PROJECT MONSTER TAMING · 기술문서

몬스터를 만드는 도구부터, 파괴되는 전장의 규칙까지.

무엇이 막혔고, 어떻게 풀었는지 적었습니다.

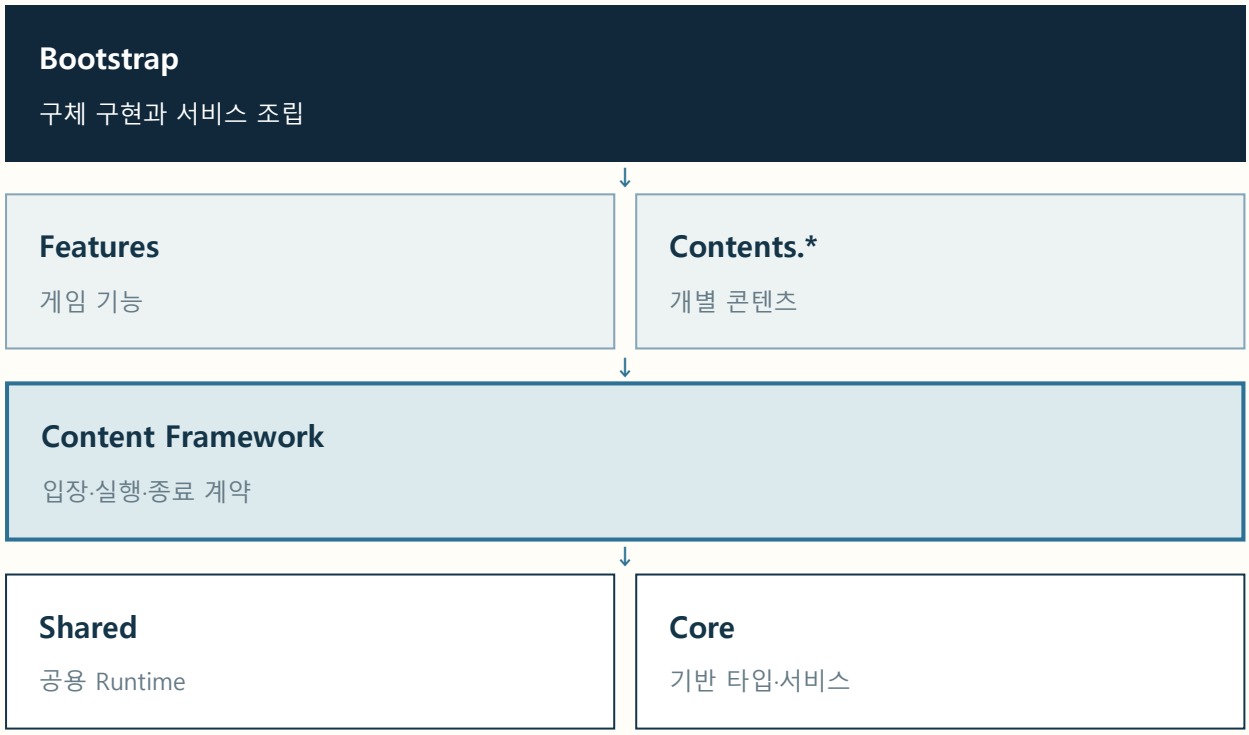
개발자	**전찬우**
프로젝트	**수집형 방치 RPG · Unity 6**
기간	**1개월 팀 프로젝트**
직접 범위	**시스템 구조 · 제작 도구 · 전투 · 공성 · 저장**

00 전체 구조

프로젝트 구조와 콘텐츠 실행 흐름

프로젝트 계층과 콘텐츠 실행 수명을 분리했습니다. 아래 두 그림은 각각 **누가 누구를 참조하는지**, 그리고 **한 콘텐츠가 어떻게 시작하고 끝나는지**를 보여줍니다.

계층별 참조 관계



의존은 아래 기반 계층을 향합니다. Bootstrap만 최상위에서 구체 구현을 조립합니다.

실행 수명

Feature Request
콘텐츠와 파티 선택

ContentFlow
중복 입장·Phase 관리

StartDataFactory

콘텐츠 입력 생성

Hosted / Separate Scene

같은 계약, 다른 실행 방식

Controller · Exit

결과 반환

ResultAdapter

공통 진행 변경으로 변환

IContentStartData·IContentController·IContentExit이 구현과 게임 흐름 사이의 경계입니다.

콘텐츠별 Manager를 늘리지 않은 이유

Hosted와 Separate Scene 모두 입장·종료·결과 전달이 필요합니다. 이를 각 콘텐츠에 복제하면 실패 처리와 복귀 규칙이 달라지므로, 공통 수명은 Framework가 소유하고 콘텐츠는 자기 Controller와 결과 타입에 집중하게 했습니다.

콘텐츠를 추가할 때 연결하는 요소

콘텐츠마다 시작 데이터와 Controller를 연결하고, 결과를 진행 데이터에 반영할 때 ResultAdapter를 사용합니다. Bootstrap은 구현과 서비스를 조립하고, ContentFlow는 입장부터 종료와 결과 전달까지의 순서를 관리합니다.

직접 맡은 범위

계층 경계, 콘텐츠 실행 계약, Hosted·Separate Scene 진입, 결과 Adapter 연결과 통합을 설계·구현했습니다.

관련 코드

콘텐츠 프레임워크의 계층 참조

ProjectMT.Contents.Framework.asmdef

+

콘텐츠 시작·결과·실행 수명 계약

ContentContracts.cs

+

Hosted와 Separate Scene을 조율하는 실행 흐름

ContentFlow.cs

+

01 Monster Maker

사람·팀원·AI가 공유하는 몬스터 제작 규격

팀원과 AI가 같은 제작 규칙으로 작업을 이어갈 수 있도록 Monster Maker를 만들었습니다. Draft와 Profile로 제작 입력을 관리하고, 미리보기에서 모션과 효과를 확인한 뒤 자산을 생성합니다.

제작 목적

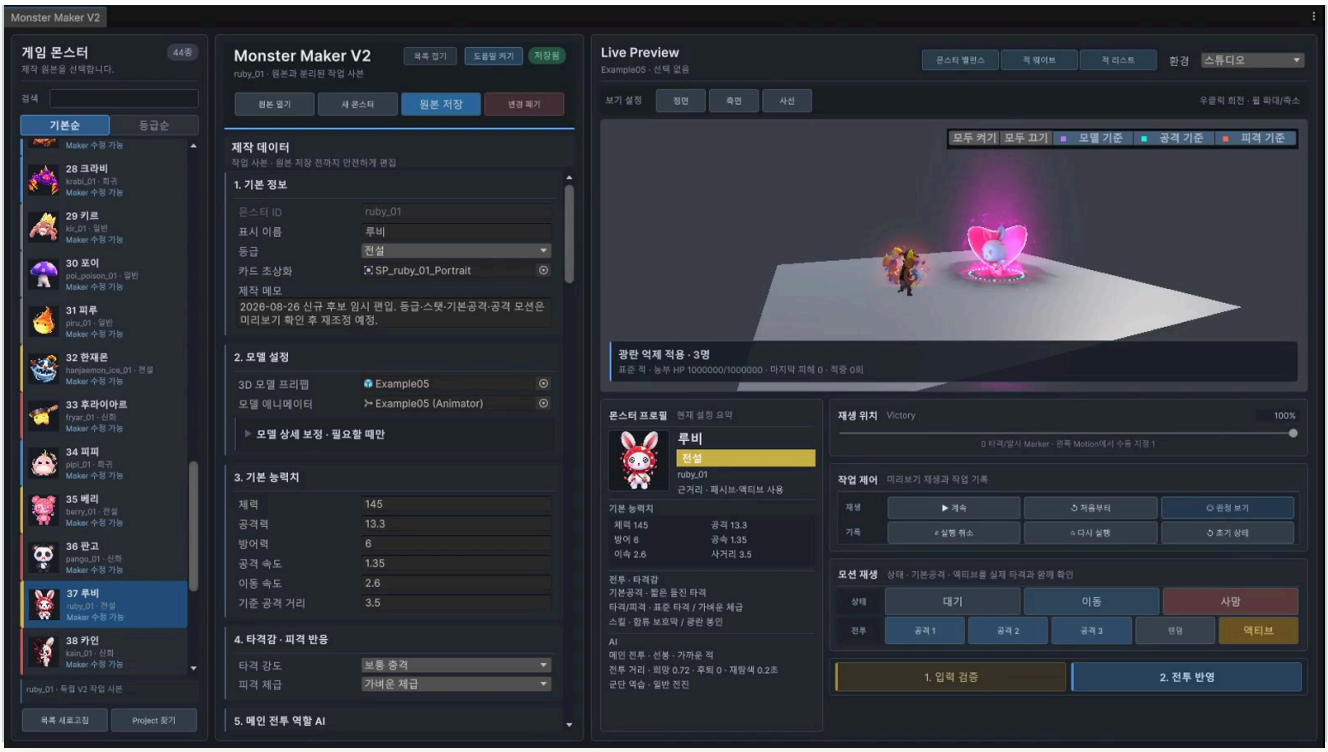
공통 제작 규격

사람이 만든 결과를 AI가 이어받고, 다른 팀원이 다시 검토할 수 있는 환경을 목표로 했습니다. 다양한 몬스터를 공통 입력 형식과 검증·저장 규칙에 맞춰 제작하도록 했습니다.

01 **같은 입력 규격**
ID·필드·Step·Binding 규칙

02 **같은 검증 단계**
검증·중복 차단·저장 경계

03 **이어지는 작업**
사람 → AI → 다른 사람



협업 구조

GUI와 API의 공통 작성 서비스

사람

Editor GUI

화면에서 설정·확인

AI

JSON Contract API

구조화된 요청 작성

↓ 입구에서 공통 데이터 계약으로 변환

SHARED AUTHORING BOUNDARY

MonsterActiveAttackAuthoringService

같은 검증 · 같은 생성/수정 · 같은 저장 위치

↓ 공통 계약을 통과한 결과만 저장

규격 검사

ID · Step · VFX/SFX

지원 조합과 중복 확인

공유 결과물

ActiveAttackProfile

누구나 다시 열어 수정

현재 AI 공통화 범위 공격 액티브 프리셋 검증·생성·수정

공격 액티브 프리셋의 GUI와 JSON API는 같은 작성 서비스를 호출합니다.

설계

문제와 해결 방식

문제

작업자마다 몬스터 생성·공격 구성·자산 등록 방식이 달라지면 결과를 이어받을 때마다 구조를 다시 해석해야 했습니다. 수작업 복제는 Runtime 참조 누락·GUID 변경·부분 저장 위험도 함께 만들었습니다.

적용 방식

Draft와 Profile을 공통 입력 계약으로 두고 Preview·Validator·Writer를 한 제작 순서로 연결했습니다. 공격 액티브 영역은 **사람용 공방과 AI용 API가 동일한 작성 서비스를 호출**하도록 구성했습니다.

다른 방법

문서와 체크리스트만 배포하거나 GUI와 API의 저장 로직을 각각 구현할 수도 있습니다. 규칙 변경 때 두 경로가 어긋날 수 있어, 작성 서비스와 검증 규칙을 코드의 공통 경계로 모았습니다.

규격화

몬스터 제작 규칙을 표준화했습니다

공통 규격을 사용하면 다른 작업자가 만든 몬스터도 익숙한 구조로 확인하고 수정할 수 있습니다.

01 · INPUT

입력 규격

ID, 입력 필드, 공격 Step과 Binding 키를 공통 형식으로 유지합니다.

02 · FLOW

작업 순서

Draft → Preview → Validate → Writer 순서로 확인과 생산을 연결합니다.

03 · QUALITY

04 · HANDOFF

검증 단계

지원 조합, 중복 ID, 소유권, 저장 결과와 Catalog 등록을 검사합니다.

작업 인계

같은 Profile과 Draft를 열어 사람과 AI의 결과를 이어서 조정합니다.

HUMAN ENTRY

Monster Workshop

`TryCreate / TryUpdate`

AI ENTRY

JSON Contract API

`Validate / Create / Update`

↓

ONE SERVICE

Authoring Service

`Validate · Save · Guard`

제작 작업

Monster Maker는 Draft에서 Runtime 자산을 만들고 Catalog에 등록합니다. JSON API는 공격 액티브 프리셋의 검증·생성·수정을 지원합니다.

처리 흐름

01

Preflight

입력·소유권 검사

02

Capture

파일·GUID·복구 범위

03

Write

대상 자산 저장

04

Reload

Definition 재검증

05

Register

Catalog·GUID 확인

06

Commit

전체 후조건 통과

실패 처리

사전 검사에서 오류가 발견되면 쓰기를 중단합니다. 생성 중 오류가 발생하면 확보한 파일의 복구를 시도합니다. 복구도 실패하면 생성 오류와 복구 오류를 함께 예외로 전달합니다.

코드에서 처리한 내용

- Draft와 Profile에 공통 입력 형식과 제작 순서를 고정했습니다.
- 공격 액티브 공방과 AI용 JSON API를 하나의 작성 서비스에 연결했습니다.
- 저장된 Definition의 참조, Catalog 등록 결과와 기존 GUID를 확인합니다.
- 후조건 통과 뒤 Commit하며, 예외와 검증 실패는 캡처된 범위의 복구 경로로 모읍니다.

GUI와 AI API가 같은 작성 서비스를 호출합니다

공격 액티브 공방은 새 프리셋 저장과 현재 프리셋 수정에 MonsterActiveAttackAuthoringService를 사용합니다. AI용 JSON API의 검증·생성·수정도 같은 서비스를 호출합니다. API가 제공하는 계약에는 지원하는 Enum, 요청 예시, Step 규칙과 VFX/SFX 생성 규칙이 포함됩니다.

제작자가 고르는 값과 도구가 확인하는 값

제작자는 모델·모션·타격 Marker·효과를 고릅니다. Writer는 입력값을 받아 Runtime에서 참조할 자산을 만들고 **저장 결과를 확인**합니다. 운영 경로에서는 기본 산출물 9개와 Catalog 4개를 다루며, 설정에 따라 고유 패시브·액티브 산출물이 추가됩니다.

쓰는 범위와 완료 조건

경계	실제 처리	막으려는 문제
쓰기 전	Draft 소유권·출력 경로를 검사하고 파일·폴더와 기존 GUID를 확보합니다.	다른 제작 원본의 산출물을 덮어쓰거나 복구 범위를 놓치는 문제
산출물 저장 후	Definition을 다시 불러와 Runtime 참조와 필수 값을 검사합니다.	메모리에서는 연결됐지만 저장 결과에는 누락된 참조
운영 등록 후	Catalog별 ID·등급·AI 연결과 기존 GUID를 확인한 뒤 Commit합니다.	파일만 있고 게임에서는 찾을 수 없는 자산
중간 실패	캡처한 파일·폴더 범위로 Rollback을 시도하고, 복구 실패도 별도 예외로 전달합니다.	부분 생성 상태를 성공으로 처리하는 문제

이번에 만든 자산만 저장합니다

BuildAndRegister는 이번 제작 대상 자산만 저장합니다. 다른 작업자의 미저장 변경은 그대로 둡니다.

남은 과제

몬스터 전체 Draft 작성과 Catalog 등록을 API로 연결하는 작업이 남아 있습니다.

담당 내용

Draft·Profile 규격, GUI·API 공용 작성 서비스, Preview, 자산 생성·검사, Catalog 등록과 Rollback을 구현했습니다.

관련 코드

생산 전 검증과 변경 범위 확보

MonsterMakerAssetWriter.cs

+

생성물·Catalog·GUID 검증과 복구

MonsterMakerAssetWriter.cs

+

몬스터 1종의 기본 산출물 경로

MonsterMakerAssetWriter.cs

+

사람용 공방이 호출하는 공통 저장 경계

MonsterWorkshopV2Window.cs

+

AI용 JSON API의 검증·생성·수정 경로

MonsterActiveAttackAuthoringApi.cs

+

GUI와 API가 공유하는 검증 서비스

MonsterActiveAttackAuthoringService.cs +

AI가 읽을 수 있는 작성 계약

MonsterActiveAttackAuthoringApi.cs +

02 공성 AI

소환한 전면을 공략하고, 가까운 병력은 경로를 공유합니다

소환 위치를 병력의 공략 방향으로 정합니다. **최대 6명의 느슨한 부대가 팀장의 경로 탐색을 공유**하고, 같은 전면 안에서 실제 이동·정렬 시간과 잔여 HP의 파괴 시간을 정수 ms로 비교합니다.

핵심 도식

가까운 병력이 팀장의 경로 탐색을 공유하는 방식



부대는 경로 탐색 결과를 공유하는 단위입니다. 임시 공격과 자리 변경 중에도 소속을 유지하고, 실제로 멀어지거나 연결이 끊어지면 다시 묶습니다.

설계 판단

소환 위치는 공략 방향이라는 약속

문제

플레이어는 눈앞의 벽을 공략하려고 그쪽에 병력을 소환합니다. 반대편 문이 싸다는 이유로 성을 크게 돌아가면 비용 계산이 빨라도 플레이어의 전략을 무시하게 됩니다.

선택

먼저 소환 측 전면의 벽·문·돌파구를 정하고, 그 안에서 실제 이동과 남은 HP의 파괴 시간을 비교합니다. 가까운 최대 6명은 왕궁에 가장 가까운 팀장의 경로를 공유합니다.

판단 기준

소환 전면을 먼저 확정한 뒤, 그 전면 안에서 이동과 파괴 비용을 비교합니다. 부대는 경로 캐시를 함께 쓰는 병력으로 묶습니다.

실제 공격 자리까지

열린 경로 이동·마지막 정렬 ms

+

현재 남은 내구도

잔여 HP ÷ 유효 DPS의 파괴 ms

처리 흐름

01
소환 전면
플레이어의 공격 방향

v

02
느슨한 부대
합류 가능한 최대 6명

v

03
팀장 경로
대표 탐색과 재사용

v

04

개인 접근

짧은 합류·개별 자리

05

지역 공격

포화 시 주변 다른 목표

06

선택적 갱신

영향받은 부대 재탐색

부대와 공격 목표

목표·예약 자리·부대 소속의 수명은 별개입니다. 목표나 공격 자리가 바뀌어도 부대 소속과 유효한 이동 계획을 유지합니다. 병력이 멀어지거나 통행 연결이 끊어지면 부대를 다시 구성합니다.

구현

- 소환 위치에서 왕궁 방향으로 향하는 전면의 첫 장애물과 인접 후보를 평가합니다. 비용은 소환 전면의 후보끼리 비교합니다.
- 같은 소환 방향에서 통행 가능한 3칸 안의 병력을 최대 6명으로 묶고, 기존 소속은 4칸까지 유지해 경계 왕복을 줄입니다. 가입 여부는 현재 위치에서 팀장까지의 통행 가능 거리로 판단합니다.
- 왕궁에 가장 가까운 현재 부대원이 팀장이며 동률은 소환 순번입니다. 사망·이탈 때 승계하고 현재 위치에서 합류 가능한 작은 부대는 병합합니다. 팀장이 바뀌어도 유효한 경로를 재사용합니다.
- 팀장의 열린 경로 탐색을 재사용하고 각 병력은 최대 3칸의 합류와 마지막 접근을 계산합니다. 접근 칸 내부의 고정 세 자리를 각각 예약하며 실제 몸체·성벽·공격선 검사는 유지합니다.
- 공격 자리는 현재 공략 중인 전면에서 찾습니다. 구조물 접근 경로는 현재 위치와 목표 사거리의 바깥 고리 중 큰 값 안에 있어야 합니다. 이 범위 밖으로 후퇴하는 자리만 남으면

기존 지역 대체 공격으로 넘어갑니다. 위협 대응·아군 지원과 마지막 몸체 회피는 별개입니다.

- 주변에 공격할 대상이 없으면 왕궁 방향으로 전진합니다. 전선이 열리면 기존 자리에서 계속 멈추지 않고, 이미 계산한 개인 합류 범위의 최대 3칸 안에서 같은 전면의 빈 공간으로 따라갑니다. 파괴되어 통행 가능한 구조물 자리도 포함하며 좁은 길목 점유 금지와 몸체·예약 검사는 유지합니다.

AI RUNTIME CONTRACT

공성 AI 판단·경로 실행 계약

사각 전장에서 다듬은 경로 공유와 느슨한 부대 구성을 육각 전장에 맞게 확장했습니다. 병력은 여섯 방향의 통행 규칙 안에서 이동하고, 공격과 수비도 같은 전장 규칙을 따릅니다.

플레이어 의도

소환한 쪽을 공략

소환 셀에서 왕궁 방향으로 만나는 첫 장애물을 돌파 대상으로 정합니다. 같은 전면의 인접 후보를 이동 시간과 남은 HP로 비교합니다.

느슨한 부대

최대 6명 · 현재 위치로 합류

개별 전체 탐색을 반복하지 않도록 근처 병력을 묶습니다. 팀장은 왕궁 최근접 병력이며 쉽게 가입·이탈·재합류합니다. 벽 양편처럼 짧은 통행이 불가능하면 분리합니다.

경로 공유

팀장 탐색 + 개인의 짧은 연결

팀장 경로의 부모 정보를 공유하고 가까운 합류점부터 재사용합니다. 사용 구간과 인접 셀의 통행 변화만 해당 부대 탐색을 무효화합니다. 합류 실패와 팀장 이동으로 새 탐색이 필요하면 해당 부대만 갱신합니다.

자리 포화

2·3순위와 주변 4순위 공격

같은 전면의 대체 목표에 실제 자리를 확보합니다. 그래도 막히면 해당 병력 주변의 같은 소환 전면에서 도달 가능한 대상을 비교합니다. 가장 짧은 이동 시간의 250ms 범위에서 HP가 낮은 대상을 고릅니다. 주변 벽도 포함하며 전략 소속은 유지합니다.

비용 재검토

현재 속도 · 남은 HP · 실제 DPS

접근·정렬 거리를 현재 속도로 나누고 잔여 HP 파괴 시간을 더해 정수 ms로 비교합니다. 1000ms 안에 합류 가능한 후보에만 자리에 정착해 실제 공격 중인 동료 DPS를 반영합니다.

행동 안정성

공격 직전의 불필요한 왕복 방지

임시 공격은 대상이 살아 있고 접근 가능한 동안 유지합니다. 위협 대응과 아군 지원을 우선하며, 대상이 사라지거나 접근에 실패하면 다시 판단합니다.

공성 병력의 실행은 enum 기반 FSM으로 관리합니다

병력마다 `초기 판단 대기 → 경로 이동 → 마지막 위치 정렬 → 공격`의 흐름을 갖습니다. 대기 계획을 받거나 목표가 사라지면 `Holding`, 이동이 막혀 진행하지 못하면 `Recovering`으로 전환해 다시 판단을 요청합니다.

전략 판단과 행동 실행의 책임도 나눴습니다. World가 목표와 경로를 결정하면 Unit은 현재 실행 상태에 맞춰 이동·정렬·공격을 처리합니다. 일곱 실행 상태의 전환 조건은 TickDynamicRuntime에서 처리합니다.

같은 전면의 벽과 닫힌문을 ms 단위로 비교합니다

`T_move=ceil(열린 경로·정렬 거리÷현재 속도×1000)`입니다. 유효한 실제 공격 동료의 DPS를 D라고 하면 도착 시 예상 HP는 `max(0, 현재 HP-D×접근 시간)`, 파괴 시간은 이를 `자신의 DPS+D`로 나누어 ms로 올림합니다. 동료 반영 조건 밖에서는 D=0이며 같은 비용은 좌표순으로 고정합니다.

열린문과 닫힌문은 게임에 설정된 통행 상태와 현재 HP를 그대로 사용합니다. 이 비교는 소환 전면을 정한 다음 실행합니다.

부대와 개별 전투의 수명을 분리합니다

부대는 경로 계산 공유 집단입니다. 슬롯 포화·대체 공격·위협 대응만으로 소속을 바꾸지 않습니다. 목표가 사라져도 몸체가 나갈 때까지 점유를 유지하며, 같은 슬롯이라도 접근 경로가 달라지면 새 이동 계획을 적용합니다.

탐색 비용과 행동 품질을 함께 봅니다

대표 전체 탐색, 공유 구간 사용, 개인 합류 탐색, 각 방문 칸 수를 따로 셉니다. 첫 타격까지 걸린 시간, 누적 이동, 2.5초 이상 멈춘 구간과 미공격 병력도 함께 봅니다.

▼ 측정 조건

일반 전투는 최대 24명, 50명과 100명은 별도 과부하 입력입니다.

측정 환경: Unity Editor.

▼ 개선 과정

개선 과정

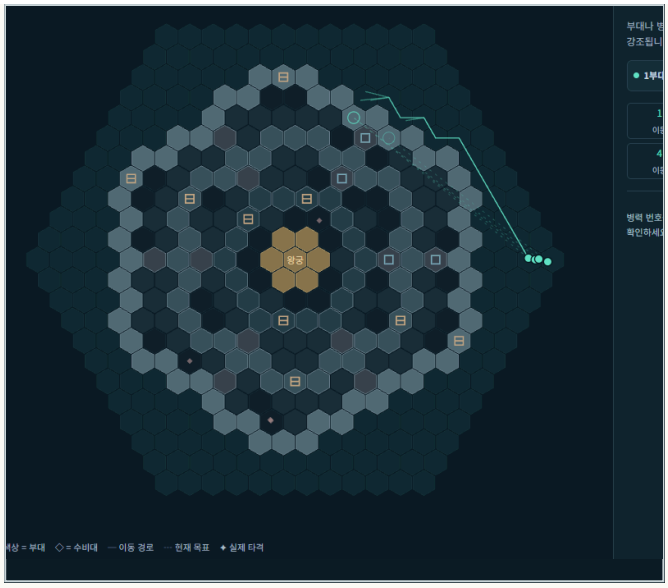
소환 방향과 경로 공유를 개선한 과정

계산상 유리한 선택이 플레이어의 의도와 어긋나는 문제가 있었습니다. 문제의 원인과 판단 기준을 바꾼 과정을 정리했습니다.

---

01. 저 벽을 부수라고 소환했는데, 왜 반대쪽으로 갈까?

성 오른쪽에 몬스터를 놓았는데 시작하자마자 위쪽 벽으로 길게 돌아갔습니다. 경로가 연결되어 있고 계산한 비용이 낮다는 조건만으로는 플레이어의 의도를 반영할 수 없었습니다. 플레이어는 앞의 벽을 공략하려고 그 위치에 소환했습니다. AI가 더 싼 다른 전면을 고르면 배치라는 전략 입력이 의미를 잃습니다.



목표를 고르는 순서를 바꿨습니다. 병력이 등록될 때 소환 셀을 전면 기준으로 저장합니다. 첫 경로는 그 셀에서 왕궁 쪽으로 한 칸씩 안쪽으로 진행하며, 옆으로 덜 벗어나는 칸을 고릅니다. 그 선에서 처음 만난 통과 불가능한 벽이나 문을 첫 돌파 대상으로 확정합니다. 이 단계에서는 멀리 있는 저HP 구조물을 전역 비용 경쟁에 포함하지 않습니다.

같은 소환 전면의 첫 장애물과 인접 돌파 후보를 다음 순서로 비교합니다. 예약 가능한 공격 자리까지의 열린 경로 이동·마지막 정렬 시간을 이동 속도로 환산하고, 현재 남은 HP를 해당 병력의 유효 DPS로 나눈 파괴 시간을 더해 정수 ms로 평가합니다. 소환 위치가 공략 방향을 정하고, 비용 계산은 그 방향 안의 빠른 목표를 정합니다.

실제 구현 경로

`HexCastleAssaultWorld`가 병력별 `deploymentFronts[InstanceId]`를 보존합니다. `HexCastleAssaultNavigation.TryResolveFrontRoute`는 매 스텝에서 `DistanceFromOrigin`이 감소하는 이웃만 허용하고, 소환점→왕궁 축과의 외적 절댓값이 가장 작은 셀을 고릅니다. 처음 만난 `IsBlocked && IsDamageable` 셀을 `FirstObstacle`로 기록합니다.

```
front = deploymentFronts[unitId]
next  = inwardNeighbors(current)
        .min(lateralDeviationFromSpawnAxis)
candidates = FirstObstacle + sameLayerAdjacentBreachCells
scoreMs = ceil(approachPathSeconds * 1000)
        + ceil(remainingHp / effectiveDps * 1000)
```

`IsInDeploymentFront`의 방향 내적은 `0.5` 이상이어야 하고, `IsOnAssaultFront`는 첫 장애물 또는 같은 방어층의 인접 성벽만 통과시킵니다. `TryEstimateAdvance`는 예약 불가능한 공격 자리를 무한 대기 후보에서 제외합니다. 점수가 같으면 육각 좌표 순으로 고정해 같은 입력의 결과가 흔들리지 않게 합니다.

네 방향 회귀에서는 반대편 벽의 HP를 1로 낮춰도 소환 전면의 벽을 먼저 선택하도록 고정했습니다. 실제 집중 배치의 첫 접근은 위쪽 벽까지 23.587m·8700ms이던 기록에서 오른쪽 전면 5.587m·2450ms로 바뀌었습니다. 후보 범위를 플레이어가 지정한 전면으로 좁혀 해결했습니다.

---

02. 가까운 병력의 경로 계산 공유



병력이 가까이 붙어 있는데도 목록에는 1명, 2명짜리 부대가 많이 생겼습니다. 가까운 병력이 경로 계산을 공유하도록 부대를 느슨하게 구성하고 싶었습니다.

기존 경로 번호와 과거 합류 시점을 소속 조건으로 사용한 것이 원인이었습니다. 지금 서로 붙어 있고 합류할 수 있어도 이전 판단 기록이 다르면 별도 부대로 남았습니다. 공격 자리나 잠깐의 임시 목표가 달라질 때도 계산 집단이 잘게 갈라졌습니다.

현재 위치의 연결 가능성을 부대 구성 기준으로 바꿨습니다. 새 병력은 소환 방향이 같은 부대 중 팀장까지 실제 열린 길로 3셀 안에 도달할 수 있는 곳에 가입합니다. 내적 0.5 미만인 다른 전면과 벽 양편처럼 실제 통행이 끊긴 병력은 합치지 않습니다. 한 부대는 공격 자리 두 묶음을 감당할 수 있는 최대 6명으로 제한했습니다.

이미 가입한 병력은 4셀까지 유지해 경계에서 가입·탈퇴가 매 프레임 뒤집히지 않게 했습니다. 더 이른 작은 부대와 합칠 때도 모든 부대원이 그 팀장에게 3셀 안으로 연결되고 합친 인원이 6명 이하일 때만 병합합니다. 공격 목표·슬롯·잠깐의 지원 행동은 부대 소속과 분리했습니다.

팀장은 매 갱신 때 살아 있는 부대원 중 왕궁에 가장 가까운 병력으로 정하고, 동률이면 소환 순서를 사용합니다. 팀장이 죽으면 즉시 다음 병력이 승계하며 유효한 공유 경로는 유지합니다. 팀장만 전체 전면 경로를 만들고 부대원은 그 경로까지 최대 3셀의 짧은 연결과 마지막 공격 자리만 계산합니다. 일반 네 배치 3679프레임 전체 검사에서 최대 인원 초과와 팀장 누락은 모두 0이었습니다.

부대와 경로 캐시의 구현 계약

```
join = sameDeploymentFront(dot >= 0.5)
    && openConnectionToLeader <= 3 cells
    && memberCount < 6
keep = openConnectionToLeader <= 4 cells
leader = min(distanceToPalace, spawnOrder)
path = localBfs(maxSteps: 3) + leaderSharedTrunk + finalAttackSlot
```

`ResolveCohort`가 가입, `MaintainCohort`가 이탈과 작은 부대 병합, `RefreshCohortLeader`가 승계를 담당합니다. `TryResolveCohortRoute`는 부대의 `SharedRoute`와 `RouteVersion`을 관리합니다. `HexAssaultApproachPlanner.BeginDecision`은 팀장 기준 전체 BFS를 부대 ID에 한 번 저장하고, 부대원에게는 최대 3셀 지역 BFS만 실행합니다.

공유 경로를 실제로 사용한 셀은 `Dependencies`에 기록합니다. 구조물의 통행 상태가 바뀌면 `InvalidateSharedSearchesAt`이 변경 셀 또는 인접 셀을 사용한 부대 캐시만 폐기합니다. HP 구간만 달라지면 비용장을 갱신하고 모든 부대의 열린 경로를 다시 만들지 않습니다.

---

03. 경로 공유 후 줄어든 계산

팀장 한 번의 탐색 결과를 부대원이 재사용하고, 각자는 공유 경로에 합류하는 짧은 구간과 마지막 공격 자리만 계산하도록 했습니다. 사용 중인 경로에 영향이 있는 부대만 갱신해 구조 변경 때 전군이 다시 계산하는 일을 막았습니다.

같은 6명의 위치·목표·능력치에서 경로 공유만 켜고 껐습니다. 선택한 자리·경로·비용은 같았고, **전체 열린 경로 탐색은 6회에서 1회로, 5회(83.3%) 줄었습니다.** 개인의 짧은 합류 탐색 6회는 그대로 남았습니다. 전체 프레임 변화는 아래 성능표에 함께 제시합니다.

같은 판단 중 복원한 경로를 재사용하고 육각 충돌 계산의 반복 상수도 한 번만 만들었습니다. 범위와 이동 시간의 하한을 먼저 확인해 결과가 바뀌지 않는 정밀 검사를 건너뛰고, 진단 과정에서 병력마다 전체 명단을 다시 찾던 중복도 없앴습니다. 공격 후보·슬롯·충돌 판정은 그대로 유지했습니다.

측정값도 계산 단계별로 분리했습니다. `FullSearchCount / FullVisitedCellCount`는 팀장 전체 BFS, `SharedPathUseCount`는 공유 구간 사용, `LocalConnectionSearchCount / LocalVisitedCellCount`는 개인 합류 탐색을 셉니다.

▼ 전후 AI 시간·프레임 지연·탐색 수치

줄어든 계산과 전체 실행 변화

팀장이 전체 열린 경로를 한 번 찾고 부대원은 그 경로까지의 짧은 합류 구간과 마지막 공격 자리만 계산합니다. 성벽이 바뀌면 그 셀을 사용하는 부대의 경로만 다시 만듭니다. 슬롯·몸체·공격선 검사는 그대로 유지했습니다.

일반 24명 전후 비교. 같은 성, 시드, 소환 순서와 능력치로 예전 코드와 현재 코드를 각각 실행했습니다. 여러 수정이 함께 들어간 결과이므로 전체 변화만 비교합니다.

배치	AI 평균 ms	관측 차이(참고)	프레임 평균 ms	프레임 p99 ms	무행동 후보
한쪽 집중	8.972 → 2.877	6.095ms	30.564 → 19.846	64.948 → 48.258	0구간
양쪽 분산	9.233 → 2.076	7.157ms	30.776 → 19.224	81.148 → 49.316	0구간
사방	8.802 → 1.823	6.979ms	29.408 → 19.748	67.592 → 50.696	0구간
무작위	8.998 → 1.067	7.931ms	30.136 → 19.304	73.521 → 53.444	0구간

여기서 무행동은 2.5초 이상 대기·복구 중 이동과 타격이 모두 없었던 구간입니다. 일반 네 배치에서는 0개였습니다. 공격하지 않았더라도 왕궁 쪽으로 움직였다면 무행동으로 세지 않았습니다.

탐색 공유 고정 대조. 같은 6명으로 공유만 켜고 껐습니다. 선택 자리·경로·비용은 같았고 전체 탐색은 6회에서 1회로 줄었습니다. 개인 합류 6회는 그대로입니다. **83.3%는 전체 탐색 호출 감소**이며 실제 프레임 변화는 위 표에서 따로 봅니다.

실행 결과 — 한쪽 집중: 왕궁 파괴 / 양쪽 분산: 왕궁 파괴 / 사방: 왕궁 파괴 / 무작위: 제한 시간 도달.

▼ 일반 24명 탐색·할당·GC

현재 네 실행의 누적 탐색 횟수와 방문 셀 수입니다. 프레임 할당량에는 전투 기록과 Editor 렌더링도 들어 있습니다.

배치	전체 탐색	공유 구간 사용	개인 합류	전체 / 지역 방문 셀	프레임 평균 할당 M
한쪽 집중	160	1275	696	32886 / 14826	0.555
양쪽 분산	149	1564	799	31807 / 14574	0.610
사방	104	1065	803	22883 / 12037	0.617
무작위	162	1414	878	36588 / 12390	0.675

▼ 24·50·100명 동시 소환

게임에서는 최대 24명을 사용합니다. 50명과 100명은 공격 자리 경쟁과 길목 혼잡을 보기 위한 과부하 조건입니다.

조건	AI 평균 ms	프레임 p95 / p99 ms	정지 후보 / 미공격	결과
focus-24	2.821	34.889 / 42.118	0 / 0	왕궁 파괴
split-24	2.348	33.941 / 45.520	2 / 0	왕궁 파괴
surround-24	2.189	36.261 / 50.629	1 / 0	왕궁 파괴
random-24	1.085	37.316 / 59.629	1 / 0	제한 시간 도달
focus-50	5.032	44.150 / 60.604	12 / 0	왕궁 파괴
split-50	5.259	49.881 / 72.160	9 / 0	왕궁 파괴
surround-50	5.406	55.568 / 94.953	4 / 0	왕궁 파괴

조건	AI 평균 ms	프레임 p95 / p99 ms	정지 후보 / 미공격	결과
random-50	3.467	47.678 / 83.305	1 / 0	왕궁 파괴
focus-100	6.686	63.697 / 88.412	430 / 3	왕궁 파괴
split-100	9.624	75.180 / 139.784	308 / 0	왕궁 파괴
surround-100	11.692	94.692 / 179.111	337 / 0	왕궁 파괴
random-100	10.682	98.898 / 175.953	265 / 1	왕궁 파괴

100명에서는 최장 9.7초의 대기·복구 구간과 높은 p99가 남았습니다. 게임 상한 24명 결과와 함께 이 수치를 다음 최적화 기준으로 사용합니다.

---

04. 평균과 p95·p99를 함께 본 이유

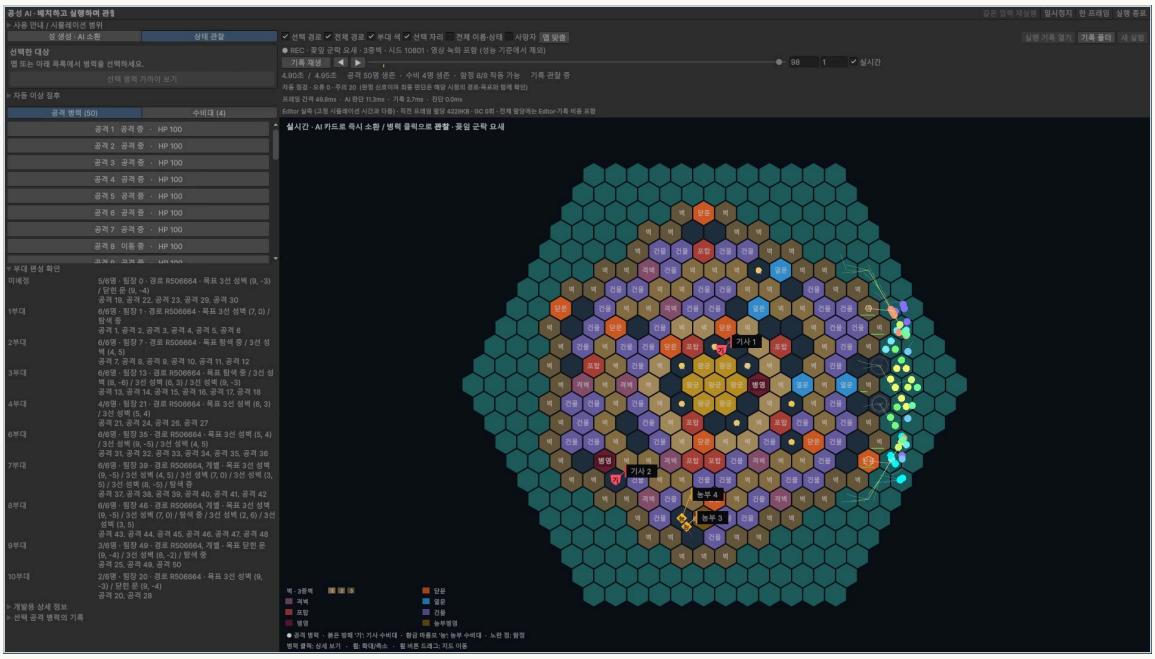
중간 집중 배치에서 AI 평균은 8.972ms에서 4.292ms로 줄었지만, 전체 프레임 p99는 64.948ms에서 78.476ms로 나빠졌습니다. 평균만 보면 놓치는 끊김이 있어 이후부터 평균과 p95·p99를 함께 기록했습니다.

> **p95**는 전체 프레임의 95%가 이 시간 안에 끝났다는 뜻이고, **p99**는 99%가 이 시간 안에 끝났다는 뜻입니다. 둘 다 낮을수록 좋으며, p99는 드물게 발생하는 큰 끊김을 확인할 때 사용합니다.

게임 상한인 24명과 별도로 50명, 100명을 한꺼번에 넣어 공격 자리 경쟁, 판단 대기, 길목 혼잡을 살폈습니다.

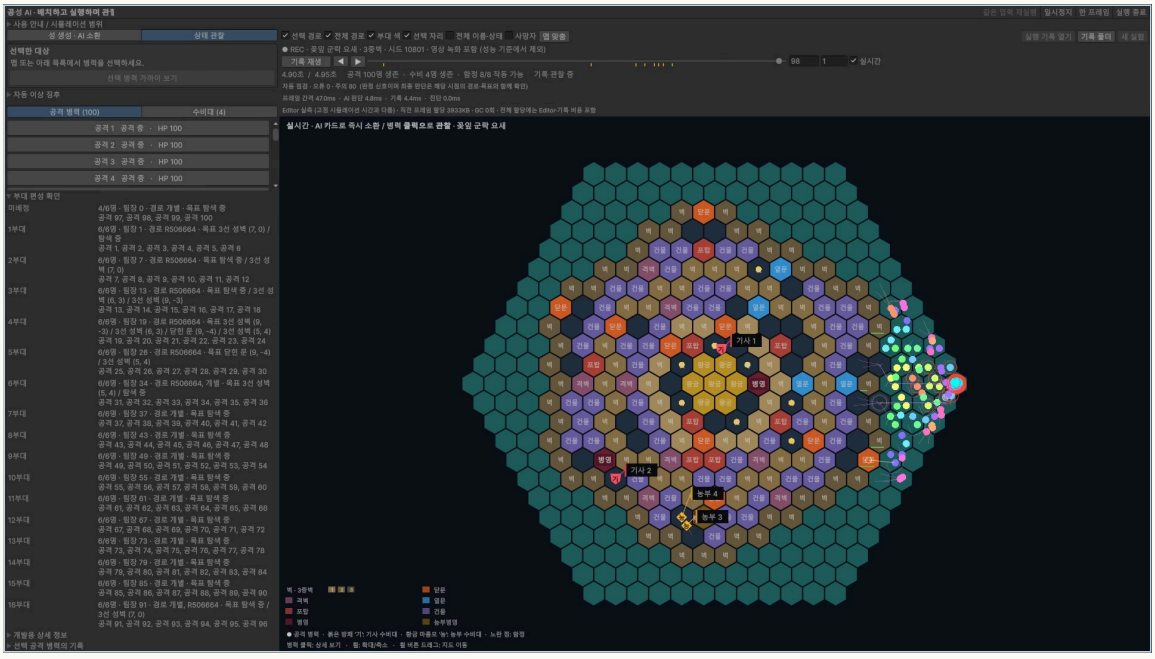
▼ 50·100명 직접 녹화

50·100명 스트레스 테스트 직접 녹화



50명 · 집중 → 양쪽 → 사방 → 무작위

약 149초 · 1배속 · 10fps 표본 · 무음입니다.



100명 · 집중 → 양쪽 → 사방 → 무작위

약 162초 · 1배속 · 10fps 표본 · 무음입니다.

05. 공격할 수 없을 때의 다음 행동

처음에는 한 번도 공격하지 못한 병력을 문제 지표로 많이 봤습니다. 기록을 따라가 보니 계속 전진한 병력도 있었고, 공격 자리로 접근하다 죽은 병력도 있었습니다. 공격 횟수 0만으로 무행동을 판단하면 다른 문제를 고치게 됩니다.

공격할 게 없으면 이동만 해도 괜찮다. 가만히 아무것도 안 하고 있는 상황만 없어지면 된다.

이 기준으로 다시 보니 실제로 고칠 분기가 보였습니다. 전선이 열렸는데도 예전 대기 자리를 계속 유지했고, 파괴되어 열린 성벽 자리도 전진 후보에서 빠졌습니다. 기존 공유 경로를 따라 가까운 빈 공간으로 전진하게 하고, 파괴된 자리는 현재 통행 가능 여부로 판단하도록 최소 수정했습니다. 좁은 길목을 점유하지 않는 조건과 몸 충돌 검사는 유지했습니다.

> **무행동 방지 분기**
>
> `TryCreateHoldPlan`은 병력이 이미 안전하게 이동 중이거나 유효한 공격 목표를 가진 경우 개입하지 않습니다. 둘 다 아닐 때 공유 전면 경로의 첫 장애물 접근점까지 남은 거리를 구하고, 현재 위치에서 최대 3셀 안에서 더 전진한 빈 자리를 예약합니다. 후보는 왕궁에서 더 멀어지지 않고 `IsInDeploymentFront`를 유지해야 합니다.
>
> `HexAssaultApproachPlanner.HoldingCell`은 현재 `Open(cell)` 값으로 통행 가능 여부를 검사합니다. 파괴된 성벽 자리도 통행 가능하면 후보가 됩니다. 이웃 6칸 중 열린 칸이 4개 미만이면 한 칸짜리 길목으로 보고 의도적인 대기 자리를 만들지 않습니다. 슬롯 예약·몸체 반경·마지막 이동 선분 검사는 공격 접근과 동일하게 유지합니다.

현재 방식에서는 공격 가능한 대상이 있으면 싸우고, 없으면 왕궁 쪽으로 움직입니다. 일반 24명 네 배치에서는 2.5초 이상 대기·복구 상태로 멈춘 구간이 없었습니다.

소환 전면을 먼저 고정하고, 그 안에서 부대 경로 공유와 개별 공격 자리를 계산하는 방식으로 정리했습니다.

혼잡 구간의 남은 과제

100명 집중 배치에서는 최장 9.7초의 대기·복구 구간이 남아 있습니다. 혼잡 구간의 공격 자리 경쟁과 재판단 대기를 줄이는 것이 다음 최적화 과제입니다.

직접 맡은 범위

소환 전면 판단, 6인 근접 부대와 팀장 경로 재사용, 개인 공격 자리, 이동·잔여 HP 비용, 지역 공격과 선택적 재검토, 전투 기록 재생 도구를 설계·구현했습니다.

관련 코드

소환 전면에서 처음 만나는 장애물의 대표 경로

`HexCastleAssaultNavigation.cs`

+

부대 접근 탐색 공유와 열린 자리로 전진

`HexAssaultApproachPlanner.cs`

+

같은 전면의 실제 이동·정렬과 잔여 HP 파괴 비용

`HexCastleAssaultWorld.cs`

+

자리 포화 시 지역 공격과 열린 전면의 대기 자리 갱신

`HexCastleAssaultWorld.cs`

+

Unity Navigation 비의존 계약

HexCastleGenerationTests.cs

+

최대 6명 근접 부대의 가입·팀장·경로 재사용

HexCastleAssaultWorld.cs

+

공성 병력의 일곱 실행 상태

HexAssaultExecutionContracts.cs

+

이동·정렬·공격·대기 상태 전환

HexCastleAssaultUnit.cs

+

03 육각 성 생성

내부 골격을 보존하고 전장 제약을 검사하는 육각 성 생성

육각 좌표에서 다양한 성의 실루엣을 만들되, 방어층 확장·진입 경로·시설 동선이 지켜야 할 조건을 별도로 검사합니다.



Unity 실행 화면

4중벽 전체 구조

성을 회전하며 네 개 방어층과 관문·시설 배치를 확인할 수 있습니다.

핵심 도식

생성과 검증을 나눈 처리 흐름



입력 조건으로 성 후보를 만든 뒤 전장 제약을 검사하고, 실행 배치와 회귀 비교값을 함께 남깁니다.

설계 판단

설계 배경

문제

2~4중 성벽, 왕궁, 관문, 격벽, 병영 출구와 배치 구역이 동시에 성립해야 했습니다. 모양만 그럴듯한 랜덤 배치는 막힌 출구나 끊어진 방어층처럼 플레이 불가능한 결과를 만들 수 있었습니다.

선택

실루엣과 방어층을 계획하고 육각 셀에 관문·시설·함정을 배치합니다. HexCastleValidator는 위상과 6방향 진입·시설 계약을 검사하며, StructureSignature와 LayoutSignature는 골격과 전체 결과의 변화를 나눠 비교합니다.

검토한 대안

검증된 성 Prefab을 미리 제작하면 안정적이지만 조합 수와 반복 플레이가 제한됩니다. 무제약 랜덤은 다양성을 늘리는 만큼 재현과 회귀 검증이 어려워, 규칙 기반 생성과 후보 검증을 함께 선택했습니다.

처리 흐름

01
Theme·Seed
재현 입력

02
Silhouette
폐곡선·방어층

03
Axial Grid
육각 셀 투영

04
Placement
관문·시설·함정

05
Validation
연결성·출구

06

Signature

회귀 비교

처리 결과

생성기는 성 배치와 배치 조건의 검사 결과를 함께 반환합니다.

구현

- 9개 테마의 생성 규칙과 2→4중벽 확장 시 안쪽 Ring 좌표 보존을 구현했습니다.
- 성벽 연결 수, 진입 방향별 방어층 통과, 병영 출구와 관문의 통행 계약을 검사합니다.
- 구조 서명과 전체 배치 서명을 나누고 같은 입력의 재현·다른 Seed의 변화를 회귀 테스트로 비교합니다.

테마와 난이도에 따라 전장을 생성합니다

Theme은 성의 윤곽을, Seed는 배치 변화를, 난이도는 방어층과 시설 조건을 정합니다. 생성 후 성벽 연결과 진입 경로, 병영 출구를 검사합니다.

2중벽을 4중벽으로 확장해도 안쪽 골격은 유지합니다

9개 테마의 실루엣 규칙을 각각 구현했습니다. 방어층을 늘릴 때 **기존 안쪽 Ring의 셀 목록은 유지**하고 바깥 Ring을 추가합니다. 방어층 확장 테스트에서 2→3→4중벽의 기존 Ring 좌표를 비교합니다. 실루엣 골격은 보존하고, 시설과 능력치는 난이도에 따라 바뀝니다.

2중벽	3중벽	4중벽
기존 Ring A · B	A · B 유지 + C	A · B · C 유지 + D

같은 Seed에서 방어층을 추가할 때의 골격 보존 관계입니다.

생성 후 확인하는 배치 조건

검사 대상	실제 검사	찾아내는 실패
성벽 위상	일반 성벽의 연결 수 2, 연결 타워의 분기 조건을 검사합니다.	끊어진 벽이나 잘못 남은 연결 타워
6방향 진입	외곽의 각 진입 방향에서 왕궁까지 경로와 방어층 통과 순서를 확인합니다.	도달 불가 또는 방어층을 건너뛰는 경로
병영 출구	인접 이동 가능 빈 셀이 2개 이상인지 확인합니다.	생성 직후 병력이 나올 수 없는 시설
관문 계약	열린 격벽 성문의 진영별 통행과 앞뒤 접근 셀을 검사합니다.	공격대·수비대의 통행 규칙 불일치
시설·함정	역할·등급·배치 Band와 난이도별 수량, 함정 중복·위치·간격을 검사합니다.	보기에는 배치됐지만 게임 규칙과 맞지 않는 구성

서명을 둘로 나눠 변경 원인을 좁힙니다

같은 Theme·Seed·난이도·생성 설정·규칙 버전으로 생성 결과를 재현합니다. 골격의 변화와 시설·능력치를 포함한 배치 변화를 두 서명으로 나눠 비교합니다.

비교값	포함 범위	사용 목적
StructureSignature	성벽 경로·왕궁 셀의 좌표와 구조·연결 상태	골격과 통행 구조가 변했는지 비교
LayoutSignature	전체 셀과 체력·보상·시설 등 배치 상태	내용까지 포함한 생성 결과를 비교

직접 맡은 범위

육각 좌표 기반 생성 규칙, 테마 실루엣, 방어층·배치 제약, 검증과 서명 흐름을 직접 설계·통합했습니다.

관련 코드

Seed·Theme·난이도에서 Layout 생성

HexCastleFoundationGenerator.cs

+

9개 테마 실루엣과 형식 검증

HexCastleSilhouettePlanner.cs

+

동일 Seed의 서명 재현과 테마 확장 보존

HexCastleGenerationTests.cs

+

테마·방어층·Seed 후보의 연결성 검사

HexCastleGenerationTests.cs

+

성벽 연결 수와 여섯 진입 경로의 방어층 순서 검사

HexCastleCandidate.cs

+

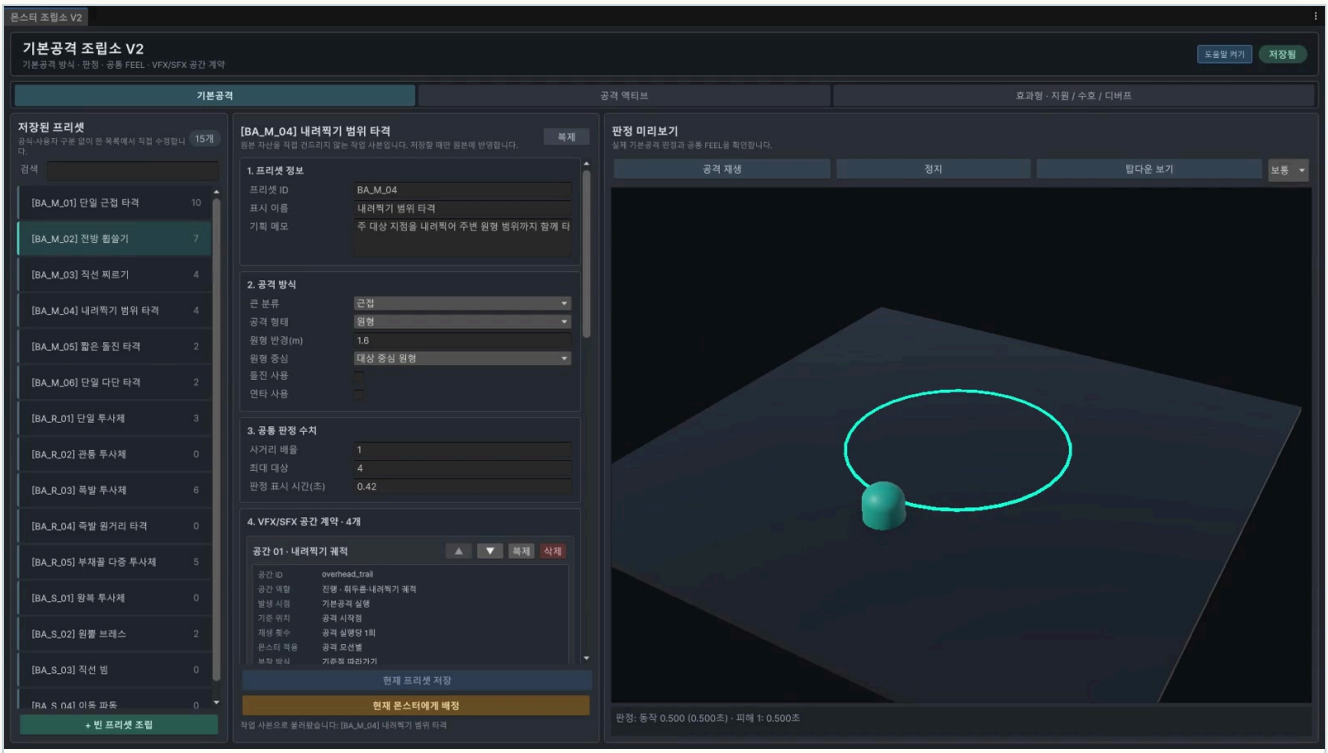
골격과 전체 배치를 서로 다른 범위로 계산하는 서명

HexCastleLayout.cs +

04 공용 전투

공격 규칙과 연출을 분리한 공용 전투 실행기

Profile은 공격 판정 규칙을, Binding은 몬스터별 모션·VFX·SFX 연결을 담당합니다. Validator에서 지원 조합을 확인한 뒤 공용 Executor로 실행합니다.



실제 화면

기본공격 조립소 · 판정 미리보기

공격 Recipe의 수치와 실제 판정 범위를 함께 확인합니다.

핵심 도식

판정 규칙과 표현 데이터의 합류

규칙

Attack Profile

전달 · 충돌 · 연타 · 이동 · 범위

표현

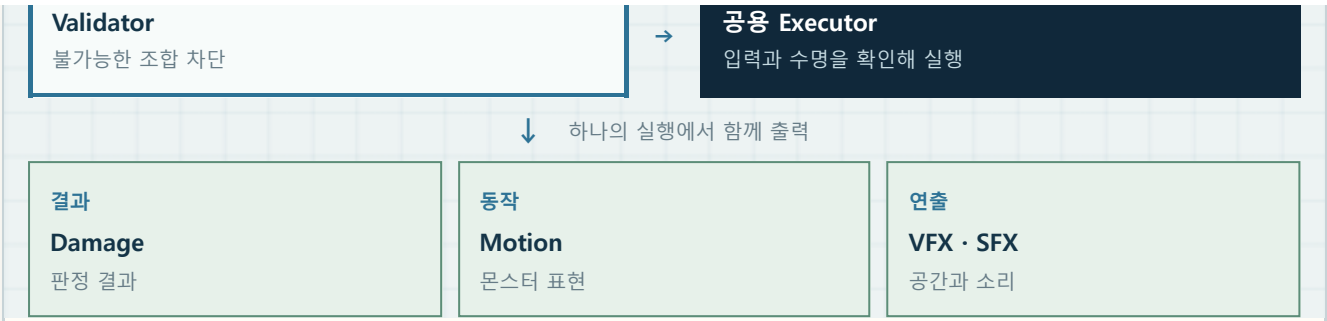
Monster Binding

모션 · VFX · SFX

↓ 지원 가능한 조합인지 먼저 확인

검증

해석



Profile은 공격 규칙을, Binding은 몬스터별 표현을 맡고, 검증된 조합만 공용 Executor가 실행합니다.

설계 판단

설계 배경

문제

몬스터마다 공격 MonoBehaviour를 만들면 피해 판정, 투사체, 다단 히트, 모션과 효과 연결이 함께 복제됩니다. 수정 범위가 커지고 실행기가 처리할 수 없는 데이터 조합도 자산에 들어갈 수 있었습니다.

선택

Delivery·Collision·Sequence·Movement·Shape를 독립 축으로 정의하고 Profile이 규칙을, Binding이 몬스터별 표현을 소유하게 했습니다. Validator가 지원하지 않는 조합을 자산 단계에서 차단한 뒤 공용 Executor가 해석합니다.

검토한 대안

공격마다 클래스를 두면 국소 수정은 쉽지만 공통 판정이 복제됩니다. 범용 노드 그래프는 제작·검증 도구의 범위를 늘립니다. 현재 필요한 동작을 명시적으로 지원하는 데이터 계약을 선택했습니다.

Delivery
Direct · Projectile · Traveling Area

Collision

Direct · Stop · Pierce · Area · Pass

Sequence

Single · Burst · Return

Movement

None · Dash

Shape

Single · Fan · Line · Circle

처리 흐름

01

Profile

판정 Recipe

∨

02

Validator

불가능 조합 차단

∨

03

Binding Key

Attack·Slot·Motion

∨

04

Executor

공용 실행 분기

∨

05

Runtime

Damage·VFX·SFX

처리 결과

Validator가 Profile의 조합을 검사하고, Executor는 통과한 입력으로 공격을 실행합니다.

구현

- Profile의 다섯 모듈 축과 피해 예산을 검증하고 Returning·Dash·Sweep 같은 조합별 불변식을 명시했습니다.
- Attack ID·Slot ID·Motion ID를 Binding Key로 사용해 같은 판정이라도 몬스터별 모션과 VFX를 연결합니다.
- Executor는 입력과 시전자 수명을 확인하며, 시간 분할 부채꼴 판정은 한 실행 안에서 중복 타깃을 제외합니다.

실제 Profile 세 개, 하나의 Executor

실제 자산	주요 조합	공용 실행 경로
BA_M_05 · 짧은 돌진 타격	Direct / DirectResolve / Single / Dash / Single	Dash 이동 처리 → 원점 재계산 → ApplyInstantHit
BA_M_06 · 단일 다단 타격	Direct / DirectResolve / Burst / None / Single, 피해 비율 0.3·0.3·0.4	첫 타격 → ContinueRepeatedHit으로 후속 단계 처리
BA_S_01 · 왕복 투사체	Projectile / PassThrough / ReturnPasses / None / Line, 피해 비율 0.6·0.4	LaunchProjectiles → Returning 이동과 왕복 피해 단계

Profile에서 지원 조합을 검사합니다

Returning은 ReturnPasses·단일 투사체·2단계 피해·PassThrough가 함께 있어야 합니다. Burst는 2단계 이상의 Direct 공격만 허용합니다. 방향성 Sweep은 Direct·Fan·Single이어야 하며 Dash는

양수 이동 거리가 필요합니다. 이 조건을 Profile 검증과 제작 흐름에서 확인하고, **Executor는 검증된 입력을 처리하는 역할에 집중**합니다.

Monster Maker와 연결되는 지점

Profile이 판정 규칙을, Binding이 Attack·Slot·Motion ID에 따른 몬스터별 모션·VFX·SFX 연결을 소유합니다. Maker에서 입력과 연결을 검증하고 Runtime에서는 같은 실행기를 사용합니다.

새 공격 동작을 추가할 때

기존 Profile로 표현할 수 없는 동작을 추가할 때는 데이터 정의와 조합 검사, Executor의 실행 처리도 함께 확장합니다.

직접 맡은 범위

공격 데이터 구조, Binding 연결, 지원 조합 검사와 공용 공격 실행기를 설계·구현했습니다.

관련 코드

공격 Recipe의 다섯 모듈 축

`MonsterBasicAttackProfile.cs`

+

지원하지 않는 모듈 조합 차단

`MonsterBasicAttackProfile.cs`

+

현재 Profile과 Motion에 필요한 Binding 투영

`MonsterBasicAttackBindingProjection.cs`

+

공격 Profile을 해석하는 공용 실행 분기

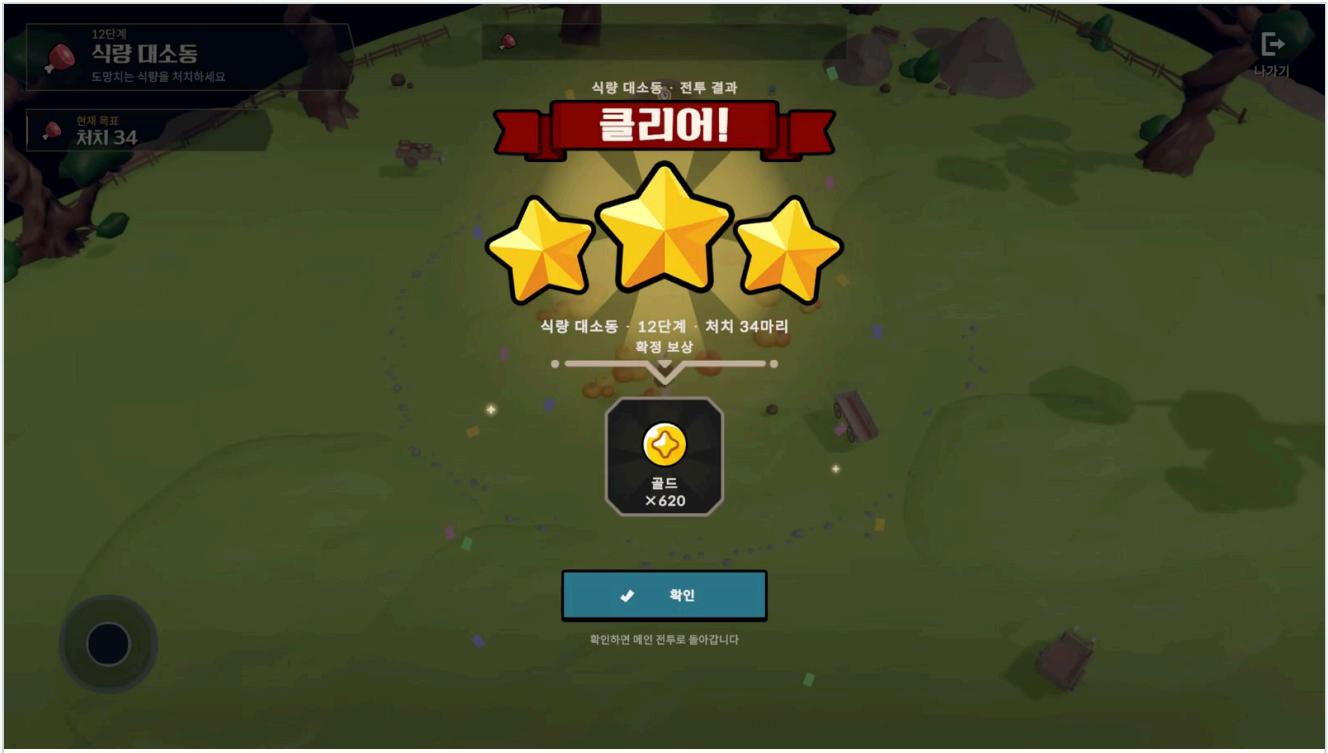
MonsterBasicAttackExecutor.cs

+

05 저장·정산

저장 완료를 기준으로 진행과 정산을 확정하는 구조

GameDataService는 현재 진행 데이터의 복제본에 변경을 적용하고 저장합니다. SaveAsync가 성공하면 복제본을 현재 데이터로 교체하고, ContentFlow가 보상 화면을 표시합니다.



실제 화면

콘텐츠 결과 · 저장 이후 보상 표시

완료 결과와 보상 UI는 정산 데이터의 저장이 끝난 뒤 열립니다.

핵심 도식

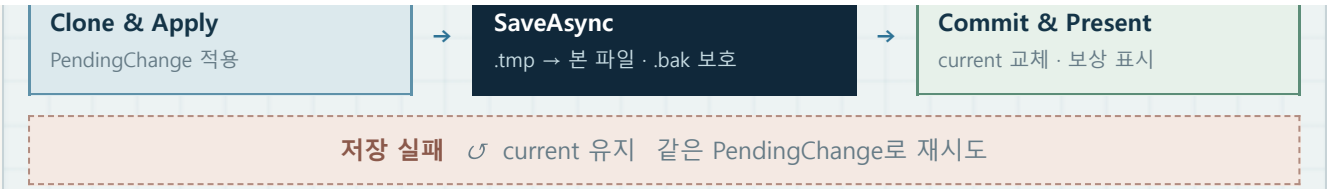
확정 상태와 변경 후보를 분리한 저장 트랜잭션

확정 영역 **current** 저장 성공 전까지 그대로 유지

변경 후보

파일 경계

확정



GameDataService가 저장 후 진행값을 확정하면 ContentFlow가 보상을 표시합니다.

설계 판단

설계 배경

문제

보상·성장·장비 변경은 여러 진행값을 함께 바꾸며, 콘텐츠 종료 콜백과 재시도가 중복 접수될 수 있습니다. 파일보다 메모리나 결과 화면을 먼저 바꾸면 쓰기 실패 뒤 손실·중복 정산이 발생할 수 있었습니다.

선택

GameDataService가 변경을 직렬화하고 current의 복제본에만 변경을 적용한 뒤 SaveAsync 완료 후 current를 교체합니다. 파일은 .tmp에 완성한 뒤 본 파일과 교체하고, ContentFlow는 종료 결과와 정산의 중복 진입을 각각 차단합니다.

검토한 대안

현재 값을 먼저 바꾸고 나중에 자동 저장하거나, 보상 화면을 먼저 보여준 뒤 저장할 수도 있습니다. 구현은 간단하지만 실패 후 어느 상태가 진실인지와 재시도에서 무엇을 다시 지급할지가 모호해져 채택하지 않았습니다.

처리 흐름

01

Exit Once

결과 중복 차단

02

Pending Change

동일 변경 유지

03

Serialize Gate

동시 변경 직렬화

04

Clone & Apply

후보 변경·검증

05

SaveAsync

파일 교체·후처리

06

Commit & Present

진행·보상 확정

처리 결과

저장이 실패하면 ContentFlow는 같은 PendingChange로 재시도합니다.

구현

- SemaphoreSlim으로 진행 변경을 한 건씩 처리하고, Clone–TryApply–Save–Commit 순서로 확정 전 상태를 분리합니다.
- SaveService가 직렬화 결과를 다시 역직렬화해 버전과 데이터를 검증한 뒤 AtomicFileStore에 전달합니다.
- ContentFlow의 ExitAccepted와 SettlementInFlight가 느린 콜백·연속 클릭의 중복 접수를 막고, 실패 시 동일 PendingChange로 재시도합니다.

두 개의 중복 방지와 하나의 확정 경계

ExitAccepted는 결과의 중복 접수를, SettlementInFlight는 정산 실행 중 재진입을 차단합니다. 저장이 실패하면 PendingChange를 유지해 동일한 변경으로 재시도합니다.

현재 남아 있는 오류 상황

상황	현재 동작과 한계
쓰기·교체 전 실패	후보가 current를 대체하지 않습니다. 콘텐츠 정산은 실패 상태를 표시하고 같은 변경으로 재시도합니다.
File.Replace 미지원	File.Copy로 덮어쓰는 대체 경로를 사용합니다. 이 경로는 원자적 교체가 아닙니다.
교체 후 정리 실패	본 파일 교체 후 .bak 삭제가 실패하면 저장 호출은 실패하지만, 디스크에는 새 데이터가 남고 메모리는 이전 값을 유지할 수 있습니다.
프로세스 중단·재실행	중복 차단은 현재 앱 실행 동안 유지됩니다. 재시작 후 중복 정산을 막는 영속 정산 ID는 아직 적용하지 않았습니다.

직접 맡은 범위

진행 데이터의 변경·저장 순서, 파일 교체 경계, ContentFlow의 중복 정산 방지와 재시도 연결을 설계·구현했습니다.

관련 코드

후보를 저장한 뒤 현재 진행을 확정 +

`GameDataService.cs`

직렬화 결과를 쓰기 전 재검증

SaveService.cs

+

.tmp에 완성한 뒤 본 파일과 교체

AtomicFileStore.cs

+

느리거나 중복된 콘텐츠 결과 무시

ContentFlow.cs

+

정산 중복 차단·재시도·저장 후 보상 표시

ContentFlow.cs

+

저장 실패 뒤 확정 진행값 유지 회귀 계약

CommanderGrowthAndStatsTests.cs

+