경희대학교

최종보고서

제목: 3D Unity 슈팅 난관 돌파 게임<좀비 위기>

전공:컴퓨터공학과(Department of computer science engingeering)

이름:SONGJINGYUAN

학번:2021105557

지도교수:CHAONING ZHANG

요약

3D Unity 슈팅 난관 돌파 게임<좀비 위기>는 Windows 기반 Unity 가 가장 엔진이며 C#은 개발 언어의 3D 게임으로 NVIDIA PhysX 의 기능과 통합 및 성능 튜닝이 가능하고 다양한 UI 기반 볼륨 조절 기능을 갖추고 있으며 UI 의 기본 기능을 갖추고 있어 게임 종료 시 설명 및 UI 가 가능하고 UI 가 가능하며 UI 가 가장 엔진의 특징을 제공합니다. 게임 카드 종료 진도를 저장합니다."게임 시작"에 들어가는 총 세 개의 관문이 있습니다. 카드 꺼짐은 선택할 수 있습니다. (그러나 이전 꺼짐의 도전을 완료한 후에야 다음 꺼짐에 들어갈 수 있습니다. 그렇지 않으면 잠금 상태가 되기 때문에 진행 상태 저장 기능을 제공합니다.) 게임에서도 언제든지"esc"를 누르고 UI 인터페이스에 들어갈 수 있으며 게임, UI 를 일시 중지할 수 있습니다.인터페이스에서도 볼륨을 조절하거나 게임으로 돌아가거나 게임을 종료할 수 있습니다 (그러나 이번 과정에서 게임 진도는 저장할 수 없습니다【같은 카드가 꺼진 진도는 저장할 수 없습니다】)

1. 서론

사격과 난관 돌파 게임은 높은 상호작용성과 도전성을 가지고 게이머들의 흥미와 주의를 끌 수 있다. 현대 게이머들은 빠른 리듬과 도전적인 게임 체험을 추구하는 경향이 있다. 사격 난관 돌파 게임은 이런 수요를 충족시킨다. 이런 게임을 개발하면 개발자들이 신기술과 새로운 게임 방법을 계속 탐색하고 기술 수준과 혁신 능력을 향상시킬 수 있다. 성공한 사격 게임도 강력한 브랜드 효과를 구축하고 게이머들의 지속적인 관심을 끌 수 있다. 그래서 나는 이 세 개의 소총을 개발하기 전에 난이도를 높인다.또한 스킬은 게임성을 제공합니다 (플레이어가 스킬을 가지고 있으면 유연성을 제공할 수 있습니다. 게임은 모두 두 가지 스킬을 설계할 수 있습니다). 또한 게임은 모든 카드 관문이 통관된 후 새로운 무기와 스킬을 잠금 해제합니다. 세 번째 관문은 종료 카드 관문으로서

boss를 설계하여 게임의 게임성과 성취감을 향상시켜야 합니다. 또한 게임은 진도를 기록할 수 있습니다. 모든 통관 후 플레이어는 이전에 잠금 해제된 강력한 무기와 스킬을 휴대하거나 사용하여 다시 도전할 수 있습니다!

2. 기존연구

개발 환경:

1. 운영체제: Windows10
2. 개발 도구: Unity Hub 3.3.2-c8
3. 편집기 버전: 2021.3.44f1c1
4. IDE : Viusal Studio Code1.94.2

이번 개발에서 사용된 각종 논리는 모두 C#을 기반으로 한 다음 unity 의 편집기로 통합하여 마지막에 실행 가능한 파일 (.exe) 으로 포장한다. 게임의 장면과 인물 (포함) 모델은 모두 unity 커뮤니티 쇼핑몰 (총기와 인물의 충돌 부피를 약간 조정한 것 제외) 에서 나온 것이다.

3. 문제정의

게임에서의 각종 구상을 실현하기 위해서는 실천이 필요할 때 항상 코드를 통해 실현해야 한다. 이 과정에서 많은 문제에 부딪혔다. 예를 들면 인물 설계 시각이 어떻게 실현되는가, 괴물 AI 추적이 어떻게 실현되는가 등등이다. 아래에 나는 개발 과정에서 사용된 각 모듈의 문제를 열거할 것이다.

3.1 인물 이동

인물 이동 부분에서 인물이 이동할 수 있도록 하는 것 외에 어떻게 이동과 동시에 스퍼트 기능을 추가합니까?

3.2 착지 검사

인물을 제작할 때 흔히 부딪히는 상황이 있다. 인물과 지도의 충돌이 호환되지 않아 테스트할 때 게임에 들어가 보이지 않지만 게임이 정상적으로 운행될수 있다. 이때 어떻게 해야만 게임에 들어갈 때 캐릭터가 자동적으로 감지되여 플랫폼아래로 떨어지지 않게 할수 있다. 게임에는 많은 장애물이 있고 충돌부피가 있다. 어떻게 하면 게임이 실시간으로 감지되여 모형을 입지 않을수 있는가?

3.3 카메라 회전

여전히 3 인칭 캐릭터의 문제이다. 사격 모듈을 설계할 때 게임의 난이도를 높이기 위해 총알의 수가 제한되어 있다는 점을 고려하여 게이머의 오발을 방지하기 위해 오른쪽 버튼으로 사격 모드를 켜도록 설계했다. 이 기능을 추가한 후 게임 체험을 더욱 사실적으로 하기 위해 어떻게 렌즈를 추가하여 따라가야 하는지, 카메라 회전을 어떻게 실현해야 하는지가 문제가 되었다.

3.4 시스템 입력

버전 문제로 인해 게이머의 일련의 조작 지령은 새로운 버전 시스템을 입력함으로써 실현해야 하는데, 어떻게 입력 시스템을 실현합니까?

3.5 인물 총기 소지 동작

3 인칭이기 때문에 카메라 추종과 인물의 총기 동작은 다소 경직되어 있다. 카메라 추종을 해결했다. 그러나 인물의 총기 동작은 카메라에 따라 업데이트될 수 없다. 위치의 변화와 카메라 회전에 따라 어떻게 손과 총기를 최대한 밀착시킬 수 있을까?

3.6 애니메이션 마스크

테스트를 할 때 사격 애니메이션과 보행 애니메이션이 충돌하는 것을 발견했습니다. 그러나 저는 이동할 때 사격을 하고 싶습니다. 즉 상체에 총을 들고 조준하고 하체에 이동 화면을 하려면 어떻게 해야 합니까?

3.7 인물 조준

인물이 조준할 때 어떻게 효과를 더욱 진실하게 하고 어떻게 인물이 조준 위치에 따라 상체를 구부릴 수 있습니까?

3.8 카메라 떨림

어떻게 총기 후좌력의 효과를 시뮬레이션합니까?어떻게 서로 다른 총기를 교체할 때 떨림 효과도 같지 않습니까?

3.9 총기 부분

총기는 모형을 제외하고 어떻게 해야만 효과적인 사격을 할 수 있습니까?

3.10 스킬 부분

첫 번째 기술 설계를 할 때, 일정한 범위에서 적에게 피해를 줄 수 있다고 구상하고, 어떻게 실현할 것인가"의 생성 범위는 어떻게 제정합니까?채혈 논리는 또 어떤가?

3.11 스킬 CD

어떻게 해야만 스킬 방출, 냉각 진입, 카운트다운 종료 후 다시 방출 가능합니까?

3.12 괴물

몬스터가 유저가 추격 범위에 진입하지 않았을 때 대기 상태에 들어간다고 가정하면 어떻게 몬스터의 스마트 추격을 실현할 수 있습니까?

4. 해결방안

4.1 인물 이동

ThirdPersonController 스크립트에서 인물 이동의 논리는 주로 Move 방법에 집중됩니다.

목표 속도 계산:플레이어가 스트로크 키를 누른지 여부에 따라 미리 설정된 이동 속도와 스트로크 속도를 결합하여 목표 속도를 계산한다.입력한 내용이 없으면 대상 속도가 0 으로 설정됩니다.이어 현재 수평 속도와 비교한 가속 또는 감속을 통해 선형 보간으로 더욱 자연스러운 속도 변화를 실현하고 속도를 3 자리 소수로 반올림한다.그런 다음 가져온 방향을 단일화합니다.

속도 부드럽게:선형 보간(Lerp)을 통해 부드럽게 변화하는 속도는 캐릭터의 움직임을 더욱 자연스럽게 만듭니다.

currentSpeed = Mathf.Lerp(currentSpeed, targetSpeed, Time.deltaTime * speedSmoothTime);

입력 방향 통합:게이머가 이동 입력이 있을 때 방향과 카메라의 각도를 입력하여 목표의 회전 각도를 계산하고 조건을 만족시킬 때 게이머를 회전시킨 후 목표의 방향을 확정하고 게이머를 이동하면 캐릭터의 방향이 이동 방향과 일치하도록 할 수 있다.

캐릭터 애니메이션 동기화:캐릭터 애니메이션을 사용하는 경우 애니메이션 매개변수를 업데이트하여 캐릭터의 애니메이션이 이동 상태와 일치하도록 해야 합니다.

```
private void Move()
{
    // set target speed based on move speed, sprint speed and if sprint is pressed
    float targetSpeed = _input.sprint ? SprintSpeed *SkillManager.instance.Rate: MoveSpeed * SkillManager.instance.Rate;

    // a simplistic acceleration and deceleration designed to be easy to remove, replace, or iterate upon

    // note: Vector2's == operator uses approximation so is not floating point error prone, and is cheaper than magnitude
    // if there is no input, set the target speed to 0
    if (_input.move == Vector2.zero) targetSpeed = 0.0f;

    // a reference to the players current horizontal velocity
    float currentHorizontalSpeed = new Vector3(_controller.velocity.x, 0.0f, _controller.velocity.z).magnitude;

    float speedOffset = 0.1f;
    float inputMagnitude = _input.analogMovement ? _input.move.magnitude : 1f;

    // accelerate or decelerate to target speed
    if (currentHorizontalSpeed < targetSpeed - speedOffset ||
        currentHorizontalSpeed > targetSpeed + speedOffset)
    {
        // creates curved result rather than a linear one giving a more organic speed change
        // note T in Lerp is clamped, so we don't need to clamp our speed
        _speed = Mathf.Lerp(currentHorizontalSpeed, targetSpeed * inputMagnitude,
            Time.deltaTime * SpeedChangeRate);

        // round speed to 3 decimal places
        _speed = Mathf.Round(_speed * 1000f) / 1000f;
    }
    else
    {
        _speed = targetSpeed;
    }

    _animationBlend = Mathf.Lerp(_animationBlend, targetSpeed, Time.deltaTime * SpeedChangeRate);
    if (_animationBlend < 0.01f) _animationBlend = 0f;

    // normalise input direction
    Vector3 inputDirection = new Vector3(_input.move.x, 0.0f, _input.move.y).normalized;

    // note: Vector2's != operator uses approximation so is not floating point error prone, and is cheaper than magnitude
    // if there is a move input rotate player when the player is moving
    if (_input.move != Vector2.zero)
    {
        _targetRotation = Mathf.Atan2(inputDirection.x, inputDirection.z) * Mathf.Rad2Deg +
                          _mainCamera.transform.eulerAngles.y;
        float rotation = Mathf.SmoothDampAngle(transform.eulerAngles.y, _targetRotation, ref _rotationVelocity,
            RotationSmoothTime);
```

4.2 착지 검사

ThirdPersonController 의 GroundedCheck 메서드에서 다음을 수행합니다.
구 위치 계산:현재 객체의 위치에서 구체적으로 수직 방향(y 축에서 아래로 GroundedOffset 값을 오프셋하여 지면에 닿았는지 여부를 감지하는 구의 위치를 결정하는 새로운 벡터 spherePosition 을 만듭니다.
구 충돌 감지:Physics 를 사용합니다.CheckSphere 메서드는 지정된 구의 위치 spherePosition 을 중심으로 GroundedRadius 를 반지름으로 특정 계층(GroundLayers 가 지정)에서 충돌 감지를 수행하여 역할이 지면에 닿았는지 여부를 판단하고 트리거의 상호 작용 감지 결과를 무시하여 현재 객체가 지면에 닿았는지 여부를 나타내는 Grounded 변수에 저장합니다.

```
private void GroundedCheck()
{
    // set sphere position, with offset
    Vector3 spherePosition = new Vector3(transform.position.x, transform.position.y - GroundedOffset,
        transform.position.z);
    Grounded = Physics.CheckSphere(spherePosition, GroundedRadius, GroundLayers,
        QueryTriggerInteraction.Ignore);

    // update animator if using character
    if (_hasAnimator)
    {
        _animator.SetBool(_animIDGrounded, Grounded);
    }
}
```

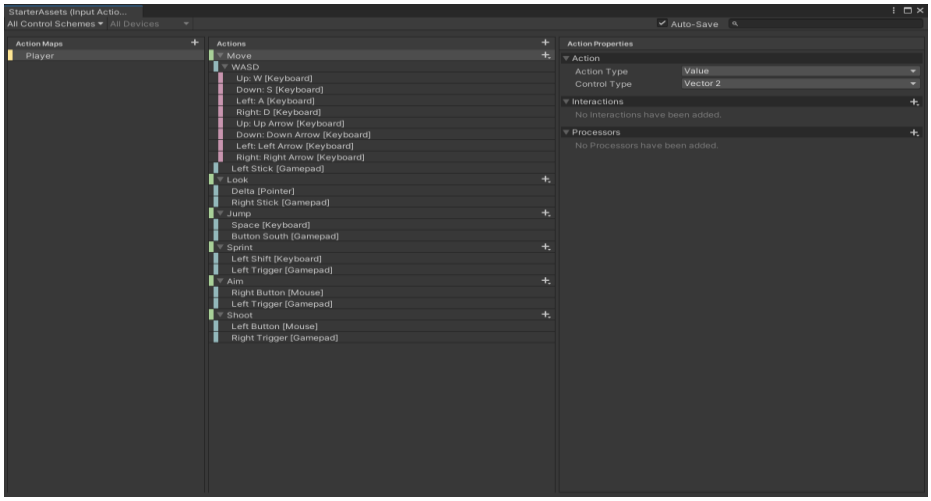
4.3 카메라 회전

ThirdPersonController 의 스크립트에서 CameraRotation 메서드는 다음과 같습니다.
입력 체크:입력이 있는지 확인하고, 조건이 충족되면 입력 장치의 유형(IsCurrentDeviceMouse 를 통해 판단)에 따라 시간 축적 계수 deltaTimeMultiplier 가 결정됩니다. 마우스 입력의 경우 이 계수는 1.0f 입니다.마우스 입력이 아닌 경우 시간 증가분 Time.deltaTime 입니다.
대상 각도 업데이트:입력 및 민감도(Sensitivity)에 따라 대상의 편향각 _cinemachineTargetYaw 및 대상 푸시업 각도 _cinemachineTargetPitch 를 업데이트한 다음 대상 편향각과 대상 푸시업 각도를 특정 범위로 제한하여 편향각에 대한 구체적인 제한 범위가 없고 푸시업 각도는 Bottoomp 과 Clamp 사이에 제한됩니다.(카메라 올려다보기 및 내려다보기 각도 제한)

적용 회전:마지막으로, CinemachineCameraTarget 의 회전을 설정하여 Cinemachine 카메라가 대상을 따라 회전하도록 합니다. 회전은 대상 푸시업 각도와 카메라 각도 오버레이 값인 CameraAngleOverride, 대상 편향각 및 고정된 z 축의 회전으로 구성된 4 원수로 결정됩니다.

```
private void CameraRotation()
{
    // if there is an input and camera position is not fixed
    if (_input.look.sqrMagnitude >= _threshold && !LockCameraPosition)
    {
        //Don't multiply mouse input by Time.deltaTime;
        float deltaTimeMultiplier = IsCurrentDeviceMouse ? 1.0f : Time.deltaTime;

        _cinemachineTargetYaw += _input.look.x * deltaTimeMultiplier*Sensitivity;
        _cinemachineTargetPitch += _input.look.y * deltaTimeMultiplier * Sensitivity;
    }

    // clamp our rotations so our values are limited 360 degrees
    _cinemachineTargetYaw = ClampAngle(_cinemachineTargetYaw, float.MinValue, float.MaxValue);
    _cinemachineTargetPitch = ClampAngle(_cinemachineTargetPitch, BottomClamp, TopClamp);

    // Cinemachine will follow this target
    CinemachineCameraTarget.transform.rotation = Quaternion.Euler(_cinemachineTargetPitch + CameraAngleOverride,
        _cinemachineTargetYaw, 0.0f);
}
```

4.4 시스템 입력



새 입력 시스템에서 move, look, jump 와 같은 몇 가지 동작을 정의했습니다. 설정 프로세스를 소개합니다.

동작 정의:행위 자산에 "move" 라는 Action 을 만듭니다.

이벤트 트리거:move Action 의 Performed 이벤트가 트리거되면 GameObject 를 호출합니다.SendMessage() 메서드, 반사 시스템을 통해 현재 GameObject 에서 "OnMove"라는 메서드를 찾아 실행합니다.

메시지 처리:StarterAssetsInputs 스크립트를 만들어 메시지를 받습니다.OnMove, OnLook, OnJump 등의 메서드를 정의하여 해당 입력 이벤트를 처리합니다.

```
public void OnMove(InputValue value)
{
    MoveInput(value.Get<Vector2>());
}

0 个引用
public void OnLook(InputValue value)
{
    if(cursorInputForLook)
    {
        LookInput(value.Get<Vector2>());
    }
}

0 个引用
public void OnJump(InputValue value)
{
    JumpInput(value.isPressed);
}

0 个引用
public void OnSprint(InputValue value)
{
    SprintInput(value.isPressed);
}
0 个引用
public void OnAim(InputValue value)
{
    AimInput(value.isPressed);
}
0 个引用
public void OnShoot(InputValue value)
{
```

4.5 인물 총기 소지 동작

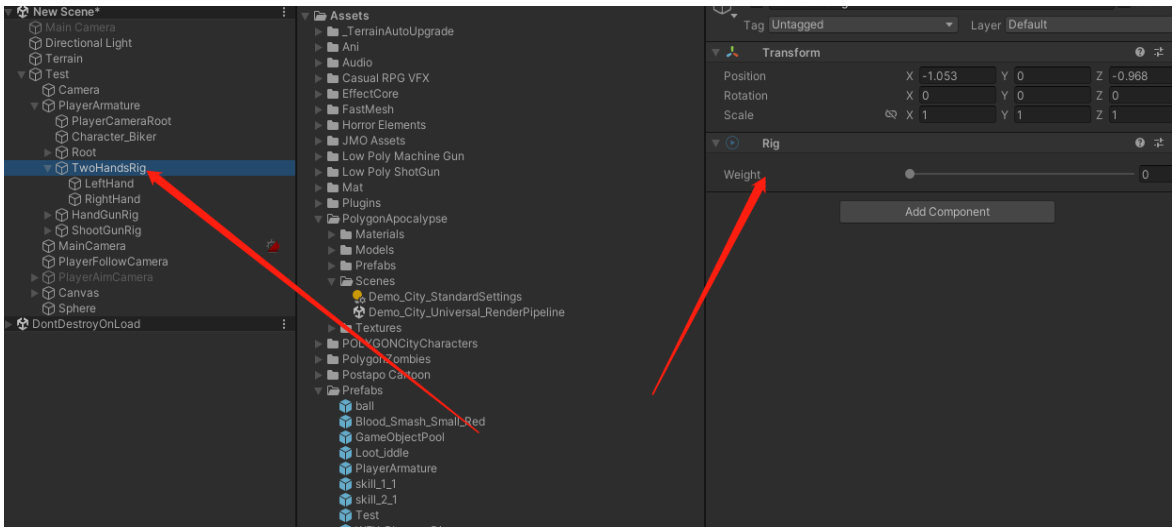
총기를 캐릭터의 손과 완벽하게 밀착시키기 위해 애니메이션과 IK 를 바인딩하는 방식을 채택했다.

IK 바인딩:IK(Inverse Kinematic 역동력학은 뼈의 끝 노드를 기준으로 다른 모 노드의 위치를 추산하는 방법입니다.예를 들어 손의 위치를 통해 손목, 팔꿈치의 뼈의 위치를 추산한다.)AnimationRig 플러그인을 가져오고 플레이어에게 RigBuilder 구성 요소를 추가하여 왼손과 오른손을 관리하는 IK 대상을 만듭니다.

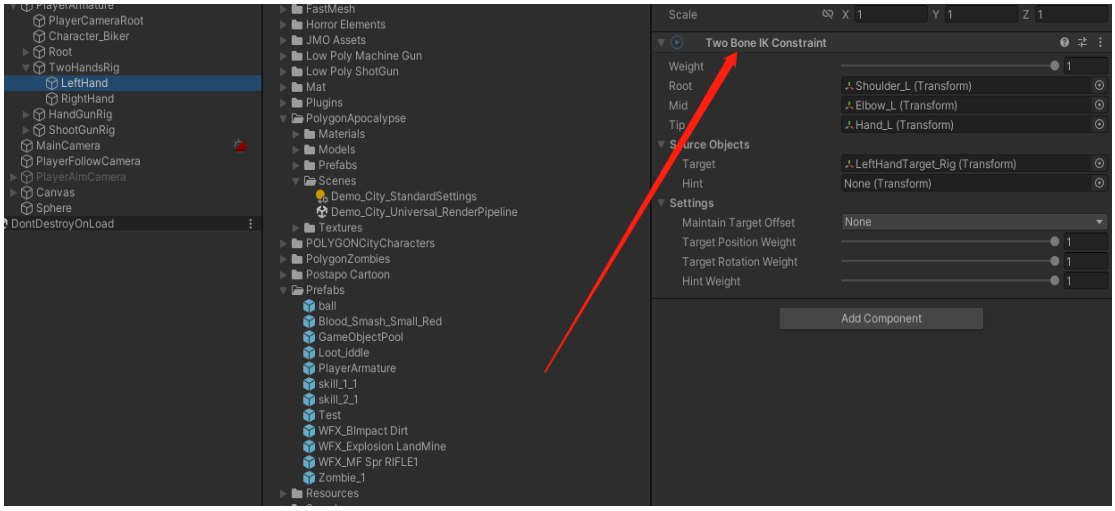
TwoBoneIKConstraint : 각 손에 각각 TwoBoneIKConstraint 구성 요소를 추가하면 TwoBone IK 구속을 통해 두 게임 객체에 대한 간단한 계층 구조 제어를 반전시켜 지체의 뾰족한 끝이 Target 위치에 도달할 수 있도록 하고, 추가 Hint GameObject 를 통해 지체가 구부러졌을 때 향해야 할 방향을 지정할 수 있으며, 손목과 팔꿈치의 뼈 위치를 손 위치에 따라 추산할 수 있다.

대상 점 설정:손이 총기와 일치하도록 표적점을 설정합니다.총기 물체 아래에 목표 지점을 만들고 스크립트에서 이 지점의 위치를 업데이트합니다. 여러 종류의 총기는 여러 종류의 IK 에 대응합니다. 이때 rig 의 가중치를 관리해야 합니다. 필요한 rig 가중치를 1, 필요하지 않은 것은 0 으로 설정해야 합니다.같은 총기는 총을 드는 동작과 대기 동작에 대응하기 때문에 전환해야 한다. 여기서 사용하는 방법은 두 가지 상태에서 target 의 위치를 기록하고 그 위치의 변화를 애니메이션으로 만드는 것이다.

인물 아래에 빈 물체 만들기, Rig 구성 요소 추가



빈 물체에 두 개의 하위 물체를 첨가하여 각각 왼손과 오른손의 IK 를 관리하는 데 사용한다



이곳의 루트에서 유저의 어깨 위치, mid 는 유저의 팔꿈치 위치, tip 는 유저의 손 위치입니다.target 은 목표 지점 (즉, 총기의 대응 위치로 총기 물체 아래에 점을 만들어야 함) 으로 지점의 위치와 회전을 끊임없이 조절하여 손과 총기를 최대한 밀착시킨다.

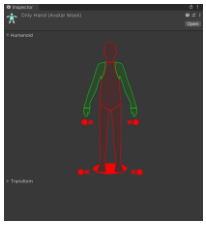
(IK 조정 전)

(IK 조정 후)

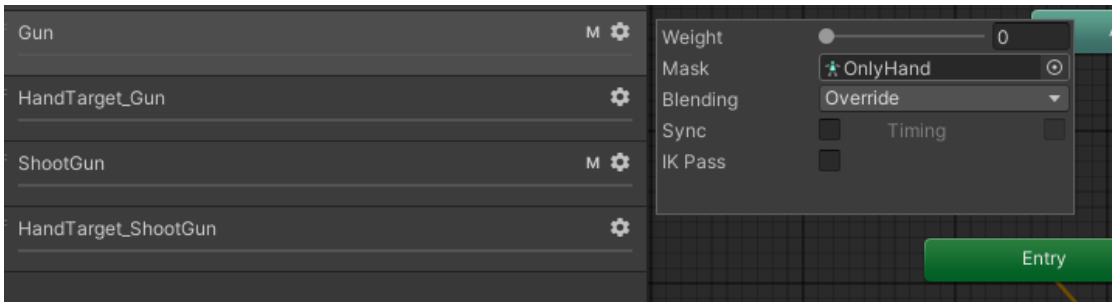
4.6 애니메이션 마스크

이동할 때 사격을 해야 한다. 이때 애니메이션을 쏘면 애니메이션과 걷기

애니메이션이 충돌한다. 내가 원하는 목표 효과는 상체는 총을 들고 조준하고 하체는 애니메이션을 이동하는 것이다. 이때 애니메이션 마스크를 사용해야 한다.
애니메이션 마스크를 만들려면 다음과 같이 하십시오.
손 애니메이션만 사용하는 애니메이션 마스크를 만듭니다.
마스크 적용:이 마스크는 특정 애니메이션 레이어에 애니메이션 컨트롤러에서 적용되어 손 애니메이션만 활성화됨



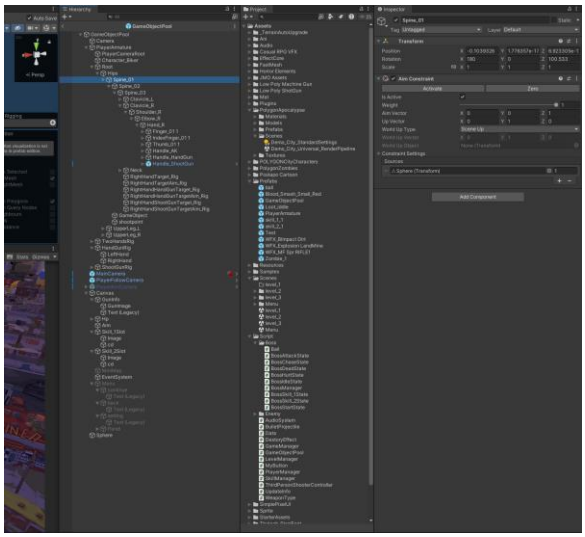
여기에서 나는 애니메이션 마스크를 만들었고 그 손 애니메이션만 사용했다.



애니메이션 컨트롤러에서 Only 마스크를 mask 에 지정하면 애니메이션의 손만 활성화됩니다.

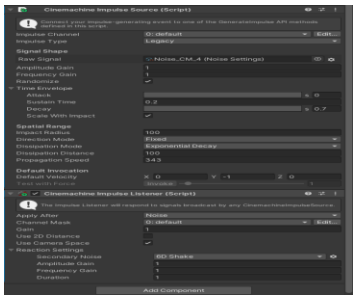
4.7 인물 조준

인물은 조준 위치에 따라 상반신이 구부러지고 실제 효과를 시뮬레이션해야 하며 인물 조준의 논리를 실현하는 방법은 AimConstraint 를 사용하는 것이다.
AimConstraint: 인물 상반신의 노드에 AimConstraint 구성 요소를 추가합니다. sources 는 빈 물체를 부여합니다. 이 빈 물체의 위치는 사격의 스크립트에서 목표를 지정하여 상반신이 조준하는 물체의 위치에 따라 조정됩니다.



4.8 카메라 떨림

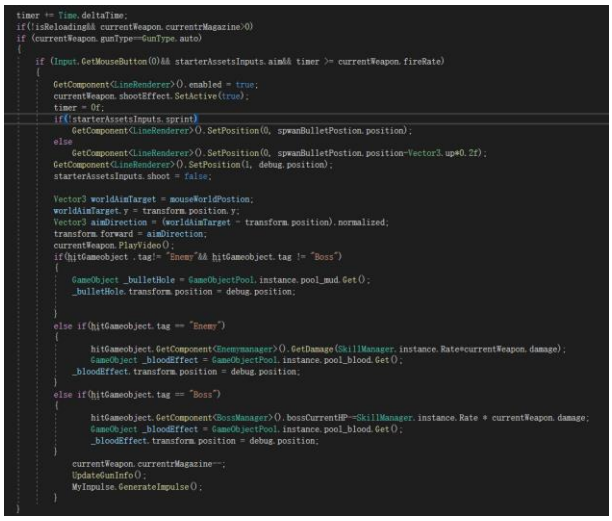
총기 후좌력 효과를 시뮬레이션하기 위해 사격 시 스크린을 흔들다:
Cinemachine 구성 요소 사용: 두 개의 Cinemachine 구성 요소를 표적 카메라에 마운트하고 사격 시 GenerateImpulse() 메서드를 호출하여 디더링 효과를 구현합니다.



4.9 총기 부분

```
private void Update()
{
    Vector3 mouseWorldPostion = Vector3.zero;
    Vector2 screenCenterPoint = new Vector2(Screen.width / 2f, Screen.height / 2f);
    Ray ray = Camera.main.ScreenPointToRay(screenCenterPoint);

    if (Physics.Raycast(ray, out RaycastHit raycastHit, 999f, aimColliderLayerMask))
    {
        debug.position = raycastHit.point;
        mouseWorldPostion = raycastHit.point;
        hitGameobject = raycastHit.collider.gameObject;
    }

    if (starterAssetsInputs.aim)
    {
        aimVirtualCamera.gameObject.SetActive(true);
        thirdPersonController.SetSensitivity(aimSensitivity);
        thirdPersonController.SetRotateOnMove(false);

        Vector3 worldAimTarget = mouseWorldPostion;
        worldAimTarget.y = transform.position.y;
        Vector3 aimDirection = (worldAimTarget - transform.position).normalized;

        transform.forward = Vector3.Lerp(transform.forward, aimDirection, Time.deltaTime * 20f);
        animator.SetBool("Aiming", true);
        aimConstraint.enabled = true;
    }
    else
    {
        aimVirtualCamera.gameObject.SetActive(false);
        thirdPersonController.SetSensitivity(normalSensitivity);
        thirdPersonController.SetRotateOnMove(true);
        animator.SetBool("Aiming", false);
        aimConstraint.enabled = false;
    }
```

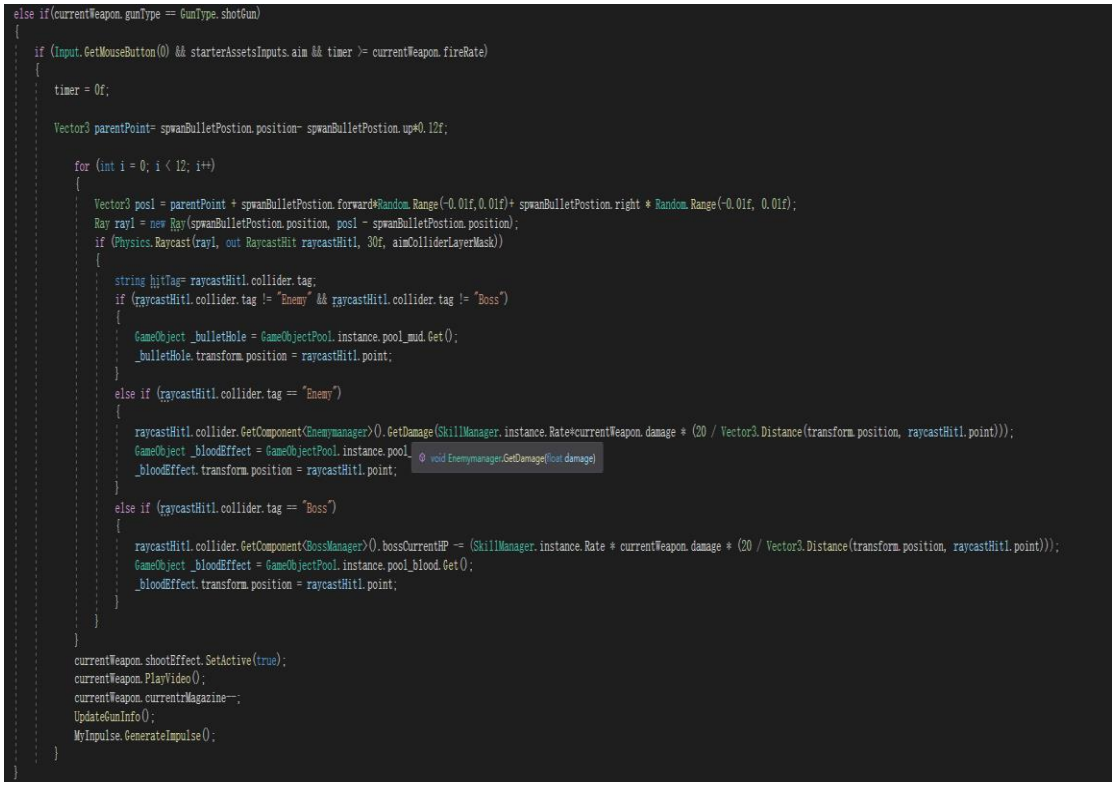
사격 부분의 논리는 ThirdPersonShooterController 스크립트에 있습니다.먼저, Vector3 유형의 변수 mouseWorldPostion 을 선언하고 0 벡터로 초기화했습니다.그런 다음 화면 중심의 좌표를 나타내는 Vector2 유형 변수 screenCenterPoint 를 생성하여 화면 폭과 높이의 절반을 가져와서 결정합니다.다음으로, Camera.main 을 통해 ScreenPointToRay(screenCenterPoint)는 메인 카메라에서 화면 중심을 가리키는 광선 ray 를 만듭니다.
Physics 를 사용하는 경우 Raycast 방법은 방사선을 따라 충돌 감지에 성공합니다 (즉, 방사선이 지정된 레이어 마스크 aimColliderLayerMask 의 물체와 최대 999f 내에서 충돌함). 충돌점의 위치를 debug.position 및 mouseWorldPostion 에 저장하고 충돌한 게임 이미지를 hitGameobject 로 기록합니다.일반 사격:
일반 사격
Shoot 유형 중 무기 유형이 권총과 AK 인 경우



```
timer += Time.deltaTime;
if(!isReloading&& currentWeapon.currentrMagazine>0)
if (currentWeapon.gunType==GunType.auto)
{
    if (Input.GetMouseButton(0)&& starterAssetsInputs.aim&& timer >= currentWeapon.fireRate)
    {
        GetComponent<LineRenderer>().enabled = true;
        currentWeapon.shootEffect.SetActive(true);
        timer = 0f;
        if(!starterAssetsInputs.sprint)
            GetComponent<LineRenderer>().SetPosition(0, spwanBulletPostion.position);
        else
            GetComponent<LineRenderer>().SetPosition(0, spwanBulletPostion.position-Vector3.up*0.2f);
        GetComponent<LineRenderer>().SetPosition(1, debug.position);
        starterAssetsInputs.shoot = false;

        Vector3 worldAimTarget = mouseWorldPostion;
        worldAimTarget.y = transform.position.y;
        Vector3 aimDirection = (worldAimTarget - transform.position).normalized;
        transform.forward = aimDirection;
        currentWeapon.PlayVideo();
        if(hitGameobject .tag!= "Enemy"&& hitGameobject.tag != "Boss")
        {
            GameObject _bulletHole = GameObjectPool.instance.pool_mud.Get();
            _bulletHole.transform.position = debug.position;

        }
        else if(hitGameobject.tag == "Enemy")
        {
            hitGameobject.GetComponent<Enemymanager>().GetDamage(SkillManager.instance.Rate*currentWeapon.damage);
            GameObject _bloodEffect = GameObjectPool.instance.pool_blood.Get();
            _bloodEffect.transform.position = debug.position;
        }
        else if(hitGameobject.tag == "Boss")
        {
            hitGameobject.GetComponent<BossManager>().bossCurrentHP-=SkillManager.instance.Rate * currentWeapon.damage;
            GameObject _bloodEffect = GameObjectPool.instance.pool_blood.Get();
            _bloodEffect.transform.position = debug.position;
        }
        currentWeapon.currentrMagazine--;
        UpdateGunInfo();
        MyInpulse.GenerateImpulse();
    }
}
```

여기서 나는 총알의 궤적을 대체하기 위해 LineRenderer 구성 요소를 사용했다. 마우스 왼쪽 버튼을 길게 눌렀을 때 linerenderer 구성 요소와 특수 효과를 활성화하고 LineRenderer 시작점을 총알의 생성 위치로 설정하고 종점은 조준하는 물체의 위치로 설정했다.다른 tag 물체를 명중시킬 때 대응하는 이벤트를 실행합니다.

산탄 사격:



```
else if(currentWeapon.gunType == GunType.shotGun)
{
    if (Input.GetMouseButton(0) && starterAssetsInputs.aim && timer >= currentWeapon.fireRate)
    {
        timer = 0f;

        Vector3 parentPoint= spwanBulletPostion.position- spwanBulletPostion.up*0.12f;

            for (int i = 0; i < 12; i++)
            {
                Vector3 pos1 = parentPoint + spwanBulletPostion.forward*Random.Range(-0.01f,0.01f)+ spwanBulletPostion.right * Random.Range(-0.01f, 0.01f);
                Ray ray1 = new Ray(spwanBulletPostion.position, pos1 - spwanBulletPostion.position);
                if (Physics.Raycast(ray1, out RaycastHit raycastHit1, 30f, aimColliderLayerMask))
                {
                    string hitTag= raycastHit1.collider.tag;
                    if (raycastHit1.collider.tag != "Enemy" && raycastHit1.collider.tag != "Boss")
                    {
                        GameObject _bulletHole = GameObjectPool.instance.pool_mud.Get();
                        _bulletHole.transform.position = raycastHit1.point;
                    }
                    else if (raycastHit1.collider.tag == "Enemy")
                    {
                        raycastHit1.collider.GetComponent<Enemymanager>().GetDamage(SkillManager.instance.Rate*currentWeapon.damage * (20 / Vector3.Distance(transform.position, raycastHit1.point)));
                        GameObject _bloodEffect = GameObjectPool.instance.pool
                        _bloodEffect.transform.position = raycastHit1.point;
                    }
                    else if (raycastHit1.collider.tag == "Boss")
                    {
                        raycastHit1.collider.GetComponent<BossManager>().bossCurrentHP -= (SkillManager.instance.Rate * currentWeapon.damage * (20 / Vector3.Distance(transform.position, raycastHit1.point)));
                        GameObject _bloodEffect = GameObjectPool.instance.pool_blood.Get();
                        _bloodEffect.transform.position = raycastHit1.point;
                    }
                }
            }
        currentWeapon.shootEffect.SetActive(true);
        currentWeapon.PlayVideo();
        currentWeapon.currentrMagazine--;
        UpdateGunInfo();
        MyInpulse.GenerateImpulse();
    }
}
```

void Enemymanager.GetDamage(float damage)

산탄총 부분의 설계는 산탄총의 근접 상해 고탄환 밀집의 특징을 모의하기 위해 전체적인 논리는 총구의 일정 거리 위치에서 동일한 평면에 있는 12 개의 점을 생성하여 총구 위치와 12 개의 지점을 통해 12 개의 방사선을 얻을 수 있는 다음 일반 사격과 유사한 논리를 진행하는 것이다.여기에는 방사선의 길이가 제한되어 있으며, 물체를 맞히는 거리에 따라 피해에 계수를 곱한다.

4,10 스킬 부분

기술 1

```
public void Skill_1()
{
    skill1CanUse = false;
    Instantiate(skill1Effect1_1, player.position, Quaternion.identity);
    Collider[] colliders = Physics.OverlapSphere(player.position, 9);
    foreach (var item in colliders)
    {
        if (item.GetComponent<Enemymanager>() != null)
        {
            item.GetComponent<Enemymanager>().GetDamage(50);
            item.GetComponent<Enemymanager>().Debuff(PlayerManager.instance.currentHealth>50?0.5f:0f);
            item.GetComponent<Enemymanager>().ResetDebuff(lockedTime);
            GameObject go = Instantiate(skill1EffectVertigo, item.transform);
            go.transform.position += Vector3.up * 1.8f;
            Destroy(go, lockedTime);

        }
        if (item.GetComponent<BossManager>() != null)
        {
            item.GetComponent<BossManager>().bossHP-=300f;
            item.GetComponent<BossManager>().Debuff();
            item.GetComponent<BossManager>().ResetDebuff(5);
            GameObject go = Instantiate(skill1EffectVertigo, item.transform);
            go.transform.position += Vector3.up * 4f;
            go.transform.localScale *= 1.5f;
            Destroy(go, 5);

        }
    }
    StartCoroutine(StartCD(skill1CD,skill1Icon1,true));
}
```

캐릭터 스킬을 추가하여 게임의 즐거움을 더하고, 스킬 주요 논리는 Physics.OverlapSphere 의 클래스, 특수 효과를 생성하고, 플레이어의 위치를 중심으로 일정한 반경에서 구형 범위 내의 모든 적을 획득하고, 그에 대한 채혈 논리를 진행하며, 서로 다른 해로운 효과 방법을 실행한다.그런 다음 스킬 CD 를 재설정하기 위해 스킬 함수를 실행합니다.

4.11 스킬 CD

```
IEnumerator StartCD(float time,Image targetIcon,bool isskill1)
{
    float skillTimer = time;
    while (skillTimer>0)
    {
        skillTimer -= Time.deltaTime;
        targetIcon.fillAmount = skillTimer / time;
        yield return null;
    }
    if (isskill1)
        skill1CanUse = true;
    else
        skill2CanUse = true;
}
```

기술이 CD (cool down) 에 들어가면 세 개의 매개 변수를 사용합니다. 하나는 부동 소수점 time 으로 냉각 시간을 나타내고, 하나는 Image 형식의 매개 변수 targetIcon (냉각 진행률을 표시하는 아이콘일 수 있음) 이고, 하나는 부울 값 isskill1 인지 기술 2 인지 구분하는 데 사용됩니다.이 템플릿에서는 먼저 들어오는 냉각 시간 타임을 로컬 변수 skillTimer 에 할당합니다.그런 다음 skillTimer 가 0 보다 크면 다음 작업을 수행하는 루프로 이동합니다.

(1) 순환마다 skillTimer 에서 시간 증가분 Time.deltaTime 을 빼면 냉각 시간이 줄어듭니다.

(2) 들어오는 targetIcon 의 충전량(fillAmount)을 현재 남은 냉각 시간 skillTimer 와 총 냉각 시간 time 의 비율로 설정하여 냉각 진행률을 시각적으로 표시합니다.

(3) yield return null 을 사용하여 다음 프레임이 계속될 때까지 진행을 중지합니다.
쿨타임 소진(skillTimer 미만은 0) 시 스킬 1 인 경우 스킬 1 을 사용할 수 있습니다.isskill1 이 거짓이면 스킬 2 를 나타냅니다.

4.12 괴물

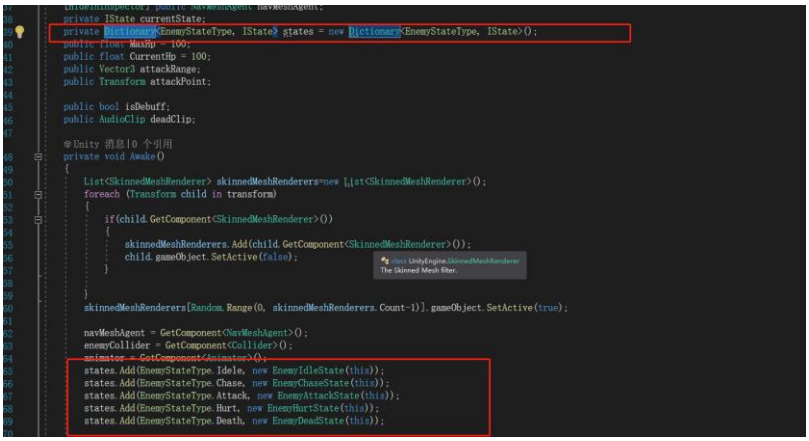
모든 몬스터 ai 논리는 사용자 정의 애니메이션 상태기를 기반으로 합니다.우선, 나는 먼저 인터페이스 (port) 를 써야 한다. 이곳의 몇 가지 함수는 각각 이 상태에서 실행하기 시작한 함수, 이 상태에서 update 함수에서 실행된 함수, 이 상태에서 fixedupdate 함수에서 실행된 함수, 이 상태에서 실행된 함수를 종료하는 것이다.

```
public interface IState
{

    void OnEnter();
    - 引用
    void OnUpdate();
    14 个引用
    void OnFixedUpdate();
    15 个引用
    void OnExit();
}
```

그런 다음 각 상태에서 적의 스크립트를 작성해야 합니다.예를 들어 추격 상태, 여기서 나는 EnemyChase 라는 스크립트를 정의한다. 이 스크립트는 Monobehaviour 클래스를 계승하지 않고 위에서 내가 정의한 인터페이스를 직접 계승한다.그런 다음 스크립트에서 인터페이스를 구현하는 방법:

```
2 个引用
public class EnemyChaseState : IState
{
    private Enemymanager enemy;
    1 个引用
    public EnemyChaseState(Enemymanager enemy)
    {
        this.enemy = enemy;
    }
    // Start is called before the first frame update
    3 个引用
    public void OnEnter()
    {
        enemy.GetPlayerTransform();
        enemy.animator.SetBool("chase", true);
        enemy.navMeshAgent.SetDestination(enemy.player.position);
        enemy.navMeshAgent.speed = enemy.chaseSpeed;
        Debug.Log("进入追击");
    }

    3 个引用
    public void OnExit()
    {

    }

    2 个引用
    public void OnFixedUpdate()
    {

    }

    3 个引用
    public void OnUpdate()
    {
        enemy.GetPlayerTransform();

        if (enemy.player != null)
        {
            if(enemy.distance < enemy.chaseDistance)
            {
                if (enemy.distance > enemy.attackDistance)
                {
未找到相关问题
错误 0   展示 127 个警告中的 0 个   展示 43 个消息中的 0 个   生成 + IntelliSense
说明
```

템플릿에 수행해야 할 논리를 쓰면 됩니다. 다른 상태도 마찬가지입니다.그런 다음 상태를 관리하기 위한 스크립트를 만들어야 합니다. 여기서 Enemy Manager 라고 명명하고 적의 다양한 상태를 매거한 다음 Dictionary 를 만들어 다양한 상태와 앞에 대응하는 인터페이스를 연결합니다.



그런 다음 상태 변환을 위한 함수 템플릿을 정의해야 합니다.

```
public void TransitionState(EnemyStateType type)
{
    if(currentState!=null)
    {
        currentState.OnExit();
    }
    currentState = states[type];
    currentState.OnEnter();
}
```

이 함수는 현재 상태의 OnExit 메서드를 실행한 다음 대상 상태로 변환하고 새 상태의 OnEnter 메서드를 실행합니다.UpDate 메서드에서 현재 상태의 OnUpdate 메서드를 실행합니다.기본 논리는 이렇다.Boss 도 마찬가지였다. 다만 로직을 약간 고쳤을뿐 전반 구조는 모두 같았다.

5. 결론 및 향후연구

이상에서는 3D Unity 사격 난관 돌파 게임인'좀비기'의 개발 과정과 실현 세부 사항을 상세히 소개하였다.Unity 엔진과 C# 프로그래밍 언어를 사용하여 NVIDIA PhysX 물리 엔진을 통합하여 게임의 사실성과 재생성을 향상시키는 Windows 플랫폼 기반 인터랙티브 게임을 성공적으로 구축했습니다. 게임 디자인에서 난이도가 점차 높아지는 세 개의 관문을 설정하고 다양한 무기 (권총, 소총, 산탄총) 와 스킬 시스템을 도입하여 게이머의 게임 경험을 풍부하게 하고 각 관문을 통과한 후 스킬 체험을 향상시켰습니다.개발 과정에서 우리는 다양한 기술적 도전에 부딪혔다. 예를 들어 캐릭터 이동, 시각 디자인, 괴물 AI 등이다. 이런 문제를 해결하기 위해 우리는 서로 다른 모듈화 디자인 사고방식을 채택했다. Unity 가 제공하는 풍부한 도구와 지역사회 자원을 이용하여 최종적으로 원활한 게임 조작과 풍부한 상호작용 체험을 성공적으로 실현했다.본 프로젝트를 통해 저는 개발 능력과 기술 수준을 향상시켰을 뿐만 아니라 슈팅 게임의 디자인과 실현에 대해 더욱 깊이 이해하게 되었습니다.앞으로 나는 이 기초에서

진일보 최적화와 확장을 진행하고 더욱 많은 혁신적인 놀이방법과 메커니즘을 모색하여 끊임없이 변화하는 시장수요와 게이머들의 기대를 만족시키기를 기대한다.