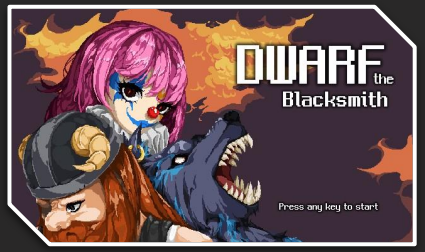
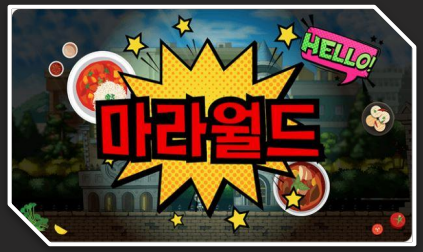
프로젝트 목록

01



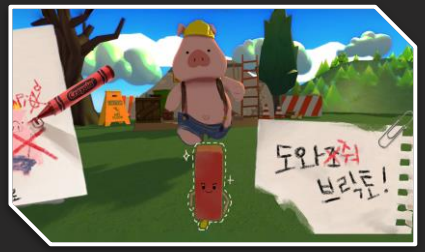
드워프 더 블랙스미스

02



마라월드

03



도와줘 브릭토!

01

드워프 더 블랙스미스

(Dwarf The Blacksmith)

* 프로그래밍
* 이펙트

2D 플랫포머 액션 어드벤처 게임

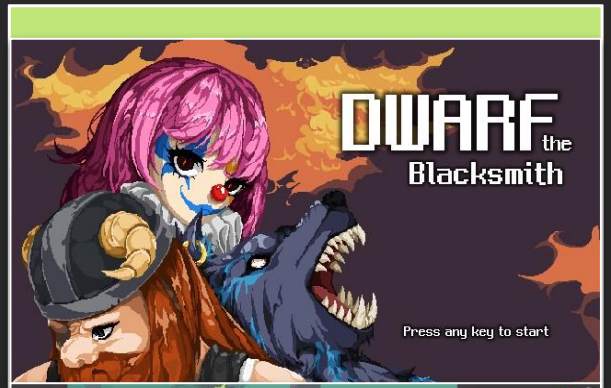
계원예술대학교 졸업작품

2024.06 - 2024.09 / 6개월 / 6인 개발

Unity Engine

C#

Aseprite



https://youtu.be/bJ7YGtzRQml

1001010101010100

1010010101001010

핵심 기술 (FSM 디자인 패턴을 PC/NPC에 적용)

PlayerMoveController.cs 중 일부

```
private void CheckMovementDirection()
{
    // 이동 방향에 따라 캐릭터 뒤집기
    if (isFacingRight && movementInputDirection < 0)
    {
        Flip();
    }
    else if (!isFacingRight && movementInputDirection > 0)
    {
        Flip();
    }

    // 캐릭터가 걷는 중인지 확인
    if (Mathf.Abs(rb.velocity.x) >= 0.01f)
    {
        isWalking = true;
    }
    else
    {
        isWalking = false;
    }

    // 대쉬 상태 처리
    if (Input.GetButtonDown("Dash") && movementInputDirection != 0)
    {
        if (Time.time >= (lastDash + dashCoolDown))
        {
            isDashing = true;
            dashTimeLeft = dashTime;
        }
    }
```

```
// 점프 상태 처리
if (Input.GetButtonDown("Jump"))
{
    if (isGrounded)
    {
        NormalJump();
    }
    else if (amountOfJumpsLeft > 0)
    {
        NormalJump();
    }
}

// 벽 슬라이딩 상태 처리
if (isTouchingWall && movementInputDirection == facingDirection && rb.velocity.y <
{
    isWallSliding = true;
}
else
{
    isWallSliding = false;
}
```

이전 코드의 문제점 ⚠

1. 상태 관리가 분산되었다.

- 다양한 상태를 한 메서드에서 처리하려다 보니 코드가 길어지고 복잡해졌다.

- 각 상태의 로직이 명확히 분리되지 않아 특정 상태를 디버깅하거나 수정하기 어려웠다.

2. 유지보수가 어려웠다.

- 여러 상태를 확인하기 위해 조건문이 중첩되어 있었다.

- 새로운 상태를 추가하거나 수정할 때 기존 조건문을 수정하거나 추가해야했다.

3. 상태 전환이 명확하지 않았다.

- 상태 전환 로직(isDashing, isWallSliding, 등)이 여러 곳에 분산되어 있어, 상태 간 전환 흐름을 파악하기 어려웠다.

핵심 기술 (FSM 디자인 패턴을 PC/NPC에 적용)

```
public class PlayerMoveState : PlayerState
{
    public PlayerMoveState(Player player, PlayerStateMachine stateMachine, string animBo
        : base(player, stateMachine, animBoolName)
    {
    }

    public override void Enter()
    {
        base.Enter();
        player.SetVelocity(player.moveSpeed * player.InputHandler.NormalizedInputX, 0);
    }

    public override void Update()
    {
        base.Update();

        // 이동 중 정지 시 Idle 상태로 전환
        if (player.InputHandler.NormalizedInputX == 0)
        {
            stateMachine.ChangeState(player.idleState);
        }

        // 대쉬 상태로 전환
        if (player.InputHandler.DashInput)
        {
            stateMachine.ChangeState(player.dashState);
        }

        // 점프 상태로 전환
        if (player.InputHandler.JumpInput && player.IsGrounded())
        {
            stateMachine.ChangeState(player.jumpState);
        }
    }
}
```

FSM 기반으로 개선된 코드

1. 상태별 동작을 분리하였다.

- 이동 상태는 PlayerMoveState에서만 관리되며, 다른 상태는 각각의 상태 클래스에서 관리한다.

- 상태 로직이 독립적이어서 이해하기 쉽고 유지보수가 용이하다.

2. 조건문을 간소화 하였다.

- 이전의 복잡했던 조건문들이 각 상태별로 나뉘어 더 간결해졌다.

- 상태 전환 조건을 명확히 정의하여 혼란을 줄였다.

3. 확장성이 높아졌다.

- 새로운 상태(예: 공격, 방어)를 추가할때 새로운 상태 클래스를 작성하고 FSM에 등록하였다.

마라월드

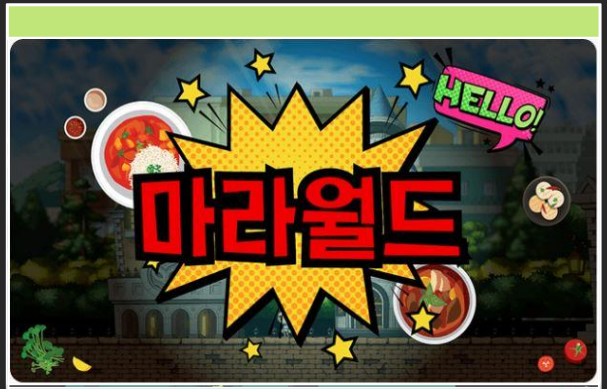
* 프로그래밍

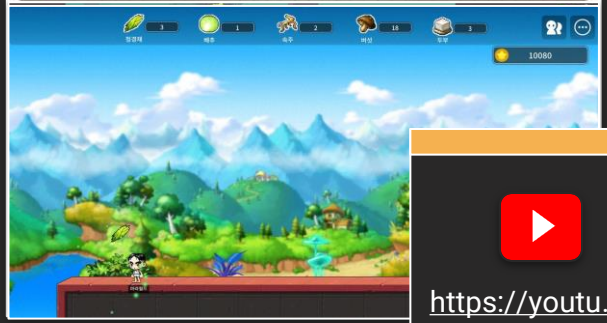
채집형 경제 시뮬레이션 게임

2024.10 - 2024.11 / 1개월 / 2인 개발

Maple World Engine

Lua





https://youtu.be/2lqeje7m_o4

핵심 기술 [채집과 제작 시스템]

채집 횟수 확인 및 초기화 로직

```
boolean CheckGatherLimit()
{
    local currentTime = os.time()
    if currentTime - self.LastGatherTime >= 86400 then
        -- 하루가 지나면 초기화
        self.GatherCount = 0
        self.LastGatherTime = currentTime
        self.DailyBokchoyCount = 0
        self.DailyCabbageCount = 0
        self.DailyMungBeanSprout = 0
        print("[INFO] 하루가 지나 채집 횟수를 초기화합니다.")
    end

    -- 최대 채집 횟수 초과 확인
    if self.GatherCount >= self.MaxGatherLimit then
        print("[ERROR] 하루 최대 채집 횟수를 초과했습니다.")
        return false
    else
        return true
    end
}
```

하루가 지나면 채집 횟수를 초기화하고, 최대 채집 횟수를 초과하지 않는지 확인한다.

재료 아이템 채집 로직

```
_TimerService:SetTimerOnce(function()
    if not self:CheckGatherLimit() then
        print("[ERROR] 채집 제한에 도달하여 더 이상 채집할 수 없습니
        return
    end

    -- 채집 횟수 증가
    self.GatherCount = self.GatherCount + 1
    print("[INFO] 현재 채집 횟수: " .. tostring(self.GatherCount))

    -- 랜덤 아이템 획득
    local itemNames = {
        {"Cabbage", "Ingredient_Cabbage"},
        {"BokChoy", "Ingredient_BokChoy"},
        {"MungBeanSprout", "Ingredient_MungBeanSprout"}
    }
    local index = math.random(1, 3)
    local getItemName = itemNames[index]
    local itemId = getItemName[2]

    -- 획득한 아이템 카운트 갱신
    if getItemName[1] == "Cabbage" then
        self.DailyCabbageCount = self.DailyCabbageCount + 1
    elseif getItemName[1] == "BokChoy" then
        self.DailyBokchoyCount = self.DailyBokchoyCount + 1
    elseif getItemName[1] == "MungBeanSprout" then
        self.DailyMungBeanSprout = self.DailyMungBeanSprout + 1
    end

    -- 아이템 인벤토리에 추가
    local SimpleInventory = TriggerBodyEntity.SimpleInventory
    SimpleInventory:AddItem(itemId, 1)
end, 0.5)
```

플레이어가 채집 구역에 들어오면 0.5초 후 재료를 랜덤으로 획득한다.

데이터 로드 / 저장

```
void OnBeginPlay()
{
    local userDataStorage = _DataStorageService:GetUserDataStorage("GatheringData")
    local errorCode, loadData = userDataStorage:GetAndWait("GatheringData")
    if loadData then
        local gatherData = _HttpService:JSONDecode(loadData)
        self.GatherCount = gatherData.GatherCount or 0
        self.LastGatherTime = gatherData.LastGatherTime or os.time()
        self.DailyBokchoyCount = gatherData.DailyBokchoyCount or 0
        self.DailyCabbageCount = gatherData.DailyCabbageCount or 0
        self.DailyMungBeanSprout = gatherData.DailyMungBeanSprout or 0
    else
        -- 데이터 초기화
        self.GatherCount = 0
        self.LastGatherTime = os.time()
    end
}
```

```
void OnEndPlay()
{
    local userDataStorage = _DataStorageService:GetUserDataStorage("GatheringData")
    local gatherData = {
        GatherCount = self.GatherCount,
        LastGatherTime = self.LastGatherTime,
        DailyBokchoyCount = self.DailyBokchoyCount,
        DailyCabbageCount = self.DailyCabbageCount,
        DailyMungBeanSprout = self.DailyMungBeanSprout
    }
    local jsonData = _HttpService:JSONEncode(gatherData)
    userDataStorage:SetAsync("GatheringData", jsonData, function()
        print("[INFO] 채집 데이터 저장 완료")
    end)
}
```

저장된 상태를 로드하고 저장함으로써 사용자 경험 유지한다.

핵심 기술 [채집과 제작 시스템]

제작 요청 처리 로직

```
void OnCraftButtonClick()
{
    log("[UIExchange] CraftButton 클릭됨 - 제작 요청 설정")
    self.isCraftRequested = true
}
```

클라이언트에서 제작 버튼을 클릭했을 때 서버로 제작 요청을 보낸다.

```
void SendCraftRequestToServer()
{
    local slotData = {}
    for itemId, slotEntity in pairs(self.CraftSlots) do
        slotData[itemId] = slotEntity.UICraftItemSlot.ItemCount or 0
    end

    -- 서버에 제작 요청과 slotData 전달
    self:SendEventToServer("CraftMaratangRequest", slotData)
    log("[UIExchange] 서버로 제작 요청 전송: ", slotData)
}
```

클라이언트가 서버로 슬롯 데이터를 전달하고 제작 요청을 처리한다.

제작 조건 확인 로직

```
boolean CheckRecipe_Server()
{
    local requiredItems = {
        {"Ingredient_Meat", 10},
        {"Ingredient_BokChoy", 10},
        {"Ingredient_Cabbage", 10},
        {"Ingredient_MungBeanSprout", 10},
        {"Ingredient_Tofu", 10},
        {"Ingredient_Mushroom", 10}
    }

    for _, required in ipairs(requiredItems) do
        local itemIdRequired = required[1]
        local itemCountRequired = required[2]
        local slot = self.ServerSlots[itemIdRequired]

        if not slot or slot.ItemId ~= itemIdRequired or slot.ItemCount < itemCountRequired then
            log_error("[UIExchange] CheckRecipe_Server: 재료가 부족합니다. 필요 아이템: " .. itemIdRequired)
            return false
        end
    end

    log("[UIExchange] 모든 재료가 충족되었습니다.")
    return true
}
```

서버에서 제작 요청을 수신한 후, 필요한 재료가 충분한지 확인한다.

도와줘 브릭토!

* 프로그래밍

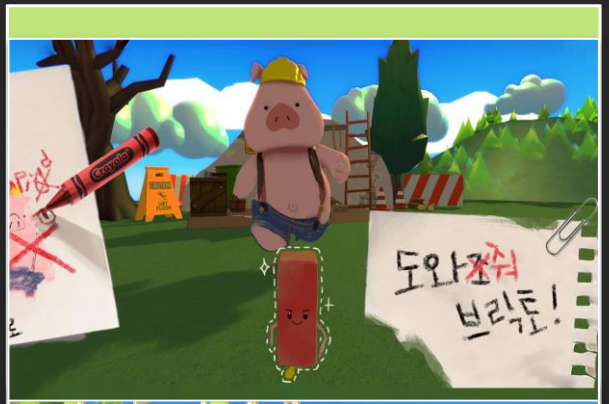
구출, 술래잡기 게임

레드브릭 게임잼

2024.11.01 - 2024.11.03 / 48시간 / 2인 개발

Red Brick Engine

Java



https://youtu.be/dGe1bIr2z5M

핵심 기술 (적 NPC의 상태와 레이더)

추적 및 이동 로직

```
function chasePlayer(dt) {
    moveTowardsTarget(dt, player.position);
    pig.lookAt(player.position.x, pig.position.y, player.position.z);
}

function moveToTarget(dt) {
    if (pig.position.distanceTo(targetBase.position) < 1) {
        visitBase(); // 목표 지점 도착 시 방문 처리
    } else {
        moveTowardsTarget(dt, targetBase.position);
        pig.lookAt(targetBase.position.x, pig.position.y, targetBase.position.z);
    }
}

function moveTowardsTarget(dt, targetPosition) {
    const direction = new THREE.Vector3();
    direction.subVectors(targetPosition, pig.position).normalize();
    direction.y = 0; // Y축 고정
    pig.position.add(direction.multiplyScalar(moveSpeed * dt));
}
```

플레이어가 감지 범위 (detectionRange) 안에 들어오면
추적 (run 애니메이션) 상태로 전환하고,
추적 중이지 않을 경우 무작위 목표 (Base) 로 이동한다.

기절상태 처리 로직

```
function applyStun(duration) {
    if (!isStunned) {
        isStunned = true;
        stunEndTime = Date.now() + duration;
        playAnimation("stun");
        targetBase = null;
        isFollowingPlayer = false;
    }
}
```

기절 시 이동 및 추적이 불가능 하고,
일정 시간 후 무작위 목표 (Base)로
복귀한다.

핵심 기술 (적 NPC의 상태와 레이더)

레이더 초기화

```
const circleGeometry = new THREE.RingGeometry(detectionRange - 0.1, detectionRange, 64);
const circleMaterial = new THREE.MeshBasicMaterial({
    color: radarColorDefault, // 기본 색상 (녹색)
    opacity: 0.2, // 투명도
    transparent: true
});
radarCircle = new THREE.Mesh(circleGeometry, circleMaterial);
radarCircle.rotation.x = -Math.PI / 2; // 평면 회전
radarCircle.position.set(pig.position.x, pig.position.y + radarHeight, pig.position.z);
WORLD.add(radarCircle);

const sweepGeometry = new THREE.CircleGeometry(detectionRange, 64, 0, Math.PI / 6);
const sweepMaterial = new THREE.MeshBasicMaterial({
    color: radarColorDefault, // 기본 색상
    opacity: radarOpacity, // 투명도
    transparent: true
});
radarSweep = new THREE.Mesh(sweepGeometry, sweepMaterial);
radarSweep.rotation.x = -Math.PI / 2; // 평면 회전
radarSweep.position.set(pig.position.x, pig.position.y + radarHeight, pig.position.z);
WORLD.add(radarSweep);
```

적 NPC의 레이더는 플레이어의 감지 여부를 시각적으로 표현하며, 레이더 색상, 투명도, 회전 등을 동적으로 업데이트합니다. 레이더는 radarCircle과 radarSweep으로 구성됩니다.

radarCircle :
원형 링 형태로 감지 범위를 나타낸다.
detectionRange에 따라 크기를 조정한다.

radarSweep :
부채꼴 형태로 레이더 스캔 효과를 구현하였다.
Math.PI / 6은 스캔 각도를 정의한다.

레이더 업데이트

```
radarCircle.position.set(pig.position.x, pig.position.y + radarHeight, pig.position.z);
radarSweep.position.set(pig.position.x, pig.position.y + radarHeight, pig.position.z);

// 레이더 회전
radarSweep.rotation.z -= dt * (isFollowingPlayer ? 2 : 1);

// 색상 변경
radarSweep.material.color.setHex(isFollowingPlayer ? radarColorAlert : radarColorDefault);
radarCircle.material.color.setHex(isFollowingPlayer ? radarColorAlert : radarColorDefault);

// 투명도 변경
radarSweep.material.opacity = radarOpacity * Math.abs(Math.cos(radarSweep.rotation.z * 2));
```

1. 레이더 위치 : 적 NPC의 현재 위치를 기반으로 radarCircle과 radarSweep의 위치를 갱신한다.

2. 스캔 회전 : radarSweep.rotation.z 값을 매 프레임마다 감소시켜 부채꼴이 회전하는 스캔 효과를 연출한다.

3. 색상 변경 : 플레이어가 감지되었을 때 빨간색(radarColorAlert), 그렇지 않으면 녹색(radarColorDefault)으로 색상을 변경한다.

4. 투명도 변화 : 레이더의 투명도는 회전 각도에 따라 주기적으로 변하며 시각적 역동성을 추가하였고, Math.abs(Math.cos(radarSweep.rotation.z * 2))는 투명도를 부드럽게 변화시킨다.