
PROJECT
STAR BRINGER
신입 게임 프로그래머 오태양
INVENTORY
SAVE
BATTLE
SEARCH
ENTER

목차

참고 링크

플레이 영상	https://youtu.be/7d41kKi3xfY	(Youtube)
모든 소스코드	https://github.com/sunio00000/StarBringer	(Github)

00 미 개요

Genre
게임 장르

모바일 슈팅 아케이드 RPG

기획 참고 - 브롤스타즈 , 패스오브엑자일

Participants
개발 인원

1인 개발

업무 - 기획, 프로그래밍, 이미지 디자인 등 전반에 걸쳐

Period
프로젝트 기간

19.03.11 착수 ~

진행 중 (19.05.08 기준 , 58일+)

Preference
개발 환경

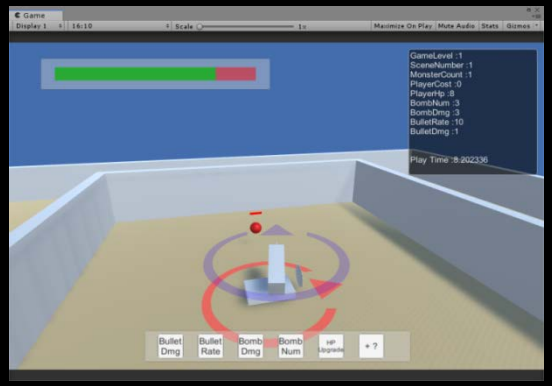
Unity3D Engine (2018.3.0f2)

Language C#
Open Source “Particle Scaler” , “Custom Outline” , “Joystick Pack”

Photoshop CS6
Windows 10, Android

01 제작동기

Why?
왜 이 게임을 개발했는가?



프로그램을 개발하며 느낄 수 있는 가장 큰 **성취**는 생각과 목적을 **구현**하거나, 성능을 **개선**하는 과정에서 생깁니다.
RPG/아케이드 장르의 컨텐츠 구성이 위를 충족할 많은 요소들로 이루어져 있고, 관심이 있는 분야이기 때문에 깊은 개념의 개발을 할 수 있을 것이라 확신했습니다.
이런 이유로, **도전적인 시도와 모방**을 통하여 많은 것을 배울 수 있었습니다.

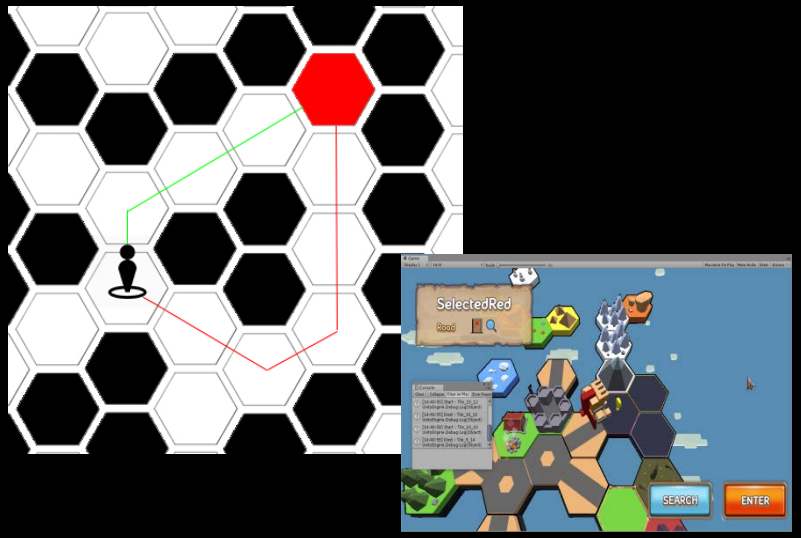
Unity 토이 프로젝트로 제작한 탱크 게임을 장르에 맞게 **재구성**하여 본 프로젝트로 확장했습니다.

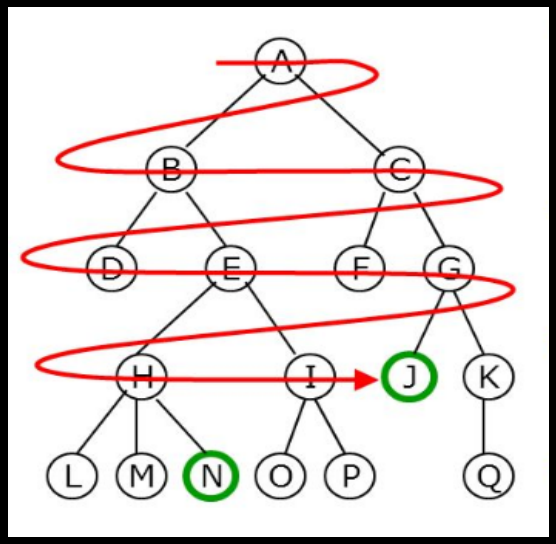
02 01 스크립트 로직 (경로 탐색)

영상 #1에 대한 설명입니다.

What?
무엇을 탐색하는가?

World Map Scene 에서 Player는 기준점에서 Hexagon 형태의 타일로 이루어진 주변 6개 타일을 통하여 목적지 까지 단위타일로 이동할 수 있는 최단 경로를 탐색합니다.





How?
어떻게 최단 경로를 탐색하는가?

너비 우선 탐색(BFS)을 통하여 Node를 탐색합니다.

Hash(String Key, String Value)에 방문 기록을 저장하고, 방문 오브젝트를 모두 Queue에 저장하고 꺼내가며, 목표점을 발견하면 탐색이 종료됩니다.

이동은 이전 경로를 저장한 Stack에서 하나씩 꺼내어 되추적합니다.

이에 대한 코드 설명은 다음 페이지에 있습니다.

02 미 스크립트 로직 (경로 탐색)

영상 #1에 대한 설명입니다.

탐색

```
//Function
private bool FindShortWayBFS(string Start, string Goal)
{
    string Default = "Tile_", center, top; string[] sNumber = new string[3]; int x , y;
    // 어디로부터 이동했는지 기록하는 자료구조 visitedFrom["curr"] => "prev"
    Dictionary<string, string> visitedFrom = new Dictionary<string, string>();
    // 이동 가능 노드를 저장하는 자료구조
    Queue<GameObject> Storage = new Queue<GameObject>();
    visitedFrom[Start] = null;
    Storage.Enqueue(GameObject.Find(Start));
    // Search..
    while (Storage.Count!=0){
        // 기준이 되는 노드를 설정한다.
        center = Storage.Dequeue().name;
        sNumber = center.Split('_');
        x = int.Parse(sNumber[2]); y = int.Parse(sNumber[1]);
        // 주변 노드의 인덱스(string)이 되는 값을 할당한다.
        /// top, topleft, topright, bottom, bottomleft, bott
        top = Default + (y - 1) + "_" + x;
        /// ...
        // 목적지를 찾았으므로 탐색이 종료된다.
        // 목적지가 아니라면, 그 곳은 노드가 존재하는가 && 존재 한다면 해당 노드가 방문했던 노드인가
        /// top, topleft, topright, bottom, bottomleft, bottomright 에 대해서
        if (top == Goal){
            visitedFrom[Goal] = center;
            SavePath(visitedFrom, Goal);
            return true;
        }
        else if (GameObject.Find(top).transform.childCount!=0 && !visitedFrom.ContainsKey(top)){
            Storage.Enqueue(GameObject.Find(top));
            visitedFrom[top] = center;
        }
        /// ...
    }
    // 저장한 노드가 없어 탐색을 종료한다.
    StartCoroutine(cautionTouch("연결된 길이 없다.."));
    return false;
}
```

경로저장

```
//Function
private void SavePath(Dictionary<string,string> dss, string Goal)
{
    string curr = Goal;
    // 목적지 부터 이전 경로를 추적하여 스택에 저장한다.
    // dss["Start"] 의 Value 는 null 이므로, 시작점이 가장 위에 쌓인 스택이 된다.
    while (dss[curr] != null)
    {
        // Stack<Transform> pathToGoal;
        pathToGoal.Push(GameObject.Find(curr).transform);
        curr = dss[curr];
    }
}
```

CODE
구현 코드

02 02 스크립트 로직 (공격 방식 구조화 & 구현)

영상 #2에 대한 설명입니다.

How?

어떻게 공격이 구현되는가?

공격 방식(SKILL)은 투사 주체 기준의 위치와 발사하는 투사체의 경로를 중점으로 구조적으로 설계되어 있습니다. (주체는 Player, Enemy, Tower가 됨)

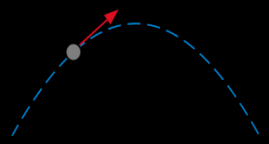
위치와 경로를 요소로 조합하여 다양한 공격방식을 구현할 수 있도록 설계했습니다.




아래 그림은 그에 대한 하나의 예시입니다.

Shooter

Projectile



CurrentSkillSystem = ShotSector;

AddComponent(ParabolaCtrl());

RESULT

구현 가능한 공격의 수 = 투사체 생성 함수의 수 X 투사체 스크립트의 수

이에 대한 코드 설명은 다음 페이지에 있습니다.

02 02 스크립트 로직 (공격 방식 구조화 & 구현)

영상 #2에 대한 설명입니다.

공격 방식구조화

```
abstract public class AttackMgr : MonoBehaviour
{
    // 투사체 오브젝트를 저장하는 리스트 입니다.
    public List<GameObject> Projectiles = new List<GameObject>();

    //투사체 발사 위치를 정하는 델리게이트를 Hash로 저장하는 공간입니다.
1.  protected readonly Dictionary<string, GameCtrl.SkillType> SkillType = new Dictionary<string, GameCtrl.SkillType>();
    //투사체 경로를 정하는 스크립트 타입을 Hash로 저장하는 공간입니다.
2.  protected readonly Dictionary<string, ProjectileMgr> SkillSystem = new Dictionary<string, ProjectileMgr>();
    //투사체 오브젝트 리스트의 내용을 Hash로 저장하는 공간입니다.
    protected readonly Dictionary<string, GameObject> SkillObject = new Dictionary<string, GameObject>();
    /// [주의] SkillType.Count == SkillSystem.CountTRUE 가 항상 성립해야 합니다.
    private void Awake(){
        for (int i = 0; i < Projectiles.Count; ++i) SkillObject.Add(Projectiles[i].name, Projectiles[i]);

        //다양한 조합의 공격을 추가합니다.
        SkillType.Add("FireBall", new GameCtrl.SkillType(AttackCtrl.instance.ShotFace));
        SkillSystem.Add("FireBall", new StraightCtrl());
        ///
        SkillType.Add("PFireBall", new GameCtrl.SkillType(AttackCtrl.instance.ShotSector));
        SkillSystem.Add("PFireBall", new ParabolaCtrl());
        ///
        SkillType.Add("LightningBall", new GameCtrl.SkillType(AttackCtrl.instance.ShotSector));
        SkillSystem.Add("LightningBall", new StraightCtrl());
        ///
        SkillType.Add("Teleport", new GameCtrl.SkillType(AttackCtrl.instance.MoveFace));
        SkillSystem.Add("Teleport", new TeleportCtrl());
        ///
        SkillType.Add("Orb", new GameCtrl.SkillType(AttackCtrl.instance.AroundSuriken));
        SkillSystem.Add("Orb", new AroundCtrl());
        ///
        SkillType.Add("Meteor", new GameCtrl.SkillType(AttackCtrl.instance.ShotMeteor));
        SkillSystem.Add("Meteor", new MeteorCtrl());
        //...
    }
}
```

CODE

구현 코드

1. SkillType 형태, 구조
2. ProjectileMgr 상속 Class 예

다음 페이지에 이어서..

02 02 스크립트 로직 (공격 방식 구조화 & 구현)

영상 #2에 대한 설명입니다.

CODE
구현 코드

Assign

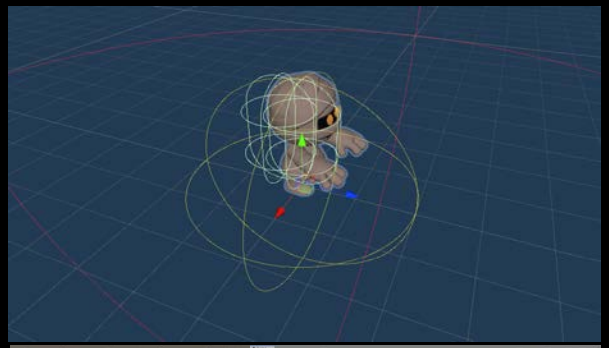
1.
```
// 투사체 생성 델리게이트, 매개 변수는 기준이 될 위치의 Transform이며, 반환 값은 쿨타임과 관련있다.
public delegate float SkillType(Transform t);
```

2.
```
//ProjectileMgr를 상속받은 스크립트가 투사체를 컨트롤한다.
//Trigger판정 혹은 Active상태 등은 부모 클래스에서 관리한다.
public class ParabolaCtrl : ProjectileMgr
{
    private float zVelocity = 2.5f , yVelocity = 3f;

    protected override void Start()
    {
        base.Start();
    }

    //움직임 및 효과는 Update 이벤트 함수나 Coroutine을 이용해 제어한다.
    protected override void Update()
    {
        base.Update();
        if (CanMove)
        {
            transform.position +=
                (new Vector3(0, (yVelocity - Gravity * (Time.time - retainedTime)) * Time.deltaTime, 0)
                + transform.forward * zVelocity * Time.deltaTime);
            if (transform.position.y <= GroundBound) CanMove = false;
        }
    }

    //오버라이드 된 추상메소드
    public override int Damage()
    {
        return (int)GameCtrl.instance.data[(int)Whose.Player].currState[(int)What.Power];
    }
}
```

```
// 델리게이트에 연결한 투사체 메소드(Default)
public float ShotDefault(Transform t)
{
    bulletRateCurr += bulletRate * Time.deltaTime;
    if (bulletRateCurr >= ShotRateBound && !AttackableJoystick.ReadyToAttack)
    {
        PlayerCtrl.instance.animator.SetTrigger("Attack");
        GameObject go = (GameObject)Instantiate(SkillObject["?"]);
        go.AddComponent(CurrSkillSystem());
        // Start Logic
        // ...
        // .
        // ...
        // End Logic
        AttackableJoystick.ReadyToAttack = true;
        bulletRateCurr = 0;
    }
    return ShotCurrRate <= ShotRateBound ? ShotCurrRate / ShotRateBound : 1;
}
```

02 03 스크립트 로직 (몬스터 추적)

영상 #3에 대한 설명입니다.



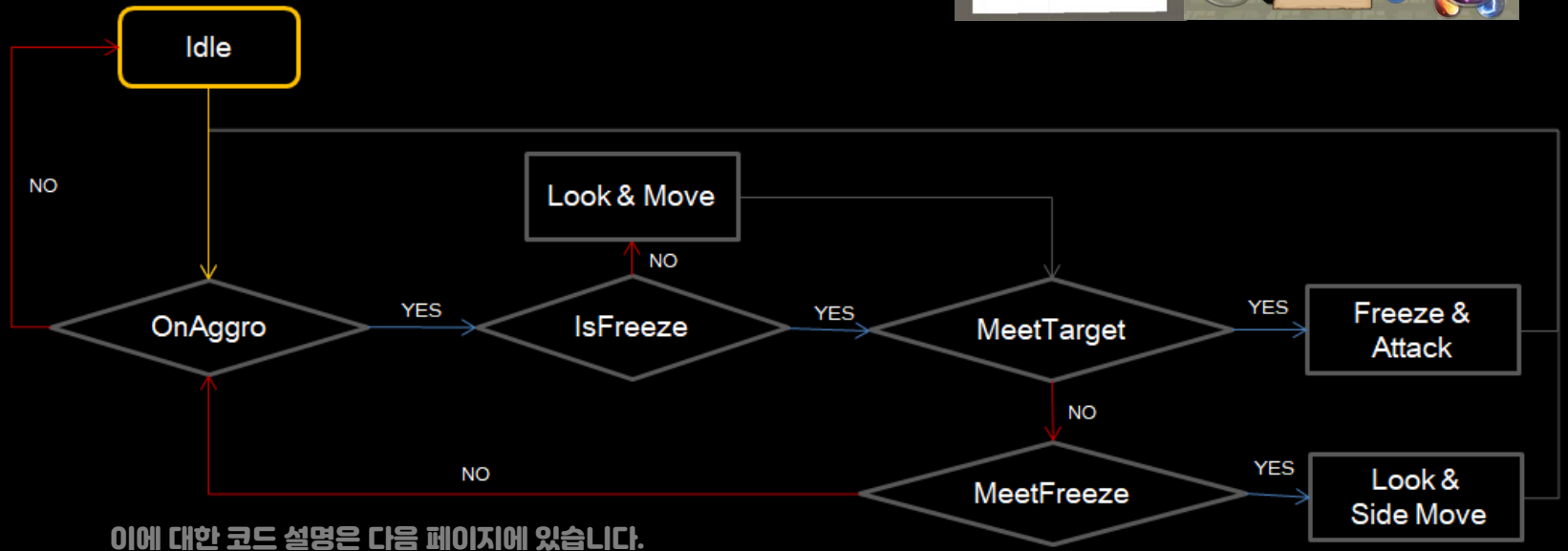
How?

어떻게 몬스터는 추적하는가?

Battle에서 몬스터는 2개의 Collider 컴퍼넌트를 가집니다.

몬스터는 아래 조건 흐름에 따라서 플레이어에게 다가갑니다.

몬스터가 움직이는 흐름은 다음과 같습니다.



이에 대한 코드 설명은 다음 페이지에 있습니다.

02 03 스크립트 로직 (몬스터 추적)

영상 #3에 대한 설명입니다.

자동 이동 함수

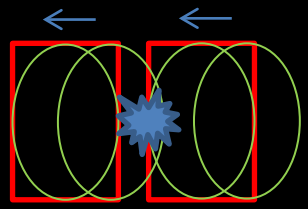
```
private void AutoMove()
{
    // 타겟이 어그로 범위 안에 들어왔다.
    if (OnAggro(go.position))
    {
        // 나의 Rigidbody가 모두 고정되어 있나?
        if (IsFreeze())
        {
            // 타겟이 공격 범위 안에 들어왔으므로, 공격한다.
            if (MeetTarget(go)) AttackFunction(true);
            else if (!MeetFreeze(cols)) transform.GetComponent<Rigidbody>().constraints = RigidbodyConstraints.FreezePositionY;
            // Rigidbody가 모두 고정된 오브젝트를 만났다면, 그 길은 갈 수 없다. 돌아가라.
            else
            {
                transform.GetComponent<Rigidbody>().constraints = RigidbodyConstraints.FreezePositionY;
                transform.position += transform.right * Time.deltaTime * 2.0f;
            }
        }
        else
        {
            AttackFunction(false); MoveFunction(true);
            // 움직임의 방해 요소가 없다면, 타겟을 향해 움직여라.
            if (!MeetFreeze(cols)){
                transform.GetComponent<Rigidbody>().constraints = RigidbodyConstraints.FreezePositionY;
                MoveToPlayer();
            }
            // 타겟을 만났다면, 내 Rigidbody를 고정시켜라.
            else if(MeetTarget(go)) transform.GetComponent<Rigidbody>().constraints = RigidbodyConstraints.FreezeAll;
        }
    }
}
```

Why?

2개의 Collider?

몬스터의 전방 Collider는 트리거되는 다른 몬스터의 후방 Collider를 리스트로 수집합니다.

리스트의 길이가 1이상 일때, 이동할 수 없습니다.



02 04 스크립트 로직 (오브젝트 풀링)

영상 #4에 대한 설명입니다.

투사체의 종류와 수로 생기는 오버헤드를 오브젝트 풀링을 통해 개선합니다.

CODE
구현 코드

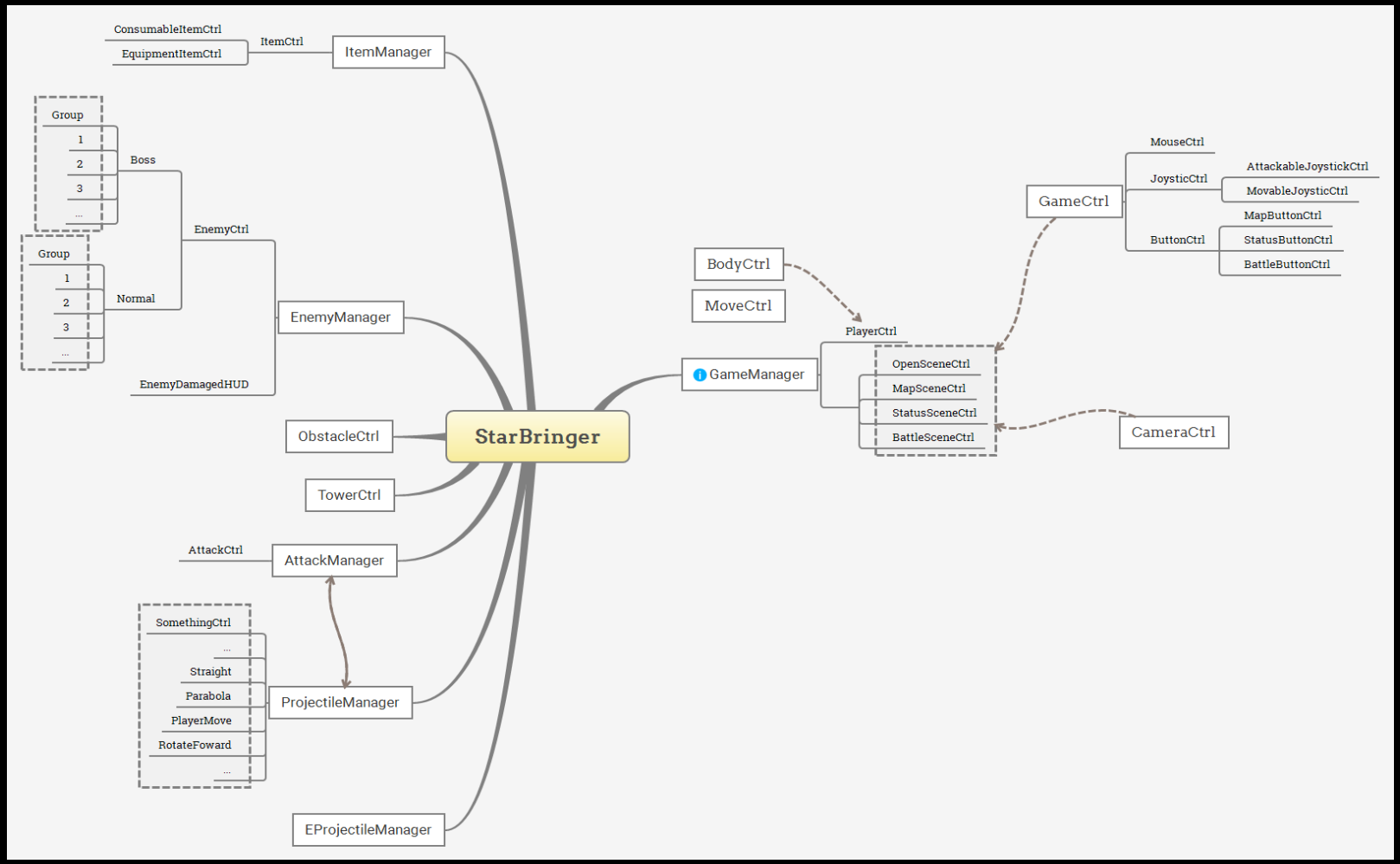
```
// static 메소드로 여러 오브젝트에서 이용 가능하다.
// 매개변수 (대상이 될 오브젝트, 대상의 트랜스폼, 임의의 이름)
static public void PoolCreateAndManage(GameObject Object, Transform Muzz, string n)
{
    // 재활용 가능한 오브젝트가 있다면 이용한다.
    if (Pooling(n + "_weapon", Muzz)) { return; }
    // 아니라면 오브젝트를 생성하고 이니셜라이즈 한다.
    else
    {
        GameObject Weapon = Instantiate(Object, Muzz.position, Muzz.rotation);
        Weapon.name = n + "_weapon";
        GameObject Basket = GameObject.Find(Weapon.name + "_Pool");
        // 담을 Pool이 없다면 새로 생성한다.
        if (Basket == null) Basket = new GameObject(Weapon.name + "_Pool");
        Weapon.transform.parent = Basket.transform;
    }
}
static public bool Pooling(string Weapon, Transform Muzz)
{
    GameObject Basket = GameObject.Find(Weapon + "_Pool");
    if (Basket == null) return false;
    // 풀이 존재한다면 리스트를 확인한다.
    for (int i = 0; i < Basket.transform.childCount; ++i) {
        // 대상 오브젝트가 있다면,
        GameObject w = Basket.transform.GetChild(i).gameObject;
        // 오브젝트가 이용 중이면 다음 것을 확인하고,
        if (w.activeSelf) continue;
        // 사용 가능한 오브젝트가 있다면, 그 것을 이용한다.
        else
        {
            w.transform.position = Muzz.position;
            w.transform.rotation = Muzz.rotation;
            w.SetActive(true);
            return true;
        }
    }
    return false;
}
```

Pooling 메소드

03 01 스크립트 구조

Structure

스크립트 구조 전개도



04 01 Self Q/A

Q
본 게임의 제작 목표

A
상용 가능한 게임으로서의 완성입니다.

Q
제작 간의 개발 중점

A
출시 게임의 기본 컨텐츠를 모방, 응용하여 직접 구현하고 최적화하여 최종적으로 다양한 컨텐츠를 개발하고, 성능을 개선하려고 노력했습니다.

Q
이 후의 개발 계획

A
코드 개선을 기본으로 구조를 응용하여 게임 컨텐츠를 제작하고, 출시 게임과 비교한 편의성을 계획 중입니다.

END….