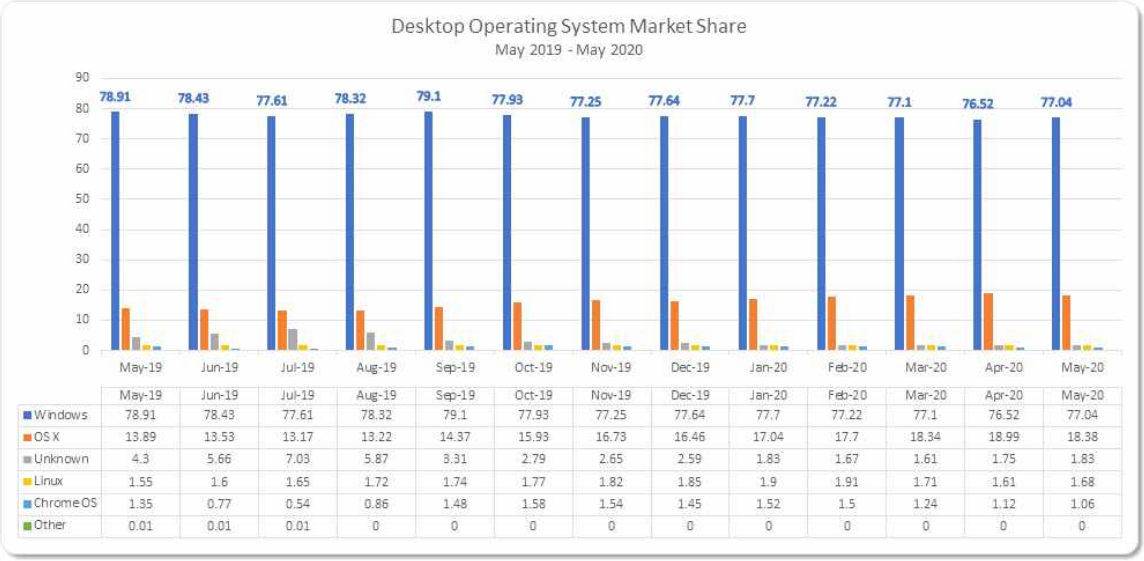
상위 무료 모바일게임에 사용된 게임 개발 엔진 점유율
출처 : Unity 홈페이지

최근 몇 년간 가장 짧은 기간 내에 가장 높은 인기와 관심을 받은 게임 엔진을 꼽으라면 Unity를 꼽을 수 있습니다. 아이폰을 비롯한 스마트폰의 등장으로 스마트폰 게임이라는 새로운 게임 플랫폼이 구축이 되었고 이를 바탕으로 한 새로운 시장이 형성되었다. 이 새로운 시장은 누구나 자유롭게 게임을 만들고 사용화 할 수 있는 기회를 제공하였으며, 게임을 만들기 위해 설립된 게임 개발사뿐만 아니라 개인이 스스로 개발한 콘텐츠를 통한 수익 창출의 길을 열어 주었고, 이는 곧 다양한 분야에 종사하던 개발자들이 스마트폰 플랫폼에서 게임을 개발하기 위에 뛰어들게 만드는 계기가 되었습니다. 이에 따라 기존 게임 개발자들과 더불어 게임을 개발한 경험이 없는 개발자들이 새로운 플랫폼을 지원하는 게임을 개발하기 위한 사용하기 쉽고 안정적인 개발 엔진이 필요하게 되었고, 이는 곧 Unity 개발엔진의 주목으로 이어지게 되었습니다. Unity 엔진의 가장 큰 장점은 다양한 플랫폼을 지원하면서 동시에 초보자도 쉽게 사용할 수 있다는 점입니다. 이러한 장점을 통해 게임 개발자들은 새로운 플랫폼을 위해 새로운 작업을 할 필요가 없어지게 되었으며 Unity 자체 제공 소스를 이용하여 다양한 결과를 도출 할 수 있게 되었습니다.

Unity 엔진은 기본적으로 엔진 자체에서 지원하는 MonoDevelop 에디터를 이용하여 엔진의 모든 기능을 자유롭게 사용할 수 있습니다. 게임의 콘텐츠 제작은 물론, 애니메이션과 이벤트 설정 등의 다양한 기능을 구현할 수 있음은 물론이고 에디터를 통해 제작된 게임의 정보를 실시간으로 플레이 해봄으로써 콘텐츠 개발 중에 보다 신속한 디버깅이 가능합니다.

이러한 편리함에 더불어 엔진 내부의 정보를 전혀 모르는 상태에서도 에디터에서의 데이터 조작만으로 빠른 시간 내에 게임을 만들어 낼 수 있으며 이 결과 역시 바로 실행하여 플레이 해봄으로써 확인해 볼 수 있다는 점을 통해서 게임 개발 프로젝트를 진행하는데 Unity 엔진을 사용하게 된 계기가 되었습니다.

나. Window 10



	May-19	Jun-19	Jul-19	Aug-19	Sep-19	Oct-19	Nov-19	Dec-19	Jan-20	Feb-20	Mar-20	Apr-20	May-20
Windows	78.91	78.43	77.61	78.32	79.1	77.93	77.25	77.64	77.7	77.22	77.1	76.52	77.04
OS X	13.89	13.53	13.17	13.22	14.37	15.93	16.73	16.46	17.04	17.7	18.34	18.99	18.38
Unknown	4.3	5.66	7.03	5.87	3.31	2.79	2.65	2.59	1.83	1.67	1.61	1.75	1.83
Linux	1.55	1.6	1.65	1.72	1.74	1.77	1.82	1.85	1.9	1.91	1.71	1.61	1.68
Chrome OS	1.35	0.77	0.54	0.86	1.48	1.58	1.54	1.45	1.52	1.5	1.24	1.12	1.06
Other	0.01	0.01	0.01	0	0	0	0	0	0	0	0	0	0

출서 : STATCOUNTER

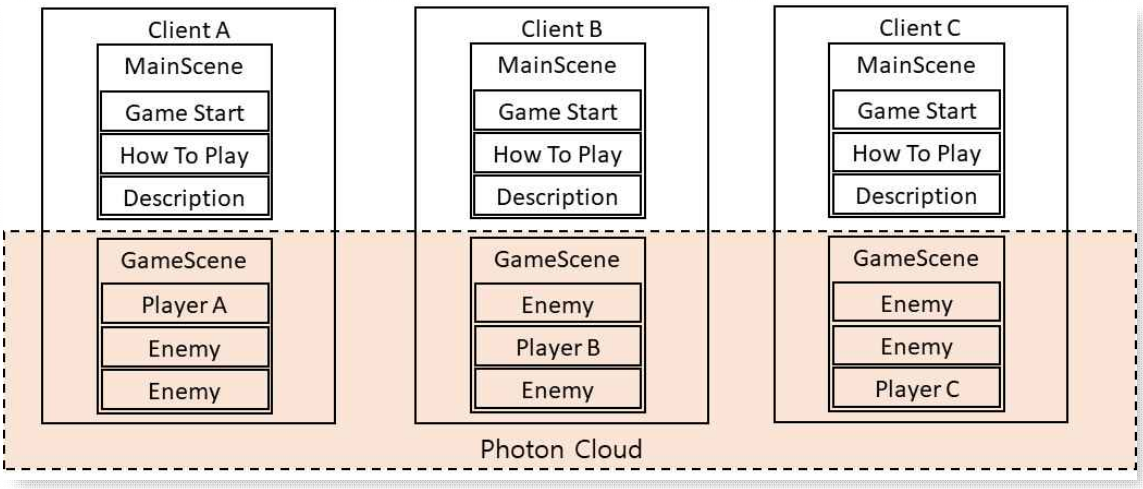
Windows OS는 20년 5월 현재 기준 가장 많이 사용되고 있는 Desktop OS입니다. 20년 5월 현재 77.04 퍼센트를 기록하고 있으며 타 OS에 비해 압도적으로 높은 점유율을 기록하고 있습니다. 위 통계를 통해서 단순한 게임 환경을 제외 하고도 많은 비율의 Desktop 보유자들이 Windows OS를 선택하고 사용하고 있는 현재, 최대한 많은 사람들이 즐길 수 있는 접근성 높은 게임을 만들기 위해서는 Windows OS 기반 게임을 제작하는 것이라는 결론을 내렸습니다.

다. Visual Studio 2017

Unity Engine 에서는 자체적으로 MonoDevelop 이라는 개발 에디터를 제공합니다. 하지만 기존에 컴퓨터 공학 전공 수업을 수강하고 다양한 프로그램 코드를 작성하기 위해 지속적으로 Visual Studio를 사용했고, 이에 대한 익숙함을 통해 보다 효율적은 프로젝트의 진행이 가능할 것이라고 판단했습니다.

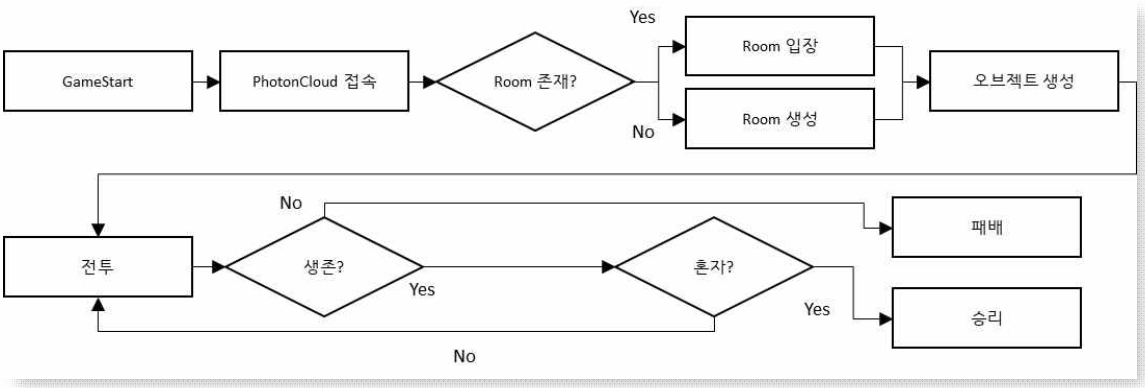
3. 게임 설명

가. 구성도



나. 이벤트 전송

다. 흐름도



게임은 독립된 여러 Client의 이벤트가 Photon Cloud에게 전달되어 처리하고 그 정보를 다시 각 Client들에게 전달함으로서 기본적으로 실시간 멀티플레이가 가능하도록 설계되었습니다.

각 Client에서 발생한 RPC(Remote Procedure Call) 이벤트는 Photon Cloud를 통해 서버에서 함수가 수행되어 Photon Cloud를 통해 연결된 모든 Client들에게 실시간으로 전송됩니다.

각 Client를 실행 후, Game Start 버튼을 클릭하게 되면 Photon Cloud 접속 과정을 거쳐 Room에 입장하게 됩니다. 여기서 기존의 Room이 존재하지 않는다면 최초 접속자가 새로운 Room을 생성하게 되고 이후의 접속자들은 이렇게 생성된 Room에 제한된 인원만큼 순서대로 접속하게 됩니다.

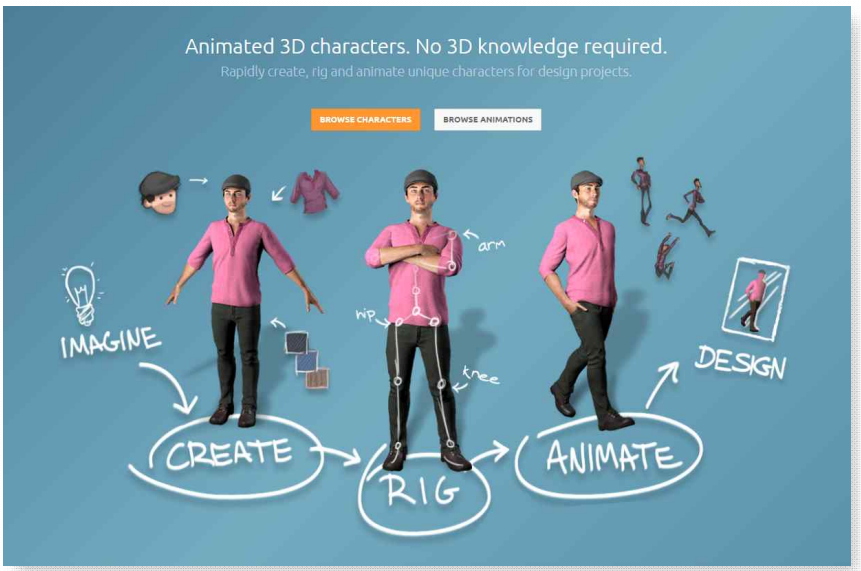
Room에 이미 접속 중인 사람들은 지형을 자유롭게 돌아다닐 수 있으나 아이템과 적대적 Object(Zombie)는 생성되지 않습니다. 하나의 Room에 게임 시작 기준 인원이 모이게 되면 게임은 자동으로 시작되며, 시작과 동시에 맵 전역에 아이템과 Zombie들이 생성되고 게임이 시작됩니다.
이후의 게임 진행은 모두 플레이어의 행동에 따라 달라집니다. 플레이어는 전투를 회피할 수 있으며, Zombie와 싸워 이겨 더 많은 양의 탄약을 얻고 이를 토대로 강한 화력을 통해 승리를 거머쥘 수 있습니다. 또한 사격 솜씨에 자신이 있다면 시작과 동시에 다른 플레이어와 싸움을 벌여 경쟁자를 빠르게 줄일 수도 있습니다.

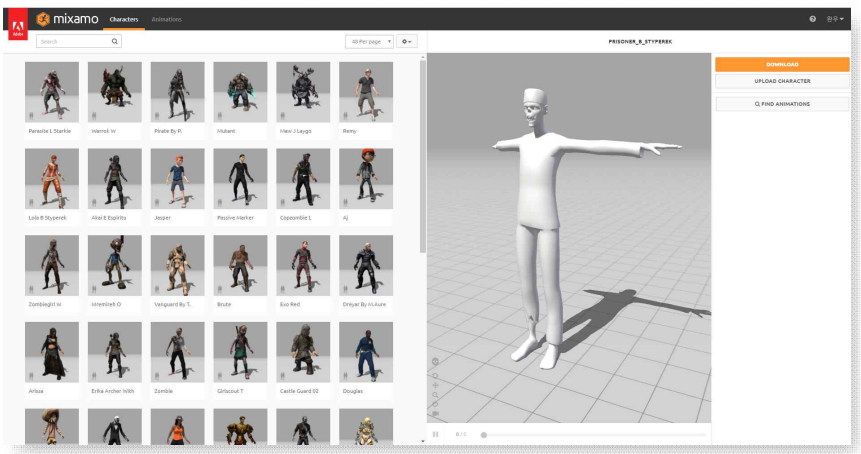
4. 소요 기술

가. 리소스

1) 3D Models

Adobe 사의 Mixamo 사이트에서 제공하는 게임 플레이에 사용되는 Character Player, Zombie, Item 등의 fbx 모델을 사용했습니다.

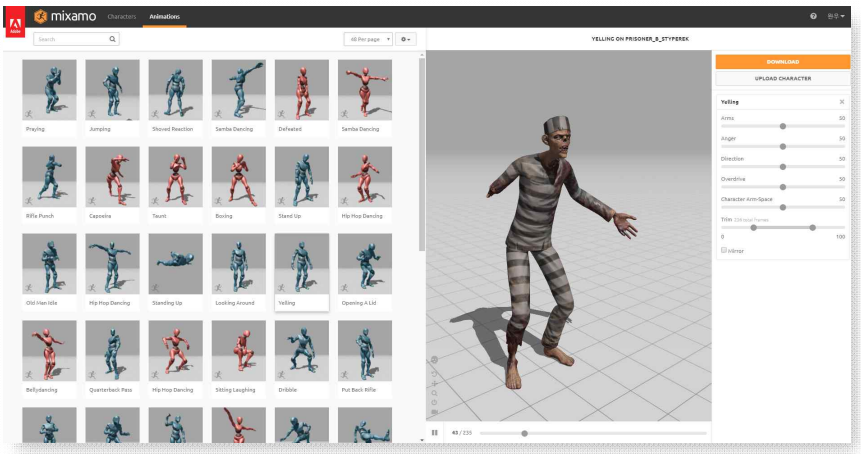




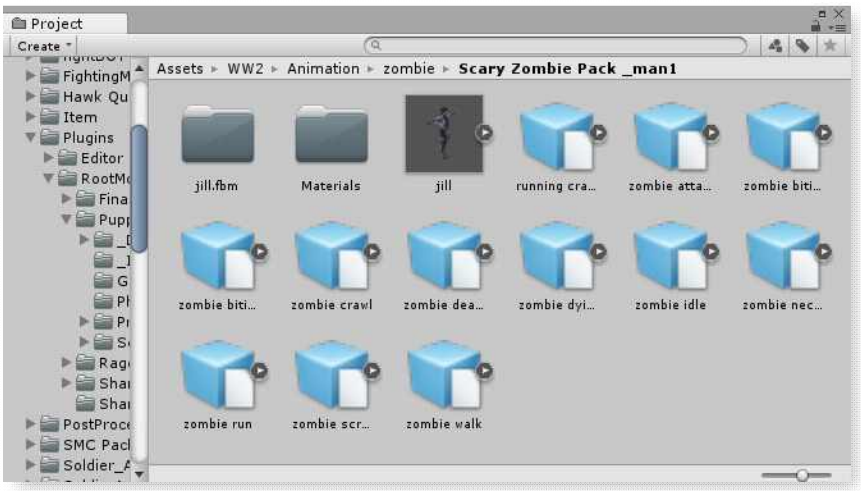
게임에 사용될 Humanoid 모델 선택화면

2) 3D Animation

Mixamo 사이트에서 제공하는 다양한 Humanoid Animation을 사용했습니다.



원하는 3D Model에 Animation을 선택하여 적용



해당 리소스를 Unity 프로젝트에 Import

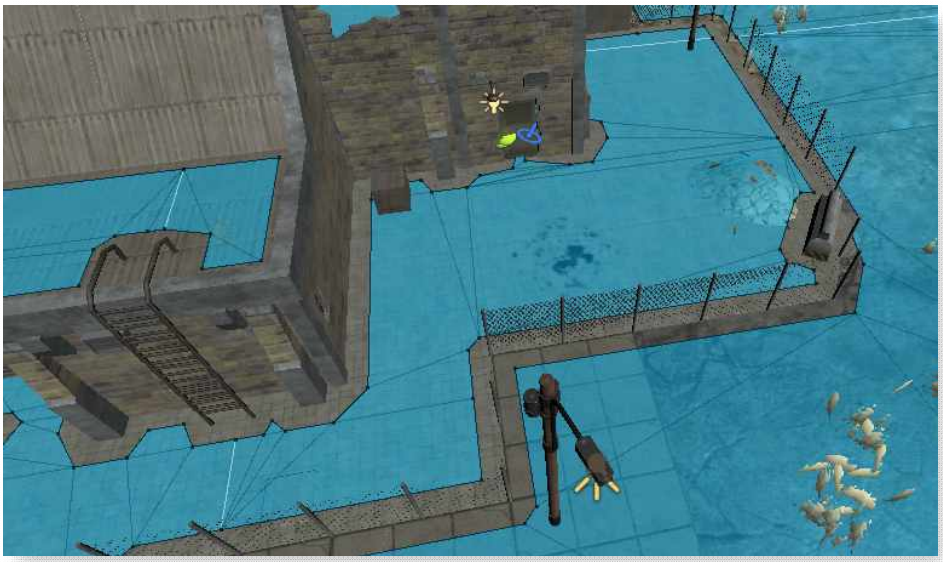
나. NavMesh

1) NavMesh란?

NavMesh란, Unity Engine에서 자체적으로 제공하는 길 찾기 알고리즘입니다.
기본적으로 Unity의 NavMesh는 A-Star 알고리즘을 바탕으로 설계되었습니다.

2) NavMesh 구현

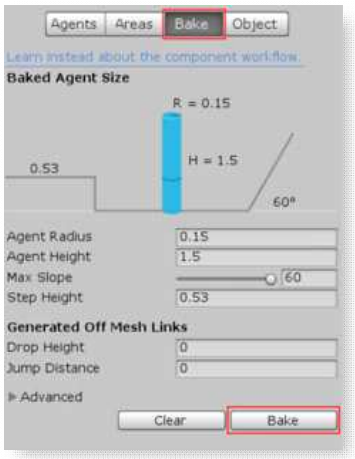
Unity Engine에서 제공하는 NavMesh 기능을 사용하여 Zombie Object의 이동 경로 탐색 기능을 구현했습니다.



Unity NavMesh를 이용해 이동가능 지역을 Bake한 상황

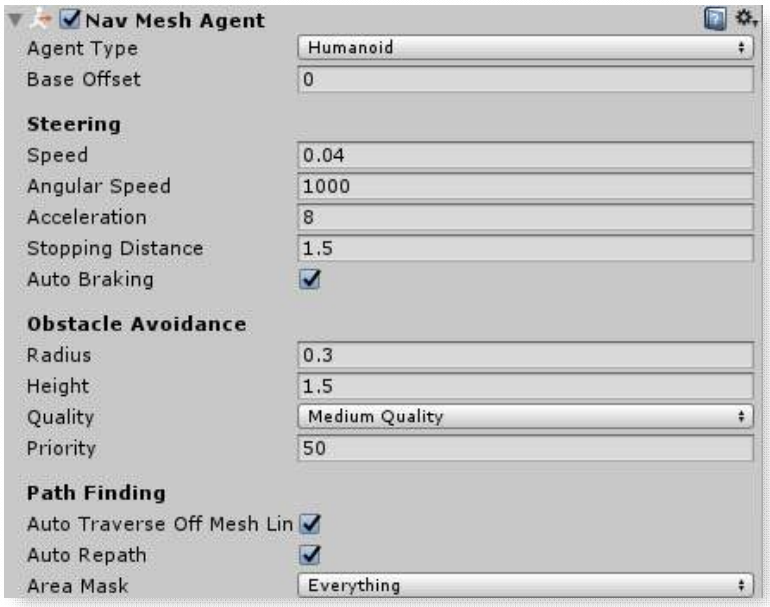
NavMesh를 사용하기 위해서는 우선 지형 지물에 대한 Bake를 수행해야 합니다.

이 과정에서 Unity Engine은 지형지물에 대한 Walkable Surface를 판단하여 파란 화면으로 표시해 줍니다. 사용자는 Bake 이전에 Agent Radius, Height, Max Slope, Step Height 항목들의 수치를 설정하여 NavMesh를 적용할 Agent의 키와 반경, 이동 가능한 경사로의 각도 및 계단의 높이 등을 상세하게 설정할 수 있습니다.



NavMesh Bake 설정 항목
출처 : Unity Documentation

또한, Bake된 지형을 이동할 Object에게 NavMesh Agent Component를 등록하여 NavMesh Agent가 제공하는 다양한 함수, 변수들을 사용하여 구체적인 이동 목표, 속도, 회전 각 등을 설정하여 원하는 움직임을 구현할 수 있습니다.



Object에 등록된 Navmesh Agent Component

```
using UnityEngine.AI;
```

```
void Start()
{
    navAgent = GetComponent<NavMeshAgent>();
```

UnityEngine.AI Module 사용하겠다고 선언 후
navAgent 변수에 NavMeshAgent 컴포넌트 할당하여 Agent로서 활용합니다.

```
void chasingTarget()
{
    if(navAgent.isStopped == true)
    {
        navAgent.isStopped = false;
    }

    navAgent.SetDestination(movePoint.position);
```

ChasingTarget() 함수가 실행 될 경우, navAgent가 비활성 상태라면 활성화 후 SetDestination 명령어를 통해 새로운 목적지를 할당합니다.
SetDestination 함수에는 목적지의 좌표 혹은 추적을 원하는 Object의 위치 좌표가 인수로 포함되며, 이는 이동 중에 지속적으로 갱신되어 이동 중인 목표물을 계속해서 추적할 수 있게 해줍니다.

다. Animation

1) Mecanim Animation

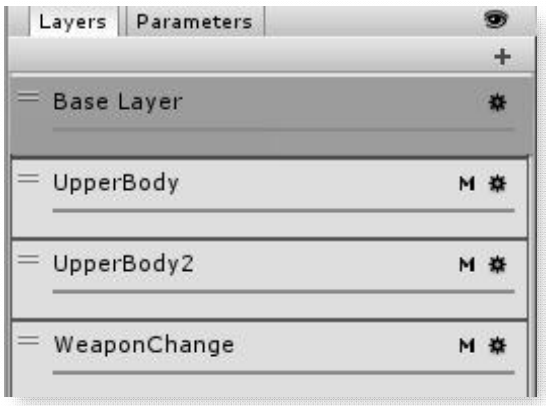
Unity Engine에서 기본 제공하는 기능인 Mecanim Animation을 사용하여 Player Character, Zombie Object의 Animation을 구현했습니다. Mixamo에서 제공하는 3D Model에 가장 최적화된 Animation을 활용하기 위해 각 Model별로 Generic Animation을 적용했습니다.

2) Animation Mask & Layer

게임을 플레이하다 보면 캐릭터의 신체 부위별로 각각 다른 애니메이션을 재생해야 하는 경우, 예를 들어 달리면서 검을 휘두르는 경우 하체는 달리고 있는 Animation을, 상체는 검을 휘두르는 Animation을 재생하며 두 Animation이 하나의 Object에서 동시에 재생해야 하는 상황이 발생합니다. 본 게임에서는 이러한 상황이 바로 Player Character가 서서 조준하는 경우와 앉아서 조준하는 경우, 하체는 서있거나 앉아있는 Idle Animation을 재생하고 상체는 플레이어가 조준하고 있는 위치의 높낮이에 맞춰 조준 Animation을 재생해야 하는 경우입니다.

Animation Layer의 경우는 두 가지 경우에 사용할 수 있는데, 첫 번째는 Animation weight에 차이를 두어 Animation을 재생하는데 우선순위를 부여하여 정확한 상황에 정확한 Animation을 재생하는 방법이고, 두 번째는 현재 재생되고 있는 다른 Layer의 Animation과 함께 재생할 수 있는 방법입니다.

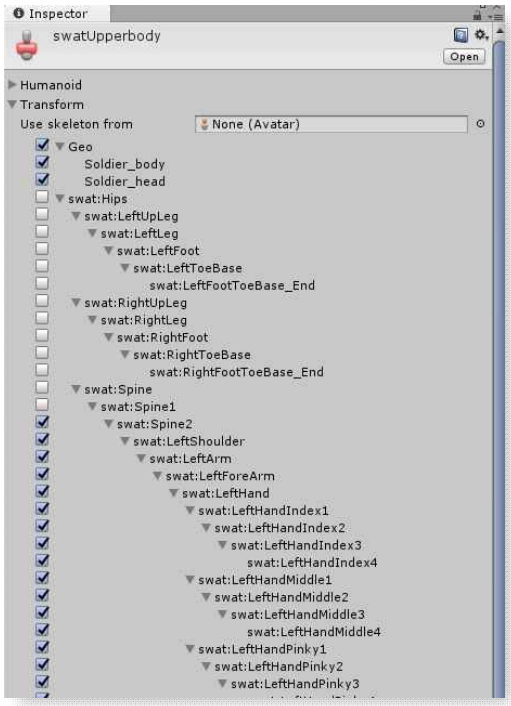
본 게임에서 Player Character Object는 총 4개의 Layer를 가지고 있으며 BaseLayer는 몸 전체, UpperBody, UpperBody2는 각각 라이플과 권총의 상체 동작, WeaponChange는 무기를 교체하는 동작을 표현하기 위해 사용하고 있습니다.
각 Layer의 weight는 모두 1로 두어 동일한 우선순위를 부여하였으며 Override 설정을 적용해 각 Layer의 Animation이 재생 될 경우 동시에 재생되고 있던 다른 Animation은 모두 무시됨으로써 좀 더 자연스러운 Animation을 구현했습니다.



플레이어 오브젝트의 애니메이션 레이어

Avatar Mask는 Animation을 재생할 때 움직임을 할당하고 싶은 부분을 직접 설정할 수 있는 기능으로 Animation Layer에서 설정하여 사용합니다.
본 게임에서 Player Character Object에 적용된 Layer 중 BaseLayer을 제외한 나머지 Layer들은 모두 상체만 재생하는 Animation이므로 그림과 같이 하체를 제외한 상체 부분만 활성화한 Avatar Mask를 적용했습니다.

이를 통해 걷기, 달리기, 앉기 등의 Animation을 재생 중에 상체의 Animation을 덮어씌움으로서 상체는 독립적인 Animation 재생이 가능하게 됩니다.



플레이어 오브젝트의 BaseLayer를 제외한 UpperBody, UpperBody2, WeaponChange 레이어에 적용된 Avatar Mask

BaseLayer를 제외한 나머지 레이어들의 설정.
swatUpperbody 아바타 마스크가 적용,
Weight는 1로 설정되어 있고, Blending 옵션이 Override로 설정되어 있는 것을 볼 수 있습니다.

라. 총알 발사 구현

1) Raycast & Coroutine

Raycast는 해석 그대로 가상의 광선을 발사하여 충돌한 Object와 상호작용 할 수 있도록 Unity Engine에서 제공하는 기술로 무궁무진한 활용성을 가지고 있습니다. 본 게임에서는 해당 기술을 통해 총알 발사를 구현하였으며, 총알을 발사할 때 마다 플레이어가 조준하고 있는 장소에 Ray를 발사, 충돌한 Object가 Zombie/EnemyPlayer 라면 해당 Object의 체력을 감소시키고, 적이 아닌 지형지물이라면 탄착지점에 구멍이 일정 시간동안 생성되도록 구현했습니다.

```
if (Input.GetMouseButton(0) && Time.time > nextFire)
{
    if (weaponHandler.GetComponent<WeaponHandler>().isReloading == false)
    {
        nextFire = Time.time + (1.0f / fireRate);

        if (GameObject.Find("Animation Controller").GetComponent<Animator>().GetBool("Aiming") == true)
        {
            StartCoroutine(Fire());
        }
    }
}
```

발사 모드가 연사일 경우

연사 모드의 경우에는 마우스 버튼을 누른 상태로 유지하는 동안 Fire() 라는 이름의 Coroutine을 반복 호출함으로써 연속된 발사를 구현했습니다.

nextFire와 fireRate변수는 총기의 연사 속도를 위해 사용한 변수로, Time.time이 nextFire보다 클 경우에만 Fire() 함수를 실행하게 됩니다. 따라서 fireRate 변수를 통해 각 총기의 연사속도를 다르게 설정할 수 있습니다.

또한, 장전 중 에는 총알을 발사할 수 없어야하고 조준한 상태에서만 총을 발사할 수 있어야 하기 때문에 isReloading 변수가 False일 경우, 그리고 Aiming 변수가 True일 경우에만 Fire() 함수를 실행하게 됩니다.

```
IEnumerator Fire()
{


    Ray();
    spread.sSpread += spread.spreadPerShooting;
    Muzzle.GetComponent<ParticleSystem>().Play();
    loadedBullet -= 1;
    yield return new WaitForSeconds(1.0f);


}
```

Update()에서 조건 만족시 호출되는 코루틴 Fire() 함수

Fire() 함수가 한 번 실행될 때 마다 Ray를 발사하는 Ray() 함수가 한 번 실행되고, 그에 맞는 Particle Effect를 한 번 재생하면서 현재 장전된 총알을 1 감소시킵니다.
Fire() 함수가 한 번 호출 될 때 마다 Ray를 발사하게 되는데, 그 방향인 FireDir을 그림과 같이 선언했습니다. 여기서 AimTarget은 PlayerCharacter가 조준하고 있는 지점을 의미하고 firePos는 실제 총알이 발사 되기 시작하는 지점을 의미합니다.

따라서 Ray()함수가 한 번 실행 될 때 마다 [AimTarget - firePos] 의 Vector 값을 통해 Ray 발사 방향을 지정하고 해당 지점에 충돌 검사를 실행하게 됩니다.

```
public void Ray()
{
    Damage = Random.Range(MinDamage, MaxDamage);
    RaycastHit hit;
    FireDir = AimTarget.transform.position - firePos.transform.position;
    Vector3 v = Random.insideUnitSphere * spread.sSpread * 0.0015f;

    if (Physics.Raycast(firePos.transform.position, FireDir.normalized + v, out hit, Mathf.Infinity, rayHitRayer))
    {
        Debug.DrawRay(firePos.transform.position, FireDir.normalized + v, Color.red, Mathf.Infinity);
        if (hit.collider.tag == "Enemy")
        {
            hitReaction = hit.transform.root.GetComponent<HitReaction>();
            hitReaction.Hit(hit.collider, transform.forward * (hitForce/10), hit.point);
            CreateBlood(hit);
            hit.transform.root.SendMessage("HitByPlayer", Damage, SendMessageOptions.DontRequireReceiver);
            hit.transform.root.SendMessage("setTarget", Player, SendMessageOptions.DontRequireReceiver);

        }
        else if(hit.collider.tag == "EnemyHead")
        {
            hitReaction = hit.transform.root.GetComponent<HitReaction>();
            hitReaction.Hit(hit.collider, transform.forward * (hitForce / 10), hit.point);
            CreateBlood(hit);
            hit.transform.root.SendMessage("HitByPlayer", Damage, SendMessageOptions.DontRequireReceiver);
            hit.transform.root.SendMessage("HitByPlayer", Damage, SendMessageOptions.DontRequireReceiver);
            hit.transform.root.SendMessage("setTarget", Player, SendMessageOptions.DontRequireReceiver);

        }
        else if (hit.collider.tag == "PlayerBody")
        {
            Debug.Log("Attack Player!");
            hitReaction = hit.transform.Find("CharacterController").GetComponent<HitReaction>();
            hitReaction.Hit(hit.collider, transform.forward * (hitForce / 10), hit.point);

            hit.transform.Find("CharacterController").SendMessage("HitByPlayer", Damage, SendMessageOptions.DontRequireReceiver);

            CreateBlood(hit);
        }
        else if (hit.collider.tag == "PlayerHead")
        {
            hitReaction = hit.transform.Find("CharacterController").GetComponent<HitReaction>();
            hitReaction.Hit(hit.collider, transform.forward * (hitForce / 10), hit.point);
            CreateBlood(hit);
            hit.transform.Find("CharacterController").SendMessage("HitByPlayer", Damage, SendMessageOptions.DontRequireReceiver);
            hit.transform.Find("CharacterController").SendMessage("HitByPlayer", Damage, SendMessageOptions.DontRequireReceiver);
        }
        else
        {
            CreateBulletHole(hit);
        }
    }
}
```

코루틴을 통해 실행되는 Ray() 함수

2) GUI 설정을 통한 반동 구현

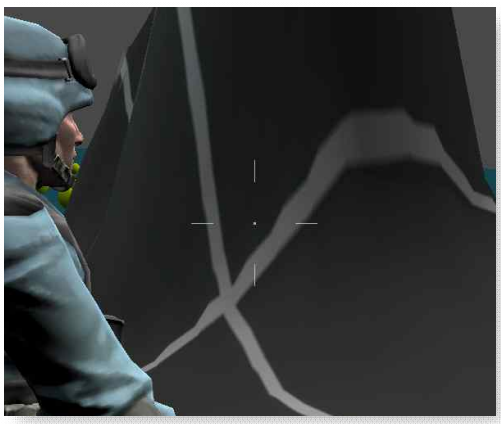
```
void OnGUI()
{
    float screenRatio = Screen.height / 100;
    newHeight = height + screenRatio;

    if (drawCrosshair)
    {
        GUI.Box(new Rect(centerPoint.x - 1, centerPoint.y - 1, 3, 3), GUIContent.none, lineStyle);
        GUI.Box(new Rect(centerPoint.x - (width / 2), centerPoint.y - (newHeight + spread.sSpread), width, newHeight), GUIContent.none, lineStyle);
        GUI.Box(new Rect(centerPoint.x - (width / 2), (centerPoint.y + spread.sSpread), width, newHeight), GUIContent.none, lineStyle);
        GUI.Box(new Rect((centerPoint.x + spread.sSpread), (centerPoint.y - (width / 2)), newHeight, width), GUIContent.none, lineStyle);
        GUI.Box(new Rect(centerPoint.x - (newHeight + spread.sSpread), (centerPoint.y - (width / 2)), newHeight, width), GUIContent.none, lineStyle);
    }
    if (Input.GetKey(KeyCode.W) || Input.GetKey(KeyCode.A) || Input.GetKey(KeyCode.S) || Input.GetKey(KeyCode.D))
    {
        spread.sSpread -= spread.decreasePerSecond * Time.deltaTime * 0.5f;
        spread.sSpread += spread.decreasePerSecond * 0.05f;
    }
    if (Input.GetKey(KeyCode.Space))
    {
        spread.sSpread = spread.maxSpread;
    }
    else
    {
        spread.sSpread -= spread.decreasePerSecond * Time.deltaTime * 1;
    }
    spread.sSpread = Mathf.Clamp(spread.sSpread, spread.minSpread, spread.maxSpread);

}
```

화면 UI 조작을 가능케 하는 OnGUI() 함수

OnGUI() 함수는 화면에 나타나는 Interface를 Script를 통해 조작할 수 있도록 Unity에서 제공하는 함수입니다. 현재 상단에 drawCrosshair 변수를 최초로 선언할 때 Ture로 설정했기 때문에 해당 Scrip를 가지고 있는 Object가 활성화 된 상태라면 if 조건문이 실행됩니다. 조건문 안에는 얇은 직사각형 형태의 박스를 4 방향으로 그려주고, 가운데 점을 찍어 흔히 FPS/TPS 게임에 사용되는 크로스헤어를 표현했습니다.



OnGUI() 함수 안에서 그려진 크로스바

함수 내에 크로스헤어를 그려주는 명령어를 살펴보면 해당 Box를 그려주는 변수에 spread.sSpread 라는 변수가 사용된 것을 볼 수 있습니다. 이는 현재 반동의 수치(sSpread)를 이용해 각 상, 하, 좌, 우 크로스헤어의 위치를 조작하기 위한 것입니다. 위에 언급한 Fire()함수를 다시 살펴보면 함수를 호출할 때 마다 sSpread 변수의 값을 spreadPerShooting 만큼 증가시키고 해당 값을 현재 GUI에 그려지고 있는 크로스헤어에 반영함으로써 firePerShooting 변수의 값을 설정하여 한 발당 증가하는 반동의 쉬를 제어할 수 있습니다.

```
FireDir = AimTarget.transform.position - firePos.transform.position;
Vector3 v = Random.insideUnitSphere * spread.sSpread * 0.0015f;

if (Physics.Raycast(firePos.transform.position, FireDir.normalized + v, out hit, Mathf.Infinity, rayHitRayer))
{
```

Ray() 함수의 반동 구현 부분

Ray() 함수에도 이를 적용했는데, FireDir Vector값에 sSpread 값에 따라 그 지름이 증가하는 가상의 구체의 한 지점에 랜덤하게 설정되는 Vector를 더해 줌으로써 총을 쏠 때 마다 sSpread변수가 spreadPerShooting만큼 증가하게 되고, 이는 탄착 지점의 구체의 지름을 키워 실제 총알의 랜덤한 탄착 반경이 반동이 커짐에 따라 함께 커지게 되는 것을 구현했습니다.

발사 초기

사격을 지속해 총기의 반동이 커져있는 상황

사진을 보면 GUI의 크로스헤어가 넓어짐에 따라 총알의 탄착 반경이 넓어지는 것을 확인할 수 있습니다.

마. Raycast를 이용한 Camera Wall Check

많은 게임에서 그러하겠지만, 특히 캐릭터의 위치와 화면을 보여주는 카메라의 위치가 다른 TPS 게임에서는 카메라와 캐릭터 사이의 장애물 처리 방식에 대한 별도의 장치가 필요합니다. 아무런 장치 없이 구현하게 된다면 캐릭터와 카메라 사이의 장애물 때문에 플레이어의 즐거운 게임 경험이 방해될 수 있습니다. 이러한 현상을 해결하기 위해 상술한 Raycast를 활용하여 PlaterCharacter Object와 카메라 사이에 Ray를 발사하여 장애물을 탐지하고, 장애물이 존재한다면 카메라의 위치를 해당 장애물 앞 쪽에 위치함으로서 어떠한 장애물이 존재하는 경우에도 플레이어는 자신의 캐릭터를 볼 수 있도록 구현했습니다.

```
void cameraRay()
{
    RaycastHit hit;
    Vector3 dir = cam.transform.position - cameraTarget.transform.position;
    Vector3 v = cameraTarget.transform.position - cam.transform.position;
    float dist = cam.distance;

    if (Physics.Raycast(cameraTarget.transform.position, dir, out hit, dist-0.6f, wallLayers))
    {
        Debug.DrawRay(cameraTarget.transform.position, dir, Color.green);
        Vector3 a = (cameraTarget.transform.position - hit.point)*0.1f;
        cam.WallCheck(dir, v, hit);
    }
    else
    {
        Debug.DrawRay(cameraTarget.transform.position, dir, Color.blue);
    }

}
```

Update()에서 지속적으로 CameraRay() 함수를 호출

```
public void WallCheck(Vector3 V1, Vector3 V2, RaycastHit H)
{
    transform.position = H.point + (V2*0.1f);
}
```

CameraRay()에서 발사한 레이저가 장애물과 충돌할 경우 호출되는
WallCheck 함수. 여기서는 카메라의 position을 조정합니다.

바. Zombie

1) 감지 범위

Zombie들은 언제나 플레이어를 추적하지 않습니다. 그들은 평소에는 얌전히 있으며 자신이 공격할 수 있는 대상, 즉 PlayerCharacter Object가 그들의 시야 범위 내에 접근해야만 공격적인 행동을 띄어야 합니다. 이를 구현하는 방법은 많이 존재하겠지만, 본 게임에서는 Trigger 기능을 사용했습니다.

Zombie Object의 하위에 MissingRange와 DetectRange Object를 각각 추가하고 Collider Component를 추가하여 Trigger로 설정하여 Collider Componect가 물리 법칙을 반영하는 상호작용을 하지 않도록 설정했습니다.

DetectRange Trigger는 공격 가능한 Target이 Trigger 범위에 들어오기 전 까지 활성화 되어 있습니다. Target이 범위에 들어오게 되면 Zomibe는 공격적으로 변해 Target을 따라 이동 후 공격하게 됩니다. Target이 발생한 시점에서 DetectRange는 비활성화 되고 MissingRange Trigger가 활성화 됩니다. MissingRange는 한 번 쫓기 시작한 Target을 감지할 수 없게 되는 범위입니다. 따라서 한 번 Zomibe에게 Target이 된 플레이어는 MissingRange의 바깥, 즉, Zombie의 인식 범위 밖으로 벗어나게 되면 해당 Zomibe는 추적을 종료하게 됩니다. MissingRange는 당연하게도 Target이 범위 바깥으로 도망갈 경우 비활성화 됩니다.

2) 행동 패턴

Zombie들은 평소에는 Idle 상태로 대기합니다. 그리고 상술한 감지 범위의 알고리즘을 통해 이동, 공격 Animation을 재생합니다. 감지 범위에 Target이 들어오면 주어진 NavMesh Agent의 기능을 이용하여 이동 경로를 탐색하며 해당 Target을 추적하게 되며, Target과 가까워져 공격 범위에 들어오게 되면 Zombie는 추적을 멈추고 공격 Animation을 실행합니다.
이를 위해서는 Zombie가 자기 자신과 Target과의 거리 정보를 획득해야 하는데, 이는 navmeshAgent에서 제공하는 remaningDistance 변수를 사용해 해결할 수 있습니다.

```
if (remainingDistance <= attackReach && remainingDistance > 0)
```

```
else if (remainingDistance > attackReach && remainingDistance <=Mathf.Infinity)
```

조건문을 사용해 Target과 Zombie의 거리가 각 Zombie의 종류별로 할당 된 사정거리 안에 들어온 경우를 판단하고 공격 Animation을 실행합니다.

플레이어를 발견하고 공격하는 무서운(?) 좀비!

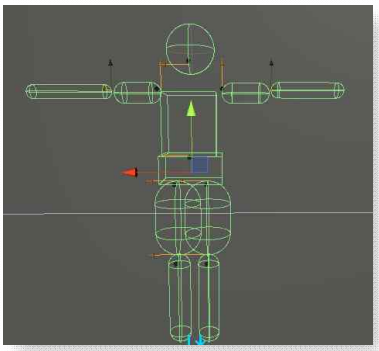
사. Ragdoll

1) 시체

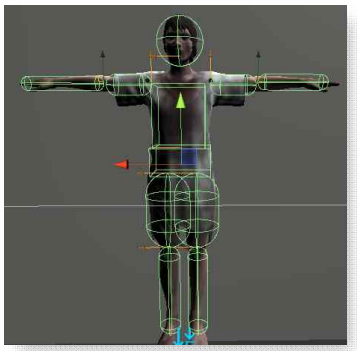
Zombie와 PlayerCharacter는 모두 Ragdoll을 이용해 체력이 0 이하가 될 경우 실제로 힘없이 쓰러지는 인형과 같이 자연스럽게 쓰러지는 모습을 연출했습니다.

2) 부위별 타격

앞서 말한 Ragdoll 시스템을 구성하는 요소 중 하나인 Collider를 실제 총알 피격 판정 도구로 사용함으로써 피격 범위를 실제 Zombie Object의 시각적 형태와 최대한 동일하게 구성했습니다.



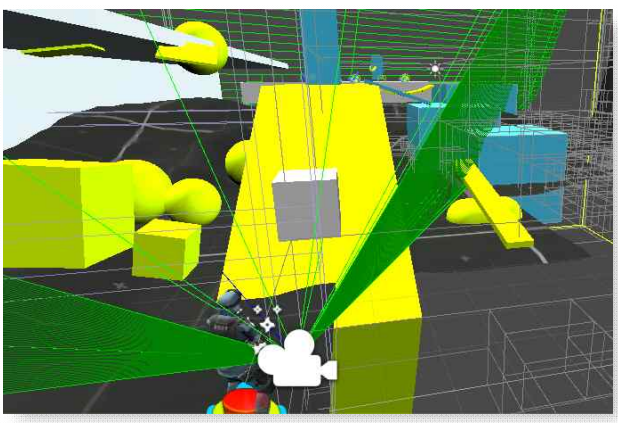
메쉬를 끈 상태



메쉬를 킨 상태

아. Occlusion Culling

Unity Engine에서 제공하는 최적화 기능인 Occlusion Culling 기술을 사용하여 카메라가 불러오는 Mesh 정보를 최적화 했습니다.
실제로 게임 플레이 시, 카메라에 보이지 않는 Object의 Mesh 정보만 활성화 하고, 보이지 않는 Object의 Mesh정보는 비활성화 하므로 메모리 관리 측면에서 효율성을 높였습니다.



초록색의 카메라범위 안에 들어오는 오브젝트의 메쉬만 활성화

5. 부록

가. Spell Selector

1) 게임 설명

본 게임의 매커니즘을 이용하여 별도의 마법 대전 게임도 제작해 보았습니다. [Spell Selector] 라는 제목의 게임으로, 제목 그대로 게임 시작 전 단계에서 원하는 Spell을 선택하고 해당 Spell을 사용하여 다른 플레이들과 전투하여 살아 남는 마지막 1인이 승리하는 게임입니다. 이 게임 역시 동일한 개발 환경에서 개발을 진행했습니다.

2) 게임 진행

게임 메인 화면입니다. Game Play 버튼을 클릭하여 다음 장면으로 넘어갑니다.

캐릭터 선택 화면입니다. 원하는 캐릭터를 선택하고 Confirm 버튼을 누르면 다음 화면으로 넘어갑니다.

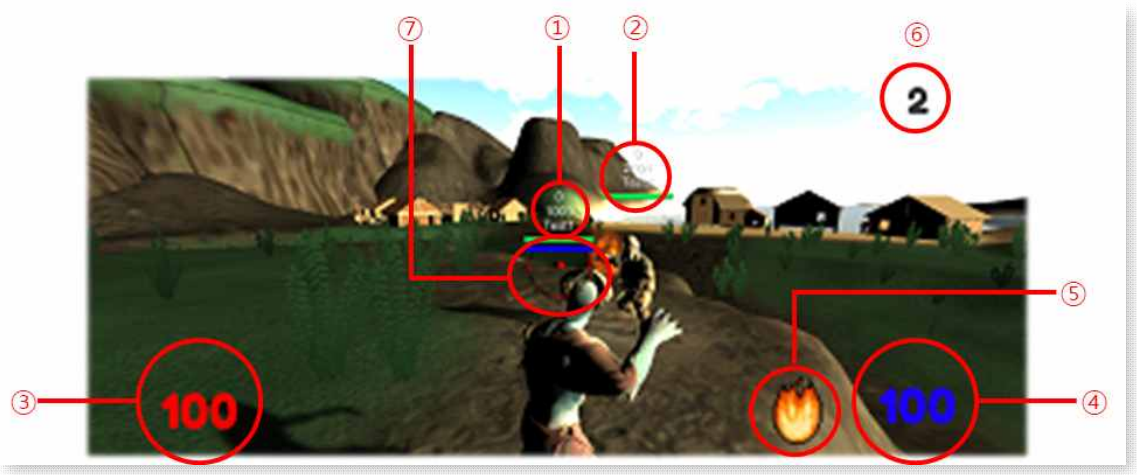
마법 선택 화면입니다. 원하는 마법 5가지를 선택하고 Confirm 버튼을 클릭하면 다음 화면으로 넘어갑니다.

닉네임 설정 화면입니다. 원하는 닉네임을 입력하고 Join Room 버튼을 클릭 시 게임 대기실로 입장합니다.

게임 대기실입니다. 2인 이상의 플레이어가 존재하고 모든 플레이어가 Ready 상태이면 게임이 시작됩니다.

실제 게임 화면입니다. 지금부터 본격적인 전투가 가능합니다.

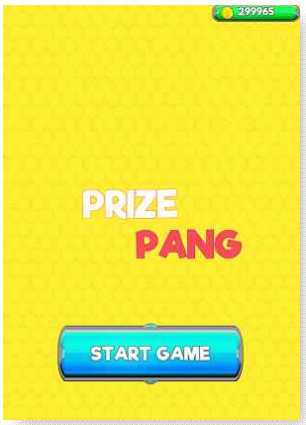
3) UI



① 플레이어 캐릭터의 UI 입니다. 포톤뷰 ID와 닉네임이 텍스트로 표시되고, 그 아래 체력과 마나를 바 형태로 나타내었습니다.
② 상대방 캐릭터의 UI 입니다. 1번과 마찬가지이나 상대방의 마나는 확인할 수 없도록 표시하지 않았습니다.
③ 플레이어 캐릭터의 체력을 텍스트로 확인할 수 있도록 표현했습니다.
④ 플레이어 캐릭터의 마나를 텍스트로 확인할 수 있도록 표현했습니다.
⑤ 현재 플레이어가 장착한 마법을 아이콘으로 표시합니다. 1~5버튼을 눌러 마법을 교체하면 아이콘이 해당 마법 아이콘으로 바뀝니다.
⑥ 자기 자신을 포함한 생존중인 플레이어 수를 보여줍니다.
⑦ 조준점을 표시합니다. 화면의 정 가운데 위치해 있습니다. 레이캐스트를 발사하는 방향입니다.

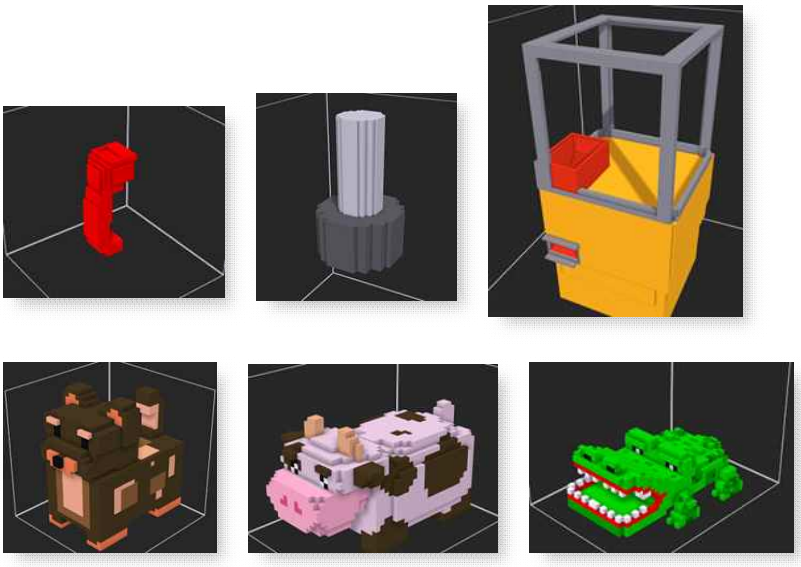
나. Pize Pang

1). 게임 설명



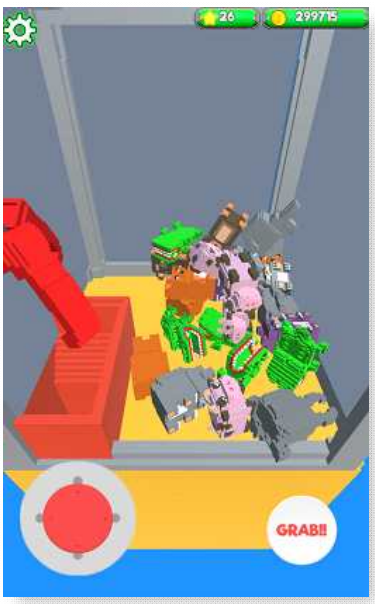
안드로이드 기반 캐주얼 인형 뽑기 게임입니다. 최초 설치 시 200코인을 지급하며 기계 안에 인형을 생성하는데 10 코인을 소모하고 인형을 뽑는데 성공하면 15코인을 획득하면서 동시에 뽑은 인형이 콜렉션 탭에 표시됩니다.

2). 리소스



게임 내 리소스는 MagicalVoxel 이라는 프로그램을 사용하여 직접 제작했습니다.
인형뽑기 기계와 집게, 다양한 종류의 인형들을 만들어 뽑는 재미를 더했습니다.

3). 게임 구성



MainScene, GameScene 두 씬으로 구성되어 있습니다.
MainScene에서 Start 버튼을 누르면 GameScene 으로 이동합니다.
GameScene에 진입하게 되면 Collection, Spawn Prize, Play, Go Back 버튼이 팝업 됩니다.
현재 기계 내에 뽑을 인형이 하나도 없다면 Play 버튼이 비활성화 되어, 게임을 시작할 수 없습니다.
Spawn Prize 버튼을 눌러 10코인을 소모해 뽑을 수 있는 Prize를 생성할 수 있으며 과도한 생성을 막기 위해 26개를 한계로 Spawn Prize 버튼이 비활성화 됩니다.
Collection 버튼을 클릭하면 지금까지 뽑은 Prize의 종류와 개수를 확인할 수 있습니다.
우측 상단에 현재 기계 내의 Prize 개수와 잔여 코인을 확인할 수 있습니다.

4). 소요 기술

(1). 유한상태기계(FSM)

FSM을 사용하여 인형 뽑기 기계 집게의 상태를 구현했습니다.

Int 형으로 State변수를 선언하고 Update() 안에서 Switch문을 사용해 각 상태에 맞는 동작을 하도록 구현했습니다.

Case 안에서 일정 조건, 예를 들면 State = 0 일 때에는 버튼의 입력이 들어오는 것과 같은 조건이 만족 되면 State를 변경합니다.

```
void Update()
{
    switch (state) {
        case 0: // Idle (기본 조종 가능한 상태)

            float h = SimpleInput.GetAxis("Horizontal");
            float v = SimpleInput.GetAxis("Vertical");


            ropePosition.transform.position = new Vector3(Mathf.Clamp(ropePositio

            movement.Set(h, 0, v);
            movement = movement.normalized * Time.deltaTime * 15;
            ropePosition.transform.position = (ropePosition.transform.position +



            break;
        case 1: // 집게 내림

            ropePosition.transform.position = (Vector3.Lerp(ropePosition.transfor
            if (claw.GetComponent<MyGame.casual.Claw>().downCheck==false)
            {
                claw.GetComponent<MyGame.casual.Claw>().downCheck = true;
            }
            if(ropePosition.transform.position.y <= 66.0f)
            {
                state = 2;
            }
            break;
        case 2: // 집게 집음
```

State = 0
조이스틱을 사용하여 집게를 자유롭게 이동할 수 있는 상태

State = 1
Grab 버튼을 눌러 집게가 하강하는 상태

State = 2
집게와 상품의 충돌 판정 후 상품을 집는 행동을 하는 상태

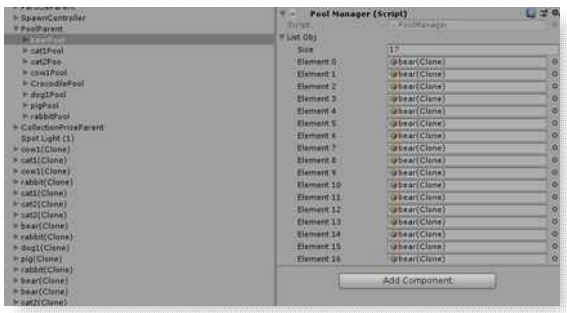
State = 3
집게가 상승하는 상태

State = 4
집게가 상품을 출구에 떨어뜨리기 위해 이동하는 상태

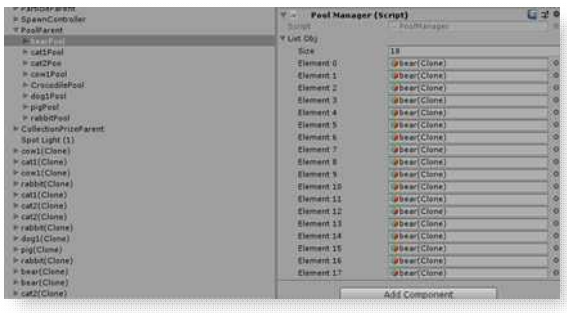
State = 5
플레이 대기 상태

(2). Object Pooling

Object Pooling 이란, 오브젝트 생성과 삭제 시에 부하 증가로 가비지 콜렉터 발생을 최소화 하는 방법입니다. 게임에 사용되는 Prize Object들을 매 번 Instantiate 하지 않고 게임 최초 실행 시 20개를 생성해 놓고 필요한 순간에 Active 하는 방법을 사용했습니다.



최초 Bear Object를 20개 생성 후 3개를 사용한 모습



집어넣은 Bear Object가 SetActive(false)가 되어 Pool로 되돌아간 모습 (리스트 1 증가)

Bear Object를 뽑는 과정