Radius-Based RPG Battle Mechanics

Share

**J**

For a unity game mechanic, and with only writing pseudocode, I'm curious how a radius based system would be for a turn based rpg battle system, where moving and attacking are only allowed in a radius around a unit and anywhere in the circle of the battle arena can be stood in. Attacks can have different radius as well as a speed Stat affecting the movement radius as well.

Edit

Here's a pseudocode implementation for a radius-based turn-based RPG battle system:

pseudocode

// Core Data Structures

class Unit {

position: Vector2

stats: {

health, maxHealth

speed // affects movement radius

attack, defense

}

abilities: List<Ability>

isAlive: boolean

team: TeamID

}

class Ability {

name: string

range: float // attack radius

damage: float

aoeRadius: float // 0 for single target

cooldown: int

currentCooldown: int

}

class BattleArena {

centerPoint: Vector2

radius: float

units: List<Unit>

currentTurnUnit: Unit

turnOrder: Queue<Unit>

}

// Core Functions

function IsWithinArena(position: Vector2, arena: BattleArena) -> boolean {

distance = Distance(position, arena.centerPoint)

return distance <= arena.radius

}

function GetMovementRadius(unit: Unit) -> float {

baseRadius = 5.0 // base movement units

speedModifier = unit.stats.speed \* 0.5 // speed affects radius

return baseRadius + speedModifier

}

function GetValidMovePositions(unit: Unit, arena: BattleArena) -> List<Vector2> {

validPositions = []

moveRadius = GetMovementRadius(unit)

// Sample points in a circle around unit

for angle in range(0, 360, 10): // sample every 10 degrees

for distance in range(0, moveRadius, 0.5): // sample distances

testPoint = unit.position + PolarToCartesian(distance, angle)

if IsWithinArena(testPoint, arena) AND

!IsOccupied(testPoint, arena.units) {

validPositions.add(testPoint)

}

return validPositions

}

function GetTargetsInRange(unit: Unit, ability: Ability, arena: BattleArena) -> List<Unit> {

targetsInRange = []

for targetUnit in arena.units:

if targetUnit == unit OR !targetUnit.isAlive:

continue

distance = Distance(unit.position, targetUnit.position)

if distance <= ability.range:

targetsInRange.add(targetUnit)

return targetsInRange

}

function MoveUnit(unit: Unit, targetPosition: Vector2, arena: BattleArena) -> boolean {

moveRadius = GetMovementRadius(unit)

distance = Distance(unit.position, targetPosition)

if distance > moveRadius:

return false // Out of movement range

if !IsWithinArena(targetPosition, arena):

return false // Outside arena bounds

if IsOccupied(targetPosition, arena.units):

return false // Position occupied

unit.position = targetPosition

return true

}

function ExecuteAbility(caster: Unit, ability: Ability, targetPosition: Vector2, arena: BattleArena) {

// Check if target is in range

distance = Distance(caster.position, targetPosition)

if distance > ability.range:

return // Target out of range

// Handle AoE

if ability.aoeRadius > 0:

affectedUnits = GetUnitsInRadius(targetPosition, ability.aoeRadius, arena)

for unit in affectedUnits:

if unit.team != caster.team:

ApplyDamage(unit, ability.damage, caster)

else:

// Single target

target = GetUnitAtPosition(targetPosition, arena)

if target AND target.team != caster.team:

ApplyDamage(target, ability.damage, caster)

ability.currentCooldown = ability.cooldown

}

// Turn Management

function InitializeBattle(arena: BattleArena) {

// Sort units by speed for turn order

sortedUnits = SortBySpeed(arena.units)

for unit in sortedUnits:

arena.turnOrder.enqueue(unit)

}

function ProcessTurn(arena: BattleArena) {

currentUnit = arena.turnOrder.dequeue()

if !currentUnit.isAlive:

return // Skip dead units

// Player/AI makes decisions here

action = GetPlayerAction(currentUnit)

switch action.type:

case MOVE:

MoveUnit(currentUnit, action.targetPosition, arena)

case ATTACK:

ExecuteAbility(currentUnit, action.ability, action.targetPosition, arena)

case MOVE\_AND\_ATTACK:

MoveUnit(currentUnit, action.movePosition, arena)

ExecuteAbility(currentUnit, action.ability, action.targetPosition, arena)

case WAIT:

// Do nothing

// Re-add to turn queue

arena.turnOrder.enqueue(currentUnit)

// Update cooldowns

for ability in currentUnit.abilities:

if ability.currentCooldown > 0:

ability.currentCooldown--

}

// Visualization Helpers

function DrawMovementRange(unit: Unit, arena: BattleArena) {

radius = GetMovementRadius(unit)

DrawCircle(unit.position, radius, BLUE\_TRANSPARENT)

// Clip to arena bounds

if Distance(unit.position, arena.centerPoint) + radius > arena.radius:

// Draw intersection with arena edge

DrawArcIntersection(unit.position, radius, arena)

}

function DrawAttackRange(unit: Unit, ability: Ability) {

DrawCircle(unit.position, ability.range, RED\_TRANSPARENT)

if ability.aoeRadius > 0:

// Show AoE indicator at mouse position

DrawCircle(mousePosition, ability.aoeRadius, ORANGE\_TRANSPARENT)

}

// Helper Functions

function Distance(a: Vector2, b: Vector2) -> float {

return sqrt((b.x - a.x)^2 + (b.y - a.y)^2)

}

function IsOccupied(position: Vector2, units: List<Unit>) -> boolean {

threshold = 0.5 // Units can't be closer than this

for unit in units:

if Distance(position, unit.position) < threshold:

return true

return false

}

function GetUnitsInRadius(center: Vector2, radius: float, arena: BattleArena) -> List<Unit> {

unitsInRadius = []

for unit in arena.units:

if Distance(center, unit.position) <= radius:

unitsInRadius.add(unit)

return unitsInRadius

}

This system provides:

1. **Flexible positioning** - Units can move anywhere within their movement radius (based on speed stat) as long as it's within the arena
2. **Range-based attacks** - Each ability has its own range, creating tactical decisions about positioning
3. **Arena boundaries** - The circular arena naturally limits movement and creates interesting edge positioning
4. **Speed matters** - Faster units can move further and act sooner in turn order
5. **AoE support** - Abilities can affect areas, not just single targets

The key advantages of this radius-based system are:

* More tactical depth than grid-based movement
* Natural feeling movement without artificial constraints
* Easy to visualize ranges with circles
* Smooth integration of speed stats affecting mobility
* Interesting positioning puzzles near arena edges

Retry

**J**

Would a system of very small units filling the arena as a sort of grid work for keeping track of where units are and for checking if a unit is inside a certain aoe attack, or should this be kept only in vector 3s?

Edit

Great question! Let me break down both approaches with their trade-offs:

**Spatial Grid Approach**

pseudocode

class SpatialGrid {

cellSize: float // e.g., 0.5 units

gridWidth, gridHeight: int

cells: Array2D<List<Unit>> // Multiple units can overlap cells

arenaCenter: Vector2

arenaRadius: float

}

function InitializeGrid(arenaRadius: float, cellSize: float) -> SpatialGrid {

grid = new SpatialGrid()

grid.cellSize = cellSize

grid.arenaRadius = arenaRadius

// Create grid covering the circular arena

gridDiameter = arenaRadius \* 2

grid.gridWidth = ceil(gridDiameter / cellSize)

grid.gridHeight = ceil(gridDiameter / cellSize)

// Initialize cells

for x in range(0, grid.gridWidth):

for y in range(0, grid.gridHeight):

grid.cells[x][y] = new List<Unit>()

return grid

}

function WorldToGrid(worldPos: Vector2, grid: SpatialGrid) -> GridCoord {

// Convert world position to grid coordinates

offsetPos = worldPos - (grid.arenaCenter - Vector2(grid.arenaRadius, grid.arenaRadius))

x = floor(offsetPos.x / grid.cellSize)

y = floor(offsetPos.y / grid.cellSize)

return GridCoord(x, y)

}

function UpdateUnitInGrid(unit: Unit, oldPos: Vector2, newPos: Vector2, grid: SpatialGrid) {

oldCell = WorldToGrid(oldPos, grid)

newCell = WorldToGrid(newPos, grid)

if oldCell != newCell:

grid.cells[oldCell.x][oldCell.y].remove(unit)

grid.cells[newCell.x][newCell.y].add(unit)

unit.position = newPos // Still store exact position

}

function GetUnitsInRadius\_Grid(center: Vector2, radius: float, grid: SpatialGrid) -> List<Unit> {

unitsInRadius = []

// Calculate which cells to check

minCell = WorldToGrid(center - Vector2(radius, radius), grid)

maxCell = WorldToGrid(center + Vector2(radius, radius), grid)

// Only check relevant cells

for x in range(max(0, minCell.x), min(grid.gridWidth, maxCell.x + 1)):

for y in range(max(0, minCell.y), min(grid.gridHeight, maxCell.y + 1)):

// Check if this cell could contain units in radius

cellCenter = GridToWorld(x, y, grid)

// Early reject cells that are completely outside radius

if !CellIntersectsCircle(cellCenter, grid.cellSize, center, radius):

continue

// Check units in this cell

for unit in grid.cells[x][y]:

exactDistance = Distance(unit.position, center)

if exactDistance <= radius:

unitsInRadius.add(unit)

return unitsInRadius

}

**Hybrid Approach (Recommended)**

pseudocode

class HybridBattleArena {

// Precise positions for gameplay

units: List<Unit> // Units store Vector2 positions

// Spatial grid for optimization

spatialGrid: SpatialGrid

// Arena properties

centerPoint: Vector2

radius: float

}

function InitializeBattle(arena: HybridBattleArena) {

// Use fine grid for optimization (not gameplay grid)

arena.spatialGrid = InitializeGrid(arena.radius, cellSize: 1.0)

// Place units with exact positions

for unit in arena.units:

AddUnitToGrid(unit, arena.spatialGrid)

}

// Fast broad-phase collision detection

function GetPotentialTargets(center: Vector2, radius: float, arena: HybridBattleArena) -> List<Unit> {

// Step 1: Use grid for broad phase

candidates = GetUnitsInRadius\_Grid(center, radius, arena.spatialGrid)

// Step 2: Precise distance check (narrow phase)

actualTargets = []

for unit in candidates:

if Distance(unit.position, center) <= radius:

actualTargets.add(unit)

return actualTargets

}

// Pathfinding with dynamic unit avoidance

function GetValidMovePosition(targetPos: Vector2, unit: Unit, arena: HybridBattleArena) -> Vector2 {

// Check exact position first

if !IsOccupied\_Precise(targetPos, arena.units, unit):

return targetPos

// Find nearest valid position using spiral search

searchRadius = 0.1

while searchRadius < 2.0:

for angle in range(0, 360, 30):

testPos = targetPos + PolarToCartesian(searchRadius, angle)

if IsWithinArena(testPos, arena) AND

!IsOccupied\_Precise(testPos, arena.units, unit):

return testPos

searchRadius += 0.1

return unit.position // Can't move

}

function IsOccupied\_Precise(position: Vector2, units: List<Unit>, excludeUnit: Unit) -> boolean {

unitRadius = 0.5 // Each unit's collision radius

for unit in units:

if unit == excludeUnit OR !unit.isAlive:

continue

if Distance(position, unit.position) < unitRadius \* 2:

return true

return false

}

**Comparison**

**Pure Vector3/Vector2 Approach:**

* ✅ Precise positioning
* ✅ Smooth movement
* ✅ Simple implementation
* ❌ O(n) for every range check
* ❌ Can be slow with many units

**Pure Grid Approach:**

* ✅ O(1) position lookups
* ✅ Fast area queries
* ❌ Less precise (snapped positions)
* ❌ Memory overhead
* ❌ Complex circle/grid intersection logic

**Hybrid Approach (Recommended):**

* ✅ Precise Vector2 positions for gameplay
* ✅ Spatial grid for optimization only
* ✅ Fast broad-phase collision detection
* ✅ Maintains smooth movement feel
* ✅ Scales well with unit count

**Recommendation**

Use the **hybrid approach**:

1. Store exact Vector2 positions on units
2. Use a spatial hash grid (with cells ~1-2 unit radius) purely for optimization
3. Grid is invisible to players - just speeds up queries

This gives you:

* Smooth, precise movement that feels good
* Fast performance for AoE checks (check grid cells, then precise distance)
* No visible grid snapping
* Scalable to hundreds of units

The grid cell size should be based on your typical query radius - if most AoEs are 5 units, cells of 1-2 units work well. This dramatically reduces the number of distance calculations needed while maintaining perfect accuracy.

Retry

**J**

How would something like a cleave attack work with a cone range, or even more unique shapes like a moon shape.

Edit

Excellent question! Here's how to handle various shaped attacks in a radius-based system:

**Cone/Cleave Attacks**

pseudocode

class ConeAttack {

origin: Vector2

direction: Vector2 // normalized

angle: float // cone angle in degrees (e.g., 60 for 60° cone)

range: float // how far the cone extends

}

function IsPointInCone(point: Vector2, cone: ConeAttack) -> boolean {

// Vector from origin to point

toPoint = point - cone.origin

distance = Length(toPoint)

// Check if within range

if distance > cone.range:

return false

// Check angle

toPointNormalized = Normalize(toPoint)

dotProduct = Dot(toPointNormalized, cone.direction)

angleToPoint = acos(dotProduct) \* (180 / PI) // convert to degrees

// Check if within half-angle on either side

return angleToPoint <= (cone.angle / 2)

}

function GetUnitsInCone(caster: Unit, direction: Vector2, angle: float, range: float, arena: Arena) -> List<Unit> {

targets = []

cone = ConeAttack(caster.position, direction, angle, range)

// Broad phase: get all units within range circle

potentialTargets = GetUnitsInRadius(caster.position, range, arena)

// Narrow phase: check cone angle

for unit in potentialTargets:

if unit.team != caster.team AND IsPointInCone(unit.position, cone):

targets.add(unit)

return targets

}

**Complex Shapes System**

pseudocode

// Base shape interface

abstract class AttackShape {

abstract function Contains(point: Vector2, origin: Vector2, rotation: float) -> boolean

abstract function GetBounds() -> BoundingBox // for optimization

abstract function Draw(origin: Vector2, rotation: float)

}

// Moon/Crescent shape

class MoonShape extends AttackShape {

outerRadius: float

innerRadius: float

offsetX: float // how much the inner circle is offset

function Contains(point: Vector2, origin: Vector2, rotation: float) -> boolean {

// Rotate point to local space

localPoint = RotatePoint(point - origin, -rotation)

// Check if in outer circle

distanceOuter = Distance(localPoint, Vector2.zero)

if distanceOuter > outerRadius:

return false

// Check if NOT in inner circle (the "bite" of the moon)

innerCenter = Vector2(offsetX, 0)

distanceInner = Distance(localPoint, innerCenter)

return distanceInner >= innerRadius

}

}

// Rectangle/Line attack

class RectangleShape extends AttackShape {

width: float

length: float

function Contains(point: Vector2, origin: Vector2, rotation: float) -> boolean {

localPoint = RotatePoint(point - origin, -rotation)

return abs(localPoint.x) <= width/2 AND

localPoint.y >= 0 AND

localPoint.y <= length

}

}

// Star burst pattern

class StarShape extends AttackShape {

numPoints: int

innerRadius: float

outerRadius: float

function Contains(point: Vector2, origin: Vector2, rotation: float) -> boolean {

localPoint = RotatePoint(point - origin, -rotation)

distance = Length(localPoint)

angle = atan2(localPoint.y, localPoint.x)

// Calculate the radius at this angle for a star

segmentAngle = (2 \* PI) / (numPoints \* 2)

segmentIndex = floor(angle / segmentAngle)

if segmentIndex % 2 == 0:

// On a point

maxRadius = outerRadius

else:

// In a valley

maxRadius = innerRadius

// Smooth interpolation for better feel

angleInSegment = angle % segmentAngle

t = angleInSegment / segmentAngle

if segmentIndex % 2 == 0:

currentRadius = Lerp(outerRadius, innerRadius, t)

else:

currentRadius = Lerp(innerRadius, outerRadius, t)

return distance <= currentRadius

}

}

// Wave/Sine wave attack

class WaveShape extends AttackShape {

length: float

amplitude: float

frequency: float

thickness: float

function Contains(point: Vector2, origin: Vector2, rotation: float) -> boolean {

localPoint = RotatePoint(point - origin, -rotation)

if localPoint.y < 0 OR localPoint.y > length:

return false

// Calculate wave position at this y

expectedX = sin(localPoint.y \* frequency) \* amplitude

distanceFromWave = abs(localPoint.x - expectedX)

return distanceFromWave <= thickness / 2

}

}

// Spiral attack

class SpiralShape extends AttackShape {

numRotations: float

maxRadius: float

thickness: float

function Contains(point: Vector2, origin: Vector2, rotation: float) -> boolean {

localPoint = RotatePoint(point - origin, -rotation)

distance = Length(localPoint)

angle = atan2(localPoint.y, localPoint.x)

// Spiral equation: r = a \* θ

spiralGrowthRate = maxRadius / (numRotations \* 2 \* PI)

// Check multiple spiral arms

for spiralOffset in range(0, 2\*PI, 2\*PI/3): // 3 arms

expectedRadius = spiralGrowthRate \* (angle + spiralOffset)

// Check all rotations of the spiral

for rotation in range(0, numRotations):

adjustedRadius = expectedRadius + (rotation \* 2 \* PI \* spiralGrowthRate)

if abs(distance - adjustedRadius) <= thickness:

return true

return false

}

}

**Composite Shapes**

pseudocode

// Combine multiple shapes

class CompositeShape extends AttackShape {

shapes: List<AttackShape>

operation: enum { UNION, INTERSECTION, DIFFERENCE }

function Contains(point: Vector2, origin: Vector2, rotation: float) -> boolean {

switch operation:

case UNION: // Any shape contains point

for shape in shapes:

if shape.Contains(point, origin, rotation):

return true

return false

case INTERSECTION: // All shapes contain point

for shape in shapes:

if !shape.Contains(point, origin, rotation):

return false

return true

case DIFFERENCE: // First shape minus others

if !shapes[0].Contains(point, origin, rotation):

return false

for i in range(1, shapes.length):

if shapes[i].Contains(point, origin, rotation):

return false

return true

}

}

// Example: X-shaped attack

function CreateXAttack() -> CompositeShape {

diagonal1 = new RectangleShape(width: 2, length: 10)

diagonal2 = new RectangleShape(width: 2, length: 10)

composite = new CompositeShape()

composite.operation = UNION

composite.shapes = [

RotatedShape(diagonal1, 45),

RotatedShape(diagonal2, -45)

]

return composite

}

**Implementation with Abilities**

pseudocode

class ShapedAbility extends Ability {

shape: AttackShape

rotationType: enum {

FIXED, // Always same direction

TOWARD\_TARGET, // Aims at target position

FREE\_AIM, // Player can rotate freely

CHARACTER\_FACING // Uses character's facing direction

}

function GetAffectedUnits(caster: Unit, targetPos: Vector2, arena: Arena) -> List<Unit> {

rotation = 0

switch rotationType:

case TOWARD\_TARGET:

direction = targetPos - caster.position

rotation = atan2(direction.y, direction.x)

case CHARACTER\_FACING:

rotation = caster.facingAngle

case FREE\_AIM:

rotation = GetPlayerAimAngle()

affected = []

// Optimization: broad phase with bounding box

bounds = shape.GetBounds()

rotatedBounds = RotateBounds(bounds, rotation)

potentialTargets = GetUnitsInBounds(rotatedBounds, arena)

// Narrow phase: exact shape check

for unit in potentialTargets:

if shape.Contains(unit.position, caster.position, rotation):

affected.add(unit)

return affected

}

}

// Visualization

function DrawAttackPreview(ability: ShapedAbility, caster: Unit, mousePos: Vector2) {

rotation = CalculateRotation(ability.rotationType, caster, mousePos)

// Draw shape outline

ability.shape.Draw(caster.position, rotation)

// Highlight affected units

affected = ability.GetAffectedUnits(caster, mousePos, arena)

for unit in affected:

DrawHighlight(unit, ENEMY\_COLOR if unit.team != caster.team else FRIENDLY\_COLOR)

}

**Cool Shape Examples**

pseudocode

// Boomerang arc

class BoomerangShape extends AttackShape {

function Contains(point: Vector2, origin: Vector2, rotation: float) -> boolean {

local = RotatePoint(point - origin, -rotation)

// Parametric curve for boomerang path

// Could use Bezier curve or simple parabola

t = local.x / maxRange // 0 to 1

expectedY = sin(t \* PI) \* curveWidth

distance = Distance(local, Vector2(local.x, expectedY))

return distance <= thickness AND t >= 0 AND t <= 1

}

}

// DNA Helix attack

class HelixShape extends AttackShape {

function Contains(point: Vector2, origin: Vector2, rotation: float) -> boolean {

local = RotatePoint(point - origin, -rotation)

// Two intertwined helixes

helix1Y = sin(local.x \* frequency) \* amplitude

helix2Y = sin(local.x \* frequency + PI) \* amplitude

dist1 = Distance(local, Vector2(local.x, helix1Y))

dist2 = Distance(local, Vector2(local.x, helix2Y))

return (dist1 <= thickness OR dist2 <= thickness) AND

local.x >= 0 AND local.x <= range

}

}

The key advantages of this system:

1. **Flexibility** - Any shape imaginable can be implemented
2. **Precision** - Pixel-perfect hit detection
3. **Visual clarity** - Easy to show exact hit areas
4. **Unique gameplay** - Different shapes create different tactical situations
5. **Composable** - Combine simple shapes for complex patterns

For performance, always use a two-phase approach:

1. Broad phase with bounding boxes or circles
2. Narrow phase with exact shape calculations

This keeps the system fast even with complex shapes and many units!

Retry

**J**

And how would you implement a timeline based turn order system? I was envisioning a system where each attack has a time to cast and a recovery time. Is that viable?

Edit

Absolutely! A timeline-based system with cast times and recovery is not only viable but creates really engaging tactical gameplay. Here's a comprehensive implementation:

**Core Timeline System**

pseudocode

class Timeline {

currentTime: float = 0

events: PriorityQueue<TimelineEvent> // sorted by executionTime

unitStates: Map<Unit, UnitTimelineState>

timeScale: float = 1.0 // for slow-mo effects

}

class TimelineEvent {

executionTime: float

eventType: enum { TURN\_READY, CAST\_COMPLETE, RECOVERY\_END, STATUS\_TICK }

unit: Unit

ability: Ability // nullable, only for cast events

targetData: TargetData // stores target position/units

}

class UnitTimelineState {

currentAction: enum { IDLE, CASTING, RECOVERING, STUNNED }

actionStartTime: float

actionEndTime: float

nextTurnTime: float

queuedAbility: Ability

queuedTarget: TargetData

}

class Ability {

name: string

castTime: float // time to cast

recoveryTime: float // time after cast before next action

range: float

damage: float

// Speed modifiers

castTimeModifier: enum { FIXED, SPEED\_MODIFIED, HASTE\_AFFECTED }

recoveryModifier: enum { FIXED, SPEED\_MODIFIED, HASTE\_AFFECTED }

// Can this be interrupted?

interruptible: boolean

// Movement while casting?

allowMovementDuringCast: boolean

castSpeedReduction: float // 0.5 = half speed while casting

}

**Timeline Management**

pseudocode

function InitializeTimeline(units: List<Unit>) -> Timeline {

timeline = new Timeline()

for unit in units:

// Initial delay based on speed/initiative

initialDelay = 100.0 / (50 + unit.stats.speed) // faster units act sooner

event = TimelineEvent(

executionTime: initialDelay,

eventType: TURN\_READY,

unit: unit

)

timeline.events.enqueue(event)

timeline.unitStates[unit] = new UnitTimelineState(

currentAction: IDLE,

nextTurnTime: initialDelay

)

return timeline

}

function ProcessTimeline(timeline: Timeline, deltaTime: float) {

timeline.currentTime += deltaTime \* timeline.timeScale

while timeline.events.peek().executionTime <= timeline.currentTime:

event = timeline.events.dequeue()

ProcessEvent(event, timeline)

}

function ProcessEvent(event: TimelineEvent, timeline: Timeline) {

state = timeline.unitStates[event.unit]

switch event.eventType:

case TURN\_READY:

if state.currentAction == IDLE:

// Unit can now act

event.unit.canAct = true

if event.unit.isAI:

action = GetAIAction(event.unit)

ExecuteAction(event.unit, action, timeline)

else:

// Wait for player input

HighlightUnit(event.unit)

case CAST\_COMPLETE:

// Execute the ability effect

ExecuteAbilityEffect(event.unit, event.ability, event.targetData)

// Start recovery

state.currentAction = RECOVERING

state.actionStartTime = timeline.currentTime

recoveryTime = CalculateRecoveryTime(event.unit, event.ability)

state.actionEndTime = timeline.currentTime + recoveryTime

// Queue recovery end event

timeline.events.enqueue(TimelineEvent(

executionTime: state.actionEndTime,

eventType: RECOVERY\_END,

unit: event.unit

))

case RECOVERY\_END:

// Unit returns to idle

state.currentAction = IDLE

// Calculate next turn time

turnDelay = CalculateTurnDelay(event.unit)

state.nextTurnTime = timeline.currentTime + turnDelay

// Queue next turn

timeline.events.enqueue(TimelineEvent(

executionTime: state.nextTurnTime,

eventType: TURN\_READY,

unit: event.unit

))

**Action Execution**

pseudocode

function ExecuteAction(unit: Unit, action: Action, timeline: Timeline) {

state = timeline.unitStates[unit]

switch action.type:

case ABILITY:

ability = action.ability

// Validate targets in range

if !IsTargetValid(unit, ability, action.target):

return false

// Start casting

state.currentAction = CASTING

state.actionStartTime = timeline.currentTime

state.queuedAbility = ability

state.queuedTarget = action.target

// Calculate actual cast time

castTime = CalculateCastTime(unit, ability)

state.actionEndTime = timeline.currentTime + castTime

// Queue cast completion

timeline.events.enqueue(TimelineEvent(

executionTime: state.actionEndTime,

eventType: CAST\_COMPLETE,

unit: unit,

ability: ability,

targetData: action.target

))

unit.canAct = false

case MOVE:

// Movement might have its own recovery

ExecuteMovement(unit, action.position)

moveRecovery = CalculateMoveRecovery(unit, action.distance)

state.nextTurnTime = timeline.currentTime + moveRecovery

timeline.events.enqueue(TimelineEvent(

executionTime: state.nextTurnTime,

eventType: TURN\_READY,

unit: unit

))

case WAIT:

// Skip turn with reduced delay

waitDelay = CalculateTurnDelay(unit) \* 0.5 // waiting is faster

state.nextTurnTime = timeline.currentTime + waitDelay

timeline.events.enqueue(TimelineEvent(

executionTime: state.nextTurnTime,

eventType: TURN\_READY,

unit: unit

))

}

function CalculateCastTime(unit: Unit, ability: Ability) -> float {

baseCastTime = ability.castTime

switch ability.castTimeModifier:

case FIXED:

return baseCastTime

case SPEED\_MODIFIED:

// Faster units cast quicker

speedMultiplier = 1.0 - (unit.stats.speed / 200.0) // up to 50% reduction

return baseCastTime \* max(0.5, speedMultiplier)

case HASTE\_AFFECTED:

// Affected by haste buffs

hasteMultiplier = unit.GetBuffMultiplier("haste")

return baseCastTime / hasteMultiplier

}

**Interruption System**

pseudocode

function InterruptUnit(target: Unit, timeline: Timeline) {

state = timeline.unitStates[target]

if state.currentAction != CASTING:

return false // Can only interrupt casting

if !state.queuedAbility.interruptible:

return false // This ability can't be interrupted

// Cancel the cast

state.currentAction = IDLE

// Remove the cast complete event

timeline.events.removeWhere(e =>

e.unit == target AND

e.eventType == CAST\_COMPLETE AND

e.executionTime == state.actionEndTime

)

// Apply interrupt penalty

penaltyTime = 2.0 // 2 second penalty for being interrupted

state.nextTurnTime = timeline.currentTime + penaltyTime

timeline.events.enqueue(TimelineEvent(

executionTime: state.nextTurnTime,

eventType: TURN\_READY,

unit: target

))

// Visual feedback

ShowInterruptEffect(target)

return true

}

**Advanced Timeline Features**

pseudocode

// Combo system - reduced cast time for chained abilities

class ComboAbility extends Ability {

comboFrom: List<Ability> // abilities that can combo into this

comboCastReduction: float // e.g., 0.5 for 50% faster cast

function GetCastTime(unit: Unit, timeline: Timeline) -> float {

lastAbility = GetLastUsedAbility(unit, timeline, timeWindow: 2.0)

if lastAbility in comboFrom:

return castTime \* comboCastReduction

return castTime

}

// Channeled abilities

class ChanneledAbility extends Ability {

channelDuration: float

tickInterval: float // how often effect applies

function StartChannel(unit: Unit, target: TargetData, timeline: Timeline) {

numTicks = floor(channelDuration / tickInterval)

for i in range(0, numTicks):

tickTime = timeline.currentTime + castTime + (i \* tickInterval)

timeline.events.enqueue(TimelineEvent(

executionTime: tickTime,

eventType: CHANNEL\_TICK,

unit: unit,

ability: this,

targetData: target

))

}

}

// Delayed abilities

class DelayedAbility extends Ability {

delayAfterCast: float // time between cast finish and effect

telegraphDuration: float // how long to show warning

function OnCastComplete(unit: Unit, target: TargetData, timeline: Timeline) {

// Show telegraph immediately

ShowTelegraph(target.position, telegraphDuration)

// Queue delayed effect

timeline.events.enqueue(TimelineEvent(

executionTime: timeline.currentTime + delayAfterCast,

eventType: DELAYED\_EFFECT,

unit: unit,

ability: this,

targetData: target

))

}

}

**Visualization**

pseudocode

class TimelineUI {

barLength: float = 800 // pixels

timeWindow: float = 10.0 // show next 10 seconds

function Draw(timeline: Timeline) {

// Draw timeline bar

DrawRectangle(x: 100, y: 50, width: barLength, height: 20)

// Draw current time marker

DrawLine(x: 100, y: 45, height: 30, color: WHITE)

// Draw future events

for event in timeline.events:

if event.executionTime > timeline.currentTime + timeWindow:

break

// Calculate position on bar

timeDelta = event.executionTime - timeline.currentTime

xPos = 100 + (timeDelta / timeWindow) \* barLength

// Draw event marker

DrawUnitIcon(event.unit, x: xPos, y: 50)

// Draw cast/recovery bars

state = timeline.unitStates[event.unit]

if state.currentAction == CASTING:

DrawCastBar(unit: event.unit,

progress: GetActionProgress(state, timeline),

x: xPos, y: 70)

}

function DrawUnitCastBar(unit: Unit, state: UnitTimelineState, timeline: Timeline) {

if state.currentAction == IDLE:

return

progress = (timeline.currentTime - state.actionStartTime) /

(state.actionEndTime - state.actionStartTime)

barColor = state.currentAction == CASTING ? BLUE : ORANGE

// Draw above unit

worldPos = unit.position + Vector2(0, 2)

screenPos = WorldToScreen(worldPos)

DrawProgressBar(screenPos, progress, barColor)

// Show time remaining

remaining = state.actionEndTime - timeline.currentTime

DrawText(remaining.toFixed(1) + "s", screenPos + Vector2(0, -10))

}

}

// Preview system for planning

function PreviewAction(unit: Unit, ability: Ability, timeline: Timeline) -> ActionPreview {

preview = new ActionPreview()

currentTime = timeline.currentTime

state = timeline.unitStates[unit]

// Calculate when this action would complete

if state.currentAction != IDLE:

startTime = state.actionEndTime // start after current action

else:

startTime = currentTime

castTime = CalculateCastTime(unit, ability)

recoveryTime = CalculateRecoveryTime(unit, ability)

preview.castStart = startTime

preview.castEnd = startTime + castTime

preview.recoveryEnd = startTime + castTime + recoveryTime

preview.nextTurnTime = preview.recoveryEnd + CalculateTurnDelay(unit)

// Show on timeline UI

DrawPreviewMarkers(preview)

return preview

}

**Strategic Considerations**

pseudocode

// AI decision making with timeline

function GetAIAction(unit: Unit, timeline: Timeline) -> Action {

bestAction = null

bestScore = -infinity

for ability in unit.abilities:

// Consider cast time vs enemy actions

castTime = CalculateCastTime(unit, ability)

// Will enemy act before our cast completes?

enemyEvents = GetEnemyEvents(timeline,

timeline.currentTime,

timeline.currentTime + castTime)

if enemyEvents.any(e => e.ability.canInterrupt):

// Risk of interruption

score -= 50

// Check if we can combo

if CanCombo(unit, ability, timeline):

score += 30

// Favor quick abilities when multiple enemies are about to act

if enemyEvents.count > 2:

score += 100 / castTime // prefer faster abilities

// Consider recovery impact

recoveryTime = CalculateRecoveryTime(unit, ability)

nextTurnTime = timeline.currentTime + castTime + recoveryTime

// How many enemy turns before we act again?

enemyTurnsBeforeNext = CountEnemyTurns(timeline,

timeline.currentTime,

nextTurnTime)

score -= enemyTurnsBeforeNext \* 10

if score > bestScore:

bestScore = score

bestAction = CreateAction(ability, FindBestTarget(unit, ability))

return bestAction

}

This system creates really interesting decisions:

* **Quick vs Powerful**: Fast abilities let you act more often, but deal less damage
* **Interruption risk**: Long cast times are vulnerable to interrupts
* **Combo planning**: Chain abilities for reduced cast times
* **Timeline manipulation**: Abilities that delay enemies or hasten allies become very valuable
* **Turn economy**: Sometimes waiting or using a quick ability is better than a slow powerful one

The visual timeline helps players plan several moves ahead, creating a chess-like strategic depth!

Retry

**J**

And would this be viable for a turn based system purely based on discrete units of time vs a real time approach

Edit

Excellent question! Yes, this works beautifully for a discrete turn-based system - in fact, it might work even better than real-time. Here's how to adapt it:

**Discrete Time Units (CTB - Charge Time Battle)**

pseudocode

class DiscreteTimeline {

currentTick: int = 0 // discrete time units

timeUnitsPerTurn: int = 100 // base "ATB" gauge

events: PriorityQueue<DiscreteTimeEvent>

unitStates: Map<Unit, UnitState>

// For smooth visualization

displayTime: float // interpolated for animations

tickDuration: float = 0.1 // seconds per tick for display

}

class DiscreteTimeEvent {

executionTick: int // when this triggers

unit: Unit

eventType: EventType

priority: int // for same-tick resolution

}

// Everything uses integer time units

class DiscreteAbility {

name: string

castTime: int // in ticks (e.g., 30 ticks)

recoveryTime: int // in ticks (e.g., 20 ticks)

// Speed affects these via multipliers

function GetCastTicks(unit: Unit) -> int {

baseTicks = castTime

speedMod = 100 - min(50, unit.stats.speed / 2) // 50-100% of base

return ceil(baseTicks \* speedMod / 100)

}

}

**Turn Processing**

pseudocode

function ProcessTurn(timeline: DiscreteTimeline) {

// Process all events at current tick

while timeline.events.peek().executionTick == timeline.currentTick:

event = timeline.events.dequeue()

if event.eventType == TURN\_READY:

unit = event.unit

// PAUSE HERE for player input

if !unit.isAI:

PauseForPlayerDecision(unit, timeline)

return // Wait for input

else:

action = GetAIAction(unit, timeline)

ExecuteDiscreteAction(unit, action, timeline)

// Advance time

timeline.currentTick++

// Check if we need player input at next tick

if NextTickHasPlayerTurn(timeline):

return // Pause before advancing

else:

ProcessTurn(timeline) // Continue recursively

}

function PauseForPlayerDecision(unit: Unit, timeline: DiscreteTimeline) {

// Show UI for player to decide

ShowActionMenu(unit)

ShowTimelinePreview(timeline)

// Game waits here

// When player chooses action, call:

// ExecuteDiscreteAction(unit, playerAction, timeline)

// Then resume: ProcessTurn(timeline)

}

**Speed/Initiative System**

pseudocode

class InitiativeSystem {

baseATB: int = 100 // ticks to get a turn

function CalculateNextTurn(unit: Unit) -> int {

// Speed determines how fast you gain ATB

speed = unit.stats.speed

// Higher speed = fewer ticks to next turn

ticksToTurn = baseATB - speed // simple version

// OR more complex:

ticksToTurn = ceil(baseATB \* (100 / (100 + speed)))

return ticksToTurn

}

function CalculateInitialDelay(unit: Unit) -> int {

// Random starting position based on speed

maxDelay = 50

speedBonus = min(unit.stats.speed / 2, 40)

return Random(0, maxDelay - speedBonus)

}

}

// Alternative: Classic ATB gauge fill

class ATBSystem {

maxGauge: int = 10000

function ProcessATB(timeline: DiscreteTimeline) {

for unit in timeline.units:

state = timeline.unitStates[unit]

if state.currentAction == IDLE:

// Fill gauge based on speed

fillRate = 100 + unit.stats.speed \* 2

state.atbGauge += fillRate

if state.atbGauge >= maxGauge:

state.atbGauge = maxGauge

// Unit gets a turn

timeline.events.enqueue(DiscreteTimeEvent(

executionTick: timeline.currentTick,

unit: unit,

eventType: TURN\_READY,

priority: unit.stats.speed // faster units go first on ties

))

timeline.currentTick++

}

}

**Discrete Action Resolution**

pseudocode

function ExecuteDiscreteAction(unit: Unit, action: Action, timeline: DiscreteTimeline) {

state = timeline.unitStates[unit]

switch action.type:

case INSTANT\_ABILITY:

// Executes immediately, has recovery

ExecuteEffect(action.ability, action.target)

recoveryTicks = action.ability.recoveryTime

state.nextTurnTick = timeline.currentTick + recoveryTicks

ScheduleTurn(unit, state.nextTurnTick, timeline)

case CAST\_ABILITY:

// Start casting

castTicks = action.ability.GetCastTicks(unit)

state.currentAction = CASTING

state.castCompleteTick = timeline.currentTick + castTicks

state.pendingAbility = action.ability

state.pendingTarget = action.target

// Schedule cast completion

timeline.events.enqueue(DiscreteTimeEvent(

executionTick: state.castCompleteTick,

unit: unit,

eventType: CAST\_COMPLETE

))

case WAIT:

// Half normal wait time

waitTicks = CalculateNextTurn(unit) / 2

ScheduleTurn(unit, timeline.currentTick + waitTicks, timeline)

}

// Clean tick-based interruption

function InterruptAtTick(target: Unit, timeline: DiscreteTimeline) -> boolean {

state = timeline.unitStates[target]

if state.currentAction != CASTING:

return false

// Cancel future cast event

RemoveEvent(timeline, target, CAST\_COMPLETE, state.castCompleteTick)

// Add stun penalty

stunTicks = 30

state.currentAction = STUNNED

state.recoverTick = timeline.currentTick + stunTicks

ScheduleTurn(unit, state.recoverTick, timeline)

return true

}

**Turn Order Display**

pseudocode

class TurnOrderDisplay {

maxFutureTurns: int = 10

function CalculateTurnOrder(timeline: DiscreteTimeline) -> List<TurnPreview> {

preview = []

simulatedTimeline = timeline.copy()

// Simulate forward to see turn order

for i in range(0, maxFutureTurns):

nextTurn = GetNextTurn(simulatedTimeline)

if !nextTurn:

break

preview.add(TurnPreview(

unit: nextTurn.unit,

tick: nextTurn.tick,

relativeTime: nextTurn.tick - timeline.currentTick

))

// Simulate unit taking average action

avgActionTime = EstimateAverageActionTime(nextTurn.unit)

ScheduleTurn(nextTurn.unit,

nextTurn.tick + avgActionTime,

simulatedTimeline)

return preview

}

function DrawTurnOrder(preview: List<TurnPreview>) {

// Show as portrait list

for i, turn in enumerate(preview):

DrawPortrait(turn.unit, x: 10, y: 100 + i \* 60)

DrawText(turn.relativeTime + " ticks", x: 80, y: 100 + i \* 60)

// Show what they're doing

state = timeline.unitStates[turn.unit]

if state.currentAction == CASTING:

DrawCastProgress(state, x: 150, y: 100 + i \* 60)

}

}

**Advanced Discrete Mechanics**

pseudocode

// Delay/Haste effects work cleanly

class TimeManipulation {

function DelayUnit(target: Unit, delayTicks: int, timeline: DiscreteTimeline) {

// Find all future events for this unit

for event in timeline.events:

if event.unit == target AND event.executionTick > timeline.currentTick:

event.executionTick += delayTicks

}

function HasteUnit(target: Unit, hasteTicks: int, timeline: DiscreteTimeline) {

for event in timeline.events:

if event.unit == target AND event.executionTick > timeline.currentTick:

// Don't move events into the past

event.executionTick = max(timeline.currentTick + 1,

event.executionTick - hasteTicks)

}

}

// Clock mechanics

class ClockMechanic {

clockSegments: int = 12

currentSegment: int = 0

function OnTickAdvance(timeline: DiscreteTimeline) {

// Every X ticks, advance clock

if timeline.currentTick % 100 == 0:

currentSegment = (currentSegment + 1) % clockSegments

// Trigger clock-based effects

for unit in timeline.units:

for buff in unit.buffs:

if buff.triggerSegment == currentSegment:

buff.Activate(unit)

}

}

// Combo system with discrete timing

class ComboSystem {

comboWindow: int = 50 // ticks to chain combo

function CheckCombo(unit: Unit, ability: Ability, timeline: DiscreteTimeline) -> boolean {

lastAction = unit.lastActionTick

if timeline.currentTick - lastAction <= comboWindow:

if ability.comboFrom.contains(unit.lastAbility):

return true

return false

}

}

**Advantages of Discrete Over Real-Time**

pseudocode

// 1. Perfect Synchronization

// No floating point errors, everything aligns perfectly

function PerfectSimultaneous() {

// These will ALWAYS execute together

event1.executionTick = 100

event2.executionTick = 100

// No worries about 99.999999 vs 100.000001

}

// 2. Save/Load Simplicity

class SaveState {

currentTick: int

events: List<SavedEvent>

// Perfectly reproducible, no float precision issues

}

// 3. Network Multiplayer

function SendAction(action: Action) {

packet = {

tick: timeline.currentTick,

action: action

}

// Everyone calculates same result

}

// 4. Clear Player Communication

function ShowAbilityTooltip(ability: Ability) {

return "Cast Time: " + ability.castTime + " ticks\n" +

"Recovery: " + ability.recoveryTime + " ticks\n" +

"Your speed reduces cast time by " + GetSpeedReduction(unit) + " ticks"

// Players can count and plan exactly

}

// 5. Predictable Gameplay

function PlayerCanCalculate() {

// "If I use this 30-tick ability, enemy acts at tick 145,

// I'll finish at tick 130 + 30 = 160, so they'll interrupt me.

// Better use the 10-tick ability instead."

}

**Discrete vs Real-Time Comparison**

**Discrete Advantages:**

* ✅ **Predictability** - Players can count exact turns ahead
* ✅ **No pause issues** - Game naturally pauses for decisions
* ✅ **Network stable** - Perfect synchronization
* ✅ **Save anywhere** - State is just integers
* ✅ **Clear feedback** - "20 ticks until your turn"
* ✅ **No animation conflicts** - Animations are just visual layer

**Real-Time Advantages:**

* ✅ Smoother visual feel
* ✅ Tension from time pressure
* ✅ No "analysis paralysis"

**Hybrid Solution:**

pseudocode

class HybridTimeline {

// Discrete for logic

logicTick: int

// Smooth for display

visualTime: float

animationSpeed: float = 1.0

function Update(deltaTime: float) {

if !waitingForPlayer:

visualTime += deltaTime \* animationSpeed

// Advance logic when visual catches up

while visualTime >= (logicTick + 1) \* TICK\_DURATION:

ProcessTick()

logicTick++

if NeedsPlayerInput():

waitingForPlayer = true

break

}

}

The discrete approach is absolutely viable and arguably better for turn-based games because it gives players perfect information and control. Games like Divinity: Original Sin, Grandia, and many SRPGs use discrete tick systems very successfully!

Retry

**J**

So, using this method, would it be possible to essentially record the battle and replay it back as an almost animation

Edit

Absolutely! This is one of the huge advantages of a discrete tick-based system - perfect replay capability. You can record battles with minimal data and replay them exactly, even with different visual settings or camera angles. Here's how:

**Battle Recording System**

pseudocode

class BattleRecording {

// Minimal data needed

initialState: BattleInitialState

actions: List<RecordedAction>

metadata: RecordingMetadata

}

class BattleInitialState {

units: List<UnitSnapshot>

arenaConfig: ArenaConfig

randomSeed: int // for any RNG

version: string // game version for compatibility

}

class UnitSnapshot {

id: int

position: Vector2

stats: StatBlock

abilities: List<AbilityId>

team: int

}

class RecordedAction {

tick: int // when it happened

unitId: int // who acted

actionType: ActionType

targetData: TargetData // position or unit IDs

// Optional: for validation/debugging

resultHash: int // hash of game state after action

}

class RecordingMetadata {

timestamp: DateTime

duration: int // total ticks

players: List<PlayerInfo>

winner: TeamId

highlights: List<HighlightMoment> // for cool moments

}

**Recording Implementation**

pseudocode

class BattleRecorder {

recording: BattleRecording

isRecording: boolean

function StartRecording(battle: Battle) {

recording = new BattleRecording()

// Snapshot initial state

recording.initialState = CaptureInitialState(battle)

recording.actions = []

recording.metadata.timestamp = Now()

isRecording = true

}

function RecordAction(action: Action, tick: int, unitId: int) {

if !isRecording:

return

recordedAction = RecordedAction(

tick: tick,

unitId: unitId,

actionType: action.type,

targetData: SerializeTarget(action.target)

)

recording.actions.add(recordedAction)

// Auto-detect highlights

if IsHighlightWorthy(action):

AddHighlight(tick, action)

}

function FinishRecording(winner: TeamId) {

recording.metadata.winner = winner

recording.metadata.duration = GetCurrentTick()

// Compress if needed

CompressRecording(recording)

SaveToFile(recording)

}

}

// Tiny file sizes!

function CalculateFileSize(recording: BattleRecording) -> int {

// Each action is roughly:

// tick: 4 bytes, unitId: 2 bytes, actionType: 1 byte,

// target: 8 bytes = ~15 bytes per action

// 100 actions in a battle = 1.5KB

// Initial state: ~2KB

// Total: ~3-4KB per battle!

}

**Replay Player**

pseudocode

class BattleReplayPlayer {

recording: BattleRecording

replayTimeline: DiscreteTimeline

currentActionIndex: int = 0

// Playback controls

playbackSpeed: float = 1.0

isPaused: boolean = false

visualTime: float = 0

function LoadReplay(recording: BattleRecording) {

// Recreate initial battle state

battle = ReconstructBattle(recording.initialState)

replayTimeline = InitializeTimeline(battle.units)

// Set RNG seed for identical results

SetRandomSeed(recording.initialState.randomSeed)

}

function Update(deltaTime: float) {

if isPaused:

return

visualTime += deltaTime \* playbackSpeed

// Process actions up to current visual time

targetTick = floor(visualTime / TICK\_DURATION)

while replayTimeline.currentTick < targetTick:

ProcessNextTick()

}

function ProcessNextTick() {

// Apply any actions at this tick

while currentActionIndex < recording.actions.length:

action = recording.actions[currentActionIndex]

if action.tick > replayTimeline.currentTick:

break // No more actions this tick

// Reconstruct and execute the action

ReplayAction(action)

currentActionIndex++

// Advance timeline

replayTimeline.currentTick++

UpdateVisuals()

}

function ReplayAction(recordedAction: RecordedAction) {

unit = GetUnitById(recordedAction.unitId)

// Reconstruct the full action

action = ReconstructAction(recordedAction)

// Execute it exactly as it was

ExecuteDiscreteAction(unit, action, replayTimeline)

// Optional: Validate state matches

if VALIDATE\_REPLAY:

currentHash = HashGameState(replayTimeline)

assert(currentHash == recordedAction.resultHash)

}

}

**Advanced Replay Features**

pseudocode

class EnhancedReplayPlayer {

// Camera controls

cameraMode: enum {

CINEMATIC, // Auto camera for best view

FOLLOW\_UNIT, // Track specific unit

FREE\_CAM, // User controlled

TACTICAL\_VIEW // Top-down overview

}

// Visual enhancements

showDamageNumbers: boolean = true

showEffectRadii: boolean = false

highlightMode: enum { NONE, CURRENT\_ACTOR, ALL\_THREATS }

function RenderReplay(tick: int) {

// Smooth interpolation between ticks

nextTick = tick + 1

t = (visualTime % TICK\_DURATION) / TICK\_DURATION

for unit in units:

// Interpolate positions

currentPos = GetPositionAtTick(unit, tick)

nextPos = GetPositionAtTick(unit, nextTick)

unit.visualPosition = Lerp(currentPos, nextPos, t)

// Interpolate health bars

currentHP = GetHealthAtTick(unit, tick)

nextHP = GetHealthAtTick(unit, nextTick)

unit.visualHealth = Lerp(currentHP, nextHP, t)

// Enhanced visuals not in original battle

if showEffectRadii:

DrawAllRangeIndicators()

if highlightMode == CURRENT\_ACTOR:

HighlightActingUnit()

}

// Jump to interesting moments

function JumpToHighlight(index: int) {

highlight = recording.metadata.highlights[index]

// Rewind or fast-forward to moment

SetReplayTick(highlight.tick - 20) // Start 2 seconds before

// Set cinematic camera

SetCameraFocus(highlight.focusPoint)

// Slow down for dramatic effect

playbackSpeed = 0.5

}

// Generate highlight reel

function CreateHighlightReel() -> BattleRecording {

highlights = []

for moment in recording.metadata.highlights:

// Extract 5 seconds around each highlight

startTick = max(0, moment.tick - 50)

endTick = min(recording.metadata.duration, moment.tick + 50)

highlights.add(ExtractSegment(recording, startTick, endTick))

return CombineHighlights(highlights)

}

}

// Highlight detection

function IsHighlightWorthy(action: Action, result: ActionResult) -> boolean {

// Multi-kills

if result.unitsKilled >= 3:

return true

// Clutch interrupts

if action.type == INTERRUPT AND result.targetWasCasting:

return true

// Last-second saves

if action.type == HEAL and result.targetHealth < 10:

return true

// Perfect dodges

if action.type == MOVE and result.dodgedAttacks > 0:

return true

// Comeback moments

if GetTeamUnitCount(action.unit.team) == 1 and result.unitsKilled > 0:

return true

return false

}

**Replay Analysis Tools**

pseudocode

class BattleAnalyzer {

function AnalyzeReplay(recording: BattleRecording) -> BattleStats {

stats = new BattleStats()

// Replay the battle collecting statistics

replayTimeline = LoadReplay(recording)

for action in recording.actions:

unit = GetUnitById(action.unitId)

// Track DPS

if action.type == ATTACK:

stats.damageDealt[unit] += CalculateDamage(action)

// Track efficiency

stats.actionsPerUnit[unit]++

stats.averageActionTime[unit] += action.tick

// Track ability usage

stats.abilityUsage[action.abilityId]++

// Generate insights

stats.mvp = GetHighestDamageUnit(stats)

stats.turnEfficiency = CalculateTurnEfficiency(stats)

stats.timeline = GenerateEventTimeline(recording)

return stats

}

// Heatmap generation

function GenerateMovementHeatmap(recording: BattleRecording) -> Heatmap {

heatmap = new Heatmap(arenaSize)

for action in recording.actions:

if action.type == MOVE:

position = action.targetData.position

heatmap.AddHeat(position, intensity: 1.0)

return heatmap

}

// What-if analysis

function SimulateAlternative(recording: BattleRecording,

changeIndex: int,

newAction: Action) -> BattleResult {

// Replay up to change point

alternateTimeline = LoadReplay(recording)

for i in range(0, changeIndex):

ReplayAction(recording.actions[i])

// Apply different action

ExecuteAction(newAction, alternateTimeline)

// Simulate rest with AI

return SimulateBattleCompletion(alternateTimeline)

}

}

**Sharing and Streaming**

pseudocode

class ReplaySharing {

// Compact URL encoding

function EncodeReplayURL(recording: BattleRecording) -> string {

// Compress to base64

compressed = CompressRecording(recording)

base64 = Base64Encode(compressed)

// Create shareable link

return "game.com/replay?data=" + base64

// Usually just a few KB - fits in a URL!

}

// Live replay streaming

function StreamReplay(recording: BattleRecording, viewers: List<Player>) {

for tick in range(0, recording.metadata.duration):

// Send next action to viewers

actions = GetActionsAtTick(recording, tick)

for viewer in viewers:

SendReplayPacket(viewer, tick, actions)

// Wait for real-time or specified speed

Wait(TICK\_DURATION / streamSpeed)

}

// Tournament replay packs

function CreateTournamentBundle(replays: List<BattleRecording>) -> ReplayBundle {

bundle = new ReplayBundle()

// Add match progression

bundle.brackets = GenerateBracketStructure(replays)

// Add commentary markers

for replay in replays:

replay.commentary = GetCommentaryEvents(replay)

// Create mega-highlights

bundle.tournamentHighlights = ExtractBestMoments(replays, count: 20)

return bundle

}

}

**Visual Enhancement Options**

pseudocode

class ReplayVisualizer {

// Different visual styles for same replay

visualStyle: enum {

ORIGINAL, // As it was played

CINEMATIC, // Movie-like cameras and effects

ANALYTICAL, // Shows all hidden info

MINIMALIST, // Clean, easy to follow

STREAMER // With space for overlay

}

function RenderWithStyle(replay: BattleRecording, style: VisualStyle) {

switch style:

case CINEMATIC:

EnableDepthOfField()

EnableMotionBlur()

SetDynamicCameraAngles()

AddParticleEffects()

case ANALYTICAL:

ShowAllRanges()

ShowDamageCalculations()

ShowAIDecisionTrees()

DisplayTimeline()

case MINIMALIST:

DisableAllEffects()

UseSimpleShapes()

HighContrastColors()

}

// Export as video

function ExportToVideo(replay: BattleRecording, settings: VideoSettings) {

videoWriter = new VideoWriter(settings)

// Render each frame

framesPerTick = settings.fps \* TICK\_DURATION

for tick in range(0, replay.metadata.duration):

for frame in range(0, framesPerTick):

t = frame / framesPerTick

RenderFrame(tick, t)

videoWriter.WriteFrame()

videoWriter.Finalize()

}

}

The beauty of this system is:

1. **Tiny file sizes** - Full battles in just a few KB
2. **Perfect reproduction** - Exact same outcome every time
3. **Flexible visualization** - Change camera, effects, speed without changing data
4. **Easy sharing** - Small enough to embed in URLs or chat
5. **Analysis tools** - Can generate statistics and what-if scenarios
6. **Spectator friendly** - Can stream replays or export as videos

This is exactly how games like StarCraft, League of Legends, and chess platforms handle replays - recording actions, not video!

Retry

**J**

Going outside of the battle system, what would a grid system for a first person dungeon crawler look like such that we can have walls that separate 2 cells without needing to have an entire cell take up that space. Ie having 2 cells directly next to eachother but not reachable due to a wall directly inbetween.

Edit

Excellent question! This is the classic "thin wall" problem in grid-based dungeon crawlers. Here's a comprehensive system that handles walls as edges between cells rather than taking up cells themselves:

**Edge-Based Wall System**

pseudocode

class DungeonGrid {

width: int

height: int

cells: Array2D<Cell>

// Walls exist between cells, not in them

horizontalWalls: Array2D<Wall> // [x][y] = wall between cell(x,y) and cell(x,y-1)

verticalWalls: Array2D<Wall> // [x][y] = wall between cell(x,y) and cell(x-1,y)

}

class Cell {

x: int

y: int

floor: FloorType

ceiling: CeilingType

elevation: float

contents: List<Entity> // monsters, items, etc

properties: CellProperties // lighting, effects, triggers

}

class Wall {

exists: boolean

type: WallType // solid, door, bars, illusory, one-way

material: Material

state: WallState // for doors: open/closed/locked

// Visual data

textureNorth: TextureId // texture when viewed from north

textureSouth: TextureId // texture when viewed from south

// Decorations on the wall

decorations: List<WallFeature> // torches, switches, signs

}

enum WallType {

NONE, // no wall, can pass

SOLID, // standard wall

DOOR, // can open/close

LOCKED\_DOOR, // needs key

BARS, // can see through, not walk

ILLUSORY, // looks solid, can walk through

ONE\_WAY, // can pass from one direction only

WINDOW, // see through, not walk

DESTROYABLE // can be destroyed

}

**Movement and Collision**

pseudocode

class MovementSystem {

function CanMove(from: GridPos, direction: Direction, grid: DungeonGrid) -> boolean {

to = from + DirectionVector(direction)

// Check bounds

if !IsInBounds(to, grid):

return false

// Check wall between cells

wall = GetWallBetween(from, to, grid)

if wall.exists:

switch wall.type:

case SOLID:

return false

case DOOR:

return wall.state == OPEN

case LOCKED\_DOOR:

return wall.state == OPEN

case BARS, WINDOW:

return false

case ILLUSORY:

return true // can walk through

case ONE\_WAY:

return CanPassOneWay(from, to, wall)

case DESTROYABLE:

return false // until destroyed

return true

}

function GetWallBetween(from: GridPos, to: GridPos, grid: DungeonGrid) -> Wall {

dx = to.x - from.x

dy = to.y - from.y

if dx == 1: // moving east

return grid.verticalWalls[to.x][to.y]

else if dx == -1: // moving west

return grid.verticalWalls[from.x][from.y]

else if dy == 1: // moving north

return grid.horizontalWalls[to.x][to.y]

else if dy == -1: // moving south

return grid.horizontalWalls[from.x][from.y]

return Wall(exists: false) // diagonal or same cell

}

}

**Rendering System**

pseudocode

class FirstPersonRenderer {

viewDistance: int = 5 // how many cells ahead to render

fov: float = 90 // field of view in degrees

function RenderView(playerPos: GridPos, playerDir: Direction, grid: DungeonGrid) {

// Render cells in view frustum

for distance in range(0, viewDistance):

for offset in range(-distance, distance + 1):

cellPos = GetCellInView(playerPos, playerDir, distance, offset)

if !IsInBounds(cellPos, grid):

continue

RenderCell(cellPos, distance, offset, grid)

RenderWalls(cellPos, playerPos, playerDir, distance, offset, grid)

}

function RenderWalls(cellPos: GridPos, viewerPos: GridPos,

viewDir: Direction, distance: int, offset: int, grid: DungeonGrid) {

// Determine which walls are visible

// Front wall (far side of cell)

frontWall = GetWallBetween(cellPos, cellPos + DirectionVector(viewDir), grid)

if frontWall.exists:

RenderWallSegment(frontWall, FRONT, distance, offset)

// Side walls

leftDir = RotateDirection(viewDir, -90)

rightDir = RotateDirection(viewDir, 90)

leftWall = GetWallBetween(cellPos, cellPos + DirectionVector(leftDir), grid)

if leftWall.exists:

RenderWallSegment(leftWall, LEFT, distance, offset)

rightWall = GetWallBetween(cellPos, cellPos + DirectionVector(rightDir), grid)

if rightWall.exists:

RenderWallSegment(rightWall, RIGHT, distance, offset)

// Back wall (only if we're in this cell)

if cellPos == viewerPos:

backWall = GetWallBetween(cellPos, cellPos - DirectionVector(viewDir), grid)

if backWall.exists and distance == 0:

// Don't render the wall we're facing away from

}

function RenderWallSegment(wall: Wall, side: WallSide, distance: int, offset: int) {

// Calculate screen position based on perspective

perspectiveScale = 1.0 / (distance + 1)

screenX = SCREEN\_CENTER\_X + (offset \* CELL\_WIDTH \* perspectiveScale)

screenY = SCREEN\_CENTER\_Y

width = CELL\_WIDTH \* perspectiveScale

height = CELL\_HEIGHT \* perspectiveScale

// Draw wall texture

texture = GetWallTexture(wall, side)

DrawTexturedQuad(texture, screenX, screenY, width, height)

// Draw wall features

for feature in wall.decorations:

RenderWallFeature(feature, screenX, screenY, perspectiveScale)

}

}

**Advanced Wall Features**

pseudocode

// Secret walls and passages

class SecretWall extends Wall {

trigger: TriggerType // how to open

revealedState: Wall // what it becomes when revealed

function OnTrigger() {

// Transform into revealed state

this.type = revealedState.type

this.texture = revealedState.texture

PlayAnimation("wall\_slide\_away")

}

}

// Multi-cell walls and features

class WallFeature {

type: enum { TORCH, LEVER, SIGN, ALCOVE, FOUNTAIN, DOOR\_FRAME }

position: float // 0.0 to 1.0 along the wall

height: float // 0.0 to 1.0 up the wall

interaction: Interaction

visualEffect: Effect

}

// Diagonal walls for non-square rooms

class DiagonalWallSystem {

diagonalWalls: Map<(GridPos, GridPos), Wall>

function AddDiagonalWall(cell: GridPos, corner: DiagonalDir, wall: Wall) {

// Store wall that cuts across corner of cell

key = (cell, corner)

diagonalWalls[key] = wall

}

function CheckDiagonalCollision(from: Vector2, to: Vector2, grid: DungeonGrid) -> boolean {

// Check if movement crosses any diagonal walls

cellsTraversed = GetCellsOnLine(from, to)

for cell in cellsTraversed:

for corner in [NE, NW, SE, SW]:

if diagonalWalls.contains((cell, corner)):

wall = diagonalWalls[(cell, corner)]

if LineIntersectsCornerWall(from, to, cell, corner):

return !CanPassWall(wall)

return true

}

}

**Visibility System**

pseudocode

class VisibilitySystem {

function CanSeeThrough(from: GridPos, to: GridPos, grid: DungeonGrid) -> boolean {

cells = GetCellsOnLine(from, to)

for i in range(0, cells.length - 1):

wall = GetWallBetween(cells[i], cells[i+1], grid)

if wall.exists:

switch wall.type:

case SOLID, DOOR, LOCKED\_DOOR, DESTROYABLE:

if wall.state != OPEN:

return false

case BARS, WINDOW:

continue // can see through

case ILLUSORY:

continue // looks solid but isn't for LOS

return true

}

function CalculateLighting(cell: GridPos, grid: DungeonGrid) -> float {

totalLight = cell.ambientLight

// Check all light sources

for source in grid.lightSources:

if CanSeeThrough(cell, source.position, grid):

distance = Distance(cell, source.position)

attenuation = 1.0 / (1.0 + distance \* source.falloff)

totalLight += source.intensity \* attenuation

return min(totalLight, 1.0)

}

}

**Door System**

pseudocode

class DoorSystem {

function InteractWithDoor(wall: Wall, player: Player) -> boolean {

if wall.type != DOOR and wall.type != LOCKED\_DOOR:

return false

switch wall.state:

case CLOSED:

if wall.type == LOCKED\_DOOR:

if player.HasKey(wall.keyId):

wall.state = OPEN

PlaySound("door\_unlock")

else:

ShowMessage("Door is locked")

PlaySound("door\_locked")

else:

wall.state = OPEN

PlaySound("door\_open")

case OPEN:

// Check if anything blocking

if !IsPathBlocked(wall):

wall.state = CLOSED

PlaySound("door\_close")

else:

ShowMessage("Something is blocking the door")

return true

}

function AutoCloseDoor(wall: Wall, timer: float) {

if wall.state == OPEN and timer >= wall.autoCloseTime:

if !IsPathBlocked(wall):

wall.state = CLOSED

}

}

**Minimap Rendering**

pseudocode

class MinimapRenderer {

function DrawMinimap(grid: DungeonGrid, playerPos: GridPos, explored: Set<GridPos>) {

scale = 8 // pixels per cell

for x in range(0, grid.width):

for y in range(0, grid.height):

if !explored.contains((x, y)):

continue // Don't show unexplored

// Draw cell

cellX = x \* scale

cellY = y \* scale

DrawRectangle(cellX, cellY, scale, scale, FLOOR\_COLOR)

// Draw walls as lines

// North wall

if grid.horizontalWalls[x][y].exists:

DrawLine(cellX, cellY, cellX + scale, cellY, WALL\_COLOR)

// West wall

if grid.verticalWalls[x][y].exists:

DrawLine(cellX, cellY, cellX, cellY + scale, WALL\_COLOR)

// Draw doors differently

wall = grid.horizontalWalls[x][y]

if wall.type == DOOR:

color = wall.state == OPEN ? OPEN\_DOOR\_COLOR : CLOSED\_DOOR\_COLOR

DrawLine(cellX + scale/4, cellY, cellX + 3\*scale/4, cellY, color)

// Draw player

DrawCircle(playerPos.x \* scale + scale/2,

playerPos.y \* scale + scale/2,

scale/3, PLAYER\_COLOR)

}

}

**Level Generation**

pseudocode

class DungeonGenerator {

function GenerateDungeon(width: int, height: int) -> DungeonGrid {

grid = new DungeonGrid(width, height)

// Initialize all walls as solid

for x in range(0, width + 1):

for y in range(0, height + 1):

grid.verticalWalls[x][y] = Wall(exists: true, type: SOLID)

grid.horizontalWalls[x][y] = Wall(exists: true, type: SOLID)

// Carve out rooms

rooms = GenerateRooms(width, height)

for room in rooms:

CarveRoom(room, grid)

// Connect rooms with corridors

for i in range(0, rooms.length - 1):

ConnectRooms(rooms[i], rooms[i+1], grid)

// Add doors

AddDoors(grid)

// Add secret passages

AddSecrets(grid)

return grid

}

function CarveRoom(room: Rectangle, grid: DungeonGrid) {

for x in range(room.x1, room.x2):

for y in range(room.y1, room.y2):

// Remove internal walls

if x > room.x1:

grid.verticalWalls[x][y].exists = false

if y > room.y1:

grid.horizontalWalls[x][y].exists = false

}

function ConnectRooms(room1: Rectangle, room2: Rectangle, grid: DungeonGrid) {

// Create L-shaped corridor

start = room1.center

end = room2.center

// Horizontal then vertical

for x in range(min(start.x, end.x), max(start.x, end.x) + 1):

RemoveWall(x, start.y, NORTH, grid)

RemoveWall(x, start.y, SOUTH, grid)

for y in range(min(start.y, end.y), max(start.y, end.y) + 1):

RemoveWall(end.x, y, EAST, grid)

RemoveWall(end.x, y, WEST, grid)

}

}

This system gives you:

1. **Efficient space usage** - Walls don't waste cells
2. **True maze-like layouts** - Walls can create complex paths
3. **Clean rendering** - Walls render as actual walls, not blocks
4. **Flexible wall types** - Doors, windows, secret passages
5. **Classic dungeon crawler feel** - Like Eye of the Beholder, Legend of Grimrock
6. **Easy minimap** - Walls show as lines between cells

The key insight is treating walls as edges of a graph rather than nodes, which is much more natural for first-person dungeon crawlers!

Retry

**J**

In that secret wall extends wall portion

Would an interface work better for that sort of thing or would this be a good case for inheritance

Edit

Excellent question! This is a perfect example of where the choice between interfaces and inheritance really matters. Let me show you both approaches and why **interfaces (composition)** would likely be better here:

**Interface/Composition Approach (Recommended)**

pseudocode

// Core wall uses composition

class Wall {

exists: boolean

type: WallType

material: Material

// Composable behaviors

passages: List<IPassage> // how things move through

visuals: List<IVisual> // how it looks

interactions: List<IInteraction> // what happens when interacted with

triggers: List<ITrigger> // what activates changes

states: List<IState> // state management

}

// Interfaces for behaviors

interface IPassage {

function CanPass(entity: Entity, direction: Direction) -> boolean

function OnPass(entity: Entity)

}

interface IVisual {

function GetTexture(viewSide: Side, state: WallState) -> Texture

function GetTransparency() -> float

function RenderEffects(renderer: Renderer)

}

interface IInteraction {

function CanInteract(entity: Entity) -> boolean

function OnInteract(entity: Entity) -> InteractionResult

function GetPrompt() -> string

}

interface ITrigger {

function CheckTrigger(grid: DungeonGrid, position: GridPos) -> boolean

function OnTrigger(wall: Wall)

}

interface IState {

function GetCurrentState() -> StateData

function TransitionTo(newState: StateData)

function Update(deltaTime: float)

}

// Concrete implementations

class SecretPassage implements IPassage, ITrigger {

revealed: boolean = false

triggerType: enum { PUSH, LEVER, SPELL, ITEM }

requiredItem: ItemId

function CanPass(entity: Entity, direction: Direction) -> boolean {

return revealed

}

function CheckTrigger(grid: DungeonGrid, position: GridPos) -> boolean {

switch triggerType:

case PUSH:

return PlayerPushedWall(position)

case LEVER:

return NearbyLeverActivated(grid, position)

case ITEM:

return PlayerHasItem(requiredItem)

}

function OnTrigger(wall: Wall) {

revealed = true

wall.visuals.add(new DissolveEffect())

PlaySound("secret\_found")

}

}

class DoorBehavior implements IPassage, IInteraction, IState {

isOpen: boolean = false

isLocked: boolean

keyId: ItemId

autoClose: boolean

autoCloseTimer: float

function CanPass(entity: Entity, direction: Direction) -> boolean {

return isOpen

}

function OnInteract(entity: Entity) -> InteractionResult {

if isLocked:

if entity.HasKey(keyId):

isLocked = false

isOpen = true

return SUCCESS("Door unlocked")

return FAILURE("Door is locked")

isOpen = !isOpen

return SUCCESS(isOpen ? "Door opened" : "Door closed")

}

function Update(deltaTime: float) {

if autoClose and isOpen:

autoCloseTimer -= deltaTime

if autoCloseTimer <= 0:

isOpen = false

autoCloseTimer = AUTO\_CLOSE\_DURATION

}

}

class IllusoryWall implements IPassage, IVisual {

function CanPass(entity: Entity, direction: Direction) -> boolean {

return true // Always passable

}

function GetTexture(viewSide: Side, state: WallState) -> Texture {

return TextureLibrary.Get("solid\_wall") // Looks solid

}

function GetTransparency() -> float {

if PlayerHasDetectIllusion():

return 0.5 // Semi-transparent for player

return 1.0 // Fully opaque

}

}

class OneWayPassage implements IPassage, IVisual {

allowedDirection: Direction

function CanPass(entity: Entity, direction: Direction) -> boolean {

return direction == allowedDirection

}

function GetTexture(viewSide: Side, state: WallState) -> Texture {

if viewSide == GetSide(allowedDirection):

return TextureLibrary.Get("passage")

return TextureLibrary.Get("solid\_wall")

}

}

**Creating Complex Walls**

pseudocode

class WallFactory {

function CreateSecretDoor() -> Wall {

wall = new Wall()

// Combine multiple behaviors

secret = new SecretPassage(triggerType: PUSH)

door = new DoorBehavior(autoClose: true)

visual = new SecretDoorVisual()

wall.passages.add(secret)

wall.interactions.add(door)

wall.visuals.add(visual)

wall.triggers.add(secret)

wall.states.add(door)

return wall

}

function CreatePuzzleWall() -> Wall {

wall = new Wall()

// Complex multi-state puzzle wall

puzzle = new PuzzleState(solution: "BLUE,RED,GREEN")

visual = new MultiTextureVisual(texturesByState: puzzleTextures)

passage = new ConditionalPassage(condition: puzzle.IsSolved)

wall.states.add(puzzle)

wall.visuals.add(visual)

wall.passages.add(passage)

wall.interactions.add(new PuzzleInteraction(puzzle))

return wall

}

function CreateMimicWall() -> Wall {

wall = new Wall()

// Wall that's actually a monster!

mimic = new MimicBehavior()

wall.passages.add(mimic)

wall.interactions.add(mimic)

wall.triggers.add(new ProximityTrigger(range: 1, target: mimic))

wall.visuals.add(new MimicVisual())

return wall

}

}

**Why Composition Wins Here**

pseudocode

// The problem with inheritance:

class Wall { ... }

class Door extends Wall { ... }

class SecretWall extends Wall { ... }

class SecretDoor extends ??? // Door or SecretWall? Can't extend both!

// Even worse with more combinations:

class LockableDoor extends Door { ... }

class TrapDoor extends Door { ... }

class LockableTrapDoor extends ??? // Explosion of classes!

class SecretLockableTrapDoor extends ??? // Getting ridiculous!

// With composition, just mix and match:

function CreateComplexWall() -> Wall {

wall = new Wall()

wall.passages.add(new SecretPassage())

wall.passages.add(new DoorBehavior(locked: true))

wall.triggers.add(new TrapTrigger())

wall.interactions.add(new LockInteraction())

// Any combination works!

return wall

}

**Runtime Modification Benefits**

pseudocode

class WallModificationSystem {

function EnchantWall(wall: Wall, enchantment: Enchantment) {

// Add new behaviors at runtime!

switch enchantment:

case PHASING:

// Wall becomes passable during full moon

wall.passages.add(new ConditionalPassage(

condition: () => IsFullMoon()

))

case ANIMATED:

// Wall can talk and give hints

wall.interactions.add(new DialogueInteraction(

lines: ["I am the guardian wall...", "Seek the key in the depths"]

))

case TELEPORTER:

// Walking through teleports you

wall.passages.add(new TeleportPassage(destination: otherSide))

}

function CurseWall(wall: Wall) {

// Modify existing behaviors

wall.triggers.add(new CurseTrigger())

wall.visuals.add(new CurseVisualEffect())

// Wrap existing passage behavior

originalPassage = wall.passages[0]

wall.passages[0] = new CursedPassage(originalPassage)

}

function TransformWall(wall: Wall, duration: float) {

// Temporarily change wall behaviors

transformation = new TemporaryTransformation(duration)

transformation.Store(wall) // Save current state

wall.passages.clear()

wall.passages.add(new PhasePassage())

wall.visuals.add(new ShimmerEffect())

// Revert after duration

ScheduleRevert(wall, transformation, duration)

}

}

**The Interface Advantage for Walls**

pseudocode

// Easy to test

class MockPassage implements IPassage {

passable: boolean

function CanPass(entity: Entity, direction: Direction) -> boolean {

return passable

}

}

// Easy to extend

class PressurePlatePassage implements IPassage {

requiredWeight: float

function CanPass(entity: Entity, direction: Direction) -> boolean {

platePressed = CheckPressurePlate(GetAdjacentPlate())

return platePressed and GetPlateWeight() >= requiredWeight

}

}

// Easy to combine

class CompositePassage implements IPassage {

requirements: List<IPassage>

mode: enum { ALL, ANY, NONE }

function CanPass(entity: Entity, direction: Direction) -> boolean {

switch mode:

case ALL:

return requirements.all(r => r.CanPass(entity, direction))

case ANY:

return requirements.any(r => r.CanPass(entity, direction))

case NONE:

return !requirements.any(r => r.CanPass(entity, direction))

}

}

// Easy to serialize

function SerializeWall(wall: Wall) -> JSON {

data = {

behaviors: []

}

for passage in wall.passages:

data.behaviors.add({

type: passage.GetType(),

config: passage.Serialize()

})

return data

}

function DeserializeWall(data: JSON) -> Wall {

wall = new Wall()

for behaviorData in data.behaviors:

behavior = BehaviorFactory.Create(behaviorData.type)

behavior.Deserialize(behaviorData.config)

wall.AddBehavior(behavior)

return wall

}

**When Inheritance Might Make Sense**

pseudocode

// Only for truly hierarchical relationships

abstract class WallBase {

// Shared core functionality that ALL walls need

position: GridPos

abstract function Render()

abstract function GetBounds() -> Bounds

}

// But even then, prefer composition for behaviors

class StandardWall extends WallBase {

behaviors: List<IWallBehavior> // Still use composition!

function Render() {

for visual in behaviors.filterType(IVisual):

visual.Render()

}

}

**Bottom Line:** For your wall system, **composition with interfaces** is definitely the way to go because:

1. **Walls can have multiple combined behaviors** (secret AND door AND trapped)
2. **Behaviors can be added/removed at runtime** (enchantments, curses, transformations)
3. **No combinatorial explosion** of classes
4. **Easy to test** individual behaviors
5. **Easy to save/load** by serializing behavior lists
6. **Modders can add new behaviors** without touching existing code
7. **Behaviors are reusable** across different game objects

The only inheritance you might want is a base WallBase class for the absolute core functionality, but all the interesting behavior should be through interfaces/composition!

Retry

**J**

How would the walls be stored in the unity engine when building them? As a graph or should each wall be placed individually as an object and have behavior applied?

Edit

Great question! In Unity, there are several approaches, each with trade-offs. Let me show you the most practical solutions:

**Hybrid Approach (Recommended): Data + Pooled GameObjects**

csharp

*// Central data structure holds the truth*

[System.Serializable]

public class DungeonData : ScriptableObject {

public int width;

public int height;

public WallData[,] horizontalWalls;

public WallData[,] verticalWalls;

public CellData[,] cells;

}

[System.Serializable]

public class WallData {

public bool exists;

public WallType type;

public string prefabId;

public List<WallBehaviorData> behaviors;

*// Serializable data only - no GameObjects here*

}

*// Wall renderer manages GameObject pool*

public class WallRenderer : MonoBehaviour {

[SerializeField] private Dictionary<string, GameObject> wallPrefabs;

[SerializeField] private Transform wallContainer;

private Dictionary<Vector2Int, GameObject> activeWalls;

private Queue<GameObject> wallPool;

private DungeonData dungeonData;

public void BuildDungeon(DungeonData data) {

dungeonData = data;

ClearExistingWalls();

*// Only create GameObjects for walls near player*

UpdateVisibleWalls(playerPosition, viewDistance);

}

void UpdateVisibleWalls(Vector3 playerPos, float distance) {

Vector2Int gridPos = WorldToGrid(playerPos);

*// Calculate visible range*

int minX = gridPos.x - Mathf.CeilToInt(distance);

int maxX = gridPos.x + Mathf.CeilToInt(distance);

*// Spawn/despawn walls as needed*

HashSet<Vector2Int> neededWalls = GetWallsInRange(minX, maxX, minY, maxY);

*// Spawn new walls*

foreach (var wallPos in neededWalls) {

if (!activeWalls.ContainsKey(wallPos)) {

SpawnWall(wallPos);

}

}

*// Return distant walls to pool*

var toRemove = new List<Vector2Int>();

foreach (var kvp in activeWalls) {

if (!neededWalls.Contains(kvp.Key)) {

ReturnToPool(kvp.Value);

toRemove.Add(kvp.Key);

}

}

foreach (var key in toRemove) {

activeWalls.Remove(key);

}

}

GameObject SpawnWall(Vector2Int dataPos) {

WallData data = GetWallData(dataPos);

GameObject wall = GetFromPool(data.prefabId);

wall.transform.position = GridToWorld(dataPos);

wall.transform.rotation = GetWallRotation(dataPos);

*// Apply behaviors*

var wallComponent = wall.GetComponent<WallController>();

wallComponent.Initialize(data, dataPos);

activeWalls[dataPos] = wall;

return wall;

}

}

**Wall GameObject Component**

csharp

public class WallController : MonoBehaviour {

private WallData data;

private Vector2Int gridPosition;

private List<IWallBehavior> behaviors = new List<IWallBehavior>();

*// Components for different wall features*

[SerializeField] private MeshRenderer meshRenderer;

[SerializeField] private Collider wallCollider;

[SerializeField] private GameObject doorMesh;

[SerializeField] private Transform pivotPoint;

public void Initialize(WallData data, Vector2Int gridPos) {

this.data = data;

this.gridPosition = gridPos;

*// Clear old behaviors*

foreach (var behavior in behaviors) {

behavior.Cleanup();

}

behaviors.Clear();

*// Add behaviors based on data*

foreach (var behaviorData in data.behaviors) {

var behavior = WallBehaviorFactory.Create(behaviorData);

behavior.Initialize(this, behaviorData);

behaviors.Add(behavior);

}

UpdateVisuals();

}

void UpdateVisuals() {

*// Set material based on wall type*

meshRenderer.material = WallMaterialLibrary.Get(data.type);

*// Enable/disable components*

doorMesh.SetActive(data.type == WallType.Door);

*// Let behaviors modify visuals*

foreach (var behavior in behaviors) {

behavior.UpdateVisuals(this);

}

}

public void OnPlayerInteract(Player player) {

foreach (var behavior in behaviors) {

if (behavior is IInteractable interactable) {

if (interactable.CanInteract(player)) {

interactable.OnInteract(player);

*// Update data model*

DungeonManager.Instance.UpdateWallData(gridPosition, data);

*// Update visuals*

UpdateVisuals();

}

}

}

}

}

**Level Editor Workflow**

csharp

[CustomEditor(typeof(DungeonBuilder))]

public class DungeonBuilderEditor : Editor {

private DungeonBuilder builder;

private WallType selectedWallType = WallType.Solid;

private bool editingMode = false;

void OnSceneGUI() {

if (!editingMode) return;

Event e = Event.current;

Ray ray = HandleUtility.GUIPointToWorldRay(e.mousePosition);

if (Physics.Raycast(ray, out RaycastHit hit, 1000f, LayerMask.GetMask("GridPlane"))) {

Vector2Int gridPos = WorldToGrid(hit.point);

WallPlacement placement = GetNearestWallPlacement(hit.point, gridPos);

*// Preview wall*

DrawWallPreview(placement);

*// Place wall on click*

if (e.type == EventType.MouseDown && e.button == 0) {

PlaceWall(placement);

e.Use();

}

}

}

void PlaceWall(WallPlacement placement) {

*// Update data*

builder.dungeonData.SetWall(placement.gridPos, placement.orientation, new WallData {

exists = true,

type = selectedWallType,

prefabId = GetPrefabId(selectedWallType),

behaviors = GetDefaultBehaviors(selectedWallType)

});

*// Create preview GameObject (for editor only)*

if (builder.showPreview) {

builder.wallRenderer.SpawnWall(placement.gridPos);

}

EditorUtility.SetDirty(builder.dungeonData);

}

}

**Prefab Structure**

csharp

*// Wall prefab hierarchy example:*

*/\**

WallPrefab

├── Mesh (with LOD group)

│ ├── LOD0\_Detailed

│ ├── LOD1\_Medium

│ └── LOD2\_Simple

├── Collider

├── InteractionTrigger

├── Effects

│ ├── Torches

│ ├── Decorations

│ └── ParticleEffects

└── DoorComponents (disabled by default)

├── DoorMesh

├── DoorPivot

└── DoorAnimator

*\*/*

public class WallPrefabSetup : MonoBehaviour {

[Header("Wall Variations")]

public GameObject solidWallPrefab;

public GameObject doorWallPrefab;

public GameObject barsWallPrefab;

public GameObject windowWallPrefab;

[Header("Optimization")]

public bool useLOD = true;

public float[] lodDistances = { 10f, 20f, 50f };

[Header("Batching")]

public bool enableBatching = true;

public Material[] sharedMaterials; *// Use same materials for batching*

}

**Optimization Strategies**

csharp

public class WallOptimizer : MonoBehaviour {

*// 1. Merge static walls into chunks*

public void GenerateStaticBatches(DungeonData data) {

*// Group walls by material*

Dictionary<Material, List<MeshFilter>> wallsByMaterial = new();

foreach (var wall in GetStaticWalls(data)) {

Material mat = GetWallMaterial(wall);

if (!wallsByMaterial.ContainsKey(mat)) {

wallsByMaterial[mat] = new List<MeshFilter>();

}

wallsByMaterial[mat].Add(wall.meshFilter);

}

*// Combine meshes per material*

foreach (var kvp in wallsByMaterial) {

CombineInstance[] combine = new CombineInstance[kvp.Value.Count];

for (int i = 0; i < kvp.Value.Count; i++) {

combine[i].mesh = kvp.Value[i].sharedMesh;

combine[i].transform = kvp.Value[i].transform.localToWorldMatrix;

}

GameObject combined = new GameObject($"WallBatch\_{kvp.Key.name}");

var meshFilter = combined.AddComponent<MeshFilter>();

meshFilter.mesh = new Mesh();

meshFilter.mesh.CombineMeshes(combine, true, true);

var renderer = combined.AddComponent<MeshRenderer>();

renderer.material = kvp.Key;

*// Mark as static for additional optimizations*

combined.isStatic = true;

}

}

*// 2. Use occlusion culling*

public void SetupOcclusionPortals(DungeonData data) {

foreach (var doorPosition in GetDoorPositions(data)) {

GameObject portal = new GameObject("OcclusionPortal");

portal.transform.position = GridToWorld(doorPosition);

var occlusionPortal = portal.AddComponent<OcclusionPortal>();

occlusionPortal.open = false; *// Closed by default*

*// Link to door behavior*

var door = GetWallAtPosition(doorPosition);

door.GetComponent<DoorBehavior>().occlusionPortal = occlusionPortal;

}

}

*// 3. LOD setup*

public void ConfigureLODs(GameObject wallPrefab) {

LODGroup lodGroup = wallPrefab.AddComponent<LODGroup>();

LOD[] lods = new LOD[3];

*// High detail*

lods[0] = new LOD(0.6f, wallPrefab.GetComponentsInChildren<Renderer>());

*// Medium detail - simplified mesh*

lods[1] = new LOD(0.3f, new Renderer[] { simplifiedMeshRenderer });

*// Low detail - billboard*

lods[2] = new LOD(0.1f, new Renderer[] { billboardRenderer });

lodGroup.SetLODs(lods);

}

}

**Runtime Modification System**

csharp

public class WallModificationManager : MonoBehaviour {

private DungeonData dungeonData;

private WallRenderer wallRenderer;

public void DestroyWall(Vector2Int position, WallOrientation orientation) {

*// Update data model*

var wallData = dungeonData.GetWall(position, orientation);

wallData.exists = false;

*// Update visual*

if (wallRenderer.HasActiveWall(position)) {

GameObject wallObj = wallRenderer.GetWall(position);

*// Play destruction effect*

var destruction = wallObj.GetComponent<DestructionEffect>();

if (destruction != null) {

destruction.Play(() => {

wallRenderer.ReturnToPool(wallObj);

});

} else {

wallRenderer.ReturnToPool(wallObj);

}

}

*// Update pathfinding*

PathfindingManager.Instance.RecalculateGrid(position);

}

public void TransformWall(Vector2Int position, WallType newType) {

var wallData = dungeonData.GetWall(position, orientation);

wallData.type = newType;

*// If wall is active, update it*

if (wallRenderer.HasActiveWall(position)) {

var wallObj = wallRenderer.GetWall(position);

wallObj.GetComponent<WallController>().Initialize(wallData, position);

}

}

}

**Serialization for Save System**

csharp

[System.Serializable]

public class SerializedDungeon {

public int width;

public int height;

public List<SerializedWall> walls;

public static SerializedDungeon FromDungeonData(DungeonData data) {

var serialized = new SerializedDungeon {

width = data.width,

height = data.height,

walls = new List<SerializedWall>()

};

*// Only serialize walls that exist*

for (int x = 0; x < data.width + 1; x++) {

for (int y = 0; y < data.height + 1; y++) {

*// Horizontal walls*

if (data.horizontalWalls[x, y].exists) {

serialized.walls.Add(new SerializedWall {

x = x, y = y,

orientation = WallOrientation.Horizontal,

data = data.horizontalWalls[x, y]

});

}

*// Vertical walls*

if (data.verticalWalls[x, y].exists) {

serialized.walls.Add(new SerializedWall {

x = x, y = y,

orientation = WallOrientation.Vertical,

data = data.verticalWalls[x, y]

});

}

}

}

return serialized;

}

}

public class SaveManager : MonoBehaviour {

public void SaveDungeon(DungeonData data, string filename) {

var serialized = SerializedDungeon.FromDungeonData(data);

string json = JsonUtility.ToJson(serialized, true);

File.WriteAllText(Path.Combine(Application.persistentDataPath, filename), json);

}

}

**Best Practices Summary**

**DO:**

* ✅ Store wall data in a central data structure (ScriptableObject or plain class)
* ✅ Use object pooling for wall GameObjects
* ✅ Only spawn walls near the player
* ✅ Batch static walls together
* ✅ Use LODs for distant walls
* ✅ Keep behaviors as data until needed

**DON'T:**

* ❌ Create GameObjects for every wall at once (memory waste)
* ❌ Store game logic in GameObjects (hard to serialize)
* ❌ Use FindObjectOfType to locate walls (slow)
* ❌ Instantiate/Destroy walls frequently (causes GC spikes)

**Architecture:**

DungeonData (ScriptableObject)

↓

WallRenderer (spawns/pools GameObjects)

↓

WallController (on each wall GameObject)

↓

IWallBehavior implementations (components)

This approach gives you the best of both worlds:

* **Data-driven** for easy serialization and modification
* **GameObject pooling** for performance
* **Component-based** behaviors for flexibility
* **Editor-friendly** for level design

The key insight is that the **data model** (graph/grid) is separate from the **visual representation** (GameObjects), which lets you optimize each independently!

Retry

**J**

And would I be able to make a tool in unity that let's me create dungeons in a more simplified way such that I can just draw lines and add props/advanced walls and floors/random encounter areas in a 2d plane that will then be transformed into the data that can turn into the dungeon in 3d?

Edit

Absolutely! This is a fantastic approach that many professional dungeon crawlers use. You can create a 2D editor tool that feels like drawing a blueprint, then transforms it into 3D dungeon data. Here's how to build it:

**Custom 2D Dungeon Editor Window**

csharp

using UnityEngine;

using UnityEditor;

using System.Collections.Generic;

public class DungeonPainterWindow : EditorWindow {

*// Drawing modes*

private enum DrawMode {

Wall,

Erase,

Door,

Props,

Encounters,

Floors,

Regions

}

private DrawMode currentMode = DrawMode.Wall;

private WallType selectedWallType = WallType.Solid;

private GameObject selectedProp;

private FloorType selectedFloor = FloorType.Stone;

private EncounterType selectedEncounter = EncounterType.Random;

*// Grid settings*

private int gridWidth = 50;

private int gridHeight = 50;

private float cellSize = 20f; *// pixels per cell*

private Vector2 scrollPosition;

private Vector2 gridOffset = new Vector2(10, 10);

*// Dungeon data*

private DungeonData dungeonData;

private Dictionary<Vector2Int, List<PropPlacement>> props = new();

*// Drawing state*

private bool isDrawing;

private Vector2Int lastGridPos;

private List<Vector2Int> currentStroke = new();

[MenuItem("Tools/Dungeon Painter")]

public static void OpenWindow() {

var window = GetWindow<DungeonPainterWindow>("Dungeon Painter");

window.minSize = new Vector2(800, 600);

}

void OnEnable() {

if (dungeonData == null) {

dungeonData = CreateInstance<DungeonData>();

dungeonData.Initialize(gridWidth, gridHeight);

}

}

void OnGUI() {

DrawToolbar();

DrawCanvas();

DrawSidebar();

HandleInput();

}

void DrawToolbar() {

EditorGUILayout.BeginHorizontal(EditorStyles.toolbar);

*// Mode buttons*

if (GUILayout.Toggle(currentMode == DrawMode.Wall, "Wall", EditorStyles.toolbarButton))

currentMode = DrawMode.Wall;

if (GUILayout.Toggle(currentMode == DrawMode.Door, "Door", EditorStyles.toolbarButton))

currentMode = DrawMode.Door;

if (GUILayout.Toggle(currentMode == DrawMode.Props, "Props", EditorStyles.toolbarButton))

currentMode = DrawMode.Props;

if (GUILayout.Toggle(currentMode == DrawMode.Floors, "Floors", EditorStyles.toolbarButton))

currentMode = DrawMode.Floors;

if (GUILayout.Toggle(currentMode == DrawMode.Encounters, "Encounters", EditorStyles.toolbarButton))

currentMode = DrawMode.Encounters;

if (GUILayout.Toggle(currentMode == DrawMode.Erase, "Erase", EditorStyles.toolbarButton))

currentMode = DrawMode.Erase;

GUILayout.FlexibleSpace();

*// Actions*

if (GUILayout.Button("Clear", EditorStyles.toolbarButton)) {

if (EditorUtility.DisplayDialog("Clear Dungeon", "Clear all data?", "Clear", "Cancel")) {

dungeonData.Clear();

}

}

if (GUILayout.Button("Generate 3D", EditorStyles.toolbarButton)) {

Generate3DDungeon();

}

EditorGUILayout.EndHorizontal();

}

void DrawCanvas() {

Rect canvasRect = new Rect(200, 20, position.width - 400, position.height - 40);

GUI.Box(canvasRect, GUIContent.none, EditorStyles.helpBox);

*// Scrollable grid area*

Rect viewRect = new Rect(0, 0, gridWidth \* cellSize, gridHeight \* cellSize);

scrollPosition = GUI.BeginScrollView(canvasRect, scrollPosition, viewRect);

*// Draw grid*

DrawGrid();

*// Draw walls*

DrawWalls();

*// Draw props*

DrawProps();

*// Draw floor types*

DrawFloorTypes();

*// Draw encounter zones*

DrawEncounterZones();

*// Draw current stroke preview*

if (isDrawing && currentStroke.Count > 0) {

DrawStrokePreview();

}

GUI.EndScrollView();

}

void DrawGrid() {

Handles.color = new Color(0.5f, 0.5f, 0.5f, 0.2f);

*// Vertical lines*

for (int x = 0; x <= gridWidth; x++) {

Vector2 start = GridToScreen(x, 0);

Vector2 end = GridToScreen(x, gridHeight);

Handles.DrawLine(new Vector3(start.x, start.y), new Vector3(end.x, end.y));

}

*// Horizontal lines*

for (int y = 0; y <= gridHeight; y++) {

Vector2 start = GridToScreen(0, y);

Vector2 end = GridToScreen(gridWidth, y);

Handles.DrawLine(new Vector3(start.x, start.y), new Vector3(end.x, end.y));

}

}

void DrawWalls() {

Handles.color = Color.white;

*// Draw horizontal walls*

for (int x = 0; x < gridWidth; x++) {

for (int y = 0; y <= gridHeight; y++) {

if (dungeonData.horizontalWalls[x, y].exists) {

DrawWallLine(x, y, x + 1, y, dungeonData.horizontalWalls[x, y].type);

}

}

}

*// Draw vertical walls*

for (int x = 0; x <= gridWidth; x++) {

for (int y = 0; y < gridHeight; y++) {

if (dungeonData.verticalWalls[x, y].exists) {

DrawWallLine(x, y, x, y + 1, dungeonData.verticalWalls[x, y].type);

}

}

}

}

void DrawWallLine(float x1, float y1, float x2, float y2, WallType type) {

Vector2 start = GridToScreen(x1, y1);

Vector2 end = GridToScreen(x2, y2);

*// Different colors/styles for different wall types*

switch (type) {

case WallType.Solid:

Handles.color = Color.white;

Handles.DrawLine(new Vector3(start.x, start.y), new Vector3(end.x, end.y), 3f);

break;

case WallType.Door:

Handles.color = Color.yellow;

DrawDottedLine(start, end, 5f);

break;

case WallType.Secret:

Handles.color = Color.magenta;

DrawDottedLine(start, end, 3f);

break;

case WallType.Bars:

Handles.color = Color.cyan;

DrawDottedLine(start, end, 2f);

break;

}

}

void HandleInput() {

Event e = Event.current;

Rect canvasRect = new Rect(200, 20, position.width - 400, position.height - 40);

if (!canvasRect.Contains(e.mousePosition)) return;

Vector2 mouseInCanvas = e.mousePosition - new Vector2(200, 20) + scrollPosition;

Vector2Int gridPos = ScreenToGrid(mouseInCanvas);

switch (e.type) {

case EventType.MouseDown:

if (e.button == 0) {

StartDrawing(gridPos);

e.Use();

}

break;

case EventType.MouseDrag:

if (isDrawing) {

ContinueDrawing(gridPos);

e.Use();

}

break;

case EventType.MouseUp:

if (isDrawing) {

FinishDrawing();

e.Use();

}

break;

}

*// Force repaint while drawing*

if (isDrawing) {

Repaint();

}

}

void StartDrawing(Vector2Int gridPos) {

isDrawing = true;

lastGridPos = gridPos;

currentStroke.Clear();

currentStroke.Add(gridPos);

if (currentMode == DrawMode.Props) {

PlaceProp(gridPos);

} else if (currentMode == DrawMode.Floors) {

PaintFloor(gridPos);

}

}

void ContinueDrawing(Vector2Int gridPos) {

if (gridPos == lastGridPos) return;

if (currentMode == DrawMode.Wall || currentMode == DrawMode.Door || currentMode == DrawMode.Erase) {

*// Draw line from last position to current*

DrawLineBetween(lastGridPos, gridPos);

} else if (currentMode == DrawMode.Floors) {

PaintFloor(gridPos);

} else if (currentMode == DrawMode.Encounters) {

*// Add to region*

currentStroke.Add(gridPos);

}

lastGridPos = gridPos;

}

void DrawLineBetween(Vector2Int start, Vector2Int end) {

*// Determine if this is a horizontal or vertical wall*

Vector2Int diff = end - start;

if (Mathf.Abs(diff.x) > Mathf.Abs(diff.y)) {

*// Horizontal movement - place vertical walls*

int y = start.y;

int startX = Mathf.Min(start.x, end.x);

int endX = Mathf.Max(start.x, end.x);

for (int x = startX; x <= endX; x++) {

PlaceWall(new Vector2Int(x, y), WallOrientation.Vertical);

}

} else {

*// Vertical movement - place horizontal walls*

int x = start.x;

int startY = Mathf.Min(start.y, end.y);

int endY = Mathf.Max(start.y, end.y);

for (int y = startY; y <= endY; y++) {

PlaceWall(new Vector2Int(x, y), WallOrientation.Horizontal);

}

}

}

void PlaceWall(Vector2Int pos, WallOrientation orientation) {

WallData wallData = null;

if (orientation == WallOrientation.Horizontal) {

if (pos.x >= 0 && pos.x < gridWidth && pos.y >= 0 && pos.y <= gridHeight) {

wallData = dungeonData.horizontalWalls[pos.x, pos.y];

}

} else {

if (pos.x >= 0 && pos.x <= gridWidth && pos.y >= 0 && pos.y < gridHeight) {

wallData = dungeonData.verticalWalls[pos.x, pos.y];

}

}

if (wallData != null) {

if (currentMode == DrawMode.Erase) {

wallData.exists = false;

} else {

wallData.exists = true;

wallData.type = currentMode == DrawMode.Door ? WallType.Door : selectedWallType;

}

}

}

}

**Smart Drawing Tools**

csharp

public class SmartDrawingTools {

*// Auto-complete rooms*

public static void AutoCompleteRoom(DungeonData data, Vector2Int start, Vector2Int end) {

*// Draw rectangle*

for (int x = start.x; x <= end.x; x++) {

PlaceWall(data, x, start.y, WallOrientation.Horizontal);

PlaceWall(data, x, end.y + 1, WallOrientation.Horizontal);

}

for (int y = start.y; y <= end.y; y++) {

PlaceWall(data, start.x, y, WallOrientation.Vertical);

PlaceWall(data, end.x + 1, y, WallOrientation.Vertical);

}

}

*// Intelligent door placement*

public static void SmartDoor(DungeonData data, Vector2Int pos) {

*// Find walls adjacent to this position*

List<WallPlacement> adjacentWalls = GetAdjacentWalls(data, pos);

if (adjacentWalls.Count == 2) {

*// Corner - place door on one wall*

var wall = adjacentWalls[0];

SetWallType(data, wall, WallType.Door);

} else if (adjacentWalls.Count == 1) {

*// Against single wall*

var wall = adjacentWalls[0];

SetWallType(data, wall, WallType.Door);

}

}

*// Pattern fill*

public static void PatternFill(DungeonData data, Vector2Int start, Vector2Int end, PatternType pattern) {

switch (pattern) {

case PatternType.Maze:

GenerateMaze(data, start, end);

break;

case PatternType.Pillars:

GeneratePillars(data, start, end, spacing: 2);

break;

case PatternType.Cells:

GenerateCells(data, start, end, cellSize: 3);

break;

}

}

}

**Prop and Decoration System**

csharp

[System.Serializable]

public class PropPlacement {

public string propId;

public Vector2Int gridPosition;

public float rotation;

public PropAnchor anchor; *// Floor, Wall, Ceiling, Corner*

[System.Serializable]

public class PropMetadata {

public bool blockMovement;

public bool isInteractable;

public string interactionScript;

public float lightRadius;

public Color lightColor;

}

public PropMetadata metadata;

}

public class PropPainter {

private PropLibrary propLibrary;

public void DrawPropPalette(Rect rect) {

GUILayout.BeginArea(rect);

GUILayout.Label("Props", EditorStyles.boldLabel);

scrollPos = GUILayout.BeginScrollView(scrollPos);

foreach (var category in propLibrary.categories) {

if (foldouts[category.name] = EditorGUILayout.Foldout(foldouts[category.name], category.name)) {

foreach (var prop in category.props) {

Rect buttonRect = GUILayoutUtility.GetRect(64, 64, GUILayout.Width(64));

if (GUI.Button(buttonRect, prop.icon)) {

selectedProp = prop;

}

*// Highlight if selected*

if (prop == selectedProp) {

EditorGUI.DrawRect(buttonRect, new Color(0, 1, 0, 0.3f));

}

}

}

}

GUILayout.EndScrollView();

GUILayout.EndArea();

}

public void PlaceProp(Vector2Int gridPos, PropData prop) {

var placement = new PropPlacement {

propId = prop.id,

gridPosition = gridPos,

rotation = currentRotation,

anchor = DetermineAnchor(prop, gridPos)

};

*// Smart placement based on prop type*

if (prop.type == PropType.WallTorch) {

*// Snap to nearest wall*

placement.anchor = PropAnchor.Wall;

placement.rotation = GetWallRotation(gridPos);

} else if (prop.type == PropType.Chest) {

*// Face away from nearest wall*

placement.rotation = GetFacingRotation(gridPos);

}

dungeonData.AddProp(placement);

}

}

**Encounter Zone Painting**

csharp

[System.Serializable]

public class EncounterZone {

public string id;

public List<Vector2Int> cells = new();

public EncounterType type;

public int difficulty;

public Color editorColor;

[System.Serializable]

public class EncounterData {

public List<string> possibleEnemies;

public int minEnemies;

public int maxEnemies;

public float triggerChance;

public bool oneTimeOnly;

public string specialScript;

}

public EncounterData data;

}

public class EncounterZonePainter {

private Color currentColor = Color.red;

private EncounterType currentType = EncounterType.Random;

public void PaintEncounterZone(List<Vector2Int> cells) {

var zone = new EncounterZone {

id = System.Guid.NewGuid().ToString(),

cells = new List<Vector2Int>(cells),

type = currentType,

editorColor = currentColor

};

dungeonData.encounterZones.Add(zone);

}

public void DrawEncounterOverlay() {

foreach (var zone in dungeonData.encounterZones) {

Color zoneColor = zone.editorColor;

zoneColor.a = 0.3f;

foreach (var cell in zone.cells) {

Rect cellRect = GetCellRect(cell);

EditorGUI.DrawRect(cellRect, zoneColor);

}

*// Draw zone label*

Vector2 center = GetZoneCenter(zone);

GUI.Label(GetCellRect(center), zone.type.ToString(), centeredStyle);

}

}

}

**3D Generation from 2D Data**

csharp

public class Dungeon3DGenerator : MonoBehaviour {

[Header("Prefabs")]

public GameObject floorPrefab;

public GameObject ceilingPrefab;

public WallPrefabSet wallPrefabs;

public PropPrefabLibrary propLibrary;

[Header("Settings")]

public float cellSize = 3f;

public float wallHeight = 3f;

public bool generateLighting = true;

public bool generateNavMesh = true;

public GameObject Generate3DDungeon(DungeonData data) {

GameObject dungeonRoot = new GameObject("Generated Dungeon");

*// Generate floors and ceilings*

GenerateFloors(data, dungeonRoot.transform);

*// Generate walls*

GenerateWalls(data, dungeonRoot.transform);

*// Place props*

PlaceProps(data, dungeonRoot.transform);

*// Setup encounter triggers*

SetupEncounterZones(data, dungeonRoot.transform);

*// Post-processing*

if (generateLighting) {

GenerateLighting(dungeonRoot);

}

if (generateNavMesh) {

GenerateNavMesh(dungeonRoot);

}

*// Optimize*

OptimizeDungeon(dungeonRoot);

return dungeonRoot;

}

void GenerateWalls(DungeonData data, Transform parent) {

GameObject wallContainer = new GameObject("Walls");

wallContainer.transform.parent = parent;

*// Generate horizontal walls*

for (int x = 0; x < data.width; x++) {

for (int y = 0; y <= data.height; y++) {

if (data.horizontalWalls[x, y].exists) {

Vector3 position = new Vector3(

x \* cellSize + cellSize/2,

wallHeight/2,

y \* cellSize

);

var wallPrefab = GetWallPrefab(data.horizontalWalls[x, y].type);

var wall = Instantiate(wallPrefab, position, Quaternion.identity, wallContainer.transform);

wall.name = $"Wall\_H\_{x}\_{y}";

ConfigureWall(wall, data.horizontalWalls[x, y]);

}

}

}

*// Similar for vertical walls...*

}

void PlaceProps(DungeonData data, Transform parent) {

GameObject propContainer = new GameObject("Props");

propContainer.transform.parent = parent;

foreach (var prop in data.props) {

var prefab = propLibrary.GetPrefab(prop.propId);

if (prefab == null) continue;

Vector3 position = GridToWorld(prop.gridPosition);

*// Adjust based on anchor*

switch (prop.anchor) {

case PropAnchor.Wall:

position += GetWallOffset(prop.gridPosition, prop.rotation);

break;

case PropAnchor.Ceiling:

position.y = wallHeight;

break;

case PropAnchor.Corner:

position += GetCornerOffset(prop.gridPosition);

break;

}

var instance = Instantiate(prefab, position, Quaternion.Euler(0, prop.rotation, 0), propContainer.transform);

*// Apply metadata*

if (prop.metadata.blockMovement) {

instance.AddComponent<NavMeshObstacle>();

}

if (prop.metadata.isInteractable) {

var interactable = instance.AddComponent<Interactable>();

interactable.scriptName = prop.metadata.interactionScript;

}

}

}

}

**Validation and Testing**

csharp

public class DungeonValidator {

public class ValidationResult {

public bool isValid;

public List<string> warnings = new();

public List<string> errors = new();

}

public static ValidationResult Validate(DungeonData data) {

var result = new ValidationResult { isValid = true };

*// Check for unreachable areas*

var reachableAreas = FindReachableAreas(data);

if (reachableAreas.Count > 1) {

result.warnings.Add($"Found {reachableAreas.Count} disconnected areas");

}

*// Check for missing walls (open boundaries)*

var openBoundaries = FindOpenBoundaries(data);

if (openBoundaries.Count > 0) {

result.warnings.Add($"Found {openBoundaries.Count} open boundaries");

}

*// Check door placement*

foreach (var door in FindDoors(data)) {

if (!HasAdjacentWalls(door)) {

result.errors.Add($"Door at {door.position} has no adjacent walls");

result.isValid = false;

}

}

*// Check encounter zones*

foreach (var zone in data.encounterZones) {

if (zone.cells.Count == 0) {

result.errors.Add($"Encounter zone {zone.id} has no cells");

result.isValid = false;

}

}

return result;

}

*// Quick play test from editor*

public static void QuickTest(DungeonData data) {

*// Save current scene*

EditorSceneManager.SaveCurrentModifiedScenesIfUserWantsTo();

*// Create test scene*

var testScene = EditorSceneManager.NewScene(NewSceneSetup.EmptyScene);

*// Generate dungeon*

var generator = new GameObject().AddComponent<Dungeon3DGenerator>();

var dungeon = generator.Generate3DDungeon(data);

*// Add player*

var playerPrefab = Resources.Load<GameObject>("Player/PlayerPrefab");

var player = Instantiate(playerPrefab);

player.transform.position = data.playerStartPosition \* generator.cellSize;

*// Enter play mode*

EditorApplication.EnterPlaymode();

}

}

This system gives you:

1. **Intuitive 2D drawing** - Like using a paint program
2. **Smart tools** - Auto-complete rooms, pattern fills
3. **Visual feedback** - See walls, props, zones immediately
4. **Prop library** - Drag and drop decorations
5. **Encounter zones** - Paint areas for battles
6. **Validation** - Catch errors before generation
7. **Quick testing** - Jump into your dungeon instantly
8. **Non-destructive** - Edit the 2D map, regenerate 3D anytime

The key is that you're editing abstract data in 2D, which is much faster than placing 3D objects, then generating the full 3D dungeon from that data. This is how tools like DungeonDraft, Tiled, and many professional level editors work!

Retry

**J**

Would this dungeon editing tool and the way the dungeon is set up allow for multiple heights of floors so ladders/stairs can be added but it wouldn't be a separate floor like in a traditional dungeon crawler

Edit

Excellent question! Yes, we can absolutely extend this system to support variable floor heights within the same level. This creates much more dynamic dungeons with platforms, pits, and vertical movement. Here's how to implement it:

**Extended Data Structure for Heights**

csharp

[System.Serializable]

public class CellData {

public float floorHeight = 0f; *// Height in world units*

public float ceilingHeight = 3f; *// Relative to floor*

public FloorType floorType;

public CeilingType ceilingType;

*// Vertical connections*

public bool hasLadderNorth;

public bool hasLadderSouth;

public bool hasLadderEast;

public bool hasLadderWest;

public bool hasStairs;

public StairDirection stairDirection;

*// Special height features*

public bool isPit; *// Can fall through*

public bool isPlatform; *// Raised area*

public float platformHeight; *// Additional height if platform*

}

[System.Serializable]

public class WallData {

public bool exists;

public WallType type;

*// Height information*

public float bottomHeight; *// Where wall starts*

public float topHeight; *// Where wall ends*

public bool extendsToFloor = true; *// Auto-adjust to floor height*

public bool extendsToCeiling = true; *// Auto-adjust to ceiling*

*// Partial walls*

public bool isHalfWall; *// Can see over but not walk through*

public float halfWallHeight = 1.5f;

*// Vertical transitions*

public bool hasLadder;

public bool hasStairs;

public float climbSpeed = 2f;

}

public enum VerticalTransitionType {

None,

Ladder,

Stairs,

Ramp,

Jump, *// One-way down*

Teleport

}

**Enhanced 2D Editor with Height Painting**

csharp

public class HeightAwareDungeonEditor : EditorWindow {

*// Height painting mode*

private float currentPaintHeight = 0f;

private float heightStep = 0.5f; *// Snap to half-unit increments*

private bool showHeightOverlay = true;

private bool autoGenerateStairs = true;

*// Visualization*

private Dictionary<float, Color> heightColors = new Dictionary<float, Color> {

{ -2f, new Color(0.2f, 0.2f, 0.5f) }, *// Deep pit*

{ -1f, new Color(0.3f, 0.3f, 0.6f) }, *// Shallow pit*

{ 0f, new Color(0.5f, 0.5f, 0.5f) }, *// Ground level*

{ 1f, new Color(0.6f, 0.5f, 0.4f) }, *// Raised*

{ 2f, new Color(0.7f, 0.6f, 0.3f) }, *// Platform*

{ 3f, new Color(0.8f, 0.7f, 0.2f) } *// High platform*

};

void DrawHeightTools() {

GUILayout.BeginVertical(EditorStyles.helpBox);

GUILayout.Label("Height Tools", EditorStyles.boldLabel);

*// Height slider*

currentPaintHeight = EditorGUILayout.Slider("Paint Height", currentPaintHeight, -3f, 5f);

*// Snap to grid*

if (GUILayout.Button($"Snap to {heightStep}m grid")) {

currentPaintHeight = Mathf.Round(currentPaintHeight / heightStep) \* heightStep;

}

*// Quick height buttons*

GUILayout.BeginHorizontal();

if (GUILayout.Button("Pit (-2)")) currentPaintHeight = -2f;

if (GUILayout.Button("Lower (-1)")) currentPaintHeight = -1f;

if (GUILayout.Button("Ground (0)")) currentPaintHeight = 0f;

if (GUILayout.Button("Raised (1)")) currentPaintHeight = 1f;

if (GUILayout.Button("Platform (2)")) currentPaintHeight = 2f;

GUILayout.EndHorizontal();

*// Height tools*

autoGenerateStairs = EditorGUILayout.Toggle("Auto-add Stairs", autoGenerateStairs);

showHeightOverlay = EditorGUILayout.Toggle("Show Heights", showHeightOverlay);

*// Vertical connectors*

GUILayout.Label("Vertical Connectors", EditorStyles.boldLabel);

GUILayout.BeginHorizontal();

if (GUILayout.Button("Ladder")) currentMode = DrawMode.Ladder;

if (GUILayout.Button("Stairs")) currentMode = DrawMode.Stairs;

if (GUILayout.Button("Ramp")) currentMode = DrawMode.Ramp;

GUILayout.EndHorizontal();

GUILayout.EndVertical();

}

void DrawHeightOverlay() {

if (!showHeightOverlay) return;

for (int x = 0; x < gridWidth; x++) {

for (int y = 0; y < gridHeight; y++) {

float height = dungeonData.cells[x, y].floorHeight;

*// Color based on height*

Color heightColor = GetHeightColor(height);

heightColor.a = 0.4f;

Rect cellRect = GetCellRect(x, y);

EditorGUI.DrawRect(cellRect, heightColor);

*// Draw height text*

if (height != 0) {

var style = new GUIStyle(GUI.skin.label);

style.alignment = TextAnchor.MiddleCenter;

style.fontSize = 10;

style.normal.textColor = Color.white;

GUI.Label(cellRect, height.ToString("F1"), style);

}

*// Draw height transitions*

DrawHeightTransitions(x, y);

}

}

}

void DrawHeightTransitions(int x, int y) {

CellData cell = dungeonData.cells[x, y];

*// Check adjacent cells for height differences*

if (x < gridWidth - 1) {

float heightDiff = dungeonData.cells[x + 1, y].floorHeight - cell.floorHeight;

if (Mathf.Abs(heightDiff) > 0.1f) {

DrawHeightTransitionLine(x, y, x + 1, y, heightDiff);

}

}

if (y < gridHeight - 1) {

float heightDiff = dungeonData.cells[x, y + 1].floorHeight - cell.floorHeight;

if (Mathf.Abs(heightDiff) > 0.1f) {

DrawHeightTransitionLine(x, y, x, y + 1, heightDiff);

}

}

}

void DrawHeightTransitionLine(int x1, int y1, int x2, int y2, float heightDiff) {

Vector2 start = GridToScreen(x1, y1);

Vector2 end = GridToScreen(x2, y2);

Vector2 mid = (start + end) / 2;

*// Different line styles for different height changes*

if (Mathf.Abs(heightDiff) <= 0.5f) {

*// Small step - can walk*

Handles.color = Color.green;

Handles.DrawLine(new Vector3(start.x, start.y), new Vector3(end.x, end.y), 2f);

} else if (Mathf.Abs(heightDiff) <= 1.5f) {

*// Need stairs/ladder*

Handles.color = Color.yellow;

Handles.DrawDottedLine(new Vector3(start.x, start.y), new Vector3(end.x, end.y), 3f);

*// Draw stair/ladder icon*

if (HasVerticalConnector(x1, y1, x2, y2)) {

DrawIcon(mid, "ladder\_icon");

}

} else {

*// Too high - need special connection*

Handles.color = Color.red;

Handles.DrawDottedLine(new Vector3(start.x, start.y), new Vector3(end.x, end.y), 5f);

}

}

void PaintHeight(Vector2Int gridPos) {

if (!IsValidCell(gridPos)) return;

float oldHeight = dungeonData.cells[gridPos.x, gridPos.y].floorHeight;

dungeonData.cells[gridPos.x, gridPos.y].floorHeight = currentPaintHeight;

*// Auto-generate vertical connectors if needed*

if (autoGenerateStairs) {

CheckAndAddVerticalConnectors(gridPos, oldHeight, currentPaintHeight);

}

*// Adjust walls to match new height*

AdjustWallHeights(gridPos);

}

void CheckAndAddVerticalConnectors(Vector2Int pos, float oldHeight, float newHeight) {

*// Check all adjacent cells*

for (int dir = 0; dir < 4; dir++) {

Vector2Int adjacent = pos + GetDirectionVector(dir);

if (!IsValidCell(adjacent)) continue;

float adjacentHeight = dungeonData.cells[adjacent.x, adjacent.y].floorHeight;

float heightDiff = Mathf.Abs(adjacentHeight - newHeight);

if (heightDiff > 0.5f && heightDiff <= 2f) {

*// Need connector*

if (heightDiff <= 1f) {

*// Add stairs*

AddStairs(pos, adjacent);

} else {

*// Add ladder*

AddLadder(pos, adjacent);

}

}

}

}

}

**3D Generation with Variable Heights**

csharp

public class HeightAwareDungeonGenerator : MonoBehaviour {

[Header("Height Settings")]

public float unitHeight = 1f; *// World units per height unit*

public bool generateRamps = true;

public float maxWalkableSlope = 30f; *// Degrees*

public GameObject Generate3DDungeon(DungeonData data) {

GameObject dungeonRoot = new GameObject("Multi-Height Dungeon");

*// Generate floors at different heights*

GenerateVariableFloors(data, dungeonRoot.transform);

*// Generate walls with proper heights*

GenerateHeightAwareWalls(data, dungeonRoot.transform);

*// Generate vertical connectors*

GenerateVerticalConnectors(data, dungeonRoot.transform);

return dungeonRoot;

}

void GenerateVariableFloors(DungeonData data, Transform parent) {

GameObject floorContainer = new GameObject("Floors");

floorContainer.transform.parent = parent;

*// Group adjacent cells of same height for optimization*

var heightGroups = GroupCellsByHeight(data);

foreach (var group in heightGroups) {

float height = group.Key;

List<Vector2Int> cells = group.Value;

*// Merge adjacent cells into larger floor pieces*

var floorRegions = MergeAdjacentCells(cells);

foreach (var region in floorRegions) {

CreateFloorMesh(region, height, floorContainer.transform);

}

}

*// Generate transition pieces*

GenerateFloorTransitions(data, floorContainer.transform);

}

void GenerateFloorTransitions(DungeonData data, Transform parent) {

for (int x = 0; x < data.width; x++) {

for (int y = 0; y < data.height; y++) {

float currentHeight = data.cells[x, y].floorHeight;

*// Check east neighbor*

if (x < data.width - 1) {

float eastHeight = data.cells[x + 1, y].floorHeight;

if (Mathf.Abs(eastHeight - currentHeight) > 0.1f) {

CreateTransition(

new Vector3(x + 1, currentHeight, y) \* cellSize,

new Vector3(x + 1, eastHeight, y + 1) \* cellSize,

parent

);

}

}

*// Check north neighbor*

if (y < data.height - 1) {

float northHeight = data.cells[x, y + 1].floorHeight;

if (Mathf.Abs(northHeight - currentHeight) > 0.1f) {

CreateTransition(

new Vector3(x, currentHeight, y + 1) \* cellSize,

new Vector3(x + 1, northHeight, y + 1) \* cellSize,

parent

);

}

}

}

}

}

void CreateTransition(Vector3 lowPoint, Vector3 highPoint, Transform parent) {

float heightDiff = Mathf.Abs(highPoint.y - lowPoint.y);

if (heightDiff <= 0.5f \* unitHeight) {

*// Small step - create a sloped transition*

CreateRamp(lowPoint, highPoint, parent);

} else if (heightDiff <= 1.5f \* unitHeight) {

*// Medium height - create stairs*

CreateStairMesh(lowPoint, highPoint, parent);

} else {

*// Large height difference - just a wall*

CreateWallTransition(lowPoint, highPoint, parent);

}

}

void GenerateHeightAwareWalls(DungeonData data, Transform parent) {

GameObject wallContainer = new GameObject("Walls");

wallContainer.transform.parent = parent;

*// Generate walls that adapt to floor heights*

for (int x = 0; x < data.width; x++) {

for (int y = 0; y <= data.height; y++) {

var wallData = data.horizontalWalls[x, y];

if (!wallData.exists) continue;

*// Get floor heights on both sides of wall*

float southFloor = (y > 0) ? data.cells[x, y - 1].floorHeight : 0f;

float northFloor = (y < data.height) ? data.cells[x, y].floorHeight : 0f;

float bottomHeight = Mathf.Min(southFloor, northFloor);

float topHeight = Mathf.Max(southFloor, northFloor) + wallHeight;

if (wallData.isHalfWall) {

topHeight = bottomHeight + wallData.halfWallHeight;

}

Vector3 position = new Vector3(

x \* cellSize + cellSize/2,

bottomHeight \* unitHeight,

y \* cellSize

);

CreateAdaptiveWall(position, bottomHeight, topHeight, wallData, wallContainer.transform);

}

}

}

void CreateAdaptiveWall(Vector3 position, float bottomHeight, float topHeight,

WallData wallData, Transform parent) {

GameObject wall = new GameObject("Adaptive Wall");

wall.transform.parent = parent;

wall.transform.position = position;

MeshFilter meshFilter = wall.AddComponent<MeshFilter>();

MeshRenderer meshRenderer = wall.AddComponent<MeshRenderer>();

*// Generate wall mesh that fits between heights*

Mesh wallMesh = GenerateWallMesh(bottomHeight, topHeight, cellSize);

meshFilter.mesh = wallMesh;

*// Add ladder if needed*

if (wallData.hasLadder) {

GameObject ladder = Instantiate(ladderPrefab, position, Quaternion.identity, wall.transform);

ladder.transform.localScale = new Vector3(1, topHeight - bottomHeight, 1);

*// Add climbing trigger*

var climbZone = ladder.AddComponent<ClimbingZone>();

climbZone.climbSpeed = wallData.climbSpeed;

}

}

}

**Movement System for Variable Heights**

csharp

public class HeightAwareMovement : MonoBehaviour {

[Header("Movement Settings")]

public float maxStepHeight = 0.5f; *// Can walk up small steps*

public float climbSpeed = 2f;

public float jumpHeight = 1.5f;

private bool isClimbing = false;

private bool isOnStairs = false;

private float currentFloorHeight;

public bool CanMoveTo(Vector2Int from, Vector2Int to, DungeonData data) {

float fromHeight = data.cells[from.x, from.y].floorHeight;

float toHeight = data.cells[to.x, to.y].floorHeight;

float heightDiff = toHeight - fromHeight;

*// Check if there's a wall blocking*

if (HasWallBetween(from, to, data)) {

var wall = GetWallBetween(from, to, data);

*// Can climb if wall has ladder*

if (wall.hasLadder && Mathf.Abs(heightDiff) <= 2f) {

return true;

}

*// Half walls can be vaulted if low enough*

if (wall.isHalfWall && wall.halfWallHeight <= jumpHeight) {

return true;

}

return false;

}

*// Check height difference*

if (heightDiff <= maxStepHeight) {

*// Can walk up*

return true;

} else if (heightDiff < 0 && Mathf.Abs(heightDiff) <= jumpHeight \* 2) {

*// Can drop down (with potential fall damage)*

return true;

} else if (HasVerticalConnector(from, to, data)) {

*// Has stairs/ladder*

return true;

}

return false;

}

void OnTriggerEnter(Collider other) {

if (other.CompareTag("Ladder")) {

*// Enable climbing mode*

isClimbing = true;

GetComponent<Rigidbody>().useGravity = false;

} else if (other.CompareTag("Stairs")) {

isOnStairs = true;

}

}

void HandleClimbing() {

if (!isClimbing) return;

float vertical = Input.GetAxis("Vertical");

transform.position += Vector3.up \* vertical \* climbSpeed \* Time.deltaTime;

*// Check if reached top/bottom*

if (Input.GetButtonDown("Jump") || ReachedLadderEnd()) {

isClimbing = false;

GetComponent<Rigidbody>().useGravity = true;

}

}

}

**Visual Feedback for Heights**

csharp

public class HeightVisualization : MonoBehaviour {

[Header("Height Indicators")]

public bool showHeightGrid = true;

public bool showDropShadows = true;

public bool showHeightFog = true;

void GenerateHeightIndicators(DungeonData data) {

if (showHeightGrid) {

*// Create grid lines at different heights*

for (float h = -3f; h <= 5f; h += 1f) {

CreateHeightGrid(h, GetColorForHeight(h));

}

}

if (showDropShadows) {

*// Add shadows below elevated platforms*

foreach (var platform in FindElevatedAreas(data)) {

CreateDropShadow(platform);

}

}

if (showHeightFog) {

*// Add fog in pits*

foreach (var pit in FindPits(data)) {

CreatePitFog(pit);

}

}

}

void CreateDropShadow(PlatformArea platform) {

GameObject shadow = GameObject.CreatePrimitive(PrimitiveType.Quad);

shadow.transform.rotation = Quaternion.Euler(90, 0, 0);

shadow.transform.position = new Vector3(

platform.center.x,

0.01f, *// Just above ground*

platform.center.z

);

*// Scale based on platform size*

shadow.transform.localScale = new Vector3(

platform.width \* cellSize,

platform.length \* cellSize,

1

);

*// Semi-transparent black material*

var mat = shadow.GetComponent<MeshRenderer>().material;

mat.color = new Color(0, 0, 0, 0.3f);

mat.renderQueue = 3000; *// Render on top*

}

}

**Smart Stair/Ladder Placement**

csharp

public class VerticalConnectorPlacer {

public static void AutoPlaceConnectors(DungeonData data) {

for (int x = 0; x < data.width; x++) {

for (int y = 0; y < data.height; y++) {

CheckAndPlaceConnector(x, y, data);

}

}

}

static void CheckAndPlaceConnector(int x, int y, DungeonData data) {

float currentHeight = data.cells[x, y].floorHeight;

*// Check all four directions*

foreach (Direction dir in System.Enum.GetValues(typeof(Direction))) {

Vector2Int neighbor = GetNeighbor(x, y, dir);

if (!IsValid(neighbor, data)) continue;

float neighborHeight = data.cells[neighbor.x, neighbor.y].floorHeight;

float heightDiff = neighborHeight - currentHeight;

if (Mathf.Abs(heightDiff) < 0.1f) continue; *// Same height*

*// Determine connector type based on height difference*

if (Mathf.Abs(heightDiff) <= 1f) {

*// Small difference - use stairs*

PlaceStairs(x, y, dir, data);

} else if (Mathf.Abs(heightDiff) <= 3f) {

*// Medium difference - use ladder*

PlaceLadder(x, y, dir, data);

} else {

*// Large difference - maybe a lift or teleporter*

PlaceSpecialConnector(x, y, dir, data);

}

}

}

static void PlaceStairs(int x, int y, Direction dir, DungeonData data) {

*// Check if there's room for stairs*

if (HasRoomForStairs(x, y, dir, data)) {

data.cells[x, y].hasStairs = true;

data.cells[x, y].stairDirection = dir;

*// Stairs take up space, mark adjacent cell*

Vector2Int stairEnd = GetNeighbor(x, y, dir);

data.cells[stairEnd.x, stairEnd.y].partiallyOccupied = true;

}

}

}

This enhanced system gives you:

1. **Seamless height variation** - Not just discrete floors but smooth transitions
2. **Visual height feedback** - Color coding and indicators in editor
3. **Smart connections** - Auto-generate stairs/ladders where needed
4. **Flexible movement** - Walk up small steps, climb ladders, use stairs
5. **Natural level design** - Platforms, pits, and elevated areas
6. **Height-aware walls** - Walls adjust to connect different floor levels
7. **Optimization** - Groups same-height areas for better performance

This creates much more dynamic and interesting dungeons than traditional flat-floor crawlers, while still being easy to design in the 2D editor!

Retry

**J**

And without writing the code for it yet, is it at all possible to add tunnels or the ability to go under higher floors as a player?

Edit

Absolutely! This is definitely possible and would add incredible depth to the dungeon design. Here's how it could work conceptually:

**The Core Concept**

Instead of just having floor height, each cell would have:

* **Floor height** (bottom of walkable space)
* **Ceiling height** (top of walkable space)
* **Multiple "layers" of walkable spaces** in the same X,Y position

Think of it like a sandwich - you could have:

* Ground level tunnel (floor: 0, ceiling: 2)
* Stone bridge above it (floor: 2, ceiling: 2.5)
* Main corridor over that (floor: 2.5, ceiling: 5.5)

**Data Structure Approach**

Each cell would contain a **stack of walkable spaces** rather than a single floor:

Cell[5,5]:

Layer 0: Floor 0m, Ceiling 2m (tunnel)

Layer 1: Floor 3m, Ceiling 6m (main floor)

Layer 2: Floor 7m, Ceiling 9m (overhead walkway)

**Editor Visualization Challenges & Solutions**

**2D View Modes:**

1. **Layer Toggle** - Show/hide specific layers like Photoshop
2. **Cross-section View** - Slice through at a specific height
3. **X-Ray Mode** - Semi-transparent overlay showing all layers
4. **Active Layer Editing** - Edit one layer while others are ghosted

**Visual Indicators:**

* Different colors/patterns for each layer
* Dotted lines for "hidden" spaces below
* Height numbers on each layer
* Special icons for vertical connections between layers

**Movement System Implications**

**New movement possibilities:**

* Crawling through low tunnels
* Walking under bridges/platforms
* Enemies dropping from above
* Looking up/down through grates
* Sound traveling between layers

**Collision detection:**

* Check current layer's ceiling height
* Prevent standing up in low spaces
* Allow dropping through holes to lower layers
* Handle partial floors (grates, gaps)

**Rendering Considerations**

**Challenges:**

* Culling geometry the player can't see
* Rendering order for overlapping spaces
* Lighting that respects layer boundaries
* Performance with multiple overlapping meshes

**Solutions:**

* Dynamic LOD based on player's current layer
* Frustum culling per layer
* Portal rendering between layers
* Selective transparency for "X-ray vision" abilities

**Gameplay Opportunities**

**Vertical puzzle design:**

* Drain water from upper level to access tunnel below
* Drop objects through holes to trigger switches
* Navigate between layers to unlock doors
* Stealth: hide in tunnels while enemies patrol above

**Environmental storytelling:**

* Ancient tunnels beneath modern dungeons
* Collapsed floors revealing hidden areas
* Multi-level architecture (sewers → basement → ground floor)

**Technical Feasibility**

**Why it works:**

* The grid system already supports vertical data
* Walls can connect between different layer heights
* Pathfinding can be extended to check layer transitions
* The 2D editor can handle layer selection

**Main complexities:**

* **Visualization** in the editor (solved with layer modes)
* **Pathfinding** across layers (A\* with vertical nodes)
* **Lighting/shadows** between layers
* **Performance** with overlapping geometry

**Implementation Strategy**

1. **Start simple** - Just two layers (main floor + tunnels)
2. **Test core mechanics** - Movement between layers
3. **Expand gradually** - Add more layers as needed
4. **Optimize later** - LOD, culling, batching per layer

The beauty is that this builds naturally on the height system you already have - instead of one floor/ceiling per cell, you just allow multiple floor/ceiling pairs. The editor would need a "current editing layer" dropdown, and the renderer would need to handle multiple meshes per X,Y position, but it's absolutely doable!

This would create incredibly rich, vertical dungeons like:

* Sewers beneath a castle
* Mine tunnels under a cave system
* Multi-story buildings with floors you can fall through
* Overlapping catacomb passages

Would you like me to detail the code implementation for any specific part of this system?

Retry

**J**

If you could detail that code it would be wonderful

Edit

Absolutely! Here's the detailed implementation for a multi-layer dungeon system with tunnels and overlapping floors:

**Core Data Structure for Layers**

csharp

[System.Serializable]

public class LayeredCellData {

public Vector2Int gridPosition;

public List<CellLayer> layers = new List<CellLayer>();

*// Quick lookup for pathfinding*

public Dictionary<int, CellLayer> layerByIndex = new Dictionary<int, CellLayer>();

public CellLayer GetLayerAtHeight(float height) {

foreach (var layer in layers) {

if (height >= layer.floorHeight && height < layer.ceilingHeight) {

return layer;

}

}

return null;

}

public List<CellLayer> GetOverlappingLayers(float bottomHeight, float topHeight) {

return layers.Where(l =>

l.floorHeight < topHeight && l.ceilingHeight > bottomHeight

).ToList();

}

}

[System.Serializable]

public class CellLayer {

public int layerIndex; *// 0 = bottom-most, increases upward*

public float floorHeight;

public float ceilingHeight;

public FloorType floorType;

public CeilingType ceilingType;

*// Layer properties*

public bool isWalkable = true;

public bool allowStanding = true; *// False for crawl spaces*

public bool hasHole; *// Can fall/climb through*

public float holeSize; *// 0-1, percentage of cell*

public Vector2 holeOffset; *// Position within cell*

*// Connections*

public List<LayerConnection> verticalConnections = new List<LayerConnection>();

*// Visibility*

public bool blocksSight = true; *// False for grates*

public bool blocksLight = true;

public float transparency = 0f; *// For glass floors*

public float GetClearance() => ceilingHeight - floorHeight;

}

[System.Serializable]

public class LayerConnection {

public enum ConnectionType {

Ladder,

Stairs,

Hole,

Ramp,

Elevator,

Grate *// Can see through but need to open*

}

public ConnectionType type;

public int fromLayer;

public int toLayer;

public Vector2 position; *// Position within cell*

public float width = 1f;

public bool bidirectional = true;

public bool requiresAction = false; *// Need to press button to use*

}

[System.Serializable]

public class LayeredDungeonData : ScriptableObject {

public int width;

public int height;

public LayeredCellData[,] cells;

*// Walls now need layer information*

public Dictionary<WallKey, LayeredWallData> walls = new Dictionary<WallKey, LayeredWallData>();

public void Initialize(int w, int h) {

width = w;

height = h;

cells = new LayeredCellData[w, h];

for (int x = 0; x < w; x++) {

for (int y = 0; y < h; y++) {

cells[x, y] = new LayeredCellData {

gridPosition = new Vector2Int(x, y)

};

*// Add default ground layer*

cells[x, y].layers.Add(new CellLayer {

layerIndex = 0,

floorHeight = 0,

ceilingHeight = 3,

floorType = FloorType.Stone

});

}

}

}

}

[System.Serializable]

public struct WallKey {

public Vector2Int position;

public WallOrientation orientation;

public int layerIndex;

public WallKey(Vector2Int pos, WallOrientation orient, int layer) {

position = pos;

orientation = orient;

layerIndex = layer;

}

}

[System.Serializable]

public class LayeredWallData {

public bool exists;

public WallType type;

public float bottomHeight;

public float topHeight;

public int connectsFromLayer;

public int connectsToLayer;

}

**Enhanced Editor with Layer Support**

csharp

public class LayeredDungeonEditor : EditorWindow {

private LayeredDungeonData dungeonData;

private int currentEditingLayer = 0;

private LayerViewMode viewMode = LayerViewMode.ActiveLayer;

private enum LayerViewMode {

ActiveLayer, *// Only show current layer*

XRay, *// Show all layers with transparency*

CrossSection, *// Slice at specific height*

StackedView *// Show layers vertically separated*

}

*// Layer editing tools*

private float newLayerFloorHeight = 3f;

private float newLayerCeilingHeight = 6f;

private bool autoConnectLayers = true;

void OnGUI() {

DrawToolbar();

DrawLayerControls();

EditorGUILayout.BeginHorizontal();

DrawLayerList();

DrawCanvas();

DrawPropertyPanel();

EditorGUILayout.EndHorizontal();

}

void DrawLayerControls() {

EditorGUILayout.BeginHorizontal(EditorStyles.toolbar);

GUILayout.Label("View Mode:", GUILayout.Width(70));

viewMode = (LayerViewMode)EditorGUILayout.EnumPopup(viewMode, GUILayout.Width(100));

GUILayout.Space(20);

GUILayout.Label("Editing Layer:", GUILayout.Width(80));

*// Layer selector with preview*

string[] layerNames = GetLayerNames();

int newLayer = EditorGUILayout.Popup(currentEditingLayer, layerNames, GUILayout.Width(150));

if (newLayer != currentEditingLayer) {

currentEditingLayer = newLayer;

Repaint();

}

if (GUILayout.Button("+", GUILayout.Width(25))) {

AddNewLayer();

}

if (GUILayout.Button("-", GUILayout.Width(25))) {

RemoveCurrentLayer();

}

GUILayout.FlexibleSpace();

*// Quick layer visibility toggles*

for (int i = 0; i < GetMaxLayerCount(); i++) {

bool visible = layerVisibility[i];

layerVisibility[i] = GUILayout.Toggle(visible, $"L{i}", GUILayout.Width(35));

}

EditorGUILayout.EndHorizontal();

}

void DrawCanvas() {

Rect canvasRect = GUILayoutUtility.GetRect(600, 600, GUILayout.ExpandWidth(true), GUILayout.ExpandHeight(true));

GUI.Box(canvasRect, GUIContent.none, EditorStyles.helpBox);

switch (viewMode) {

case LayerViewMode.ActiveLayer:

DrawSingleLayer(canvasRect, currentEditingLayer);

break;

case LayerViewMode.XRay:

DrawXRayView(canvasRect);

break;

case LayerViewMode.CrossSection:

DrawCrossSection(canvasRect);

break;

case LayerViewMode.StackedView:

DrawStackedView(canvasRect);

break;

}

}

void DrawSingleLayer(Rect canvasRect, int layerIndex) {

*// Draw grid for this layer*

for (int x = 0; x < dungeonData.width; x++) {

for (int y = 0; y < dungeonData.height; y++) {

var cellData = dungeonData.cells[x, y];

CellLayer layer = cellData.layers.FirstOrDefault(l => l.layerIndex == layerIndex);

if (layer == null) {

*// No layer here - draw as void*

DrawVoidCell(x, y, canvasRect);

} else {

DrawLayerCell(x, y, layer, canvasRect, 1.0f);

*// Draw vertical connections*

DrawVerticalConnections(x, y, layer, canvasRect);

}

}

}

*// Draw walls for this layer*

DrawLayerWalls(layerIndex, canvasRect);

}

void DrawXRayView(Rect canvasRect) {

*// Draw all layers with decreasing opacity*

int maxLayers = GetMaxLayerCount();

for (int layerIndex = 0; layerIndex < maxLayers; layerIndex++) {

if (!layerVisibility[layerIndex]) continue;

float opacity = 1.0f - (layerIndex \* 0.3f);

opacity = Mathf.Max(opacity, 0.2f);

for (int x = 0; x < dungeonData.width; x++) {

for (int y = 0; y < dungeonData.height; y++) {

var cellData = dungeonData.cells[x, y];

CellLayer layer = cellData.layers.FirstOrDefault(l => l.layerIndex == layerIndex);

if (layer != null) {

DrawLayerCell(x, y, layer, canvasRect, opacity);

*// Highlight connections between layers*

if (layer.verticalConnections.Count > 0) {

DrawConnectionHighlight(x, y, layer, canvasRect);

}

}

}

}

}

}

void DrawStackedView(Rect canvasRect) {

float layerSpacing = 100f; *// Pixels between layer visualizations*

int maxLayers = GetMaxLayerCount();

for (int layerIndex = 0; layerIndex < maxLayers; layerIndex++) {

if (!layerVisibility[layerIndex]) continue;

*// Offset each layer vertically*

float yOffset = layerIndex \* layerSpacing;

*// Draw layer label*

GUI.Label(new Rect(canvasRect.x, canvasRect.y + yOffset - 20, 200, 20),

$"Layer {layerIndex} (Height: {GetLayerHeightRange(layerIndex)})");

*// Draw mini-map for this layer*

Rect layerRect = new Rect(canvasRect.x, canvasRect.y + yOffset,

canvasRect.width, layerSpacing - 10);

DrawLayerMinimap(layerIndex, layerRect);

}

}

void DrawLayerCell(int x, int y, CellLayer layer, Rect canvasRect, float opacity) {

Rect cellRect = GetCellRect(x, y, canvasRect);

*// Color based on layer properties*

Color cellColor = GetLayerColor(layer);

cellColor.a = opacity;

EditorGUI.DrawRect(cellRect, cellColor);

*// Draw icons for special features*

if (layer.hasHole) {

DrawHoleIcon(cellRect, layer.holeSize, layer.holeOffset);

}

if (!layer.allowStanding) {

DrawCrawlSpaceIcon(cellRect);

}

if (!layer.blocksSight) {

DrawGratePattern(cellRect);

}

*// Draw height text*

var style = new GUIStyle(GUI.skin.label) {

alignment = TextAnchor.MiddleCenter,

fontSize = 9,

normal = { textColor = Color.white }

};

GUI.Label(cellRect, $"{layer.floorHeight:F1}-{layer.ceilingHeight:F1}", style);

}

void DrawVerticalConnections(int x, int y, CellLayer layer, Rect canvasRect) {

foreach (var connection in layer.verticalConnections) {

Rect cellRect = GetCellRect(x, y, canvasRect);

Vector2 iconPos = cellRect.center + connection.position \* cellRect.width / 2;

*// Draw connection icon*

Texture2D icon = GetConnectionIcon(connection.type);

GUI.DrawTexture(new Rect(iconPos.x - 8, iconPos.y - 8, 16, 16), icon);

*// Draw direction arrow*

if (connection.toLayer > layer.layerIndex) {

DrawArrow(iconPos, Vector2.up, Color.green);

} else {

DrawArrow(iconPos, Vector2.down, Color.yellow);

}

}

}

}

**Layer Manipulation Tools**

csharp

public class LayerManipulationTools {

public static void AddLayerAbove(LayeredCellData cell, int baseLayerIndex) {

var baseLayer = cell.layers.FirstOrDefault(l => l.layerIndex == baseLayerIndex);

if (baseLayer == null) return;

var newLayer = new CellLayer {

layerIndex = baseLayerIndex + 1,

floorHeight = baseLayer.ceilingHeight + 0.5f, *// Small gap*

ceilingHeight = baseLayer.ceilingHeight + 3.5f,

floorType = FloorType.Wood,

isWalkable = true

};

*// Shift existing layers up*

foreach (var layer in cell.layers.Where(l => l.layerIndex > baseLayerIndex)) {

layer.layerIndex++;

}

cell.layers.Add(newLayer);

cell.layers = cell.layers.OrderBy(l => l.layerIndex).ToList();

}

public static void CreateTunnel(LayeredDungeonData data, Vector2Int start, Vector2Int end,

float tunnelFloor, float tunnelCeiling) {

var path = GetPath(start, end);

foreach (var pos in path) {

var cell = data.cells[pos.x, pos.y];

*// Check if tunnel would intersect existing layers*

var overlapping = cell.GetOverlappingLayers(tunnelFloor, tunnelCeiling);

if (overlapping.Count == 0) {

*// Create new tunnel layer*

var tunnelLayer = new CellLayer {

layerIndex = GetNextLayerIndex(cell),

floorHeight = tunnelFloor,

ceilingHeight = tunnelCeiling,

floorType = FloorType.Dirt,

allowStanding = tunnelCeiling - tunnelFloor >= 2f

};

cell.layers.Add(tunnelLayer);

} else {

*// Carve through existing layer*

foreach (var layer in overlapping) {

if (layer.floorHeight < tunnelFloor && layer.ceilingHeight > tunnelCeiling) {

*// Tunnel goes through middle of layer - split it*

SplitLayerForTunnel(cell, layer, tunnelFloor, tunnelCeiling);

}

}

}

}

}

public static void CreateBridge(LayeredDungeonData data, Vector2Int start, Vector2Int end,

float bridgeHeight, float bridgeThickness = 0.5f) {

var path = GetPath(start, end);

foreach (var pos in path) {

var cell = data.cells[pos.x, pos.y];

*// Add bridge as a thin layer*

var bridgeLayer = new CellLayer {

layerIndex = GetNextLayerIndex(cell),

floorHeight = bridgeHeight,

ceilingHeight = bridgeHeight + bridgeThickness,

floorType = FloorType.StoneBridge,

isWalkable = true,

blocksSight = false *// Can see through/under bridge*

};

cell.layers.Add(bridgeLayer);

*// Ensure there's walkable space under the bridge*

var underBridge = cell.GetLayerAtHeight(bridgeHeight - 1f);

if (underBridge != null && underBridge.ceilingHeight > bridgeHeight) {

*// Adjust ceiling to not intersect bridge*

underBridge.ceilingHeight = bridgeHeight - 0.1f;

}

}

}

public static void CreateVerticalConnection(LayeredCellData cell, int fromLayer, int toLayer,

LayerConnection.ConnectionType type) {

var from = cell.layers.FirstOrDefault(l => l.layerIndex == fromLayer);

var to = cell.layers.FirstOrDefault(l => l.layerIndex == toLayer);

if (from == null || to == null) return;

var connection = new LayerConnection {

type = type,

fromLayer = fromLayer,

toLayer = toLayer,

position = Vector2.zero *// Center of cell*

};

*// Add connection to both layers*

from.verticalConnections.Add(connection);

*// Create hole if needed*

if (type == LayerConnection.ConnectionType.Hole ||

type == LayerConnection.ConnectionType.Ladder) {

if (from.ceilingHeight == to.floorHeight) {

from.hasHole = true;

from.holeSize = 0.3f; *// 30% of cell*

}

}

*// Add reverse connection if bidirectional*

if (connection.bidirectional) {

var reverseConnection = new LayerConnection {

type = type,

fromLayer = toLayer,

toLayer = fromLayer,

position = connection.position,

bidirectional = true

};

to.verticalConnections.Add(reverseConnection);

}

}

}

**3D Generation with Layers**

csharp

public class LayeredDungeonGenerator : MonoBehaviour {

[Header("Prefabs")]

public GameObject floorPrefab;

public GameObject ceilingPrefab;

public GameObject holePrefab;

public GameObject ladderPrefab;

public GameObject gratePrefab;

private Dictionary<int, GameObject> layerContainers = new Dictionary<int, GameObject>();

public GameObject GenerateLayeredDungeon(LayeredDungeonData data) {

GameObject dungeonRoot = new GameObject("Layered Dungeon");

*// Generate each layer*

int maxLayers = GetMaxLayerCount(data);

for (int layerIndex = 0; layerIndex < maxLayers; layerIndex++) {

GenerateLayer(data, layerIndex, dungeonRoot.transform);

}

*// Generate vertical connections*

GenerateVerticalConnections(data, dungeonRoot.transform);

*// Optimize overlapping geometry*

OptimizeLayeredGeometry(dungeonRoot);

return dungeonRoot;

}

void GenerateLayer(LayeredDungeonData data, int layerIndex, Transform parent) {

GameObject layerContainer = new GameObject($"Layer\_{layerIndex}");

layerContainer.transform.parent = parent;

layerContainers[layerIndex] = layerContainer;

*// Group connected cells for mesh combining*

var layerRegions = GetConnectedRegions(data, layerIndex);

foreach (var region in layerRegions) {

GenerateLayerRegion(region, layerContainer.transform);

}

}

void GenerateLayerRegion(LayerRegion region, Transform parent) {

*// Create combined mesh for floor*

Mesh floorMesh = CreateRegionFloorMesh(region);

GameObject floor = new GameObject("RegionFloor");

floor.transform.parent = parent;

var meshFilter = floor.AddComponent<MeshFilter>();

meshFilter.mesh = floorMesh;

var meshRenderer = floor.AddComponent<MeshRenderer>();

meshRenderer.material = GetFloorMaterial(region.floorType);

*// Add collision*

var meshCollider = floor.AddComponent<MeshCollider>();

meshCollider.sharedMesh = floorMesh;

*// Generate ceiling (might have holes)*

if (region.hasCeiling) {

GenerateCeilingWithHoles(region, parent);

}

*// Handle special cases*

if (!region.allowStanding) {

*// Add crawl space trigger*

AddCrawlSpaceTrigger(region, parent);

}

if (!region.blocksSight) {

*// Use grate material*

meshRenderer.material = grateMaterial;

}

}

void GenerateCeilingWithHoles(LayerRegion region, Transform parent) {

*// Create ceiling mesh with holes cut out*

Mesh ceilingMesh = new Mesh();

List<Vector3> vertices = new List<Vector3>();

List<int> triangles = new List<int>();

foreach (var cell in region.cells) {

if (cell.layer.hasHole) {

*// Generate ceiling with hole*

AddCeilingWithHole(cell, vertices, triangles);

} else {

*// Normal ceiling quad*

AddCeilingQuad(cell, vertices, triangles);

}

}

ceilingMesh.vertices = vertices.ToArray();

ceilingMesh.triangles = triangles.ToArray();

ceilingMesh.RecalculateNormals();

GameObject ceiling = new GameObject("RegionCeiling");

ceiling.transform.parent = parent;

ceiling.AddComponent<MeshFilter>().mesh = ceilingMesh;

ceiling.AddComponent<MeshRenderer>().material = GetCeilingMaterial(region.ceilingType);

}

void GenerateVerticalConnections(LayeredDungeonData data, Transform parent) {

GameObject connectionContainer = new GameObject("VerticalConnections");

connectionContainer.transform.parent = parent;

for (int x = 0; x < data.width; x++) {

for (int y = 0; y < data.height; y++) {

var cell = data.cells[x, y];

foreach (var layer in cell.layers) {

foreach (var connection in layer.verticalConnections) {

GenerateConnection(x, y, connection, connectionContainer.transform);

}

}

}

}

}

void GenerateConnection(int x, int y, LayerConnection connection, Transform parent) {

Vector3 worldPos = GridToWorld(x, y);

switch (connection.type) {

case LayerConnection.ConnectionType.Ladder:

GenerateLadder(worldPos, connection, parent);

break;

case LayerConnection.ConnectionType.Stairs:

GenerateStairs(worldPos, connection, parent);

break;

case LayerConnection.ConnectionType.Hole:

*// Hole is handled in ceiling generation*

break;

case LayerConnection.ConnectionType.Grate:

GenerateGrate(worldPos, connection, parent);

break;

}

}

void GenerateLadder(Vector3 position, LayerConnection connection, Transform parent) {

*// Get actual heights from layer data*

float bottomHeight = GetLayerFloorHeight(connection.fromLayer);

float topHeight = GetLayerFloorHeight(connection.toLayer);

GameObject ladder = Instantiate(ladderPrefab, parent);

ladder.transform.position = position + new Vector3(connection.position.x, bottomHeight, connection.position.y);

*// Scale ladder to fit height*

float ladderHeight = topHeight - bottomHeight;

ladder.transform.localScale = new Vector3(1, ladderHeight / ladderPrefab.transform.localScale.y, 1);

*// Add climbing zone*

var climbZone = ladder.AddComponent<BoxCollider>();

climbZone.isTrigger = true;

climbZone.size = new Vector3(1, ladderHeight, 1);

var climbable = ladder.AddComponent<ClimbableArea>();

climbable.bottomLayer = connection.fromLayer;

climbable.topLayer = connection.toLayer;

}

}

**Movement System for Layers**

csharp

public class LayeredMovementController : MonoBehaviour {

[Header("Layer Settings")]

public int currentLayer = 0;

public float currentHeight;

public bool isCrawling = false;

public bool isClimbing = false;

private LayeredDungeonData dungeonData;

private CharacterController controller;

void Update() {

UpdateCurrentLayer();

HandleLayerTransitions();

HandleMovementConstraints();

}

void UpdateCurrentLayer() {

Vector2Int gridPos = WorldToGrid(transform.position);

if (!IsValidPosition(gridPos)) return;

var cell = dungeonData.cells[gridPos.x, gridPos.y];

currentHeight = transform.position.y;

*// Find which layer we're in*

var layer = cell.GetLayerAtHeight(currentHeight);

if (layer != null) {

currentLayer = layer.layerIndex;

*// Check if we need to crawl*

float clearance = layer.GetClearance();

if (clearance < standingHeight) {

StartCrawling();

} else if (isCrawling && clearance >= standingHeight) {

*// Can stand up*

if (Input.GetKeyDown(KeyCode.LeftControl)) {

StopCrawling();

}

}

}

}

void HandleLayerTransitions() {

if (isClimbing) {

HandleClimbing();

return;

}

*// Check for holes to fall through*

Vector2Int gridPos = WorldToGrid(transform.position);

var cell = dungeonData.cells[gridPos.x, gridPos.y];

var currentCellLayer = cell.layers.FirstOrDefault(l => l.layerIndex == currentLayer);

if (currentCellLayer != null && currentCellLayer.hasHole) {

*// Check if player is over the hole*

Vector2 localPos = GetLocalPositionInCell(transform.position);

float distToHole = Vector2.Distance(localPos, currentCellLayer.holeOffset);

if (distToHole < currentCellLayer.holeSize / 2) {

*// Fall through hole*

FallToLowerLayer();

}

}

}

void HandleClimbing() {

float vertical = Input.GetAxis("Vertical");

if (vertical != 0) {

*// Move up/down ladder*

transform.position += Vector3.up \* vertical \* climbSpeed \* Time.deltaTime;

*// Check if reached destination layer*

if (HasReachedLayer(targetLayer)) {

ExitClimbing();

}

}

*// Allow jumping off ladder*

if (Input.GetButtonDown("Jump")) {

ExitClimbing();

GetComponent<Rigidbody>().AddForce(transform.forward \* jumpForce);

}

}

void OnTriggerEnter(Collider other) {

if (other.CompareTag("Ladder")) {

var climbable = other.GetComponent<ClimbableArea>();

if (climbable != null) {

canClimb = true;

availableLadder = climbable;

*// Show climb prompt*

UIManager.ShowPrompt("Press E to climb");

}

} else if (other.CompareTag("LayerTransition")) {

var transition = other.GetComponent<LayerTransition>();

HandleAutomaticTransition(transition);

}

}

void HandleMovementConstraints() {

*// Get current layer properties*

Vector2Int gridPos = WorldToGrid(transform.position);

var cell = dungeonData.cells[gridPos.x, gridPos.y];

var layer = cell.layers.FirstOrDefault(l => l.layerIndex == currentLayer);

if (layer == null) return;

*// Prevent standing in low areas*

if (!layer.allowStanding && !isCrawling) {

StartCrawling();

}

*// Check ceiling collision*

float headHeight = transform.position.y + controller.height;

if (headHeight > layer.ceilingHeight) {

*// Push player down*

transform.position = new Vector3(

transform.position.x,

layer.ceilingHeight - controller.height,

transform.position.z

);

}

*// Check floor*

if (transform.position.y < layer.floorHeight) {

*// Either fell through or entering from below*

if (HasLowerLayer(gridPos, transform.position.y)) {

TransitionToLowerLayer();

} else {

*// Hit bottom, stop falling*

transform.position = new Vector3(

transform.position.x,

layer.floorHeight,

transform.position.z

);

}

}

}

}

**Visibility and Rendering**

csharp

public class LayeredVisibilitySystem : MonoBehaviour {

public int playerLayer;

public float viewDistance = 20f;

public bool enableXRayVision = false;

private Dictionary<int, float> layerOpacity = new Dictionary<int, float>();

private Camera mainCamera;

void Update() {

UpdateLayerVisibility();

UpdateLayerOpacity();

CullDistantLayers();

}

void UpdateLayerVisibility() {

*// Hide layers above player by default*

for (int layer = 0; layer < maxLayers; layer++) {

if (layer > playerLayer + 1) {

*// Hide layers more than 1 above player*

SetLayerVisible(layer, false);

} else if (layer == playerLayer + 1) {

*// Make layer directly above semi-transparent*

SetLayerOpacity(layer, 0.3f);

} else {

*// Show layers at and below player*

SetLayerVisible(layer, true);

SetLayerOpacity(layer, 1.0f);

}

}

*// X-ray vision mode*

if (enableXRayVision) {

for (int layer = playerLayer + 1; layer < maxLayers; layer++) {

SetLayerVisible(layer, true);

SetLayerOpacity(layer, 0.2f);

}

}

}

void SetLayerOpacity(int layer, float opacity) {

*// Get all renderers in layer*

GameObject layerContainer = GetLayerContainer(layer);

if (layerContainer == null) return;

foreach (var renderer in layerContainer.GetComponentsInChildren<MeshRenderer>()) {

*// Use transparent shader variant*

foreach (var material in renderer.materials) {

if (opacity < 1.0f) {

*// Switch to transparent shader*

material.shader = transparentShader;

Color color = material.color;

color.a = opacity;

material.color = color;

} else {

*// Switch to opaque shader*

material.shader = opaqueShader;

}

}

}

}

void CullDistantLayers() {

*// Cull geometry on different layers based on distance*

Vector3 playerPos = transform.position;

foreach (var layerContainer in layerContainers) {

int layer = layerContainer.Key;

GameObject container = layerContainer.Value;

*// Different cull distances for different layer relationships*

float cullDistance = viewDistance

Retry

**J**

Continue

Edit

csharp

*// Different cull distances for different layer relationships*

float cullDistance = viewDistance;

if (layer != playerLayer) {

*// Reduce view distance for other layers*

cullDistance \*= 0.7f;

*// Even less for layers far above/below*

int layerDiff = Mathf.Abs(layer - playerLayer);

cullDistance \*= 1.0f / (1.0f + layerDiff \* 0.3f);

}

*// LOD system per layer*

foreach (var lodGroup in container.GetComponentsInChildren<LODGroup>()) {

float distance = Vector3.Distance(playerPos, lodGroup.transform.position);

if (distance > cullDistance) {

lodGroup.gameObject.SetActive(false);

} else {

lodGroup.gameObject.SetActive(true);

*// Adjust LOD bias based on layer*

if (layer != playerLayer) {

lodGroup.lodBias = 0.5f; *// Use lower detail for other layers*

} else {

lodGroup.lodBias = 1.0f;

}

}

}

}

}

}

**Pathfinding Across Layers**

csharp

public class LayeredPathfinding : MonoBehaviour {

private LayeredDungeonData dungeonData;

public class LayeredNode {

public Vector2Int gridPosition;

public int layer;

public float height;

public LayeredNode parent;

public float gCost;

public float hCost;

public float fCost => gCost + hCost;

public LayeredNode(Vector2Int pos, int layer, float height) {

this.gridPosition = pos;

this.layer = layer;

this.height = height;

}

public override bool Equals(object obj) {

if (obj is LayeredNode other) {

return gridPosition == other.gridPosition && layer == other.layer;

}

return false;

}

public override int GetHashCode() {

return gridPosition.GetHashCode() ^ (layer << 16);

}

}

public List<LayeredNode> FindPath(Vector3 start, Vector3 end) {

Vector2Int startGrid = WorldToGrid(start);

Vector2Int endGrid = WorldToGrid(end);

var startCell = dungeonData.cells[startGrid.x, startGrid.y];

var endCell = dungeonData.cells[endGrid.x, endGrid.y];

var startLayer = startCell.GetLayerAtHeight(start.y);

var endLayer = endCell.GetLayerAtHeight(end.y);

if (startLayer == null || endLayer == null) return null;

var startNode = new LayeredNode(startGrid, startLayer.layerIndex, start.y);

var endNode = new LayeredNode(endGrid, endLayer.layerIndex, end.y);

return AStar(startNode, endNode);

}

List<LayeredNode> AStar(LayeredNode start, LayeredNode end) {

var openSet = new List<LayeredNode> { start };

var closedSet = new HashSet<LayeredNode>();

while (openSet.Count > 0) {

var current = openSet.OrderBy(n => n.fCost).First();

openSet.Remove(current);

closedSet.Add(current);

if (current.Equals(end)) {

return RetracePath(start, current);

}

foreach (var neighbor in GetNeighbors(current)) {

if (closedSet.Contains(neighbor)) continue;

float newGCost = current.gCost + GetDistance(current, neighbor);

var existingNode = openSet.FirstOrDefault(n => n.Equals(neighbor));

if (existingNode != null) {

if (newGCost < existingNode.gCost) {

existingNode.gCost = newGCost;

existingNode.parent = current;

}

} else {

neighbor.gCost = newGCost;

neighbor.hCost = GetHeuristic(neighbor, end);

neighbor.parent = current;

openSet.Add(neighbor);

}

}

}

return null; *// No path found*

}

List<LayeredNode> GetNeighbors(LayeredNode node) {

var neighbors = new List<LayeredNode>();

var cell = dungeonData.cells[node.gridPosition.x, node.gridPosition.y];

var layer = cell.layers.FirstOrDefault(l => l.layerIndex == node.layer);

if (layer == null) return neighbors;

*// Horizontal neighbors on same layer*

foreach (var dir in new[] { Vector2Int.up, Vector2Int.down, Vector2Int.left, Vector2Int.right }) {

var neighborPos = node.gridPosition + dir;

if (!IsValidPosition(neighborPos)) continue;

var neighborCell = dungeonData.cells[neighborPos.x, neighborPos.y];

var neighborLayer = neighborCell.layers.FirstOrDefault(l => l.layerIndex == node.layer);

if (neighborLayer != null && neighborLayer.isWalkable) {

*// Check if there's a wall between*

if (!HasWallBetween(node.gridPosition, neighborPos, node.layer)) {

neighbors.Add(new LayeredNode(neighborPos, node.layer, neighborLayer.floorHeight));

}

}

}

*// Vertical neighbors (layer transitions)*

foreach (var connection in layer.verticalConnections) {

*// Same grid position, different layer*

var targetLayer = cell.layers.FirstOrDefault(l => l.layerIndex == connection.toLayer);

if (targetLayer != null && targetLayer.isWalkable) {

neighbors.Add(new LayeredNode(node.gridPosition, connection.toLayer, targetLayer.floorHeight));

}

}

return neighbors;

}

float GetDistance(LayeredNode a, LayeredNode b) {

if (a.layer != b.layer) {

*// Vertical transition cost*

return Mathf.Abs(b.height - a.height) \* 2f; *// Higher cost for vertical movement*

} else {

*// Horizontal distance*

return Vector2Int.Distance(a.gridPosition, b.gridPosition);

}

}

float GetHeuristic(LayeredNode a, LayeredNode b) {

*// 3D distance heuristic*

Vector3 posA = new Vector3(a.gridPosition.x, a.height, a.gridPosition.y);

Vector3 posB = new Vector3(b.gridPosition.x, b.height, b.gridPosition.y);

return Vector3.Distance(posA, posB);

}

}

**Audio System for Layers**

csharp

public class LayeredAudioSystem : MonoBehaviour {

[Header("Layer Audio Settings")]

public AnimationCurve verticalOcclusion; *// How sound dampens through floors*

public AnimationCurve horizontalOcclusion; *// How sound travels horizontally*

private AudioSource playerAudio;

private List<LayeredAudioSource> environmentSounds = new List<LayeredAudioSource>();

[System.Serializable]

public class LayeredAudioSource {

public AudioSource source;

public Vector2Int gridPosition;

public int layer;

public float baseVolume;

public bool occludeThroughFloors = true;

}

void Update() {

UpdateAudioOcclusion();

UpdateReverbZones();

}

void UpdateAudioOcclusion() {

Vector2Int playerGrid = WorldToGrid(player.position);

int playerLayer = GetPlayerLayer();

foreach (var layeredAudio in environmentSounds) {

float occlusion = CalculateOcclusion(

playerGrid, playerLayer,

layeredAudio.gridPosition, layeredAudio.layer

);

*// Apply occlusion*

layeredAudio.source.volume = layeredAudio.baseVolume \* (1f - occlusion);

*// Apply low-pass filter for muffled sound through floors*

var filter = layeredAudio.source.GetComponent<AudioLowPassFilter>();

if (filter != null) {

filter.cutoffFrequency = Mathf.Lerp(22000, 500, occlusion);

}

}

}

float CalculateOcclusion(Vector2Int listenerPos, int listenerLayer,

Vector2Int sourcePos, int sourceLayer) {

float occlusion = 0f;

*// Vertical occlusion (through floors/ceilings)*

if (listenerLayer != sourceLayer) {

int layerDiff = Mathf.Abs(sourceLayer - listenerLayer);

*// Check each layer between for blocking*

for (int l = Mathf.Min(listenerLayer, sourceLayer);

l < Mathf.Max(listenerLayer, sourceLayer); l++) {

var cell = dungeonData.cells[sourcePos.x, sourcePos.y];

var layer = cell.layers.FirstOrDefault(lay => lay.layerIndex == l);

if (layer != null) {

if (layer.blocksSound) {

occlusion += 0.7f; *// Heavy occlusion*

} else if (layer.hasHole) {

occlusion += 0.1f; *// Sound travels through hole*

} else {

occlusion += 0.4f; *// Normal floor occlusion*

}

}

}

}

*// Horizontal occlusion (walls on same layer)*

if (listenerLayer == sourceLayer) {

var path = GetLineOfSight(listenerPos, sourcePos, listenerLayer);

foreach (var wall in path.walls) {

if (wall.type == WallType.Solid) {

occlusion += 0.5f;

} else if (wall.type == WallType.Door) {

occlusion += wall.state == DoorState.Closed ? 0.3f : 0.1f;

}

}

}

return Mathf.Clamp01(occlusion);

}

void UpdateReverbZones() {

*// Adjust reverb based on current layer properties*

var currentCell = GetCurrentCell();

var currentLayer = currentCell.GetLayerAtHeight(player.position.y);

if (currentLayer != null) {

float spaceSize = currentLayer.GetClearance();

bool isTunnel = !currentLayer.allowStanding;

bool isOpenAbove = currentLayer.hasHole;

*// Apply appropriate reverb*

if (isTunnel) {

AudioReverbPreset.SetPreset(AudioReverbPreset.Cave);

} else if (isOpenAbove) {

AudioReverbPreset.SetPreset(AudioReverbPreset.Hall);

} else if (spaceSize < 2f) {

AudioReverbPreset.SetPreset(AudioReverbPreset.Closet);

} else {

AudioReverbPreset.SetPreset(AudioReverbPreset.Room);

}

}

}

}

**Lighting System for Layers**

csharp

public class LayeredLightingSystem : MonoBehaviour {

[Header("Light Settings")]

public float lightBleedThoughHoles = 0.8f;

public float lightThroughGrates = 0.6f;

public bool enableVolumetricFog = true;

private Dictionary<int, List<Light>> layerLights = new Dictionary<int, List<Light>>();

public void SetupLayerLighting(LayeredDungeonData data) {

*// Create light probes for each layer*

foreach (var layerIndex in GetAllLayers(data)) {

SetupLayerLightProbes(layerIndex);

}

*// Setup light bleeding between layers*

SetupInterLayerLighting(data);

}

void SetupInterLayerLighting(LayeredDungeonData data) {

for (int x = 0; x < data.width; x++) {

for (int y = 0; y < data.height; y++) {

var cell = data.cells[x, y];

foreach (var layer in cell.layers) {

if (layer.hasHole) {

*// Create light shaft through hole*

CreateLightShaft(x, y, layer);

}

if (!layer.blocksLight) {

*// Allow light to pass through (grates, glass)*

SetupTranslucentLighting(x, y, layer);

}

}

}

}

}

void CreateLightShaft(int x, int y, CellLayer layer) {

Vector3 position = GridToWorld(x, y);

position.y = layer.floorHeight;

*// Find light source above*

var upperLayer = GetLayerAbove(x, y, layer.layerIndex);

if (upperLayer != null) {

*// Create volumetric light shaft*

GameObject lightShaft = new GameObject("LightShaft");

lightShaft.transform.position = position;

var light = lightShaft.AddComponent<Light>();

light.type = LightType.Spot;

light.angle = layer.holeSize \* 90f; *// Size based on hole*

light.intensity = GetUpperLayerBrightness(upperLayer) \* lightBleedThoughHoles;

light.range = upperLayer.ceilingHeight - layer.floorHeight;

*// Point downward*

lightShaft.transform.rotation = Quaternion.Euler(90, 0, 0);

*// Add volumetric fog for visible shaft*

if (enableVolumetricFog) {

var volume = lightShaft.AddComponent<VolumetricLight>();

volume.intensity = 0.5f;

}

}

}

public void UpdateDynamicLighting(int playerLayer) {

*// Adjust lighting based on player layer*

foreach (var kvp in layerLights) {

int layer = kvp.Key;

var lights = kvp.Value;

foreach (var light in lights) {

if (layer == playerLayer) {

*// Full brightness on player layer*

light.intensity = light.GetComponent<LayeredLight>().baseIntensity;

} else if (Mathf.Abs(layer - playerLayer) == 1) {

*// Dim adjacent layers*

light.intensity = light.GetComponent<LayeredLight>().baseIntensity \* 0.3f;

} else {

*// Turn off distant layers for performance*

light.enabled = false;

}

}

}

}

}

**Enemy AI for Layers**

csharp

public class LayeredEnemyAI : MonoBehaviour {

[Header("Layer Awareness")]

public int currentLayer;

public bool canUseVerticalConnections = true;

public bool canHearBetweenLayers = true;

public float verticalDetectionRange = 5f;

private LayeredPathfinding pathfinding;

private Transform player;

void Update() {

UpdateLayerAwareness();

if (CanDetectPlayer()) {

PursuePlayer();

} else {

Patrol();

}

}

bool CanDetectPlayer() {

*// Check if player is on same layer*

if (player.GetComponent<LayeredMovementController>().currentLayer == currentLayer) {

return CheckLineOfSight(transform.position, player.position);

}

*// Check if can hear player on different layer*

if (canHearBetweenLayers) {

int playerLayer = player.GetComponent<LayeredMovementController>().currentLayer;

float verticalDistance = Mathf.Abs(playerLayer - currentLayer) \* averageLayerHeight;

if (verticalDistance <= verticalDetectionRange) {

*// Check for holes/connections*

if (HasVerticalConnection(transform.position, player.position)) {

return PlayerMakingNoise();

}

}

}

return false;

}

void PursuePlayer() {

var path = pathfinding.FindPath(transform.position, player.position);

if (path != null && path.Count > 0) {

var nextNode = path[0];

*// Check if need to change layers*

if (nextNode.layer != currentLayer) {

*// Use vertical connection*

UseVerticalConnection(nextNode);

} else {

*// Normal movement*

MoveToward(GridToWorld(nextNode.gridPosition));

}

}

}

void UseVerticalConnection(LayeredPathfinding.LayeredNode targetNode) {

*// Find the connection to use*

var cell = dungeonData.cells[gridPosition.x, gridPosition.y];

var currentLayerData = cell.layers.FirstOrDefault(l => l.layerIndex == currentLayer);

var connection = currentLayerData.verticalConnections

.FirstOrDefault(c => c.toLayer == targetNode.layer);

if (connection != null) {

switch (connection.type) {

case LayerConnection.ConnectionType.Ladder:

StartClimbing(connection);

break;

case LayerConnection.ConnectionType.Stairs:

UseStairs(connection);

break;

case LayerConnection.ConnectionType.Hole:

if (connection.toLayer < currentLayer) {

DropThroughHole(connection);

}

break;

}

}

}

}

This layered system creates incredibly rich gameplay possibilities:

1. **Vertical Exploration** - Discover tunnels beneath dungeons
2. **Strategic Combat** - Drop on enemies from above, take cover below
3. **Environmental Puzzles** - Drain water between layers, use vertical connections
4. **Stealth Gameplay** - Hide in crawlspaces, listen for enemies above
5. **Atmospheric Design** - Light shafts through holes, echoing sounds between layers
6. **Performance Optimization** - Only render/process relevant layers

The key is that it builds naturally on your existing grid system while adding a whole new dimension of gameplay!

Retry

**J**

Without writing code, this layer system is still comparable with the advanced walls with composition, right?

Edit

Absolutely yes! The layer system and the composition-based wall system are not only compatible, they actually complement each other perfectly. Here's why:

**Perfect Compatibility**

**Walls exist at layer boundaries:** Each wall can span multiple layers or exist within a single layer, and the composition system handles this beautifully:

* A **solid wall** might extend from Layer 0's floor to Layer 1's ceiling
* A **half-wall** only exists in Layer 0, allowing visibility to Layer 1
* A **grate floor** in Layer 1 is also a "ceiling wall" for Layer 0 with special properties

**Behaviors adapt to layers:**

* A **secret door** on Layer 0 might reveal stairs to Layer -1 (basement)
* An **illusory wall** could hide a pit that drops through multiple layers
* A **one-way wall** might only work from the upper layer (drop down but can't climb up)

**Enhanced Possibilities**

The composition system actually becomes MORE powerful with layers:

**Vertical Behaviors:**

* IPassage can check which layer you're coming from/going to
* IVisual can render differently when viewed from above vs below
* ITrigger can activate based on weight from the layer above
* IState can track different states per layer (door locked on Layer 0, open on Layer 1)

**New Composite Behaviors:**

* **Collapsing Floor:** Wall that's a floor on one layer, becomes a hole when triggered
* **Elevator Wall:** Wall that moves between layers, carrying players
* **Flooding Behavior:** Wall that holds back water on upper layer until breached
* **Sound Pipe:** Wall that transmits sound between specific layers

**Data Structure Harmony**

The wall's composition list simply gets a layer context:

* Wall exists at position (x,y) between layers 0 and 1
* Has FloorBehavior for Layer 1 (walkable)
* Has CeilingBehavior for Layer 0 (visible from below)
* Has GrateBehavior (can see/hear through)
* Has WeightTrigger (breaks if too much weight from above)

**Editor Integration**

The 2D editor can show walls with layer information:

* Different line styles for walls that span multiple layers
* Icons showing special vertical behaviors
* Color coding for which layers a wall affects

**Runtime Flexibility**

The composition system means you can dynamically modify walls between layers:

* Blow a hole in the floor (remove FloorBehavior, add HoleBehavior)
* Freeze water to create an ice bridge (add WalkableSurface behavior)
* Activate ancient machinery (walls reconfigure between layers)

The layer system essentially gives your walls a vertical dimension to work with, while the composition system defines what they DO in that dimension. They're made for each other!

Retry

**J**

What about a dynamic camera system for the battle system, such that the camera is over the shoulder but allows for fancy zooms and camera tilts that changes depending on the ui menu selected (attack, item, targeting an enemy) that can point at enemies dynamically as well.

Edit

Absolutely! A dynamic camera system would add tremendous polish and game feel to the turn-based battles. Here's a comprehensive design for such a system:

**Core Camera System Architecture**

csharp

public class BattleCameraController : MonoBehaviour {

[Header("Camera Modes")]

public CameraMode currentMode = CameraMode.OverShoulder;

public CameraProfile defaultProfile;

public Dictionary<CameraMode, CameraProfile> cameraProfiles;

[Header("Current State")]

private Unit focusUnit; *// Current acting unit*

private Unit targetUnit; *// Current target being considered*

private Vector3 pointOfInterest; *// Environmental focus point*

private float currentFOV;

private float currentDutch; *// Camera tilt/roll*

public enum CameraMode {

OverShoulder, *// Default combat view*

Targeting, *// Aiming at enemy*

AbilitySelect, *// Choosing abilities*

ItemBrowse, *// Inventory view*

AreaTargeting, *// AoE ability aiming*

Cinematic, *// Special moves*

TacticalOverview, *// Pull back to see battlefield*

CloseUp, *// Dramatic moments*

MultiTarget, *// Showing multiple enemies*

Counter, *// Enemy's perspective during counter*

Critical *// Critical hit cinematics*

}

[System.Serializable]

public class CameraProfile {

*// Position*

public Vector3 offsetFromUnit;

public float height;

public float distance;

*// Rotation*

public float pitch; *// Look up/down*

public float yawOffset; *// Horizontal offset*

public float dutch; *// Camera roll/tilt*

*// Lens settings*

public float fieldOfView = 60f;

public float nearClipPlane = 0.3f;

*// Behavior*

public float transitionSpeed = 2f;

public AnimationCurve transitionCurve;

public bool trackTarget = true;

public bool useDepthOfField = false;

public float focusDistance;

*// Shake/Movement*

public float idleSway = 0f; *// Subtle movement*

public float breathingAmount = 0f;

}

}

**Menu-Specific Camera Behaviors**

csharp

public class MenuCameraDirector : MonoBehaviour {

private BattleCameraController cameraController;

private UIMenuSystem menuSystem;

[Header("Menu-Specific Settings")]

public MenuCameraSettings attackMenuSettings;

public MenuCameraSettings itemMenuSettings;

public MenuCameraSettings targetSelectSettings;

[System.Serializable]

public class MenuCameraSettings {

public CameraMode cameraMode;

public float zoomLevel;

public Vector3 menuOffset; *// Offset camera to frame UI better*

public bool blurBackground;

public float dutchAngle; *// Tilt for style*

public bool focusOnOptions; *// Pan between menu options*

}

void OnMenuChanged(MenuType newMenu) {

switch (newMenu) {

case MenuType.Attack:

TransitionToAttackView();

break;

case MenuType.Target:

TransitionToTargetingView();

break;

case MenuType.Item:

TransitionToItemView();

break;

case MenuType.Magic:

TransitionToMagicView();

break;

}

}

void TransitionToAttackView() {

var settings = attackMenuSettings;

*// Slightly pull back to show character and menu*

var profile = new CameraProfile {

offsetFromUnit = new Vector3(2f, 1.5f, -3f),

height = 1.8f,

distance = 4f,

pitch = 15f,

yawOffset = -25f, *// Angle to show character's weapon*

dutch = 5f, *// Slight tilt for dynamism*

fieldOfView = 55f,

useDepthOfField = true,

focusDistance = 3f

};

cameraController.TransitionToProfile(profile, 0.3f);

*// Subtle idle animation*

StartCoroutine(AttackMenuIdleAnimation());

}

void TransitionToTargetingView() {

*// Dynamic based on number of targets*

List<Unit> possibleTargets = GetSelectableTargets();

if (possibleTargets.Count == 1) {

*// Single target - dramatic zoom*

FocusOnSingleTarget(possibleTargets[0]);

} else if (possibleTargets.Count <= 3) {

*// Few targets - frame them all*

FrameMultipleTargets(possibleTargets);

} else {

*// Many targets - tactical overview*

ShowTacticalView();

}

}

void FocusOnSingleTarget(Unit target) {

*// Calculate position between attacker and target*

Vector3 midpoint = (focusUnit.position + target.position) / 2f;

Vector3 perpendicular = Vector3.Cross(

(target.position - focusUnit.position).normalized,

Vector3.up

);

var profile = new CameraProfile {

*// Position camera to the side for dramatic angle*

offsetFromUnit = perpendicular \* 3f + Vector3.up \* 2f,

pitch = 20f,

fieldOfView = 45f, *// Zoom in*

dutch = -10f, *// Dutch angle for tension*

useDepthOfField = true,

focusDistance = Vector3.Distance(midpoint, Camera.main.transform.position)

};

*// Look at midpoint between units*

cameraController.SetPointOfInterest(midpoint);

cameraController.TransitionToProfile(profile, 0.5f);

}

}

**Dynamic Target Tracking**

csharp

public class TargetingCameraSystem : MonoBehaviour {

[Header("Targeting Behavior")]

public float targetSwitchSpeed = 0.3f;

public bool showTargetLine = true;

public bool pulseOnLock = true;

private Unit currentHighlight;

private Queue<Unit> targetQueue;

private Coroutine trackingCoroutine;

public void OnTargetHighlightChanged(Unit newTarget) {

if (trackingCoroutine != null) {

StopCoroutine(trackingCoroutine);

}

trackingCoroutine = StartCoroutine(SmoothTrackToTarget(newTarget));

}

IEnumerator SmoothTrackToTarget(Unit target) {

float elapsed = 0;

*// Store initial camera state*

Vector3 startPos = Camera.main.transform.position;

Quaternion startRot = Camera.main.transform.rotation;

float startFOV = Camera.main.fieldOfView;

*// Calculate target camera state*

var targetProfile = CalculateTargetingProfile(focusUnit, target);

Vector3 endPos = CalculatePosition(targetProfile);

Quaternion endRot = CalculateRotation(targetProfile, target.position);

*// Smooth transition*

while (elapsed < targetSwitchSpeed) {

elapsed += Time.deltaTime;

float t = targetingCurve.Evaluate(elapsed / targetSwitchSpeed);

Camera.main.transform.position = Vector3.Lerp(startPos, endPos, t);

Camera.main.transform.rotation = Quaternion.Slerp(startRot, endRot, t);

*// Slight zoom when hovering on target*

float targetFOV = target.isBoss ? 40f : 50f;

Camera.main.fieldOfView = Mathf.Lerp(startFOV, targetFOV, t);

yield return null;

}

*// Pulse effect when locked*

if (pulseOnLock) {

yield return PulseFOV(2f, 0.1f);

}

*// Continue tracking while selected*

while (currentHighlight == target) {

TrackMovingTarget(target);

yield return null;

}

}

CameraProfile CalculateTargetingProfile(Unit attacker, Unit target) {

float distance = Vector3.Distance(attacker.position, target.position);

*// Adjust camera based on distance*

if (distance < 5f) {

*// Close combat - intimate camera*

return new CameraProfile {

offsetFromUnit = new Vector3(1.5f, 1.8f, -2f),

pitch = 25f,

yawOffset = -35f,

fieldOfView = 50f,

dutch = 8f

};

} else if (distance < 15f) {

*// Mid range - balanced view*

return new CameraProfile {

offsetFromUnit = new Vector3(3f, 2.5f, -5f),

pitch = 30f,

yawOffset = -20f,

fieldOfView = 55f,

dutch = 5f

};

} else {

*// Long range - wide view*

return new CameraProfile {

offsetFromUnit = new Vector3(5f, 4f, -8f),

pitch = 35f,

yawOffset = -15f,

fieldOfView = 60f,

dutch = 0f

};

}

}

}

**Cinematic Action Cameras**

csharp

public class CinematicBattleCamera : MonoBehaviour {

[Header("Action Cameras")]

public Dictionary<AbilityType, CinematicSequence> cinematicSequences;

[System.Serializable]

public class CinematicSequence {

public List<CinematicShot> shots;

public float totalDuration;

public bool matchActionTiming; *// Sync with animation*

}

[System.Serializable]

public class CinematicShot {

public ShotType type;

public float duration;

public AnimationCurve intensityCurve;

*// Position*

public Vector3 positionOffset;

public bool orbitAroundTarget;

public float orbitSpeed;

*// Movement*

public Vector3 dollyDirection; *// Camera movement*

public float dollySpeed;

public bool handheldShake;

public float shakeIntensity;

*// Focus*

public FocusTarget focusTarget;

public bool rackFocus; *// Shift focus during shot*

public float depthOfFieldAperture;

*// Effects*

public float motionBlur;

public float chromaticAberration;

public float dutchAngle;

public float zoomSpeed;

}

public enum ShotType {

EstablishingShot, *// Wide shot showing battlefield*

OverShoulder, *// Classic OTS*

LowAngle, *// Hero shot from below*

HighAngle, *// Vulnerable shot from above*

DutchAngle, *// Tilted for unease*

WhipPan, *// Fast pan between subjects*

PushIn, *// Zoom toward subject*

PullOut, *// Zoom away*

Orbit, *// Circle around action*

HandHeld, *// Shaky documentary style*

POV, *// First person view*

Crash, *// Fast zoom with shake*

Tracking, *// Follow movement*

Static *// Fixed position*

}

public void PlayAbilitySequence(Ability ability, Unit caster, List<Unit> targets) {

if (!cinematicSequences.ContainsKey(ability.type)) {

*// Use default sequence*

PlayDefaultSequence(ability, caster, targets);

return;

}

var sequence = cinematicSequences[ability.type];

StartCoroutine(ExecuteCinematicSequence(sequence, caster, targets));

}

IEnumerator ExecuteCinematicSequence(CinematicSequence sequence, Unit caster, List<Unit> targets) {

*// Store original camera state*

var originalState = CaptureCurrentCameraState();

foreach (var shot in sequence.shots) {

yield return ExecuteShot(shot, caster, targets);

}

*// Return to gameplay camera*

yield return RestoreCameraState(originalState, 0.5f);

}

IEnumerator ExecuteShot(CinematicShot shot, Unit caster, List<Unit> targets) {

float elapsed = 0;

Vector3 startPos = Camera.main.transform.position;

while (elapsed < shot.duration) {

float t = elapsed / shot.duration;

float intensity = shot.intensityCurve.Evaluate(t);

switch (shot.type) {

case ShotType.Orbit:

OrbitAroundPoint(GetFocusPoint(shot, caster, targets),

shot.orbitSpeed \* Time.deltaTime);

break;

case ShotType.WhipPan:

if (t < 0.3f) {

*// Fast pan from caster to target*

WhipPanBetween(caster.position, targets[0].position, t / 0.3f);

}

break;

case ShotType.Crash:

*// Rapid zoom with increasing shake*

CrashZoom(targets[0].position, intensity);

ApplyCameraShake(intensity \* shot.shakeIntensity);

break;

case ShotType.LowAngle:

*// Hero shot*

PositionLowAngle(caster, intensity);

break;

}

*// Apply shot effects*

if (shot.handheldShake) {

ApplyHandheldMotion(shot.shakeIntensity \* intensity);

}

if (shot.dutchAngle != 0) {

ApplyDutchAngle(shot.dutchAngle \* intensity);

}

if (shot.rackFocus) {

RackFocusBetween(caster, targets[0], t);

}

elapsed += Time.deltaTime;

yield return null;

}

}

}

**Smooth Transitions and Blending**

csharp

public class CameraTransitionSystem : MonoBehaviour {

[Header("Transition Settings")]

public AnimationCurve defaultTransitionCurve;

public float minimumTransitionTime = 0.2f;

public float maximumTransitionTime = 2f;

private CameraState currentState;

private CameraState targetState;

private float transitionProgress;

[System.Serializable]

public struct CameraState {

public Vector3 position;

public Quaternion rotation;

public float fieldOfView;

public float dutchAngle;

public float depthOfFieldDistance;

public float depthOfFieldAperture;

public static CameraState Lerp(CameraState a, CameraState b, float t) {

return new CameraState {

position = Vector3.Lerp(a.position, b.position, t),

rotation = Quaternion.Slerp(a.rotation, b.rotation, t),

fieldOfView = Mathf.Lerp(a.fieldOfView, b.fieldOfView, t),

dutchAngle = Mathf.Lerp(a.dutchAngle, b.dutchAngle, t),

depthOfFieldDistance = Mathf.Lerp(a.depthOfFieldDistance, b.depthOfFieldDistance, t),

depthOfFieldAperture = Mathf.Lerp(a.depthOfFieldAperture, b.depthOfFieldAperture, t)

};

}

}

public void BlendToState(CameraState newState, float duration, AnimationCurve curve = null) {

if (curve == null) curve = defaultTransitionCurve;

StartCoroutine(TransitionCoroutine(newState, duration, curve));

}

IEnumerator TransitionCoroutine(CameraState newState, float duration, AnimationCurve curve) {

float elapsed = 0;

CameraState startState = GetCurrentCameraState();

*// Calculate easing based on distance*

float positionDelta = Vector3.Distance(startState.position, newState.position);

float rotationDelta = Quaternion.Angle(startState.rotation, newState.rotation);

*// Adjust duration based on change magnitude*

float adjustedDuration = Mathf.Lerp(minimumTransitionTime, duration,

(positionDelta + rotationDelta) / 100f);

while (elapsed < adjustedDuration) {

elapsed += Time.deltaTime;

float t = curve.Evaluate(elapsed / adjustedDuration);

currentState = CameraState.Lerp(startState, newState, t);

ApplyCameraState(currentState);

yield return null;

}

currentState = newState;

ApplyCameraState(currentState);

}

void ApplyCameraState(CameraState state) {

Camera.main.transform.position = state.position;

Camera.main.transform.rotation = state.rotation;

Camera.main.fieldOfView = state.fieldOfView;

*// Apply dutch angle (camera roll)*

Camera.main.transform.Rotate(0, 0, state.dutchAngle);

*// Apply post-processing effects*

if (postProcessVolume != null) {

DepthOfField dof;

if (postProcessVolume.profile.TryGet(out dof)) {

dof.focusDistance.value = state.depthOfFieldDistance;

dof.aperture.value = state.depthOfFieldAperture;

}

}

}

}

**Smart Framing System**

csharp

public class SmartFramingSystem : MonoBehaviour {

[Header("Framing Rules")]

public float ruleOfThirdsStrength = 0.8f;

public float headroomPercentage = 0.15f;

public float leadRoomMultiplier = 1.5f;

public Vector3 CalculateFramingPosition(Unit subject, Unit target = null) {

Bounds subjectBounds = GetUnitBounds(subject);

*// Rule of thirds positioning*

Vector3 framePosition = ApplyRuleOfThirds(subjectBounds);

*// Headroom adjustment*

framePosition.y += CalculateHeadroom(subject);

*// Lead room if targeting*

if (target != null) {

Vector3 direction = (target.position - subject.position).normalized;

framePosition += direction \* leadRoomMultiplier;

}

*// Ensure nothing blocks the shot*

framePosition = AvoidObstructions(subject.position, framePosition);

return framePosition;

}

Vector3 ApplyRuleOfThirds(Bounds bounds) {

*// Position subject at 1/3 or 2/3 of frame*

float horizontalOffset = Camera.main.aspect \* ruleOfThirdsStrength;

float verticalOffset = ruleOfThirdsStrength;

return bounds.center + new Vector3(horizontalOffset, verticalOffset, 0);

}

Vector3 AvoidObstructions(Vector3 subject, Vector3 cameraPos) {

RaycastHit hit;

if (Physics.Linecast(subject, cameraPos, out hit, obstructionMask)) {

*// Adjust camera position to avoid obstruction*

Vector3 adjustment = Vector3.Cross(hit.normal, Vector3.up) \* 2f;

return AvoidObstructions(subject, cameraPos + adjustment);

}

return cameraPos;

}

}

This system creates a cinematic, dynamic camera that:

1. **Responds to UI context** - Different angles for different menus
2. **Frames action dramatically** - Low angles for power, high for vulnerability
3. **Smoothly tracks targets** - No jarring movements
4. **Creates tension** - Dutch angles, depth of field, slow zooms
5. **Handles groups** - Intelligently frames multiple enemies
6. **Adds production value** - Crash zooms, whip pans, orbit shots
7. **Maintains clarity** - Never obscures important information
8. **Enhances abilities** - Special cameras for ultimate moves

The key is making the camera feel like an invisible cinematographer who knows exactly where to be for maximum drama!

Retry

**J**

Instead of having the walls effect floors, can the floors have their own behaviour and composition instead?

Edit

Absolutely! That's actually a much cleaner design. Floors should definitely be their own entities with their own behaviors and composition. This makes way more sense architecturally and opens up tons of possibilities:

**Floor as Independent Entities**

csharp

[System.Serializable]

public class FloorTile {

public Vector2Int gridPosition;

public int layer;

public float height;

*// Composable behaviors*

public List<IFloorBehavior> behaviors = new List<IFloorBehavior>();

public List<IFloorVisual> visuals = new List<IFloorVisual>();

public List<IFloorTrigger> triggers = new List<IFloorTrigger>();

public List<IFloorState> states = new List<IFloorState>();

}

*// Floor-specific interfaces*

public interface IFloorBehavior {

bool CanWalkOn(Entity entity);

void OnStep(Entity entity);

void OnStand(Entity entity, float deltaTime);

void OnLeave(Entity entity);

float GetMovementSpeedModifier();

float GetFriction();

}

public interface IFloorVisual {

Material GetMaterial(FloorRenderContext context);

void RenderEffects(Vector3 position);

float GetReflectivity();

bool IsTransparent();

}

public interface IFloorTrigger {

bool CheckTrigger(Entity entity, float weight);

void OnTrigger(FloorTile floor);

void OnReset();

}

public interface IFloorState {

FloorStateType GetCurrentState();

void UpdateState(float deltaTime);

bool CanTransitionTo(FloorStateType newState);

}

**Floor Behavior Examples**

csharp

*// Ice floor*

public class IceFloorBehavior : IFloorBehavior {

public float slipperiness = 0.9f;

public float slideSpeed = 2f;

public bool CanWalkOn(Entity entity) {

return true; *// But they'll slide!*

}

public void OnStep(Entity entity) {

*// Continue momentum in current direction*

entity.AddSlideForce(entity.velocity.normalized \* slideSpeed);

}

public float GetFriction() => 1f - slipperiness;

}

*// Pressure plate*

public class PressurePlateBehavior : IFloorBehavior, IFloorTrigger {

public float requiredWeight = 50f;

public string triggerId;

private bool isPressed = false;

public void OnStep(Entity entity) {

float totalWeight = CalculateTotalWeight();

if (totalWeight >= requiredWeight && !isPressed) {

isPressed = true;

TriggerConnectedMechanism(triggerId);

}

}

public void OnLeave(Entity entity) {

float totalWeight = CalculateTotalWeight();

if (totalWeight < requiredWeight && isPressed) {

isPressed = false;

ResetConnectedMechanism(triggerId);

}

}

}

*// Crumbling floor*

public class CrumblingFloorBehavior : IFloorBehavior, IFloorState {

public float stability = 100f;

public float degradeRate = 20f;

public float regenRate = 5f;

private float currentStability;

private bool isCrumbling = false;

public void OnStand(Entity entity, float deltaTime) {

currentStability -= degradeRate \* deltaTime \* entity.weight;

if (currentStability <= 0 && !isCrumbling) {

StartCrumbling();

}

}

void StartCrumbling() {

isCrumbling = true;

*// Floor becomes non-walkable after delay*

StartCoroutine(CollapseAfterDelay(1f));

}

}

*// Teleportation pad*

public class TeleportFloorBehavior : IFloorBehavior {

public Vector2Int destination;

public int destinationLayer;

public float chargeTime = 1f;

private Dictionary<Entity, float> standingTime = new Dictionary<Entity, float>();

public void OnStand(Entity entity, float deltaTime) {

if (!standingTime.ContainsKey(entity)) {

standingTime[entity] = 0;

}

standingTime[entity] += deltaTime;

if (standingTime[entity] >= chargeTime) {

TeleportEntity(entity);

standingTime[entity] = 0;

}

}

}

*// Conveyor belt*

public class ConveyorFloorBehavior : IFloorBehavior {

public Vector2 direction;

public float speed = 2f;

public bool canResist = true;

public void OnStand(Entity entity, float deltaTime) {

Vector3 push = new Vector3(direction.x, 0, direction.y) \* speed \* deltaTime;

if (canResist) {

*// Player can move against it but slower*

entity.AddMovementModifier(push);

} else {

*// Forced movement*

entity.transform.position += push;

}

}

}

**Complex Floor Compositions**

csharp

public class FloorFactory {

public FloorTile CreateLavaFloor() {

var floor = new FloorTile();

*// Damages over time*

floor.behaviors.Add(new DamageFloorBehavior {

damagePerSecond = 10f,

damageType = DamageType.Fire

});

*// Glows and bubbles*

floor.visuals.Add(new LavaVisual {

glowIntensity = 2f,

bubbleRate = 5f

});

*// Lights entities on fire*

floor.behaviors.Add(new IgniteBehavior {

burnDuration = 5f

});

*// Can temporarily cool down*

floor.states.Add(new TemperatureState {

currentTemp = 1000f,

coolingRate = 50f

});

return floor;

}

public FloorTile CreatePuzzleFloor() {

var floor = new FloorTile();

*// Changes color based on pattern*

floor.states.Add(new ColorPatternState {

colors = new[] { Color.red, Color.blue, Color.green },

switchInterval = 2f

});

*// Only walkable when correct color*

floor.behaviors.Add(new ConditionalWalkable {

condition = () => floor.GetState<ColorPatternState>().currentColor == Color.green

});

*// Visual feedback*

floor.visuals.Add(new PulsingVisual {

pulseSpeed = 1f

});

return floor;

}

public FloorTile CreateMimicFloor() {

var floor = new FloorTile();

*// Looks like normal floor*

floor.visuals.Add(new StandardFloorVisual());

*// But it's actually a monster!*

floor.triggers.Add(new ProximityTrigger {

range = 0.5f,

onTrigger = () => TransformIntoMonster(floor)

});

return floor;

}

}

**Floor-Specific Visual Effects**

csharp

public class FloorVisualSystem : MonoBehaviour {

*// Water that reacts to movement*

public class WaterFloorVisual : IFloorVisual {

public float rippleStrength = 1f;

public float waveHeight = 0.2f;

private ComputeShader rippleShader;

public void OnEntityMove(Vector3 position, Vector3 velocity) {

*// Create ripples at position*

CreateRipple(position, velocity.magnitude \* rippleStrength);

}

public Material GetMaterial(FloorRenderContext context) {

var mat = new Material(waterShader);

mat.SetFloat("\_WaveHeight", waveHeight);

mat.SetTexture("\_RippleMap", GetRippleRenderTexture());

return mat;

}

}

*// Glass floor showing layer below*

public class GlassFloorVisual : IFloorVisual {

public float transparency = 0.8f;

public float crackThreshold = 100f;

private float totalDamage = 0f;

public void OnImpact(float force) {

totalDamage += force;

if (totalDamage > crackThreshold) {

AddCrackDecal();

}

}

public bool IsTransparent() => true;

}

}

**Floor State Machines**

csharp

public class FloorStateMachine {

*// Trap floor with multiple states*

public class TrapFloorState : IFloorState {

public enum State {

Safe,

Armed,

Triggered,

Cooldown

}

private State currentState = State.Safe;

private float stateTimer = 0f;

public void UpdateState(float deltaTime) {

stateTimer += deltaTime;

switch (currentState) {

case State.Safe:

if (stateTimer > 5f) {

TransitionTo(State.Armed);

}

break;

case State.Armed:

*// Waiting for victim*

break;

case State.Triggered:

*// Spikes are up*

if (stateTimer > 2f) {

TransitionTo(State.Cooldown);

}

break;

case State.Cooldown:

if (stateTimer > 3f) {

TransitionTo(State.Safe);

}

break;

}

}

}

}

**Environmental Floor Interactions**

csharp

public class EnvironmentalFloorBehavior : IFloorBehavior {

*// Floors that interact with environment*

public class PlantGrowthFloor : IFloorBehavior {

public float growthRate = 0.1f;

public GameObject plantPrefab;

private List<GameObject> plants = new List<GameObject>();

public void OnStand(Entity entity, float deltaTime) {

if (entity.HasComponent<NatureAffinity>()) {

GrowPlants(deltaTime \* growthRate \* 2f);

}

}

void GrowPlants(float amount) {

foreach (var plant in plants) {

plant.transform.localScale += Vector3.one \* amount;

}

}

}

*// Floors affected by abilities*

public class ElementalFloor : IFloorBehavior, IFloorState {

public ElementType currentElement = ElementType.Neutral;

public void OnAbilityHit(Ability ability) {

if (ability.element == ElementType.Fire && currentElement == ElementType.Ice) {

*// Melt ice floor into water*

TransformToWaterFloor();

} else if (ability.element == ElementType.Ice && currentElement == ElementType.Water) {

*// Freeze water floor*

TransformToIceFloor();

} else if (ability.element == ElementType.Electric && currentElement == ElementType.Water) {

*// Electrify water*

AddBehavior(new ElectricFloorBehavior());

}

}

}

}

**Benefits of Separate Floor System**

**1. Clean Separation of Concerns**

* Walls handle vertical boundaries
* Floors handle horizontal surfaces
* No confusing overlap of responsibilities

**2. Rich Gameplay Possibilities**

* Puzzle floors that need specific sequences
* Environmental hazards (lava, poison swamps)
* Movement modifiers (ice, mud, conveyor belts)
* Interactive floors (switches, teleporters)

**3. Visual Variety**

* Different materials per tile
* Dynamic effects (ripples, cracks, wear)
* Transparency for seeing through layers
* Procedural damage and weathering

**4. Performance Benefits**

* Can batch similar floor types
* LOD system specifically for floors
* Culling based on layer visibility

**5. Designer Friendly**

* Paint floor types in editor
* Mix and match behaviors
* Test floor puzzles independently
* Clear visual feedback

This separation makes the system much more maintainable and allows floors to be as complex and interesting as walls, creating richer dungeon experiences where the ground itself is part of the gameplay!

Retry

**J**

can this be tailored more towards a grid based step system instead of having smooth movement?

Edit

Absolutely! A grid-based step system actually makes floors even MORE important since each tile becomes a discrete gameplay decision. Here's how to tailor it:

**Grid-Based Floor System**

csharp

[System.Serializable]

public class GridFloorTile {

public Vector2Int gridPosition;

public int layer;

*// Grid-specific behaviors*

public List<IGridFloorBehavior> behaviors = new List<IGridFloorBehavior>();

public List<IGridFloorTrigger> triggers = new List<IGridFloorTrigger>();

public List<IGridFloorState> states = new List<IGridFloorState>();

*// Occupancy*

public bool isOccupied = false;

public Entity currentOccupant = null;

public bool blocksMovement = false;

public bool blocksVision = false;

}

*// Grid-specific interfaces*

public interface IGridFloorBehavior {

bool CanEnter(Entity entity, Direction fromDirection);

void OnEnter(Entity entity, Direction fromDirection);

void OnStay(Entity entity); *// Called each turn while standing*

void OnExit(Entity entity, Direction toDirection);

int GetMovementCost(); *// How many action points to cross*

bool ForceExit(Direction direction); *// For conveyor belts, etc.*

}

public interface IGridFloorTrigger {

bool CheckTrigger(GridFloorTile tile);

void OnTrigger(GridFloorTile tile);

int GetTriggerDelay(); *// Turns before activation*

}

**Turn-Based Floor Behaviors**

csharp

*// Ice floor - slide until hitting wall or non-ice tile*

public class GridIceFloorBehavior : IGridFloorBehavior {

public bool CanEnter(Entity entity, Direction fromDirection) {

return true;

}

public void OnEnter(Entity entity, Direction fromDirection) {

*// Continue sliding in same direction*

StartCoroutine(SlideEntity(entity, fromDirection));

}

IEnumerator SlideEntity(Entity entity, Direction direction) {

Vector2Int currentPos = entity.gridPosition;

Vector2Int nextPos = currentPos + DirectionToVector(direction);

while (CanSlideToPosition(nextPos)) {

*// Animate slide to next position*

yield return MoveEntityToPosition(entity, nextPos);

currentPos = nextPos;

nextPos = currentPos + DirectionToVector(direction);

*// Check if next tile is also ice*

var nextTile = GetTileAt(nextPos);

if (nextTile == null || !nextTile.HasBehavior<GridIceFloorBehavior>()) {

break; *// Stop at non-ice tiles*

}

}

entity.gridPosition = currentPos;

entity.OnMovementComplete();

}

}

*// Rotating floor - changes facing direction*

public class GridRotatingFloorBehavior : IGridFloorBehavior {

public RotationType rotationType = RotationType.Clockwise;

public int rotationDegrees = 90;

public void OnEnter(Entity entity, Direction fromDirection) {

Direction newFacing = RotateDirection(entity.facing, rotationType, rotationDegrees);

entity.SetFacing(newFacing);

*// May affect next movement*

if (entity.HasQueuedMovement()) {

entity.UpdateQueuedDirection(newFacing);

}

}

}

*// Teleporter with charging*

public class GridTeleporterBehavior : IGridFloorBehavior, IGridFloorState {

public Vector2Int destination;

public int chargeTimeInTurns = 3;

private Dictionary<Entity, int> chargeProgress = new Dictionary<Entity, int>();

public void OnStay(Entity entity) {

if (!chargeProgress.ContainsKey(entity)) {

chargeProgress[entity] = 0;

}

chargeProgress[entity]++;

*// Visual feedback*

ShowChargeProgress(entity, chargeProgress[entity], chargeTimeInTurns);

if (chargeProgress[entity] >= chargeTimeInTurns) {

TeleportEntity(entity, destination);

chargeProgress.Remove(entity);

}

}

public void OnExit(Entity entity, Direction toDirection) {

*// Reset charge if entity moves away*

if (chargeProgress.ContainsKey(entity)) {

chargeProgress.Remove(entity);

HideChargeProgress(entity);

}

}

}

*// Crumbling floor with turn countdown*

public class GridCrumblingFloorBehavior : IGridFloorBehavior, IGridFloorState {

public int turnsUntilCollapse = 2;

private int turnsRemaining;

private bool isTriggered = false;

public void OnEnter(Entity entity, Direction fromDirection) {

if (!isTriggered) {

isTriggered = true;

turnsRemaining = turnsUntilCollapse;

ShowCrumblingWarning(turnsRemaining);

}

}

public void OnTurnEnd() {

if (isTriggered) {

turnsRemaining--;

UpdateCrumblingVisual(turnsRemaining);

if (turnsRemaining <= 0) {

CollapseFloor();

}

}

}

void CollapseFloor() {

*// Floor becomes a pit*

tile.behaviors.Clear();

tile.behaviors.Add(new GridPitBehavior());

tile.blocksMovement = true;

*// Drop any entity still on it*

if (tile.currentOccupant != null) {

tile.currentOccupant.FallInPit();

}

}

}

**Pattern-Based Floor Mechanics**

csharp

*// Dance floor puzzle - must step in pattern*

public class GridPatternFloorBehavior : IGridFloorBehavior {

public Color[] colorPattern = { Color.red, Color.blue, Color.green };

public int currentPatternIndex = 0;

private List<Vector2Int> patternSequence = new List<Vector2Int>();

public bool CanEnter(Entity entity, Direction fromDirection) {

*// Can only enter if it's the next in sequence*

Color expectedColor = colorPattern[currentPatternIndex];

return GetCurrentColor() == expectedColor;

}

public void OnEnter(Entity entity, Direction fromDirection) {

patternSequence.Add(gridPosition);

currentPatternIndex++;

if (currentPatternIndex >= colorPattern.Length) {

*// Pattern complete!*

OnPatternComplete();

currentPatternIndex = 0;

}

*// Update visual to show progress*

HighlightTile(GetNextExpectedTile());

}

}

*// Chess-based movement floor*

public class GridChessFloorBehavior : IGridFloorBehavior {

public ChessPiece restrictedTo = ChessPiece.Knight;

public bool CanEnter(Entity entity, Direction fromDirection) {

Vector2Int from = entity.gridPosition;

Vector2Int to = this.gridPosition;

switch (restrictedTo) {

case ChessPiece.Knight:

*// L-shaped movement only*

int dx = Mathf.Abs(to.x - from.x);

int dy = Mathf.Abs(to.y - from.y);

return (dx == 2 && dy == 1) || (dx == 1 && dy == 2);

case ChessPiece.Bishop:

*// Diagonal only*

return Mathf.Abs(to.x - from.x) == Mathf.Abs(to.y - from.y);

case ChessPiece.Rook:

*// Straight lines only*

return to.x == from.x || to.y == from.y;

}

return false;

}

}

**Turn-Based Environmental Effects**

csharp

*// Spreading hazards*

public class GridSpreadingFireBehavior : IGridFloorBehavior, IGridFloorState {

public int spreadTurns = 3;

public float spreadChance = 0.5f;

private int burnTurnsRemaining = 5;

public void OnTurnEnd() {

burnTurnsRemaining--;

if (burnTurnsRemaining <= 0) {

*// Fire burns out*

RemoveBehavior(this);

return;

}

*// Try to spread to adjacent tiles*

if (CurrentTurn % spreadTurns == 0) {

foreach (var adjacent in GetAdjacentTiles()) {

if (Random.value < spreadChance && !adjacent.HasBehavior<GridSpreadingFireBehavior>()) {

adjacent.AddBehavior(new GridSpreadingFireBehavior());

}

}

}

}

public void OnStay(Entity entity) {

entity.TakeDamage(10, DamageType.Fire);

}

}

*// Rhythm-based floors*

public class GridRhythmFloorBehavior : IGridFloorBehavior {

public int safeEveryNTurns = 3;

public int offset = 0;

public bool CanEnter(Entity entity, Direction fromDirection) {

int currentTurn = TurnManager.Instance.currentTurn;

bool isSafe = (currentTurn + offset) % safeEveryNTurns == 0;

if (!isSafe) {

*// Show spikes or hazard*

ShowHazard();

return false; *// Can't enter on dangerous turns*

}

return true;

}

public void OnEnter(Entity entity, Direction fromDirection) {

*// Might become dangerous next turn*

int nextTurn = TurnManager.Instance.currentTurn + 1;

bool nextTurnSafe = (nextTurn + offset) % safeEveryNTurns == 0;

if (!nextTurnSafe) {

ShowWarning(); *// Alert player they need to move*

}

}

}

**Movement Cost System**

csharp

public class GridMovementCostSystem {

*// Different floors cost different amounts to traverse*

public class DifficultTerrainBehavior : IGridFloorBehavior {

public int movementCost = 2; *// Normal is 1*

public TerrainType terrainType = TerrainType.Mud;

public int GetMovementCost() {

return movementCost;

}

public bool CanEnter(Entity entity, Direction fromDirection) {

*// Check if entity has enough movement points*

return entity.currentMovementPoints >= movementCost;

}

public void OnEnter(Entity entity, Direction fromDirection) {

entity.currentMovementPoints -= movementCost;

*// Special effects based on terrain*

switch (terrainType) {

case TerrainType.Mud:

entity.ApplyStatus(StatusEffect.Slowed, 1);

break;

case TerrainType.Vines:

if (Random.value < 0.3f) {

entity.ApplyStatus(StatusEffect.Entangled, 1);

}

break;

}

}

}

}

**Grid-Based Floor Triggers**

csharp

*// Switch that requires specific weight*

public class GridWeightSwitchBehavior : IGridFloorBehavior, IGridFloorTrigger {

public float requiredWeight = 100f;

public string connectedDoorId;

private bool isPressed = false;

public void OnEnter(Entity entity, Direction fromDirection) {

float weight = entity.weight;

*// Check for pushed blocks*

if (entity.IsPushing()) {

weight += entity.pushedObject.weight;

}

if (weight >= requiredWeight && !isPressed) {

isPressed = true;

ActivateConnectedDoor(connectedDoorId);

ShowPressedVisual();

}

}

public void OnExit(Entity entity, Direction toDirection) {

*// Check if anything else is keeping it pressed*

if (!HasSufficientWeight() && isPressed) {

isPressed = false;

DeactivateConnectedDoor(connectedDoorId);

ShowUnpressedVisual();

}

}

}

*// Trap that activates after delay*

public class GridDelayedTrapBehavior : IGridFloorBehavior, IGridFloorTrigger {

public int triggerDelayTurns = 1;

public int resetDelayTurns = 3;

private Queue<ScheduledTrigger> scheduledTriggers = new Queue<ScheduledTrigger>();

public void OnEnter(Entity entity, Direction fromDirection) {

*// Schedule trap to spring*

var trigger = new ScheduledTrigger {

triggerTurn = CurrentTurn + triggerDelayTurns,

targetEntity = entity

};

scheduledTriggers.Enqueue(trigger);

ShowTrapWarning(triggerDelayTurns);

}

public void OnTurnEnd() {

while (scheduledTriggers.Count > 0 &&

scheduledTriggers.Peek().triggerTurn <= CurrentTurn) {

var trigger = scheduledTriggers.Dequeue();

SpringTrap(trigger.targetEntity);

}

}

}

**Visual Feedback for Grid Floors**

csharp

public class GridFloorVisualSystem : MonoBehaviour {

[Header("Visual Indicators")]

public GameObject movementPreviewPrefab;

public GameObject dangerIndicatorPrefab;

public GameObject interactableHighlightPrefab;

public void ShowMovementRange(Entity entity) {

int movementPoints = entity.currentMovementPoints;

var reachableTiles = GetReachableTiles(entity.gridPosition, movementPoints);

foreach (var tile in reachableTiles) {

*// Color based on movement cost*

int cost = tile.GetMovementCost();

Color color = GetCostColor(cost);

var preview = Instantiate(movementPreviewPrefab, tile.worldPosition, Quaternion.identity);

preview.GetComponent<Renderer>().material.color = color;

*// Show special indicators*

if (tile.HasBehavior<GridIceFloorBehavior>()) {

ShowIceSlidePreview(tile, entity.facing);

}

if (tile.HasBehavior<GridTeleporterBehavior>()) {

ShowTeleportDestination(tile);

}

}

}

public void ShowFloorState(GridFloorTile tile) {

*// Show different states visually*

foreach (var state in tile.states) {

if (state is GridRhythmFloorBehavior rhythm) {

bool isSafe = IsRhythmTileSafe(rhythm, CurrentTurn);

SetTileColor(tile, isSafe ? Color.green : Color.red);

}

if (state is GridCrumblingFloorBehavior crumbling) {

ShowCrackOverlay(tile, crumbling.turnsRemaining);

}

}

}

}

**Grid Movement Controller**

csharp

public class GridMovementController : MonoBehaviour {

public Entity controlledEntity;

public int baseMovementPoints = 3;

private int currentMovementPoints;

public bool TryMove(Direction direction) {

Vector2Int targetPos = controlledEntity.gridPosition + DirectionToVector(direction);

GridFloorTile targetTile = GetTileAt(targetPos);

if (targetTile == null) return false;

*// Check if can enter*

foreach (var behavior in targetTile.behaviors) {

if (!behavior.CanEnter(controlledEntity, direction)) {

ShowBlockedFeedback(behavior.GetBlockReason());

return false;

}

}

*// Check movement cost*

int cost = targetTile.GetMovementCost();

if (currentMovementPoints < cost) {

ShowInsufficientMovementFeedback();

return false;

}

*// Execute movement*

GridFloorTile currentTile = GetTileAt(controlledEntity.gridPosition);

*// Exit current tile*

foreach (var behavior in currentTile.behaviors) {

behavior.OnExit(controlledEntity, direction);

}

*// Move entity*

controlledEntity.gridPosition = targetPos;

currentMovementPoints -= cost;

*// Enter new tile*

foreach (var behavior in targetTile.behaviors) {

behavior.OnEnter(controlledEntity, GetOppositeDirection(direction));

}

*// Check for forced movement (ice, conveyor, etc)*

CheckForcedMovement(targetTile);

return true;

}

}

This grid-based approach makes floors incredibly strategic:

1. **Every step matters** - Each tile is a decision
2. **Predictable mechanics** - Players can count turns and plan
3. **Clear visual feedback** - See exactly what each tile does
4. **Turn-based triggers** - Traps activate on specific turns
5. **Movement puzzles** - Ice slides, conveyors, teleporters work perfectly
6. **Resource management** - Movement points become valuable
7. **Pattern recognition** - Rhythm floors, color patterns, etc.

The discrete nature of grid movement actually makes floor behaviors MORE impactful since you can't just walk around hazards - you must deal with them directly!

Retry

**J**

can this be integrated into that wall building tool as well?

Edit

Absolutely! The floor system integrates perfectly into the dungeon building tool. Here's how to add floor painting and editing to the same editor:

**Enhanced Dungeon Editor with Floors**

csharp

public class DungeonPainterWindow : EditorWindow {

*// Drawing modes (extended)*

private enum DrawMode {

Wall,

Door,

Floor, *// NEW: Floor painting mode*

FloorBehavior, *// NEW: Add behaviors to floors*

Props,

Encounters,

Regions,

Erase

}

*// Floor editing*

private FloorType selectedFloorType = FloorType.Stone;

private FloorBehaviorType selectedBehavior = FloorBehaviorType.Normal;

private int currentEditingLayer = 0;

private bool showFloorGrid = true;

private bool showFloorBehaviors = true;

*// Floor palette*

private FloorPalette floorPalette;

private FloorBehaviorPalette behaviorPalette;

void DrawToolbar() {

EditorGUILayout.BeginHorizontal(EditorStyles.toolbar);

*// Mode buttons (with new floor modes)*

if (GUILayout.Toggle(currentMode == DrawMode.Wall, EditorTools.wallIcon, EditorStyles.toolbarButton))

currentMode = DrawMode.Wall;

if (GUILayout.Toggle(currentMode == DrawMode.Door, EditorTools.doorIcon, EditorStyles.toolbarButton))

currentMode = DrawMode.Door;

if (GUILayout.Toggle(currentMode == DrawMode.Floor, EditorTools.floorIcon, EditorStyles.toolbarButton))

currentMode = DrawMode.Floor;

if (GUILayout.Toggle(currentMode == DrawMode.FloorBehavior, EditorTools.behaviorIcon, EditorStyles.toolbarButton))

currentMode = DrawMode.FloorBehavior;

*// ... other modes*

EditorGUILayout.EndHorizontal();

}

void DrawCanvas() {

Rect canvasRect = new Rect(200, 20, position.width - 400, position.height - 40);

GUI.Box(canvasRect, GUIContent.none, EditorStyles.helpBox);

scrollPosition = GUI.BeginScrollView(canvasRect, scrollPosition, viewRect);

*// Draw grid*

DrawGrid();

*// Draw floors FIRST (bottom layer)*

if (showFloorGrid) {

DrawFloors();

}

*// Draw walls on top of floors*

DrawWalls();

*// Draw floor behaviors as overlays*

if (showFloorBehaviors) {

DrawFloorBehaviorOverlay();

}

*// Other elements...*

GUI.EndScrollView();

}

void DrawFloors() {

for (int x = 0; x < gridWidth; x++) {

for (int y = 0; y < gridHeight; y++) {

var floorData = dungeonData.GetFloor(x, y, currentEditingLayer);

if (floorData != null) {

DrawFloorTile(x, y, floorData);

}

}

}

}

void DrawFloorTile(int x, int y, GridFloorTile floorData) {

Rect cellRect = GetCellRect(x, y);

*// Base floor texture/color*

Color floorColor = GetFloorColor(floorData.floorType);

*// Darken if not walkable*

if (floorData.blocksMovement) {

floorColor \*= 0.3f;

}

EditorGUI.DrawRect(cellRect, floorColor);

*// Draw floor type icon*

if (floorTypeIcons.ContainsKey(floorData.floorType)) {

GUI.DrawTexture(

new Rect(cellRect.x + 2, cellRect.y + 2, 16, 16),

floorTypeIcons[floorData.floorType]

);

}

*// Draw behavior indicators*

DrawFloorBehaviorIcons(cellRect, floorData);

}

void DrawFloorBehaviorIcons(Rect cellRect, GridFloorTile floorData) {

float iconSize = 12f;

float padding = 2f;

float currentX = cellRect.x + cellRect.width - iconSize - padding;

float currentY = cellRect.y + padding;

foreach (var behavior in floorData.behaviors) {

Texture2D icon = GetBehaviorIcon(behavior);

if (icon != null) {

GUI.DrawTexture(

new Rect(currentX, currentY, iconSize, iconSize),

icon

);

currentX -= iconSize + padding;

*// Wrap to next row if needed*

if (currentX < cellRect.x) {

currentX = cellRect.x + cellRect.width - iconSize - padding;

currentY += iconSize + padding;

}

}

}

}

}

**Floor Painting Tools**

csharp

public class FloorPaintingTools {

private DungeonData dungeonData;

private int currentLayer;

public void HandleFloorPainting(Event e, Vector2Int gridPos) {

switch (e.type) {

case EventType.MouseDown:

if (e.button == 0) {

StartFloorPainting(gridPos);

} else if (e.button == 1) {

ShowFloorContextMenu(gridPos);

}

break;

case EventType.MouseDrag:

if (isPainting) {

ContinueFloorPainting(gridPos);

}

break;

}

}

void StartFloorPainting(Vector2Int gridPos) {

if (currentMode == DrawMode.Floor) {

PaintFloorType(gridPos);

} else if (currentMode == DrawMode.FloorBehavior) {

AddFloorBehavior(gridPos);

}

}

void PaintFloorType(Vector2Int gridPos) {

var floor = dungeonData.GetOrCreateFloor(gridPos.x, gridPos.y, currentLayer);

*// Record for undo*

Undo.RecordObject(dungeonData, "Paint Floor");

floor.floorType = selectedFloorType;

*// Auto-add default behaviors based on type*

floor.behaviors.Clear();

switch (selectedFloorType) {

case FloorType.Ice:

floor.behaviors.Add(new GridIceFloorBehavior());

break;

case FloorType.Lava:

floor.behaviors.Add(new GridDamageFloorBehavior {

damagePerTurn = 10,

damageType = DamageType.Fire

});

break;

case FloorType.Pit:

floor.blocksMovement = true;

floor.behaviors.Add(new GridPitBehavior());

break;

}

EditorUtility.SetDirty(dungeonData);

}

void AddFloorBehavior(Vector2Int gridPos) {

var floor = dungeonData.GetFloor(gridPos.x, gridPos.y, currentLayer);

if (floor == null) return;

Undo.RecordObject(dungeonData, "Add Floor Behavior");

*// Add selected behavior if not already present*

var newBehavior = CreateBehaviorInstance(selectedBehavior);

if (!floor.HasBehavior(newBehavior.GetType())) {

floor.behaviors.Add(newBehavior);

}

EditorUtility.SetDirty(dungeonData);

}

}

**Floor Palette Panel**

csharp

public class FloorPalettePanel {

private Vector2 scrollPosition;

private string searchFilter = "";

*// Floor categories*

private Dictionary<string, List<FloorPreset>> floorCategories = new Dictionary<string, List<FloorPreset>> {

["Basic"] = new List<FloorPreset> {

new FloorPreset("Stone", FloorType.Stone),

new FloorPreset("Wood", FloorType.Wood),

new FloorPreset("Dirt", FloorType.Dirt),

new FloorPreset("Grass", FloorType.Grass)

},

["Hazards"] = new List<FloorPreset> {

new FloorPreset("Lava", FloorType.Lava, new[] { typeof(GridDamageFloorBehavior) }),

new FloorPreset("Poison", FloorType.Poison, new[] { typeof(GridPoisonFloorBehavior) }),

new FloorPreset("Spikes", FloorType.Spikes, new[] { typeof(GridSpikeTrapBehavior) }),

new FloorPreset("Pit", FloorType.Pit, new[] { typeof(GridPitBehavior) })

},

["Special"] = new List<FloorPreset> {

new FloorPreset("Ice", FloorType.Ice, new[] { typeof(GridIceFloorBehavior) }),

new FloorPreset("Teleporter", FloorType.Teleporter, new[] { typeof(GridTeleporterBehavior) }),

new FloorPreset("Conveyor", FloorType.Conveyor, new[] { typeof(ConveyorFloorBehavior) }),

new FloorPreset("Pressure Plate", FloorType.PressurePlate, new[] { typeof(GridWeightSwitchBehavior) })

}

};

public void DrawFloorPalette(Rect rect) {

GUILayout.BeginArea(rect);

GUILayout.Label("Floor Types", EditorStyles.boldLabel);

*// Search bar*

searchFilter = EditorGUILayout.TextField("Search:", searchFilter);

scrollPosition = GUILayout.BeginScrollView(scrollPosition);

foreach (var category in floorCategories) {

if (EditorGUILayout.Foldout(foldouts[category.Key], category.Key)) {

DrawFloorCategory(category.Value);

}

}

GUILayout.EndScrollView();

*// Floor behavior section*

GUILayout.Space(10);

DrawBehaviorPalette();

GUILayout.EndArea();

}

void DrawFloorCategory(List<FloorPreset> presets) {

int columns = 3;

int current = 0;

GUILayout.BeginHorizontal();

foreach (var preset in presets) {

if (!string.IsNullOrEmpty(searchFilter) &&

!preset.name.ToLower().Contains(searchFilter.ToLower())) {

continue;

}

if (current > 0 && current % columns == 0) {

GUILayout.EndHorizontal();

GUILayout.BeginHorizontal();

}

DrawFloorPresetButton(preset);

current++;

}

GUILayout.EndHorizontal();

}

void DrawFloorPresetButton(FloorPreset preset) {

bool isSelected = (selectedFloorType == preset.floorType);

GUI.backgroundColor = isSelected ? Color.green : Color.white;

if (GUILayout.Button(new GUIContent(preset.icon, preset.name),

GUILayout.Width(50), GUILayout.Height(50))) {

SelectFloorPreset(preset);

}

GUI.backgroundColor = Color.white;

}

void DrawBehaviorPalette() {

GUILayout.Label("Floor Behaviors", EditorStyles.boldLabel);

EditorGUILayout.HelpBox(

"Select a behavior and click on floors to add it. " +

"Right-click floors to remove behaviors.",

MessageType.Info

);

foreach (FloorBehaviorType behaviorType in System.Enum.GetValues(typeof(FloorBehaviorType))) {

bool isSelected = (selectedBehavior == behaviorType);

EditorGUILayout.BeginHorizontal();

if (GUILayout.Toggle(isSelected, GetBehaviorIcon(behaviorType),

GUILayout.Width(30), GUILayout.Height(30))) {

selectedBehavior = behaviorType;

}

GUILayout.Label(behaviorType.ToString());

*// Quick settings for behavior*

DrawBehaviorQuickSettings(behaviorType);

EditorGUILayout.EndHorizontal();

}

}

}

**Floor Inspector Panel**

csharp

public class FloorInspectorPanel {

private GridFloorTile selectedFloor;

private Vector2Int selectedPosition;

public void DrawInspector(Rect rect) {

if (selectedFloor == null) {

GUILayout.BeginArea(rect);

GUILayout.Label("No floor selected", EditorStyles.centeredGreyMiniLabel);

GUILayout.EndArea();

return;

}

GUILayout.BeginArea(rect);

GUILayout.Label($"Floor [{selectedPosition.x}, {selectedPosition.y}]", EditorStyles.boldLabel);

EditorGUILayout.Space();

*// Floor type*

selectedFloor.floorType = (FloorType)EditorGUILayout.EnumPopup("Type:", selectedFloor.floorType);

*// Basic properties*

selectedFloor.blocksMovement = EditorGUILayout.Toggle("Blocks Movement", selectedFloor.blocksMovement);

selectedFloor.blocksVision = EditorGUILayout.Toggle("Blocks Vision", selectedFloor.blocksVision);

EditorGUILayout.Space();

*// Behaviors list*

GUILayout.Label("Behaviors:", EditorStyles.boldLabel);

for (int i = 0; i < selectedFloor.behaviors.Count; i++) {

var behavior = selectedFloor.behaviors[i];

EditorGUILayout.BeginVertical(EditorStyles.helpBox);

EditorGUILayout.BeginHorizontal();

GUILayout.Label(behavior.GetType().Name);

if (GUILayout.Button("Settings", GUILayout.Width(60))) {

ShowBehaviorSettings(behavior);

}

if (GUILayout.Button("X", GUILayout.Width(20))) {

selectedFloor.behaviors.RemoveAt(i);

i--;

}

EditorGUILayout.EndHorizontal();

*// Show behavior-specific properties*

DrawBehaviorProperties(behavior);

EditorGUILayout.EndVertical();

}

*// Add behavior button*

if (GUILayout.Button("+Add Behavior")) {

ShowAddBehaviorMenu(selectedFloor);

}

GUILayout.EndArea();

}

void DrawBehaviorProperties(IGridFloorBehavior behavior) {

if (behavior is GridTeleporterBehavior teleporter) {

EditorGUI.indentLevel++;

teleporter.destination = EditorGUILayout.Vector2IntField("Destination:", teleporter.destination);

teleporter.chargeTimeInTurns = EditorGUILayout.IntField("Charge Turns:", teleporter.chargeTimeInTurns);

EditorGUI.indentLevel--;

}

else if (behavior is GridDamageFloorBehavior damage) {

EditorGUI.indentLevel++;

damage.damagePerTurn = EditorGUILayout.IntField("Damage/Turn:", damage.damagePerTurn);

damage.damageType = (DamageType)EditorGUILayout.EnumPopup("Type:", damage.damageType);

EditorGUI.indentLevel--;

}

else if (behavior is GridWeightSwitchBehavior weightSwitch) {

EditorGUI.indentLevel++;

weightSwitch.requiredWeight = EditorGUILayout.FloatField("Required Weight:", weightSwitch.requiredWeight);

weightSwitch.connectedDoorId = EditorGUILayout.TextField("Door ID:", weightSwitch.connectedDoorId);

EditorGUI.indentLevel--;

}

}

}

**Smart Floor Tools**

csharp

public class SmartFloorTools {

*// Fill area with floor pattern*

public static void FillArea(DungeonData data, Vector2Int start, Vector2Int end, FloorPattern pattern) {

for (int x = start.x; x <= end.x; x++) {

for (int y = start.y; y <= end.y; y++) {

var floor = data.GetOrCreateFloor(x, y, currentLayer);

switch (pattern) {

case FloorPattern.Checkerboard:

floor.floorType = ((x + y) % 2 == 0) ? FloorType.White : FloorType.Black;

break;

case FloorPattern.ConcentricSquares:

int distFromEdge = Mathf.Min(

Mathf.Min(x - start.x, end.x - x),

Mathf.Min(y - start.y, end.y - y)

);

floor.floorType = floorTypes[distFromEdge % floorTypes.Length];

break;

case FloorPattern.RandomHazards:

if (Random.value < 0.1f) {

floor.floorType = FloorType.Spikes;

floor.behaviors.Add(new GridSpikeTrapBehavior());

}

break;

}

}

}

}

*// Create puzzle room*

public static void CreatePuzzleRoom(DungeonData data, Vector2Int center, PuzzleType type) {

switch (type) {

case PuzzleType.IceSliding:

CreateIceSlidePuzzle(data, center);

break;

case PuzzleType.PressurePlates:

CreatePressurePlatePuzzle(data, center);

break;

case PuzzleType.Teleporter:

CreateTeleporterMaze(data, center);

break;

}

}

static void CreateIceSlidePuzzle(DungeonData data, Vector2Int center) {

*// Create ice floor with strategic non-ice spots*

int roomSize = 7;

for (int x = -roomSize/2; x <= roomSize/2; x++) {

for (int y = -roomSize/2; y <= roomSize/2; y++) {

var pos = center + new Vector2Int(x, y);

var floor = data.GetOrCreateFloor(pos.x, pos.y, 0);

*// Most tiles are ice*

floor.floorType = FloorType.Ice;

floor.behaviors.Add(new GridIceFloorBehavior());

*// Add strategic stopping points*

if ((x == 0 || y == 0) && Random.value < 0.3f) {

floor.floorType = FloorType.Stone;

floor.behaviors.Clear();

}

}

}

}

}

**Visual Preview System**

csharp

public class FloorPreviewSystem {

private GameObject previewContainer;

private Dictionary<Vector2Int, GameObject> floorPreviews = new Dictionary<Vector2Int, GameObject>();

public void UpdateFloorPreview(DungeonData data, int layer) {

ClearPreviews();

for (int x = 0; x < data.width; x++) {

for (int y = 0; y < data.height; y++) {

var floor = data.GetFloor(x, y, layer);

if (floor != null) {

CreateFloorPreview(x, y, floor);

}

}

}

}

void CreateFloorPreview(int x, int y, GridFloorTile floor) {

Vector3 position = new Vector3(x \* cellSize, 0, y \* cellSize);

*// Create floor mesh*

GameObject floorObj = GameObject.CreatePrimitive(PrimitiveType.Quad);

floorObj.transform.position = position;

floorObj.transform.rotation = Quaternion.Euler(90, 0, 0);

*// Apply material based on type*

var renderer = floorObj.GetComponent<MeshRenderer>();

renderer.material = GetFloorMaterial(floor.floorType);

*// Add behavior visualizers*

foreach (var behavior in floor.behaviors) {

AddBehaviorVisualizer(floorObj, behavior);

}

floorPreviews[new Vector2Int(x, y)] = floorObj;

}

void AddBehaviorVisualizer(GameObject floorObj, IGridFloorBehavior behavior) {

if (behavior is GridTeleporterBehavior) {

*// Add particle effect*

var particles = floorObj.AddComponent<ParticleSystem>();

ConfigureTeleporterParticles(particles);

}

else if (behavior is GridDamageFloorBehavior) {

*// Add danger glow*

var light = floorObj.AddComponent<Light>();

light.color = Color.red;

light.intensity = 0.5f;

light.range = 1f;

}

}

}

**Keyboard Shortcuts**

csharp

void HandleKeyboardShortcuts() {

Event e = Event.current;

if (e.type == EventType.KeyDown) {

*// Quick floor type selection*

if (e.control) {

switch (e.keyCode) {

case KeyCode.Alpha1:

selectedFloorType = FloorType.Stone;

break;

case KeyCode.Alpha2:

selectedFloorType = FloorType.Wood;

break;

case KeyCode.Alpha3:

selectedFloorType = FloorType.Ice;

break;

case KeyCode.Alpha4:

selectedFloorType = FloorType.Lava;

break;

}

}

*// Mode switching*

switch (e.keyCode) {

case KeyCode.F:

currentMode = DrawMode.Floor;

break;

case KeyCode.B:

currentMode = DrawMode.FloorBehavior;

break;

case KeyCode.W:

currentMode = DrawMode.Wall;

break;

}

Repaint();

}

}

This integration gives you:

1. **Unified tool** - Walls and floors in same editor
2. **Visual layers** - See floors under walls clearly
3. **Smart tools** - Fill patterns, puzzle generators
4. **Behavior painting** - Add ice, teleporters, traps easily
5. **Inspector panel** - Fine-tune individual tiles
6. **Preview system** - See how it looks in 3D
7. **Context menus** - Right-click for quick edits
8. **Keyboard shortcuts** - Fast workflow

The key is that floors and walls complement each other - you can paint an ice puzzle room, then add walls around it, all in the same intuitive interface!

Retry

**J**

how would i create the 3d prefabs so i can integrate them in with the tool to where they're placed when i'm using the tool

Edit

Great question! Here's how to create and integrate 3D prefabs with your dungeon building tool:

**Prefab Structure and Organization**

csharp

*// ScriptableObject to manage all prefabs*

[CreateAssetMenu(fileName = "DungeonPrefabLibrary", menuName = "Dungeon/Prefab Library")]

public class DungeonPrefabLibrary : ScriptableObject {

[Header("Wall Prefabs")]

public WallPrefabSet walls;

[Header("Floor Prefabs")]

public FloorPrefabSet floors;

[Header("Props")]

public PropPrefabSet props;

[Header("Doors")]

public DoorPrefabSet doors;

[Header("Special")]

public SpecialPrefabSet special;

}

[System.Serializable]

public class WallPrefabSet {

public GameObject defaultWall;

public GameObject doorFrame;

public GameObject secretWall;

public GameObject brokenWall;

public GameObject windowWall;

public GameObject barsWall;

*// Variations*

public GameObject[] wallVariations; *// For randomization*

public GameObject GetWallPrefab(WallType type) {

switch (type) {

case WallType.Door: return doorFrame;

case WallType.Secret: return secretWall;

case WallType.Window: return windowWall;

case WallType.Bars: return barsWall;

default: return defaultWall;

}

}

}

[System.Serializable]

public class FloorPrefabSet {

[System.Serializable]

public class FloorPrefabMapping {

public FloorType floorType;

public GameObject prefab;

public GameObject[] variations;

}

public List<FloorPrefabMapping> floorMappings;

public GameObject defaultFloor;

public GameObject GetFloorPrefab(FloorType type) {

var mapping = floorMappings.Find(m => m.floorType == type);

if (mapping != null) {

if (mapping.variations.Length > 0) {

return mapping.variations[Random.Range(0, mapping.variations.Length)];

}

return mapping.prefab;

}

return defaultFloor;

}

}

**Creating the Prefabs in Unity**

**Basic Wall Prefab Structure:**

WallPrefab

├── Mesh (with LOD Group)

│ ├── LOD0\_Detailed

│ ├── LOD1\_Medium

│ └── LOD2\_Simple

├── Collider (Box Collider)

├── WallPrefabMarker (Script)

├── Sockets (Empty GameObjects for attachments)

│ ├── TopSocket

│ ├── BottomSocket

│ ├── LeftSocket

│ └── RightSocket

└── Decorations (Optional)

├── Torch\_Spawn\_Point

└── Ivy\_Spawn\_Point

**Script for Prefab Setup:**

csharp

[ExecuteInEditMode]

public class WallPrefabMarker : MonoBehaviour {

[Header("Prefab Info")]

public WallType wallType;

public Vector3 gridSize = new Vector3(3, 3, 0.3f); *// Width, Height, Thickness*

[Header("Connection Points")]

public Transform topSocket;

public Transform bottomSocket;

public Transform leftSocket;

public Transform rightSocket;

[Header("Behavior Components")]

public bool hasDoor;

public DoorController doorController;

public bool isDestructible;

public float health = 100f;

*// Visual guide in editor*

void OnDrawGizmos() {

Gizmos.color = Color.yellow;

Gizmos.DrawWireCube(transform.position, gridSize);

*// Draw connection points*

if (topSocket) Gizmos.DrawWireSphere(topSocket.position, 0.1f);

if (bottomSocket) Gizmos.DrawWireSphere(bottomSocket.position, 0.1f);

if (leftSocket) Gizmos.DrawWireSphere(leftSocket.position, 0.1f);

if (rightSocket) Gizmos.DrawWireSphere(rightSocket.position, 0.1f);

}

#if UNITY\_EDITOR

[Button("Auto-Setup Sockets")]

void AutoSetupSockets() {

*// Create socket points if they don't exist*

if (!topSocket) topSocket = CreateSocket("TopSocket", Vector3.up \* gridSize.y/2);

if (!bottomSocket) bottomSocket = CreateSocket("BottomSocket", Vector3.down \* gridSize.y/2);

if (!leftSocket) leftSocket = CreateSocket("LeftSocket", Vector3.left \* gridSize.x/2);

if (!rightSocket) rightSocket = CreateSocket("RightSocket", Vector3.right \* gridSize.x/2);

}

Transform CreateSocket(string name, Vector3 localPosition) {

GameObject socket = new GameObject(name);

socket.transform.parent = transform;

socket.transform.localPosition = localPosition;

return socket.transform;

}

#endif

}

**Floor Prefab Structure:**

FloorPrefab

├── Mesh (Quad or custom mesh)

├── Collider (Box or Mesh Collider)

├── FloorPrefabMarker (Script)

├── Surface Effects

│ ├── ParticleSystem (for lava, water, etc.)

│ └── Light (for glowing floors)

└── Trigger Volume (for special behaviors)

csharp

public class FloorPrefabMarker : MonoBehaviour {

[Header("Floor Info")]

public FloorType floorType;

public Vector2 gridSize = new Vector2(3, 3);

[Header("Behavior")]

public bool isWalkable = true;

public float movementSpeedModifier = 1f;

public bool triggersOnStep;

[Header("Visual")]

public Material[] materialVariations;

public bool randomizeRotation = true;

public bool hasEdgeBlending = true;

void OnValidate() {

*// Auto-configure based on floor type*

switch (floorType) {

case FloorType.Ice:

movementSpeedModifier = 2f;

break;

case FloorType.Mud:

movementSpeedModifier = 0.5f;

break;

case FloorType.Pit:

isWalkable = false;

break;

}

}

}

**Editor Integration - Live Preview**

csharp

public class DungeonPreviewManager : MonoBehaviour {

[Header("References")]

public DungeonPrefabLibrary prefabLibrary;

public Transform previewContainer;

private Dictionary<Vector3Int, GameObject> previewObjects = new Dictionary<Vector3Int, GameObject>();

private DungeonData currentData;

*// Called by the editor tool*

public void UpdatePreview(DungeonData data, int currentLayer) {

currentData = data;

*// Clear old preview*

ClearPreview();

*// Generate new preview*

GenerateWallPreviews(data, currentLayer);

GenerateFloorPreviews(data, currentLayer);

}

void GenerateWallPreviews(DungeonData data, int layer) {

for (int x = 0; x < data.width; x++) {

for (int y = 0; y <= data.height; y++) {

*// Horizontal walls*

var wallData = data.horizontalWalls[x, y];

if (wallData.exists) {

CreateWallPreview(

new Vector3(x \* cellSize + cellSize/2, layer \* layerHeight, y \* cellSize),

Quaternion.identity,

wallData

);

}

}

}

*// Similar for vertical walls...*

}

void CreateWallPreview(Vector3 position, Quaternion rotation, WallData data) {

GameObject prefab = prefabLibrary.walls.GetWallPrefab(data.type);

if (prefab == null) return;

GameObject preview = Instantiate(prefab, position, rotation, previewContainer);

*// Make it semi-transparent in preview*

SetPreviewMaterial(preview);

*// Store for updates*

Vector3Int key = Vector3Int.RoundToInt(position);

previewObjects[key] = preview;

*// Configure based on data*

var marker = preview.GetComponent<WallPrefabMarker>();

if (marker != null && marker.hasDoor) {

*// Set door state*

if (marker.doorController != null) {

marker.doorController.isLocked = data.isLocked;

}

}

}

void GenerateFloorPreviews(DungeonData data, int layer) {

for (int x = 0; x < data.width; x++) {

for (int y = 0; y < data.height; y++) {

var floorData = data.GetFloor(x, y, layer);

if (floorData != null) {

CreateFloorPreview(

new Vector3(x \* cellSize, layer \* layerHeight, y \* cellSize),

floorData

);

}

}

}

}

void SetPreviewMaterial(GameObject obj) {

foreach (var renderer in obj.GetComponentsInChildren<MeshRenderer>()) {

var materials = renderer.sharedMaterials;

for (int i = 0; i < materials.Length; i++) {

materials[i] = GetPreviewMaterial(materials[i]);

}

renderer.sharedMaterials = materials;

}

}

}

**Custom Editor Window Integration**

csharp

public partial class DungeonPainterWindow : EditorWindow {

[Header("Prefab Settings")]

private DungeonPrefabLibrary prefabLibrary;

private GameObject previewRoot;

private DungeonPreviewManager previewManager;

private bool showLivePreview = true;

void OnEnable() {

*// Load prefab library*

prefabLibrary = AssetDatabase.LoadAssetAtPath<DungeonPrefabLibrary>(

"Assets/DungeonAssets/PrefabLibrary.asset"

);

if (prefabLibrary == null) {

Debug.LogError("Prefab Library not found! Please create one.");

}

*// Setup preview*

SetupPreviewScene();

}

void SetupPreviewScene() {

*// Create preview container in scene*

previewRoot = GameObject.Find("DungeonPreview");

if (previewRoot == null) {

previewRoot = new GameObject("DungeonPreview");

}

previewManager = previewRoot.GetComponent<DungeonPreviewManager>();

if (previewManager == null) {

previewManager = previewRoot.AddComponent<DungeonPreviewManager>();

}

previewManager.prefabLibrary = prefabLibrary;

previewManager.previewContainer = previewRoot.transform;

}

void OnPaintFloor(Vector2Int gridPos) {

*// Paint in data*

var floor = dungeonData.GetOrCreateFloor(gridPos.x, gridPos.y, currentLayer);

floor.floorType = selectedFloorType;

*// Update live preview if enabled*

if (showLivePreview) {

UpdateSingleFloorPreview(gridPos);

}

EditorUtility.SetDirty(dungeonData);

}

void UpdateSingleFloorPreview(Vector2Int gridPos) {

Vector3 worldPos = new Vector3(

gridPos.x \* cellSize,

currentLayer \* layerHeight,

gridPos.y \* cellSize

);

*// Remove old preview at this position*

previewManager.RemovePreviewAt(worldPos);

*// Create new preview*

var floorData = dungeonData.GetFloor(gridPos.x, gridPos.y, currentLayer);

if (floorData != null) {

GameObject prefab = prefabLibrary.floors.GetFloorPrefab(floorData.floorType);

if (prefab != null) {

GameObject preview = PrefabUtility.InstantiatePrefab(prefab) as GameObject;

preview.transform.position = worldPos;

preview.transform.parent = previewRoot.transform;

}

}

}

}

**Prefab Creation Helper Tools**

csharp

[CustomEditor(typeof(WallPrefabMarker))]

public class WallPrefabMarkerEditor : Editor {

public override void OnInspectorGUI() {

DrawDefaultInspector();

WallPrefabMarker marker = (WallPrefabMarker)target;

EditorGUILayout.Space();

EditorGUILayout.LabelField("Setup Tools", EditorStyles.boldLabel);

if (GUILayout.Button("Auto-Configure Colliders")) {

AutoConfigureColliders(marker);

}

if (GUILayout.Button("Create LOD Group")) {

CreateLODGroup(marker);

}

if (GUILayout.Button("Align to Grid")) {

AlignToGrid(marker);

}

if (GUILayout.Button("Save as Prefab Variant")) {

SaveAsPrefabVariant(marker);

}

}

void AutoConfigureColliders(WallPrefabMarker marker) {

BoxCollider collider = marker.GetComponent<BoxCollider>();

if (collider == null) {

collider = marker.gameObject.AddComponent<BoxCollider>();

}

collider.size = marker.gridSize;

collider.center = Vector3.zero;

EditorUtility.SetDirty(marker);

}

void CreateLODGroup(WallPrefabMarker marker) {

LODGroup lodGroup = marker.GetComponent<LODGroup>();

if (lodGroup == null) {

lodGroup = marker.gameObject.AddComponent<LODGroup>();

}

*// Find child meshes*

MeshRenderer[] renderers = marker.GetComponentsInChildren<MeshRenderer>();

LOD[] lods = new LOD[3];

lods[0] = new LOD(0.6f, renderers); *// Full detail at 60%*

lods[1] = new LOD(0.3f, renderers); *// Medium at 30%*

lods[2] = new LOD(0.1f, renderers); *// Low at 10%*

lodGroup.SetLODs(lods);

lodGroup.RecalculateBounds();

}

}

**Runtime Spawning from Tool**

csharp

public class DungeonBuilder : MonoBehaviour {

public DungeonPrefabLibrary prefabLibrary;

[Button("Build Dungeon From Data")]

public void BuildDungeon(DungeonData data) {

ClearExistingDungeon();

GameObject dungeonRoot = new GameObject("GeneratedDungeon");

*// Create floors*

Transform floorContainer = new GameObject("Floors").transform;

floorContainer.parent = dungeonRoot.transform;

BuildFloors(data, floorContainer);

*// Create walls*

Transform wallContainer = new GameObject("Walls").transform;

wallContainer.parent = dungeonRoot.transform;

BuildWalls(data, wallContainer);

*// Optimize*

StaticBatchingUtility.Combine(dungeonRoot);

}

void BuildFloors(DungeonData data, Transform container) {

for (int layer = 0; layer < data.numLayers; layer++) {

for (int x = 0; x < data.width; x++) {

for (int y = 0; y < data.height; y++) {

var floorData = data.GetFloor(x, y, layer);

if (floorData != null) {

SpawnFloor(floorData, new Vector3(x \* cellSize, layer \* layerHeight, y \* cellSize), container);

}

}

}

}

}

GameObject SpawnFloor(GridFloorTile floorData, Vector3 position, Transform parent) {

GameObject prefab = prefabLibrary.floors.GetFloorPrefab(floorData.floorType);

if (prefab == null) return null;

GameObject floor = Instantiate(prefab, position, Quaternion.identity, parent);

*// Apply behaviors*

foreach (var behavior in floorData.behaviors) {

ApplyBehaviorToPrefab(floor, behavior);

}

return floor;

}

void ApplyBehaviorToPrefab(GameObject obj, IGridFloorBehavior behavior) {

*// Add runtime components based on behavior*

if (behavior is GridTeleporterBehavior teleporter) {

var component = obj.AddComponent<TeleporterRuntime>();

component.destination = teleporter.destination;

component.chargeTime = teleporter.chargeTimeInTurns;

}

*// ... etc for other behaviors*

}

}

**Prefab Validation System**

csharp

public class PrefabValidator : EditorWindow {

[MenuItem("Tools/Dungeon/Validate Prefabs")]

static void ShowWindow() {

GetWindow<PrefabValidator>("Prefab Validator");

}

void OnGUI() {

if (GUILayout.Button("Validate All Dungeon Prefabs")) {

ValidateAllPrefabs();

}

}

void ValidateAllPrefabs() {

string[] guids = AssetDatabase.FindAssets("t:Prefab", new[] { "Assets/DungeonPrefabs" });

foreach (string guid in guids) {

string path = AssetDatabase.GUIDToAssetPath(guid);

GameObject prefab = AssetDatabase.LoadAssetAtPath<GameObject>(path);

ValidatePrefab(prefab, path);

}

}

void ValidatePrefab(GameObject prefab, string path) {

*// Check for required components*

if (prefab.GetComponent<Collider>() == null) {

Debug.LogWarning($"Prefab {path} missing Collider!");

}

*// Check scale*

if (prefab.transform.localScale != Vector3.one) {

Debug.LogWarning($"Prefab {path} has non-uniform scale!");

}

*// Check pivot point*

MeshFilter mesh = prefab.GetComponent<MeshFilter>();

if (mesh != null && mesh.sharedMesh != null) {

if (mesh.sharedMesh.bounds.center != Vector3.zero) {

Debug.LogWarning($"Prefab {path} pivot not centered!");

}

}

}

}

Key setup steps:

1. **Create prefabs** with consistent scale (1 unit = 1 grid cell)
2. **Add marker scripts** for metadata
3. **Set up sockets** for modular connections
4. **Configure colliders** to match visual mesh
5. **Create LODs** for performance
6. **Link to ScriptableObject** library
7. **Test in editor** with live preview

The prefabs will automatically spawn as you paint in the editor tool!

Retry

**J**

when you mention sockets with modular connections, what do you mean

Edit

Great question! Sockets are connection points that allow prefabs to snap together perfectly and know what can connect where. Think of them like LEGO connection points. Let me explain with examples:

**Socket System Concept**

csharp

[System.Serializable]

public class Socket {

public string socketType; *// "wall\_connector", "door\_frame", "torch\_mount"*

public Transform transform;

public SocketGender gender; *// Male, Female, Universal*

public List<string> compatibleTypes; *// What can connect here*

public enum SocketGender {

Male, *// Sticks out*

Female, *// Receives*

Universal *// Both*

}

}

public class ModularPrefab : MonoBehaviour {

[Header("Connection Points")]

public List<Socket> sockets = new List<Socket>();

[Header("Visual Debugging")]

public bool showSockets = true;

public float socketGizmoSize = 0.2f;

void OnDrawGizmos() {

if (!showSockets) return;

foreach (var socket in sockets) {

if (socket.transform == null) continue;

*// Different colors for different socket types*

switch (socket.socketType) {

case "wall\_connector":

Gizmos.color = Color.blue;

break;

case "door\_frame":

Gizmos.color = Color.green;

break;

case "torch\_mount":

Gizmos.color = Color.yellow;

break;

default:

Gizmos.color = Color.white;

break;

}

*// Draw socket*

Gizmos.DrawWireCube(socket.transform.position, Vector3.one \* socketGizmoSize);

*// Draw direction arrow*

Gizmos.DrawRay(socket.transform.position, socket.transform.forward \* socketGizmoSize \* 2);

}

}

}

**Practical Examples**

**Wall-to-Wall Connections**

csharp

public class WallPrefabSetup : MonoBehaviour {

[Header("Wall Sockets")]

public Socket leftConnector; *// Connects to adjacent wall*

public Socket rightConnector;

public Socket topConnector; *// For walls above (multi-story)*

public Socket bottomConnector; *// For walls below*

[Header("Decoration Sockets")]

public Socket[] torchMounts; *// Where torches can attach*

public Socket[] windowFrames; *// For window inserts*

public Socket doorFrame; *// For door placement*

void SetupWallSockets() {

*// Create socket at left edge*

GameObject leftSocket = new GameObject("LeftSocket");

leftSocket.transform.parent = transform;

leftSocket.transform.localPosition = new Vector3(-1.5f, 0, 0); *// Half wall width*

leftSocket.transform.localRotation = Quaternion.Euler(0, -90, 0);

leftConnector = new Socket {

socketType = "wall\_edge",

transform = leftSocket.transform,

gender = Socket.SocketGender.Universal,

compatibleTypes = new List<string> { "wall\_edge", "corner\_piece", "door\_frame" }

};

*// Similar for other edges...*

}

}

**Modular Dungeon Pieces**

csharp

public class ModularDungeonPiece : MonoBehaviour {

*// Different piece types with specific sockets*

*// Straight corridor piece*

public class CorridorStraight : ModularPrefab {

void Setup() {

*// Front and back connections*

CreateSocket("front", new Vector3(0, 0, 3), Vector3.forward, "corridor");

CreateSocket("back", new Vector3(0, 0, -3), Vector3.back, "corridor");

*// Side wall decorations*

CreateSocket("left\_wall", new Vector3(-3, 1.5f, 0), Vector3.left, "wall\_decoration");

CreateSocket("right\_wall", new Vector3(3, 1.5f, 0), Vector3.right, "wall\_decoration");

}

}

*// L-shaped corner piece*

public class CorridorCorner : ModularPrefab {

void Setup() {

*// Entry/exit connections at 90 degrees*

CreateSocket("entry", new Vector3(0, 0, -3), Vector3.back, "corridor");

CreateSocket("exit", new Vector3(3, 0, 0), Vector3.right, "corridor");

*// Corner might have special socket for corner decorations*

CreateSocket("corner\_decor", new Vector3(0, 1.5f, 0), Vector3.up, "corner\_decoration");

}

}

*// T-junction piece*

public class CorridorTJunction : ModularPrefab {

void Setup() {

CreateSocket("north", new Vector3(0, 0, 3), Vector3.forward, "corridor");

CreateSocket("east", new Vector3(3, 0, 0), Vector3.right, "corridor");

CreateSocket("west", new Vector3(-3, 0, 0), Vector3.left, "corridor");

}

}

}

**Smart Snapping System**

csharp

public class SocketSnapping : MonoBehaviour {

[Header("Snapping Settings")]

public float snapDistance = 0.5f;

public bool autoRotate = true;

public bool showConnectionPreview = true;

public bool TrySnapPrefabs(GameObject prefabA, GameObject prefabB) {

var socketsA = prefabA.GetComponent<ModularPrefab>().sockets;

var socketsB = prefabB.GetComponent<ModularPrefab>().sockets;

*// Find compatible sockets*

foreach (var socketA in socketsA) {

foreach (var socketB in socketsB) {

if (AreSocketsCompatible(socketA, socketB)) {

*// Calculate snap position*

Vector3 snapPosition = CalculateSnapPosition(socketA, socketB);

Quaternion snapRotation = CalculateSnapRotation(socketA, socketB);

*// Move prefabB to connect with prefabA*

prefabB.transform.position = snapPosition;

prefabB.transform.rotation = snapRotation;

*// Create connection*

CreateConnection(socketA, socketB);

return true;

}

}

}

return false;

}

bool AreSocketsCompatible(Socket a, Socket b) {

*// Check type compatibility*

if (!a.compatibleTypes.Contains(b.socketType)) return false;

if (!b.compatibleTypes.Contains(a.socketType)) return false;

*// Check gender compatibility*

if (a.gender == Socket.SocketGender.Male && b.gender == Socket.SocketGender.Male) return false;

if (a.gender == Socket.SocketGender.Female && b.gender == Socket.SocketGender.Female) return false;

return true;

}

Vector3 CalculateSnapPosition(Socket socketA, Socket socketB) {

*// Position prefabB so socketB aligns with socketA*

Vector3 offset = socketA.transform.position - socketB.transform.localPosition;

return socketB.transform.parent.position + offset;

}

Quaternion CalculateSnapRotation(Socket socketA, Socket socketB) {

*// Rotate so sockets face each other*

Quaternion targetRotation = Quaternion.LookRotation(

-socketA.transform.forward,

socketA.transform.up

);

return targetRotation \* Quaternion.Inverse(socketB.transform.localRotation);

}

}

**Prop Attachment System**

csharp

public class PropSocket : Socket {

[Header("Prop Settings")]

public PropType allowedPropType;

public bool randomizeRotation = true;

public Vector3 rotationRange = new Vector3(0, 360, 0);

public enum PropType {

WallTorch,

WallBanner,

FloorVase,

CeilingChain,

CornerStatue,

DoorDecoration

}

}

public class WallWithPropSockets : MonoBehaviour {

public PropSocket[] propSockets;

public void PopulateProps(float propDensity = 0.5f) {

foreach (var socket in propSockets) {

if (Random.value < propDensity) {

AttachProp(socket);

}

}

}

void AttachProp(PropSocket socket) {

*// Get appropriate prop for this socket type*

GameObject propPrefab = GetPropForSocket(socket);

if (propPrefab == null) return;

GameObject prop = Instantiate(propPrefab);

prop.transform.position = socket.transform.position;

prop.transform.rotation = socket.transform.rotation;

*// Apply randomization*

if (socket.randomizeRotation) {

prop.transform.Rotate(

Random.Range(-socket.rotationRange.x, socket.rotationRange.x),

Random.Range(-socket.rotationRange.y, socket.rotationRange.y),

Random.Range(-socket.rotationRange.z, socket.rotationRange.z)

);

}

*// Parent to socket*

prop.transform.parent = socket.transform;

}

}

**Dynamic Connection at Runtime**

csharp

public class RuntimeSocketConnection : MonoBehaviour {

*// For procedural dungeon generation*

public void BuildCorridorSection(List<CorridorPiece> pieces) {

GameObject previousPiece = null;

Socket previousExitSocket = null;

foreach (var pieceData in pieces) {

GameObject piece = Instantiate(pieceData.prefab);

if (previousPiece != null) {

*// Connect to previous piece*

var entrySockets = piece.GetComponent<ModularPrefab>().sockets

.Where(s => s.socketType == "corridor\_entry").ToList();

if (entrySockets.Count > 0 && previousExitSocket != null) {

SnapToSocket(piece, entrySockets[0], previousExitSocket);

}

}

*// Find exit socket for next piece*

previousExitSocket = piece.GetComponent<ModularPrefab>().sockets

.FirstOrDefault(s => s.socketType == "corridor\_exit");

previousPiece = piece;

}

}

void SnapToSocket(GameObject piece, Socket pieceSocket, Socket targetSocket) {

*// Calculate offset from piece origin to socket*

Vector3 socketOffset = pieceSocket.transform.localPosition;

*// Position piece so its socket aligns with target*

piece.transform.position = targetSocket.transform.position - socketOffset;

*// Rotate to match socket orientations (sockets should face opposite directions)*

Vector3 targetForward = -targetSocket.transform.forward;

Vector3 pieceForward = pieceSocket.transform.forward;

Quaternion rotation = Quaternion.FromToRotation(pieceForward, targetForward);

piece.transform.rotation = rotation \* piece.transform.rotation;

}

}

**Visual Socket Editor**

csharp

[CustomEditor(typeof(ModularPrefab))]

public class ModularPrefabEditor : Editor {

private bool isPlacingSocket = false;

private string newSocketType = "wall\_connector";

public override void OnInspectorGUI() {

DrawDefaultInspector();

ModularPrefab prefab = (ModularPrefab)target;

EditorGUILayout.Space();

EditorGUILayout.LabelField("Socket Tools", EditorStyles.boldLabel);

EditorGUILayout.BeginHorizontal();

newSocketType = EditorGUILayout.TextField("Socket Type:", newSocketType);

if (GUILayout.Button("Add Socket", GUILayout.Width(100))) {

AddNewSocket(prefab);

}

EditorGUILayout.EndHorizontal();

if (GUILayout.Button("Auto-Detect Edge Sockets")) {

AutoDetectSockets(prefab);

}

if (GUILayout.Button("Test Socket Connections")) {

TestSocketConnections(prefab);

}

}

void OnSceneGUI() {

ModularPrefab prefab = (ModularPrefab)target;

*// Visual socket editing in scene view*

foreach (var socket in prefab.sockets) {

if (socket.transform == null) continue;

*// Draw socket handle*

EditorGUI.BeginChangeCheck();

Vector3 newPos = Handles.PositionHandle(

socket.transform.position,

socket.transform.rotation

);

if (EditorGUI.EndChangeCheck()) {

Undo.RecordObject(socket.transform, "Move Socket");

socket.transform.position = newPos;

}

*// Label*

Handles.Label(socket.transform.position, socket.socketType);

}

}

void AutoDetectSockets(ModularPrefab prefab) {

Bounds bounds = GetPrefabBounds(prefab.gameObject);

*// Create sockets at edges*

CreateEdgeSocket(prefab, "left", new Vector3(bounds.min.x, 0, 0), Vector3.left);

CreateEdgeSocket(prefab, "right", new Vector3(bounds.max.x, 0, 0), Vector3.right);

CreateEdgeSocket(prefab, "front", new Vector3(0, 0, bounds.max.z), Vector3.forward);

CreateEdgeSocket(prefab, "back", new Vector3(0, 0, bounds.min.z), Vector3.back);

}

}

**Benefits of Socket System**

1. **Automatic Alignment** - Pieces snap together perfectly
2. **Validation** - Only compatible pieces can connect
3. **Flexibility** - Mix and match different piece types
4. **Props** - Torches, banners, etc. know where they can attach
5. **Procedural Generation** - Algorithm knows valid connections
6. **Visual Feedback** - See connection points in editor
7. **Runtime Assembly** - Build dungeons dynamically
8. **Prevents Errors** - Can't connect incompatible pieces

This is how games like Skyrim's dungeon kits work - each piece has defined connection points that ensure everything fits together seamlessly!

Retry

**J**

does this allow the floor to connect seamlessly too or just walls?

Edit

Great observation! The socket system I described was primarily for walls, but floors absolutely need their own connection system to look seamless. Here's how to extend it for floors:

**Floor Connection System**

csharp

public class FloorSocket : Socket {

[Header("Floor-Specific")]

public EdgeType edgeType;

public bool supportsBlending = true;

public Material edgeBlendMaterial;

public enum EdgeType {

Flat, *// Simple edge*

Beveled, *// Angled edge*

Interlocking, *// Puzzle-piece style*

Overlapping *// One floor overlaps another*

}

}

public class FloorPrefab : ModularPrefab {

[Header("Floor Connections")]

public FloorSocket northEdge;

public FloorSocket southEdge;

public FloorSocket eastEdge;

public FloorSocket westEdge;

[Header("Floor Blending")]

public bool hasEdgeGeometry = true;

public GameObject[] edgePieces; *// Separate meshes for edges*

public Material[] blendMaterials; *// For transitions between floor types*

[Header("Height Matching")]

public float floorHeight = 0f;

public bool autoAdjustHeight = true;

public float heightTolerance = 0.01f;

}

**Seamless Floor Connections**

csharp

public class FloorConnectionSystem : MonoBehaviour {

[System.Serializable]

public class FloorConnection {

public FloorPrefab floorA;

public FloorPrefab floorB;

public ConnectionType type;

public GameObject transitionPiece; *// Optional blend piece*

public enum ConnectionType {

Direct, *// Floors touch directly*

Blended, *// Smooth transition*

Strip, *// Transition strip between*

Gap, *// Small gap (like real tiles)*

Overlapped *// One slightly over other*

}

}

public void ConnectFloors(FloorPrefab floor1, FloorPrefab floor2, Direction direction) {

FloorSocket socket1 = GetFloorSocket(floor1, direction);

FloorSocket socket2 = GetFloorSocket(floor2, GetOppositeDirection(direction));

*// Align heights*

if (Mathf.Abs(floor1.floorHeight - floor2.floorHeight) < floor1.heightTolerance) {

*// Same height - create seamless connection*

CreateSeamlessConnection(socket1, socket2);

} else {

*// Different heights - need transition*

CreateHeightTransition(socket1, socket2);

}

}

void CreateSeamlessConnection(FloorSocket socket1, FloorSocket socket2) {

*// Adjust positions to eliminate gaps*

Vector3 gap = socket2.transform.position - socket1.transform.position;

float gapDistance = gap.magnitude;

if (gapDistance > 0.01f && gapDistance < 0.1f) {

*// Small gap - adjust position slightly*

socket2.transform.parent.position -= gap \* 0.5f;

}

*// Handle different floor types meeting*

FloorType type1 = socket1.GetComponentInParent<FloorPrefab>().floorType;

FloorType type2 = socket2.GetComponentInParent<FloorPrefab>().floorType;

if (type1 != type2) {

CreateTransitionStrip(socket1, socket2, type1, type2);

}

}

void CreateTransitionStrip(FloorSocket socket1, FloorSocket socket2,

FloorType type1, FloorType type2) {

*// Create a blend piece between different floor types*

GameObject transitionPrefab = GetTransitionPrefab(type1, type2);

if (transitionPrefab != null) {

Vector3 position = (socket1.transform.position + socket2.transform.position) / 2f;

GameObject transition = Instantiate(transitionPrefab, position,

socket1.transform.rotation);

*// Scale to fit gap if needed*

float length = Vector3.Distance(socket1.transform.position,

socket2.transform.position);

transition.transform.localScale = new Vector3(1, 1, length);

}

}

}

**Advanced Floor Blending**

csharp

public class FloorBlendingSystem : MonoBehaviour {

[Header("Blend Settings")]

public bool useVertexBlending = true;

public bool useTextureBlending = true;

public float blendWidth = 0.5f;

public void BlendFloorEdges(FloorPrefab floor, List<FloorPrefab> neighbors) {

MeshFilter meshFilter = floor.GetComponent<MeshFilter>();

if (meshFilter == null) return;

Mesh mesh = meshFilter.mesh;

Vector3[] vertices = mesh.vertices;

Color[] colors = new Color[vertices.Length];

*// For each vertex, check proximity to edges*

for (int i = 0; i < vertices.Length; i++) {

Vector3 worldPos = floor.transform.TransformPoint(vertices[i]);

*// Check distance to each neighbor*

foreach (var neighbor in neighbors) {

float distance = GetDistanceToFloorEdge(worldPos, neighbor);

if (distance < blendWidth) {

*// Set vertex color for blending*

float blendFactor = 1f - (distance / blendWidth);

colors[i] = Color.Lerp(colors[i], Color.white, blendFactor);

}

}

}

mesh.colors = colors;

*// Apply blend material that uses vertex colors*

ApplyBlendMaterial(floor, neighbors);

}

void ApplyBlendMaterial(FloorPrefab floor, List<FloorPrefab> neighbors) {

Material blendMat = new Material(blendShader);

*// Set textures from both floor types*

blendMat.SetTexture("\_MainTex", floor.floorMaterial.mainTexture);

if (neighbors.Count > 0) {

blendMat.SetTexture("\_BlendTex", neighbors[0].floorMaterial.mainTexture);

}

floor.GetComponent<MeshRenderer>().material = blendMat;

}

}

**Modular Floor Tiles**

csharp

public class ModularFloorTile : FloorPrefab {

[Header("Tile Configuration")]

public TileShape shape;

public TileConnectionMode connectionMode;

public enum TileShape {

Square, *// Standard square tile*

Hexagon, *// Hex grid*

Octagon, *// With small squares at corners*

Interlocking, *// Puzzle piece edges*

Irregular *// Natural stone look*

}

public enum TileConnectionMode {

Seamless, *// No visible seams*

Grouted, *// Visible grout lines*

Overlapped, *// Slight overlap*

Gapped *// Small gaps between*

}

protected override void SetupSockets() {

switch (shape) {

case TileShape.Square:

SetupSquareSockets();

break;

case TileShape.Hexagon:

SetupHexagonSockets();

break;

case TileShape.Interlocking:

SetupInterlockingSockets();

break;

}

}

void SetupInterlockingSockets() {

*// Create male/female connectors for puzzle-piece style*

*// North edge - male connector*

GameObject northSocket = CreateSocket("north",

new Vector3(0, 0, tileSize/2), Vector3.forward);

northSocket.GetComponent<FloorSocket>().gender = Socket.SocketGender.Male;

*// Add geometry for the interlocking tab*

GameObject tab = GameObject.CreatePrimitive(PrimitiveType.Cube);

tab.transform.parent = northSocket.transform;

tab.transform.localScale = new Vector3(0.3f, 0.1f, 0.1f);

tab.transform.localPosition = Vector3.forward \* 0.05f;

*// South edge - female connector*

GameObject southSocket = CreateSocket("south",

new Vector3(0, 0, -tileSize/2), Vector3.back);

southSocket.GetComponent<FloorSocket>().gender = Socket.SocketGender.Female;

*// Add geometry for the slot*

GameObject slot = GameObject.CreatePrimitive(PrimitiveType.Cube);

slot.transform.parent = southSocket.transform;

slot.transform.localScale = new Vector3(0.3f, 0.1f, 0.1f);

slot.transform.localPosition = Vector3.back \* 0.05f;

*// Make slot a trigger for receiving the tab*

slot.GetComponent<Collider>().isTrigger = true;

}

}

**Floor-to-Wall Connections**

csharp

public class FloorWallConnection : MonoBehaviour {

[Header("Baseboard/Trim")]

public GameObject baseboardPrefab;

public float baseboardHeight = 0.1f;

public bool autoGenerateBaseboard = true;

public void ConnectFloorToWall(FloorPrefab floor, WallPrefabMarker wall) {

*// Check if floor and wall meet properly*

float floorEdge = floor.transform.position.z + floor.tileSize/2;

float wallBase = wall.transform.position.z;

float gap = Mathf.Abs(floorEdge - wallBase);

if (gap < 0.01f) {

*// Perfect alignment*

if (autoGenerateBaseboard) {

CreateBaseboard(floor, wall);

}

} else if (gap < 0.1f) {

*// Small gap - create filler*

CreateFloorWallFiller(floor, wall, gap);

} else {

*// Large gap - might need special transition*

Debug.LogWarning($"Large gap between floor and wall: {gap}");

}

}

void CreateBaseboard(FloorPrefab floor, WallPrefabMarker wall) {

Vector3 position = wall.transform.position;

position.y = floor.transform.position.y + baseboardHeight/2;

GameObject baseboard = Instantiate(baseboardPrefab, position,

wall.transform.rotation);

*// Scale to match wall length*

baseboard.transform.localScale = new Vector3(wall.gridSize.x,

baseboardHeight,

0.05f);

*// Parent to wall for organization*

baseboard.transform.parent = wall.transform;

}

}

**Runtime Floor Assembly**

csharp

public class FloorAssemblySystem : MonoBehaviour {

[Header("Assembly Settings")]

public bool autoAlign = true;

public bool createTransitions = true;

public bool mergeStaticFloors = true;

public void AssembleFloorGrid(FloorTile[,] floorData, Transform container) {

Dictionary<Vector2Int, GameObject> floorObjects = new Dictionary<Vector2Int, GameObject>();

*// First pass - create all floor tiles*

for (int x = 0; x < floorData.GetLength(0); x++) {

for (int y = 0; y < floorData.GetLength(1); y++) {

if (floorData[x,y] != null) {

GameObject floor = CreateFloorTile(floorData[x,y], new Vector2Int(x, y));

floorObjects[new Vector2Int(x, y)] = floor;

}

}

}

*// Second pass - connect adjacent floors*

foreach (var kvp in floorObjects) {

Vector2Int pos = kvp.Key;

GameObject floor = kvp.Value;

*// Check all four directions*

ConnectToNeighbor(floor, floorObjects, pos, Vector2Int.up);

ConnectToNeighbor(floor, floorObjects, pos, Vector2Int.down);

ConnectToNeighbor(floor, floorObjects, pos, Vector2Int.left);

ConnectToNeighbor(floor, floorObjects, pos, Vector2Int.right);

}

*// Third pass - optimize*

if (mergeStaticFloors) {

MergeStaticFloors(floorObjects);

}

}

void ConnectToNeighbor(GameObject floor, Dictionary<Vector2Int, GameObject> floors,

Vector2Int pos, Vector2Int direction) {

Vector2Int neighborPos = pos + direction;

if (floors.ContainsKey(neighborPos)) {

GameObject neighbor = floors[neighborPos];

*// Get appropriate sockets*

FloorSocket mySocket = GetSocketForDirection(floor, direction);

FloorSocket theirSocket = GetSocketForDirection(neighbor, -direction);

if (mySocket != null && theirSocket != null) {

*// Create seamless connection*

AlignFloorSockets(mySocket, theirSocket);

*// Add transition if different floor types*

if (createTransitions) {

CheckAndCreateTransition(floor, neighbor, direction);

}

}

}

}

void MergeStaticFloors(Dictionary<Vector2Int, GameObject> floors) {

*// Group floors by material*

Dictionary<Material, List<GameObject>> floorsByMaterial =

new Dictionary<Material, List<GameObject>>();

foreach (var floor in floors.Values) {

Material mat = floor.GetComponent<MeshRenderer>().sharedMaterial;

if (!floorsByMaterial.ContainsKey(mat)) {

floorsByMaterial[mat] = new List<GameObject>();

}

floorsByMaterial[mat].Add(floor);

}

*// Merge meshes for each material*

foreach (var kvp in floorsByMaterial) {

MergeMeshes(kvp.Value, kvp.Key);

}

}

}

**Edge Case Handling**

csharp

public class FloorEdgeCases : MonoBehaviour {

[Header("Special Cases")]

public GameObject cornerPiecePrefab;

public GameObject tJunctionPrefab;

public GameObject fourWayPrefab;

public void HandleCorners(Vector2Int position, FloorTile[,] grid) {

*// Check what type of corner/junction this is*

bool hasNorth = HasFloorAt(grid, position + Vector2Int.up);

bool hasSouth = HasFloorAt(grid, position + Vector2Int.down);

bool hasEast = HasFloorAt(grid, position + Vector2Int.right);

bool hasWest = HasFloorAt(grid, position + Vector2Int.left);

int connections = (hasNorth ? 1 : 0) + (hasSouth ? 1 : 0) +

(hasEast ? 1 : 0) + (hasWest ? 1 : 0);

GameObject piecePrefab = null;

Quaternion rotation = Quaternion.identity;

switch (connections) {

case 2:

if (hasNorth && hasSouth) {

*// Straight piece*

piecePrefab = straightPrefab;

} else if (hasEast && hasWest) {

*// Straight piece rotated*

piecePrefab = straightPrefab;

rotation = Quaternion.Euler(0, 90, 0);

} else {

*// L-corner*

piecePrefab = cornerPiecePrefab;

rotation = GetCornerRotation(hasNorth, hasSouth, hasEast, hasWest);

}

break;

case 3:

*// T-junction*

piecePrefab = tJunctionPrefab;

rotation = GetTJunctionRotation(hasNorth, hasSouth, hasEast, hasWest);

break;

case 4:

*// Four-way junction*

piecePrefab = fourWayPrefab;

break;

}

if (piecePrefab != null) {

Vector3 worldPos = GridToWorld(position);

Instantiate(piecePrefab, worldPos, rotation);

}

}

}

The key additions for seamless floors:

1. **Edge Matching** - Floors align perfectly at edges
2. **Height Alignment** - Floors match heights or create smooth transitions
3. **Material Blending** - Different floor types blend naturally
4. **Corner Pieces** - Special prefabs for junctions
5. **Floor-to-Wall** - Baseboards and trim pieces
6. **Grout/Gaps** - Realistic tile appearance
7. **Mesh Merging** - Combines floors for performance

This ensures your dungeon floors look professional with no visible seams or gaps!

Retry

**J**

And does this socket system work mainly with manual placement or is this specifically for the dungeon creation tool?

Edit

Great question! The socket system is actually designed to work in **BOTH** scenarios - it's flexible enough for manual placement in the Unity editor AND automated placement through the dungeon tool. Let me show you how:

**Dual-Purpose Socket System**

csharp

public class UniversalSocketSystem : MonoBehaviour {

[Header("Usage Mode")]

public PlacementMode mode = PlacementMode.Hybrid;

public enum PlacementMode {

Manual, *// Editor drag-and-drop only*

Automated, *// Tool-based only*

Hybrid *// Works with both*

}

*// The sockets work regardless of how the prefab is placed*

public List<Socket> sockets = new List<Socket>();

}

**Manual Placement in Unity Editor**

csharp

[ExecuteInEditMode]

public class ManualSocketSnapping : MonoBehaviour {

[Header("Manual Placement Settings")]

public bool enableSnapping = true;

public float snapRadius = 2f;

public bool showSnapPreview = true;

public Color validConnectionColor = Color.green;

public Color invalidConnectionColor = Color.red;

*// This runs while dragging prefabs in Scene view*

void OnDrawGizmos() {

if (!enableSnapping) return;

if (!Application.isEditor || Application.isPlaying) return;

*// Find nearby sockets while dragging*

if (Selection.activeGameObject == gameObject) {

FindAndHighlightNearbyConnections();

}

}

void FindAndHighlightNearbyConnections() {

ModularPrefab[] allPrefabs = FindObjectsOfType<ModularPrefab>();

foreach (var prefab in allPrefabs) {

if (prefab == this) continue;

foreach (var theirSocket in prefab.sockets) {

foreach (var mySocket in GetComponent<ModularPrefab>().sockets) {

float distance = Vector3.Distance(mySocket.transform.position,

theirSocket.transform.position);

if (distance < snapRadius) {

*// Show connection preview*

bool compatible = AreSocketsCompatible(mySocket, theirSocket);

Gizmos.color = compatible ? validConnectionColor : invalidConnectionColor;

Gizmos.DrawLine(mySocket.transform.position,

theirSocket.transform.position);

if (compatible) {

*// Show snap position*

Gizmos.DrawWireCube(theirSocket.transform.position,

Vector3.one \* 0.3f);

}

}

}

}

}

}

}

*// Custom Editor for manual placement*

[CustomEditor(typeof(ModularPrefab))]

public class ModularPrefabManualEditor : Editor {

private bool isSnapping = false;

private Socket selectedSocket = null;

public override void OnInspectorGUI() {

DrawDefaultInspector();

EditorGUILayout.Space();

EditorGUILayout.LabelField("Manual Placement", EditorStyles.boldLabel);

if (GUILayout.Button("Snap to Nearest Socket")) {

SnapToNearestCompatibleSocket();

}

if (GUILayout.Button("Auto-Align with Neighbors")) {

AutoAlignWithNeighbors();

}

EditorGUILayout.Space();

EditorGUILayout.LabelField("Quick Connect", EditorStyles.boldLabel);

*// Dropdown of nearby compatible pieces*

ShowNearbyCompatiblePieces();

}

void OnSceneGUI() {

ModularPrefab prefab = (ModularPrefab)target;

Event e = Event.current;

*// Ctrl+Drag for socket snapping*

if (e.control && e.type == EventType.MouseDrag) {

*// Find nearest socket*

Socket nearestSocket = FindNearestSocket(e.mousePosition);

if (nearestSocket != null) {

*// Preview snap position*

DrawSnapPreview(nearestSocket);

*// Snap on release*

if (e.type == EventType.MouseUp) {

PerformSnap(prefab, nearestSocket);

}

}

}

*// Right-click context menu*

if (e.type == EventType.ContextClick) {

GenericMenu menu = new GenericMenu();

menu.AddItem(new GUIContent("Snap to Socket"), false, SnapToNearestCompatibleSocket);

menu.AddItem(new GUIContent("Break Connection"), false, BreakCurrentConnection);

menu.AddSeparator("");

menu.AddItem(new GUIContent("Duplicate and Connect"), false, DuplicateAndConnect);

menu.ShowAsContext();

e.Use();

}

}

void DuplicateAndConnect() {

ModularPrefab original = (ModularPrefab)target;

*// Find an open socket*

Socket openSocket = original.sockets.FirstOrDefault(s => !IsSocketConnected(s));

if (openSocket == null) return;

*// Duplicate the prefab*

GameObject duplicate = Instantiate(original.gameObject);

duplicate.name = original.name + "\_Connected";

*// Find compatible socket on duplicate*

ModularPrefab dupPrefab = duplicate.GetComponent<ModularPrefab>();

Socket dupSocket = dupPrefab.sockets.FirstOrDefault(s =>

AreSocketsCompatible(openSocket, s));

if (dupSocket != null) {

*// Snap them together*

SnapSockets(openSocket, dupSocket);

*// Select the new object*

Selection.activeGameObject = duplicate;

}

}

}

**Automated Placement via Tool**

csharp

public class DungeonToolSocketPlacer : MonoBehaviour {

private DungeonData dungeonData;

private DungeonPrefabLibrary prefabLibrary;

public void PlaceWithSockets(Vector2Int gridPos, PrefabType type) {

GameObject prefab = GetPrefabForType(type);

Vector3 worldPos = GridToWorld(gridPos);

GameObject instance = Instantiate(prefab, worldPos, Quaternion.identity);

*// Auto-connect to adjacent pieces*

AutoConnectToNeighbors(instance, gridPos);

}

void AutoConnectToNeighbors(GameObject newPiece, Vector2Int gridPos) {

ModularPrefab modular = newPiece.GetComponent<ModularPrefab>();

if (modular == null) return;

*// Check all four directions*

CheckAndConnect(modular, gridPos + Vector2Int.up, Direction.North);

CheckAndConnect(modular, gridPos + Vector2Int.down, Direction.South);

CheckAndConnect(modular, gridPos + Vector2Int.left, Direction.West);

CheckAndConnect(modular, gridPos + Vector2Int.right, Direction.East);

}

void CheckAndConnect(ModularPrefab piece, Vector2Int neighborPos, Direction direction) {

GameObject neighbor = GetPrefabAt(neighborPos);

if (neighbor == null) return;

ModularPrefab neighborModular = neighbor.GetComponent<ModularPrefab>();

if (neighborModular == null) return;

*// Find matching sockets*

Socket mySocket = piece.GetSocketForDirection(direction);

Socket theirSocket = neighborModular.GetSocketForDirection(GetOpposite(direction));

if (mySocket != null && theirSocket != null) {

if (AreSocketsCompatible(mySocket, theirSocket)) {

*// Perfect alignment since we're on a grid*

ConnectSockets(mySocket, theirSocket);

}

}

}

}

**Hybrid Smart Placement**

csharp

public class HybridSocketPlacer : MonoBehaviour {

[Header("Hybrid Mode Settings")]

public bool assistManualPlacement = true;

public bool validateToolPlacement = true;

public bool suggestOptimalConnections = true;

*// Works with both manual and automated placement*

public void OnPrefabPlaced(GameObject prefab, PlacementSource source) {

ModularPrefab modular = prefab.GetComponent<ModularPrefab>();

if (modular == null) return;

switch (source) {

case PlacementSource.ManualDrag:

if (assistManualPlacement) {

AssistManualPlacement(modular);

}

break;

case PlacementSource.DungeonTool:

if (validateToolPlacement) {

ValidateAndFixPlacement(modular);

}

break;

case PlacementSource.Runtime:

AutoConnectAllSockets(modular);

break;

}

*// Always try to optimize connections*

if (suggestOptimalConnections) {

SuggestBetterConnections(modular);

}

}

void AssistManualPlacement(ModularPrefab piece) {

*// Find best orientation*

float bestScore = 0;

Quaternion bestRotation = piece.transform.rotation;

Vector3 bestPosition = piece.transform.position;

*// Try different rotations*

for (int angle = 0; angle < 360; angle += 90) {

piece.transform.rotation = Quaternion.Euler(0, angle, 0);

float score = CalculateConnectionScore(piece);

if (score > bestScore) {

bestScore = score;

bestRotation = piece.transform.rotation;

bestPosition = CalculateBestPosition(piece);

}

}

*// Apply best placement*

piece.transform.rotation = bestRotation;

piece.transform.position = bestPosition;

*// Show what connected*

HighlightConnections(piece);

}

float CalculateConnectionScore(ModularPrefab piece) {

float score = 0;

foreach (var socket in piece.sockets) {

*// Check for nearby compatible sockets*

Collider[] nearby = Physics.OverlapSphere(socket.transform.position, 1f);

foreach (var col in nearby) {

Socket otherSocket = col.GetComponent<Socket>();

if (otherSocket != null && AreSocketsCompatible(socket, otherSocket)) {

float distance = Vector3.Distance(socket.transform.position,

otherSocket.transform.position);

score += (1f - distance); *// Closer = better*

}

}

}

return score;

}

}

**Editor Tools for Both Workflows**

csharp

public class SocketSystemWindow : EditorWindow {

[MenuItem("Tools/Socket System Manager")]

static void Init() {

GetWindow<SocketSystemWindow>("Socket System");

}

private enum WorkflowMode {

Manual,

Automated,

Both

}

private WorkflowMode currentMode = WorkflowMode.Both;

void OnGUI() {

currentMode = (WorkflowMode)EditorGUILayout.EnumPopup("Workflow:", currentMode);

EditorGUILayout.Space();

switch (currentMode) {

case WorkflowMode.Manual:

DrawManualTools();

break;

case WorkflowMode.Automated:

DrawAutomatedTools();

break;

case WorkflowMode.Both:

DrawManualTools();

EditorGUILayout.Space();

DrawAutomatedTools();

break;

}

}

void DrawManualTools() {

EditorGUILayout.LabelField("Manual Placement Tools", EditorStyles.boldLabel);

if (GUILayout.Button("Enable Socket Snapping")) {

EnableSocketSnappingInScene();

}

if (GUILayout.Button("Connect Selected Objects")) {

ConnectSelectedObjects();

}

if (GUILayout.Button("Create Prefab Chain")) {

CreatePrefabChain();

}

}

void DrawAutomatedTools() {

EditorGUILayout.LabelField("Dungeon Tool Integration", EditorStyles.boldLabel);

if (GUILayout.Button("Validate All Sockets")) {

ValidateAllPrefabSockets();

}

if (GUILayout.Button("Generate Socket Map")) {

GenerateSocketConnectionMap();

}

if (GUILayout.Button("Test Random Assembly")) {

TestRandomAssembly();

}

}

void CreatePrefabChain() {

*// Let user manually build by clicking*

GameObject lastPlaced = Selection.activeGameObject;

if (lastPlaced != null) {

ModularPrefab modular = lastPlaced.GetComponent<ModularPrefab>();

if (modular != null) {

*// Show preview of where next piece would go*

foreach (var socket in modular.sockets) {

if (!IsSocketConnected(socket)) {

DrawSocketGuide(socket);

}

}

}

}

}

}

**Runtime Socket Usage**

csharp

public class RuntimeSocketManager : MonoBehaviour {

*// Works during gameplay too!*

public void OnPlayerPlacesPiece(GameObject piece, Vector3 position) {

*// Player manually placing during gameplay*

ModularPrefab modular = piece.GetComponent<ModularPrefab>();

*// Assist with placement*

Vector3 snappedPosition = FindBestSnapPosition(position);

piece.transform.position = snappedPosition;

*// Auto-connect*

ConnectToNearbyPieces(modular);

*// Validate placement*

if (!IsValidPlacement(modular)) {

*// Show invalid placement feedback*

HighlightInvalidPlacement(modular);

}

}

public void OnProceduralGeneration() {

*// Automated dungeon generation at runtime*

List<GameObject> pieces = GenerateDungeonPieces();

foreach (var piece in pieces) {

ModularPrefab modular = piece.GetComponent<ModularPrefab>();

AutoConnectAllSockets(modular);

}

}

}

**Benefits of Dual-Purpose System**

**For Manual Placement:**

* Visual feedback while dragging
* Snap-to guides
* Right-click quick connections
* Rotation helpers
* Connection validation

**For Automated Tool:**

* Guarantees proper connections
* Validates placement rules
* Optimizes orientations
* Handles complex junctions
* Batch processing

**For Both:**

* Same prefabs work everywhere
* Consistent connection rules
* Reusable validation logic
* Mix manual and automated in same dungeon
* Runtime modification support

This means you can:

1. **Build by hand** in Unity scene view with snapping assistance
2. **Paint in the tool** and get automatic connections
3. **Mix both** - tool generates base, manually tweak details
4. **Runtime building** - players can build with same rules

The sockets are just metadata on the prefabs - they work regardless of how the prefab gets placed!

Retry

**J**

So, if i place walls and floors manually, does the tool recognize my manual placement?

Edit

Great question! By default, NO - the tool won't automatically recognize manually placed prefabs. But we can definitely make it work both ways with a synchronization system. Here's how:

**Bidirectional Synchronization System**

csharp

[ExecuteInEditMode]

public class DungeonDataSynchronizer : MonoBehaviour {

[Header("Sync Settings")]

public DungeonData dungeonData;

public bool autoSyncToData = true;

public bool autoSyncFromData = true;

public float syncInterval = 0.5f; *// How often to check*

[Header("Recognition")]

public string wallTag = "DungeonWall";

public string floorTag = "DungeonFloor";

private Dictionary<GameObject, Vector2Int> trackedObjects = new Dictionary<GameObject, Vector2Int>();

void OnEnable() {

if (Application.isEditor) {

EditorApplication.update += CheckForManualChanges;

}

}

void CheckForManualChanges() {

if (!autoSyncToData) return;

*// Find all manually placed dungeon pieces*

ScanSceneForDungeonPieces();

*// Update data to match scene*

UpdateDataFromScene();

}

void ScanSceneForDungeonPieces() {

*// Find all walls*

GameObject[] walls = GameObject.FindGameObjectsWithTag(wallTag);

foreach (var wall in walls) {

RecognizeWall(wall);

}

*// Find all floors*

GameObject[] floors = GameObject.FindGameObjectsWithTag(floorTag);

foreach (var floor in floors) {

RecognizeFloor(floor);

}

}

void RecognizeWall(GameObject wallObject) {

WallPrefabMarker marker = wallObject.GetComponent<WallPrefabMarker>();

if (marker == null) return;

*// Convert world position to grid position*

Vector2Int gridPos = WorldToGrid(wallObject.transform.position);

WallOrientation orientation = GetWallOrientation(wallObject.transform.rotation);

*// Check if this wall exists in data*

var wallData = dungeonData.GetWall(gridPos, orientation);

if (wallData == null || !wallData.exists) {

*// This is a manually placed wall - add to data*

AddManualWallToData(gridPos, orientation, marker);

}

}

void AddManualWallToData(Vector2Int gridPos, WallOrientation orientation, WallPrefabMarker marker) {

*// Create wall data from the manual placement*

WallData newWall = new WallData {

exists = true,

type = marker.wallType,

prefabGuid = GetPrefabGuid(marker.gameObject)

};

*// Add to dungeon data*

dungeonData.SetWall(gridPos, orientation, newWall);

*// Mark as tracked*

trackedObjects[marker.gameObject] = gridPos;

*// Log for debugging*

Debug.Log($"Recognized manually placed wall at {gridPos}");

*// Mark data as dirty so it saves*

EditorUtility.SetDirty(dungeonData);

}

}

**Manual Placement Recognition Component**

csharp

[RequireComponent(typeof(WallPrefabMarker))]

public class ManualPlacementTracker : MonoBehaviour {

[Header("Tracking Info")]

public bool isTracked = false;

public Vector2Int gridPosition;

public int layer = 0;

public DungeonData linkedData;

*// Automatically register when placed*

void Start() {

if (Application.isEditor && !Application.isPlaying) {

RegisterWithDungeonData();

}

}

void RegisterWithDungeonData() {

*// Find dungeon data in scene*

DungeonDataSynchronizer sync = FindObjectOfType<DungeonDataSynchronizer>();

if (sync == null) {

Debug.LogWarning("No DungeonDataSynchronizer found - manual placement won't be tracked");

return;

}

linkedData = sync.dungeonData;

*// Calculate grid position*

gridPosition = WorldToGrid(transform.position);

layer = GetLayerFromHeight(transform.position.y);

*// Register based on type*

if (GetComponent<WallPrefabMarker>()) {

RegisterAsWall();

} else if (GetComponent<FloorPrefabMarker>()) {

RegisterAsFloor();

}

isTracked = true;

}

void RegisterAsWall() {

WallPrefabMarker marker = GetComponent<WallPrefabMarker>();

WallOrientation orientation = DetermineWallOrientation();

linkedData.SetWall(gridPosition, orientation, new WallData {

exists = true,

type = marker.wallType,

prefabGuid = GetPrefabGuid(gameObject)

});

Debug.Log($"Manual wall registered at {gridPosition}");

}

*// Update data when moved manually*

void OnTransformChanged() {

if (!isTracked || linkedData == null) return;

Vector2Int newGridPos = WorldToGrid(transform.position);

if (newGridPos != gridPosition) {

*// Remove from old position*

RemoveFromData(gridPosition);

*// Add to new position*

gridPosition = newGridPos;

RegisterWithDungeonData();

}

}

}

**Enhanced Dungeon Editor Tool**

csharp

public partial class DungeonPainterWindow : EditorWindow {

[Header("Scene Sync")]

private bool syncWithScene = true;

private List<GameObject> sceneObjects = new List<GameObject>();

void OnGUI() {

*// Add sync controls to toolbar*

EditorGUILayout.BeginHorizontal(EditorStyles.toolbar);

syncWithScene = GUILayout.Toggle(syncWithScene, "Sync Scene", EditorStyles.toolbarButton);

if (GUILayout.Button("Import from Scene", EditorStyles.toolbarButton)) {

ImportManualPlacements();

}

if (GUILayout.Button("Export to Scene", EditorStyles.toolbarButton)) {

ExportToScene();

}

EditorGUILayout.EndHorizontal();

*// Rest of UI...*

}

void ImportManualPlacements() {

int wallsFound = 0;

int floorsFound = 0;

*// Find all dungeon pieces in scene*

WallPrefabMarker[] walls = FindObjectsOfType<WallPrefabMarker>();

foreach (var wall in walls) {

if (!IsTrackedByTool(wall.gameObject)) {

Vector2Int gridPos = WorldToGrid(wall.transform.position);

WallOrientation orientation = GetOrientation(wall.transform);

dungeonData.SetWall(gridPos, orientation, new WallData {

exists = true,

type = wall.wallType

});

wallsFound++;

}

}

FloorPrefabMarker[] floors = FindObjectsOfType<FloorPrefabMarker>();

foreach (var floor in floors) {

if (!IsTrackedByTool(floor.gameObject)) {

Vector2Int gridPos = WorldToGrid(floor.transform.position);

int layer = GetLayer(floor.transform.position.y);

dungeonData.SetFloor(gridPos, layer, new GridFloorTile {

floorType = floor.floorType

});

floorsFound++;

}

}

Debug.Log($"Imported {wallsFound} walls and {floorsFound} floors from scene");

EditorUtility.SetDirty(dungeonData);

Repaint();

}

void DrawCanvas() {

*// Previous canvas drawing code...*

*// Overlay manually placed objects*

if (syncWithScene) {

DrawSceneObjectOverlay();

}

}

void DrawSceneObjectOverlay() {

*// Show manually placed objects in the 2D view*

Handles.color = new Color(0, 1, 0, 0.3f); *// Green overlay*

WallPrefabMarker[] walls = FindObjectsOfType<WallPrefabMarker>();

foreach (var wall in walls) {

Vector2Int gridPos = WorldToGrid(wall.transform.position);

Rect cellRect = GetCellRect(gridPos.x, gridPos.y);

*// Draw indicator*

EditorGUI.DrawRect(cellRect, new Color(0, 1, 0, 0.2f));

GUI.Label(cellRect, "M", EditorStyles.miniLabel); *// M for Manual*

}

}

}

**Smart Recognition System**

csharp

public class SmartDungeonRecognizer : EditorWindow {

[MenuItem("Tools/Dungeon/Recognize Scene Layout")]

static void ShowWindow() {

GetWindow<SmartDungeonRecognizer>("Scene Recognizer");

}

private DungeonData targetData;

private bool includeProps = true;

private bool preserveManualTweaks = true;

void OnGUI() {

targetData = EditorGUILayout.ObjectField("Dungeon Data", targetData,

typeof(DungeonData), false) as DungeonData;

includeProps = EditorGUILayout.Toggle("Include Props", includeProps);

preserveManualTweaks = EditorGUILayout.Toggle("Preserve Manual Tweaks", preserveManualTweaks);

if (GUILayout.Button("Analyze Scene", GUILayout.Height(30))) {

AnalyzeScene();

}

if (GUILayout.Button("Smart Import", GUILayout.Height(30))) {

SmartImport();

}

}

void SmartImport() {

*// Intelligently recognize dungeon layout*

*// Step 1: Find floor bounds*

Bounds dungeonBounds = CalculateDungeonBounds();

*// Step 2: Detect grid alignment*

float cellSize = DetectGridSize();

Vector3 gridOrigin = DetectGridOrigin(cellSize);

*// Step 3: Categorize all objects*

Dictionary<Vector2Int, List<GameObject>> objectsByGrid = new Dictionary<Vector2Int, List<GameObject>>();

GameObject[] allObjects = FindObjectsOfType<GameObject>();

foreach (var obj in allObjects) {

if (IsDungeonPiece(obj)) {

Vector2Int gridPos = WorldToGrid(obj.transform.position, gridOrigin, cellSize);

if (!objectsByGrid.ContainsKey(gridPos)) {

objectsByGrid[gridPos] = new List<GameObject>();

}

objectsByGrid[gridPos].Add(obj);

}

}

*// Step 4: Convert to dungeon data*

foreach (var kvp in objectsByGrid) {

ConvertCellToData(kvp.Key, kvp.Value);

}

EditorUtility.SetDirty(targetData);

Debug.Log($"Imported {objectsByGrid.Count} cells from scene");

}

float DetectGridSize() {

*// Find the most common spacing between objects*

List<float> spacings = new List<float>();

GameObject[] objects = FindObjectsOfType<GameObject>();

for (int i = 0; i < objects.Length - 1; i++) {

for (int j = i + 1; j < Mathf.Min(i + 10, objects.Length); j++) {

Vector3 diff = objects[j].transform.position - objects[i].transform.position;

*// Check X spacing*

if (Mathf.Abs(diff.z) < 0.1f && diff.x > 0.1f) {

spacings.Add(diff.x);

}

*// Check Z spacing*

if (Mathf.Abs(diff.x) < 0.1f && diff.z > 0.1f) {

spacings.Add(diff.z);

}

}

}

*// Find most common spacing (this is likely our grid size)*

return GetMostCommonValue(spacings, 0.1f);

}

}

**Live Sync Component**

csharp

[ExecuteInEditMode]

public class DungeonLiveSync : MonoBehaviour {

public DungeonData dungeonData;

public bool bidirectionalSync = true;

*// Track changes*

private Dictionary<GameObject, ObjectState> lastKnownStates = new Dictionary<GameObject, ObjectState>();

[System.Serializable]

class ObjectState {

public Vector3 position;

public Quaternion rotation;

public bool exists;

}

void Update() {

if (!Application.isEditor || Application.isPlaying) return;

*// Check for changes every frame in editor*

DetectChanges();

}

void DetectChanges() {

*// Check for new objects*

var currentObjects = FindObjectsOfType<WallPrefabMarker>();

foreach (var marker in currentObjects) {

if (!lastKnownStates.ContainsKey(marker.gameObject)) {

*// New object detected*

OnNewObjectDetected(marker.gameObject);

} else {

*// Check if moved*

var lastState = lastKnownStates[marker.gameObject];

if (Vector3.Distance(lastState.position, marker.transform.position) > 0.01f) {

OnObjectMoved(marker.gameObject, lastState.position);

}

}

}

*// Check for deleted objects*

var toRemove = new List<GameObject>();

foreach (var kvp in lastKnownStates) {

if (kvp.Key == null) {

OnObjectDeleted(kvp.Value);

toRemove.Add(kvp.Key);

}

}

foreach (var obj in toRemove) {

lastKnownStates.Remove(obj);

}

}

void OnNewObjectDetected(GameObject obj) {

Debug.Log($"New manual placement detected: {obj.name}");

*// Add to dungeon data*

SyncObjectToData(obj);

*// Track it*

lastKnownStates[obj] = new ObjectState {

position = obj.transform.position,

rotation = obj.transform.rotation,

exists = true

};

}

}

**Setup Instructions**

To make manual placements work with the tool:

1. **Add Components to Prefabs:**

csharp

*// On each wall/floor prefab, add:*

- WallPrefabMarker or FloorPrefabMarker

- ManualPlacementTracker

- Tag as "DungeonWall" or "DungeonFloor"

1. **Setup Scene:**

csharp

*// In your scene, create a GameObject with:*

- DungeonDataSynchronizer

- Link your DungeonData ScriptableObject

- Set sync settings

1. **Enable in Tool:**

csharp

*// In the Dungeon Painter window:*

- Click "Sync Scene" toggle

- Use "Import from Scene" to grab existing objects

- They'll now appear in your 2D tool view

Now you can:

* **Place manually** → See in tool
* **Paint in tool** → Updates scene
* **Move objects** → Data updates
* **Delete objects** → Data updates
* **Mix workflows** → Everything stays in sync

The key is having components that track the relationship between GameObjects and the data structure!

Retry

[Claude can make mistakes. Please double-check responses.](https://support.anthropic.com/en/articles/8525154-claude-is-providing-incorrect-or-misleading-responses-what-s-going-on)

'

Research

Opus 4.1