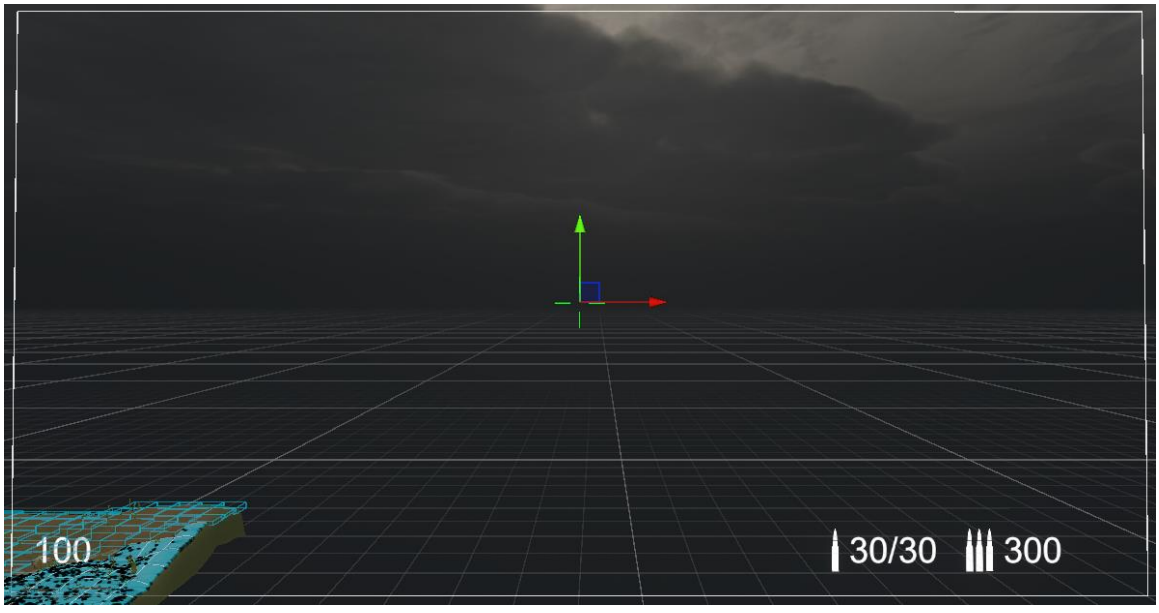
Silent Land

Brief Summary:

Overview:
Set in a harsh, post-nuclear world overrun by monstrous creatures and the undead, the game follows a lone survivor on a desperate journey to escape to the far East. The world is desolate and dangerous, filled with lurking threats at every turn.

Gameplay:
Players must navigate the ruined landscape, battling relentless monsters using an arsenal of weapons. Survival depends on strategy, skill, and resource management. In future updates, NPCs will provide missions, adding depth to the player's journey and expanding the game’s narrative.

User Interface:



Controls:

Movement:

By pressing WASD, player moves towards four directions. Player can jump by pressing Space.

Weapons:

Press left mouse click to open fire. Player has three weapons can switch. Pressing 1 switches to pistol, pressing 2 switches to sniper rifle and 3 to knife. Player can also switch back to the last weapon they used by pressing key Q.

Design Patterns:

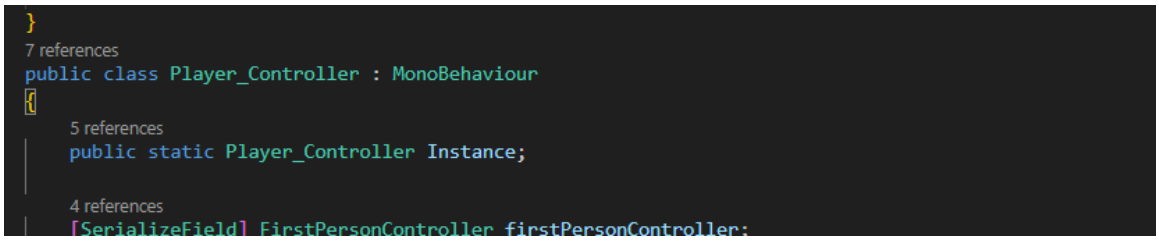
Singleton Pattern

Application Scenario:

Ensures globally unique manager classes (such as GameManager, ZombieManager) can be easily accessed by other components.

```
2 references
public class ZombieManager : MonoBehaviour
{
    2 references
    public static ZombieManager Instance;
    1 reference
    public GameObject prefab_Zombie;
    4 references
    public List<ZombieController> zombies;//当前场景中僵尸

    3 references
    private Queue<ZombieController> zombiePool = new Queue<ZombieController>();//备用僵尸
    1 reference
    public Transform Pool;
    0 references
    private void Awake()
    {
        Instance = this;
    }
    0 references
    void Start()
```



```
}
7 references
public class Player_Controller : MonoBehaviour
{
    5 references
    public static Player_Controller Instance;

    4 references
    [SerializeField] FirstPersonController firstPersonController;
```

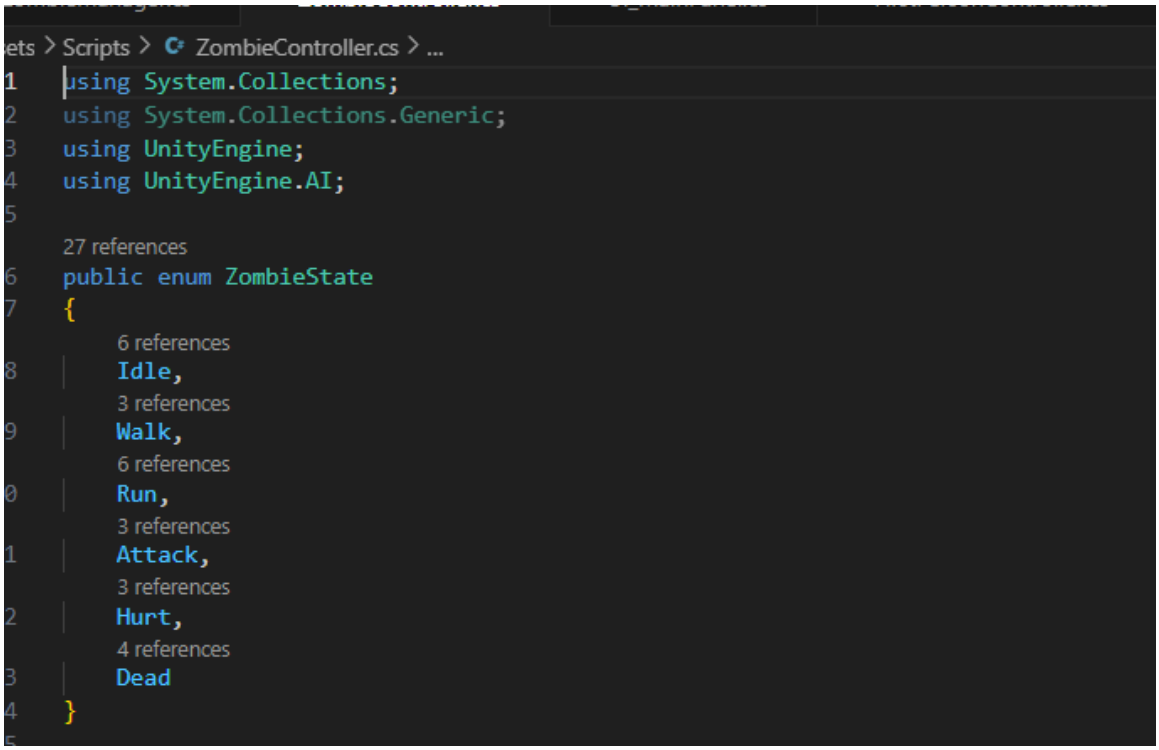
Function:

- **Directly access manager functions via GameManager.Instance.GetPoints() or ZombieManager.Instance.ZombieDead() without frequently searching for objects.**
- **Ensures global uniqueness, preventing duplicate instantiation.**

State Pattern

Application Scenario:

Dynamic switching of zombie behaviors (chase, patrol, attack) and player states (move, shoot, reload).



Function:

- **Decouples states from behaviors, managing logic centrally via switch-case.**
- **Supports flexible expansion of new states.**

Factory Method Pattern

Application Scenario:

Zombie creation and recycling are centrally managed by ZombieManager.

```
1 reference
IEnumerator CheckZombie()
{
    while (true)
    {
        yield return new WaitForSeconds(1);
        // 僵尸数量不够，产生僵尸
        if (zombies.Count<3)
        {
            // 池子里面有，从池子拿
            if (zombiePool.Count>0)
            {
                ZombieController zb = zombiePool.Dequeue();
                zb.transform.SetParent(transform);
                zb.transform.position = GameManager.Instance.GetPoints();
                zombies.Add(zb);
                zb.gameObject.SetActive(true);
                zb.Init();
                yield return new WaitForSeconds(2);
            }
            // 池子没有，就实例化
            else
            {
                GameObject zb = Instantiate(prefab_Zombie, GameManager.Instance.GetPoints(), Quaternion.identity, transform);
                zombies.Add(zb.GetComponent<ZombieController>());
            }
        }
    }
}
```

Function:

- **Encapsulates object creation logic within ZombieManager, so callers do not need to manage instantiation details.**
- **Supports expanding different generation strategies (e.g., adjusting zombie types based on difficulty).**

Object Pool Pattern

Application Scenario:

Frequent generation and recycling of zombies to reduce the performance overhead of Instantiate/Destroy.

```
2 references
public class ZombieManager : MonoBehaviour
{
    2 references
    public static ZombieManager Instance;
    1 reference
    public GameObject prefab_Zombie;
    4 references
    public List<ZombieController> zombies;//当前场景中僵尸

    3 references
    private Queue<ZombieController> zombiePool = new Queue<ZombieController>();//备用僵尸
    1 reference
    public Transform Pool;
    0 references
    private void Awake()
    {
        Instance = this;
    }
```

```
    while (true)
    {
        yield return new WaitForSeconds(1);
        // 僵尸数量不够，产生僵尸
        if (zombies.Count<3)
        {
            // 池子里面有，从池子拿
            if (zombiePool.Count>0)
            {
                ZombieController zb = zombiePool.Dequeue();
                zb.transform.SetParent(transform);
                zb.transform.position = GameManager.Instance.GetPoints();
                zombies.Add(zb);
                zb.gameObject.SetActive(true);
                zb.Init();
                yield return new WaitForSeconds(2);
            }
            // 池子没有，就实例化
            else
            {
                GameObject zb = Instantiate(prefab_Zombie, GameManager.Instance.GetPoints(), Quaternion.identity, transform);
                zombies.Add(zb.GetComponent<ZombieController>());
            }
        }
    }
}
```

Function:

- **Uses a queue to cache dead zombie objects, reactivating them when needed.**
- **Significantly optimizes performance, preventing memory fragmentation and garbage collection (GC) pressure.**

Observer Pattern

Application Scenario:

Automatically updates the UI when ammo count changes.

```
3 references
public void UpdateBulletUI(int curr_BulletNum,int curr_MaxBulletNum,int standby_BulletNum)
{
    UI_MainPanel.Instance.UpdateCurrBullet_Text(curr_BulletNum,curr_MaxBulletNum);
    UI_MainPanel.Instance.UpdateStandByBullet_Text(standby_BulletNum);

}
```

```
#endregion
1 reference
protected virtual void OnLeftAttack()
{
    if(wantBullet)
    {
        curr_BulletNum--;
        player.UpdateBulletUI(curr_BulletNum,curr_MaxBulletNum,standby_BulletNum);


    }
```

```
0 references
public void UpdateHP_Text(int hp)
{
    Hp_Text.text = hp.ToString();
    if (hp>30)
    {
        Hp_Text.color = Color.white;
    }
    else
    {
        Hp_Text.color = Color.red;
    }
}
1 reference
public void UpdateCurrBullet_Text(int cur,int max)
{
    CurrBullet_Text.text = cur + "/" + max;
    if (cur < 5)
    {
        CurrBullet_Text.color = Color.red;
    }
    else
    {
        CurrBullet_Text.color = Color.white;
    }
}

1 reference
public void UpdateStandByBullet_Text(int num)
{
    StandByBullet_Text.text = num.ToString();
    if (num < 30)
    {
        StandByBullet_Text.color = Color.red;
    }
    else
    {
        StandByBullet_Text.color = Color.white;
    }
}

/// <summary>
```

Function:

- **When ammo changes, the weapon directly calls UpdateBulletUI() to notify the UI component for an update.**
- **The UI listens for ammo changes and updates automatically when the player shoots.**

---

1. Zombie Management (ZombieManager)

Function:

- Responsible for generating and managing zombies to ensure a constant number of zombies in the scene.
- Uses the Object Pool Pattern to avoid frequent creation and destruction of zombies, improving performance.

Design Patterns Used:

- **Singleton Pattern:** Ensures that ZombieManager has only one instance to manage zombies centrally.
- **Object Pool Pattern:** Reuses zombie objects to reduce memory allocation and improve performance.

---

2. Zombie Behavior (ZombieController)

Function:

- Controls zombie AI, including idle, patrol, chase, attack, hurt, and death states.
- Uses the State Pattern to execute different behaviors in different states.

Design Patterns Used:

- **State Pattern:** Zombie behavior (Idle, Walk, Run, Attack, Hurt, Dead) is switched using a state system for clear management.
- **Singleton Pattern:** Zombies can call GameManager.Instance to get the player's location.

---

3. Player Control (Player_Controller)

Function:

- Handles player input (movement, shooting, reloading).
- Manages weapon switching.
- Uses the State Pattern to control player behavior.

Design Patterns Used:

- **Singleton Pattern:** Player_Controller.Instance ensures the player controller is unique in the game.
- **State Pattern:** Manages Move, Shoot, and Reload states, making the logic clear.

---

4. UI Management (UI_MainPanel)

Function:

- Displays UI information such as health, bullet count, and crosshair changes.
- Uses the Observer Pattern so that the UI updates automatically when data changes.

Design Patterns Used:

- **Singleton Pattern:** The UI is a global manager, and UI_MainPanel.Instance ensures uniqueness.
- **Observer Pattern:** UI updates automatically when health and bullet count change.

---

5. Weapon System (WeaponBase)

Function:

- Manages weapon switching, shooting, and recoil.
- Uses the Factory Pattern to generate different weapons.

Design Patterns Used:

- **Factory Pattern:** Different weapon types (such as rifles and pistols) are managed by WeaponFactory.
- **Strategy Pattern:** Different weapons have different shooting methods (automatic, semi-automatic, unable to fire).

---

6. Player Movement Logic (FirstPersonController.cs)

Input Handling:

- Uses CrossPlatformInputManager to get horizontal and vertical input.
- transform.forward * vertical and transform.right * horizontal convert input into movement direction in the character coordinate system.

Speed Control:

- m_IsWalking is toggled by the Left Shift key (default is walk, press to run).
- m_WalkSpeed and m_RunSpeed are set to 5 and 10 respectively to control movement speed.

Physical Movement:

- Uses CharacterController.Move() to move the character while supporting collision detection and slope handling.

View Rotation:

- MouseLook.LookRotation() processes mouse input to achieve smooth view rotation (implemented in the MouseLook component).

Effects:

- **Footsteps:** PlayFootStepAudio() plays random footstep sounds from m_FootstepSounds array.
- **Head Bobbing:** The HeadBob component dynamically adjusts camera position based on movement speed to simulate head motion while walking.

---

7. Zombie Movement Logic (ZombieController.cs)

Explanation:

State Machine Driven:

- ZombieState enum defines six states (Idle, Walk, Run, Attack, Hurt, Dead).
- The ZombieState property setter switches states and triggers corresponding behaviors (such as animations and navigation updates).

Navigation System:

- The NavMeshAgent component handles pathfinding and movement (requires pre-baking of the scene's NavMesh).
- In the **Run** state, the target point continuously updates to the player's position, enabling chase logic.

Random Walking:

- In the **Idle** state, Invoke("GoWalk", Random.Range(1,3)) randomly switches to the **Walk** state, moving towards preset path points in GameManager.

Effects:

- **Animation Switching:** animator.SetBool("Walk", true) triggers the walking animation.

- **Sound Effects:** The animation event FootStep() randomly plays footstep sounds from the FootstepAudioClips array.

---

8. Player Shooting Logic (WeaponBase.cs)

Explanation:

Ammo Management:

- curr_BulletNum represents the remaining bullets in the magazine, while standby_BulletNum is the reserve ammo count.
- Each shot calls player.UpdateBulletUI() to update the UI display.

Animation & Effects:

- animator.SetTrigger("Shoot") triggers the shooting animation.
- shootEF is a particle system attached to the gun barrel, activated during firing to display muzzle flash.

Raycasting:

- Shoots a ray from the center of the screen (ScreenPointToRay) to detect collisions within 1500 units.
- If a zombie is hit, ZombieController.Hurt() is called to reduce health and spawn blood effects (prefab_BulletEF[1]).

Effects:

- **Recoil Effect:** The coroutine ShootRecoil_Camera causes camera shake.
- **Crosshair Spread:** The coroutine ShootRecoil_Cross dynamically adjusts the crosshair size.

---

9. Weapon Switching Logic (Player_Controller.cs + WeaponBase.cs)

Explanation:

Switching Process:

1. Calls the old weapon's Exit() method to play the exit animation and registers callback OnWeaponExitOver.
2. When the exit animation completes, triggers the callback to enable the new weapon's Enter() method.

Animation Binding:

- animator.SetTrigger("Exit") and animator.SetTrigger("Enter") control weapon switch animations.
- The last frame of the animation calls ExitOver() and EnterOver() to finalize the switch.

Effects:

- **Entry/Exit Animations:** For example, pistol holstering and drawing animations.
- **UI Synchronization:** Calls InitForEnterWeapon to update crosshair and ammo display.

---

10. Zombie Damage and Death Logic (ZombieController.cs)

Explanation:

Damage Handling:

- hp starts at 100; each hit reduces attackValue (defined by the weapon).
- When health reaches zero, the zombie enters the **Dead** state, disabling its collider and playing the death animation.

Knockback Effect:

- Temporarily disables NavMeshAgent, plays the **Hurt** animation (animator.SetTrigger("Hurt")).

Object Pool Recycling:

- After 5 seconds, ZombieManager.ZombieDead(this) moves the zombie to the zombiePool queue for reuse.

Effects:

- **Death Animation:** animator.SetTrigger("Dead") triggers the collapse animation.
- **Sound Effects:** Plays a random sound from HurtAudioClips on hit.

Tasks Breakdown:

Two team members. Bohao is charge on coding mainly and animators, Xuejian is charge on Map design, UI, Visual and post processing and coding.