메이플스토리 모작

C++ · Win32 API 기반 2D 액션 RPG 개인 프로젝트

조현준

담당	게임 코드와 맵 편집 도구 단독 구현
게임 코드	게임 루프·장면/객체 관리, 이동·지형 충돌·전투·피격, 퀘스트·인벤토리·장비/UI
편집 도구	선 지형 입력·편집·저장·불러오기
사용 기술	C++ · Win32 API · GDI · STL · FMOD
실행	Windows x64 Release
자료 구성	전체 C++ 소스 · 실행 파일 · 맵 데이터

그림 1. Town 실행 화면

문서 구성

실행 흐름과 플레이 범위

마을·필드·보스전·엔딩을 연결하고, 이동·전투·퀘스트·인벤토리를 구현했습니다. 지형을 입력하고 저장하는 편집 장면도 함께 구성했습니다.

Logo → Menu → Town ⇄ Field · Town → BossRoom → Town → Ending

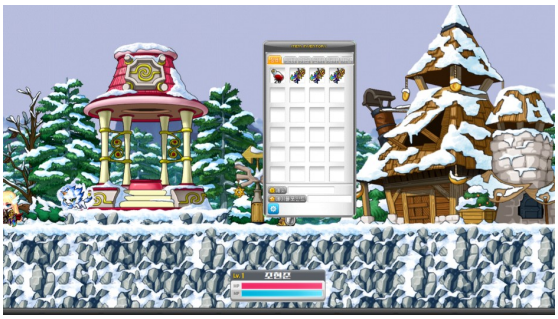

그림 2. 인벤토리 · 아이템과 장비

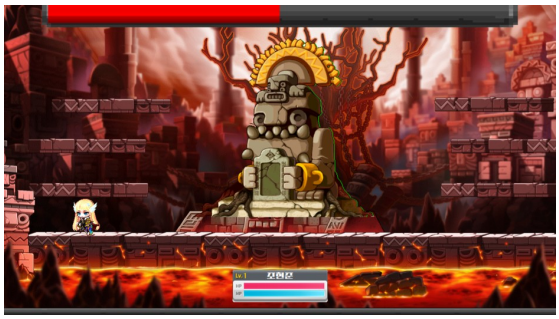

그림 3. BossRoom · 2페이즈 전투

주요 구현 범위

월드

5종 선 지형, 퀘스트 진행에 따라 열리는 포털, 몬스터 생성·아이템 드롭·플레이어 부활

전투

플레이어·몬스터 상태 전환, 기본 공격·Trinity·Soul Seeker, 이동 스킬, 보스 2페이즈

RPG·도구

퀘스트·24칸 인벤토리·장비/성장 UI, 지형 저장·불러오기, 지형/충돌 영역 디버그 표시

실행 구조와 역할 분담

루프와 공통 관리자

메시지가 없는 동안 MainGame이 Update → Late_Update → Render를 호출합니다. 시간·입력·이미지·사운드는 싱글톤 관리자로 구성해 여러 장면과 객체에서 같은 인스턴스를 사용합니다.

장면과 객체 생성

SceneMgr는 Edit를 포함한 7개 장면의 초기화·갱신·종료를 관리합니다. 각 장면은 필요한 지형과 객체를 준비합니다. 주요 객체는 템플릿 객체 팩토리로 생성하며, 위치·인덱스 설정 뒤 Initialize()를 호출하는 순서를 묶었습니다.

객체 갱신과 그리기

ObjMgr는 객체를 종류별로 등록하고, CObj*의 가상 함수를 통해 서로 다른 객체를 같은 순회 코드로 갱신합니다. Late_Update에서는 충돌과 렌더 후보 구성을 처리하고, Render에서는 렌더 그룹별 Y 정렬과 그리기를 수행합니다. 메모리 DC에 그린 결과는 BitBlt로 화면에 복사합니다.

프레임 시간과 상태 전환

프레임마다 달라지던 이동

프레임에 따라 이동이 달라지는 문제를 겪었습니다. 플레이어의 걷기·점프·낙하에 경과 시간을 반영했습니다.

TimeMgr는 QPC로 측정한 카운터 차이를 초 단위로 환산합니다. 플레이어의 주요 이동은 속도에 이 경과 시간을 곱해 계산합니다.

TimeMgr.cpp · Update_Time() 일부

```
QueryPerformanceCounter(&m_CurrentTime);
m_fTimeDelta = (float)(m_CurrentTime.QuadPart
    - m_LastTime.QuadPart) / (float)m_CpuTick.QuadPart;
m_LastTime = m_CurrentTime;
if (m_fTimeDelta > 0.1f)
{
    m_fTimeDelta = 0.1f;
}
```

한 번의 갱신에 반영하는 경과 시간은 최대 0.1초이며, 초과분은 이후에 보상하지 않습니다. Soul Seeker의 이동에는 프레임 단위 계산이 남아 있어 시간 기준을 맞추는 것이 개선 과제입니다.

상태의 시작·갱신·종료를 나누기

상태별 Initialize·Update·Release로 시작·갱신·종료를 구분했습니다. 상태가 바뀔 때는 이전 상태의 종료 처리 뒤 새 상태의 초기화를 호출합니다.

FSM.cpp · ChangeState() 일부

```
if (m_pPreState)
{
    m_pPreState->Release(pObject);
}
m_pCurrentState = newState;
m_pCurrentState->Initialize(pObject);
m_pPreState = m_pCurrentState;
```

Release()는 상태 종료 처리 함수로, 상태 객체를 삭제하지는 않습니다.

상태 클래스는 동작 가능 여부와 애니메이션을 설정하고, 플레이어 클래스는 걷기·점프·공격의 실제 처리와 일부 전환 조건을 담당합니다.

지형 판정과 편집 데이터

이전·현재 위치를 이용한 지형 판정

경사면에서 캐릭터가 예상한 경로를 벗어나는 문제를 겪었고, 이전·현재 위치를 활용해 판정을 보완했습니다.

지형 종류에 따라 판정 방식을 나눴습니다. 수평선과 벽에서는 이전·현재 캐릭터 경계가 선을 지났는지 비교합니다. 경사면에서는 선의 기울기와 캐릭터 위치로 지형 위에 있는지를 구하고, 이전·현재 판정이 바뀌는 조건에서 발 위치를 보정합니다.

Player.cpp · LineCollision()의 수직 벽 판정 일부

```
if (iter->eLineColList == LC_VERTICALLEFT &&
    m_fPrevX + m_tInfo.fCX / 2 <= m_ptLineStartPoint.x)
{
    if (m_tInfo.fX + m_tInfo.fCX / 2 >= m_ptLineStartPoint.x)
        m_tInfo.fX = (float)m_ptLineStartPoint.x - m_tInfo.fCX / 2;
}
```

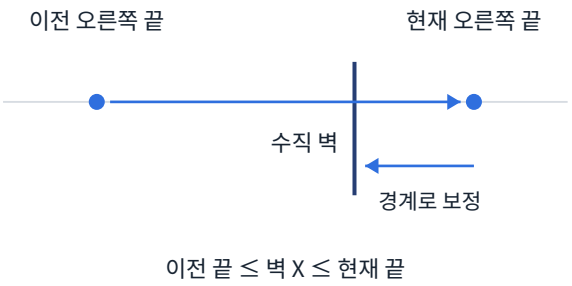

그림 4. 수직 벽 판정과 위치 보정

이전에는 캐릭터의 오른쪽 경계가 벽 왼쪽에 있었고, 현재는 벽에 닿거나 넘어간 경우 X 위치를 벽 경계로 맞춥니다.

선 지형별 조건에 맞춘 판정으로, 프레임 사이의 전체 이동 경로를 검사하지는 않습니다.

지형 편집과 게임 데이터 연결

편집 장면에서는 배경 위에 선의 두 끝점을 지정합니다. 마우스 좌표에서 스크롤 값을 빼 월드 좌표로 바꾸고, 선의 종류와 함께 저장합니다. 게임에서는 저장한 끝점과 종류를 읽어 판정용 선을 구성합니다.

수평선, 좌우 벽, 경사면, 줄의 다섯 종류를 지정할 수 있습니다.

그림 5. 지형 편집 화면
왼쪽 위에는 편집 정보, 바닥과 발판에는 입력한 지형선이 표시됩니다.

전투 규칙과 피격 처리

공격 설정과 피격 기록

피해량·치명타·반복 간격·횟수·수명은 HITBOXSTATS로 전달합니다. 공격 영역의 생성 시점과 이동은 공격별 코드로 처리합니다.

공격	판정 영역의 생성	공격별 동작
기본 공격	캐릭터 앞에 판정 영역 생성	공격 상태에서 생성 시점과 위치를 갱신
Trinity	스킬 동작에 맞춰 전방에 판정 영역 생성	스킬 모션과 이펙트에 맞춘 생성·위치 처리
Soul Seeker	한 번에 유도체 5개 생성, 각 유도체가 판정 영역 보유	가까운 대상 탐색과 추적 이동

피격 시점의 값을 기록하기

피격 등록 조건을 통과하면 Hitbox 포인터, 충돌 시각, 공격 설정의 복사본을 기록합니다. 이후 대상은 이 기록을 기준으로 피해 적용 간격과 횟수를 갱신합니다.

Grupin.cpp · ObjectCollision()의 피격 등록 일부

```
OBJCOLLISIONLIST tTemp = {};
m_pHitboxStats = pHitbox->Get_HitboxStats();
m_pHitboxStats->iCurPiercingCount++;
tTemp.CollisionTime = m_CollisionTime;
tTemp.pHitbox = pHitbox;
tTemp.tHitboxStats = *m_pHitboxStats;
m_listObjCollision.push_back(tTemp);
m_bHasHitboxCollision = true;
```

등록 조건 검사를 통과한 뒤 실행되는 부분입니다.

공격 설정의 형식은 공통이지만, 피격 처리는 몬스터와 보스 클래스에서 각각 수행합니다. 현재는 피격 처리 중인 대상에 새 Hitbox 등록이 제한됩니다. 여러 공격을 동시에 독립적으로 누적하려면 등록 조건과 기록 방식을 보완해야 합니다.

유도체의 대상 탐색과 이동

Soul Seeker는 Boss·Monster 목록의 유효한 대상을 거리 제곱으로 비교해 가까운 대상을 선택합니다. 이후 방향 벡터를 정규화하고, 속도를 누적하면서 최대 속도를 제한합니다. 대상 탐색과 이동은 유도체가 담당하고, 피해 설정은 다른 공격과 같은 형식으로 전달합니다.

사용 패턴으로 다시 본 STL 선택

현재 코드를 다시 보며, 삽입·삭제·조회 방식에 맞는 자료구조인지와 어떤 대안을 생각해 볼 수 있는지 정리했습니다.

STL 사용 예	현재 사용 방식	장단점 비교	대안과 확인할 점
list 게임 객체 렌더 목록	종류별 객체를 순회하며 사망 객체를 지웁니다. 렌더 목록은 매 프레임 모아 정렬합니다.	삭제한 원소 외의 반복자는 유지됩니다. vector::erase와 달리 뒤쪽 포인터를 옮기지 않습니다.	렌더 목록은 vector와 stable_sort 조합도 대안입니다. 객체 삭제 시 렌더 후보에서도 제거하는 처리는 유지해야 합니다.
vector 지형선 충돌 결과	선을 끝에 추가·삭제합니다. 충돌 검사에서는 선을 순회하고 결과를 모아 반환합니다.	list의 중간 삽입 이점보다 순회와 끝부분 편집이 중요합니다. 선 포인터와 충돌 결과를 각각 연속 보관합니다.	결과를 반환할 때 복사를 줄일 여지가 있습니다. 호출자 소유 버퍼를 재사용하는 방식도 대안입니다.
vector 인벤토리 24칸	슬롯 번호로 접근하며 빈 칸을 유지합니다. 드래그한 아이템은 빈 슬롯으로 옮깁니다.	슬롯 번호로 바로 접근하므로 list의 순차 접근이나 map의 키 검색이 필요하지 않습니다.	고정된 24칸에는 array<CItem*, 24>도 대안입니다. vector로 슬롯 수를 늘릴 때는 재할당에 따른 참조 무효화를 살펴야 합니다.
map BMP 이미지 자원	문자열 키로 등록하지만, 현재 조회는 find_if로 원소를 순회합니다.	키와 자원을 대응시키는 데 적합합니다. 정렬 순서는 쓰지 않아 unordered_map도 비교 대상입니다.	컨테이너 교체보다 map::find 사용이 먼저라고 봅니다. unordered_map으로 바꿀지는 실제 조회 비용을 비교할 문제입니다.
unordered_map 사운드 자원	이름으로 조회·재생합니다. 같은 키의 재로드가 성공하면 기존 Sound를 해제하지 않고 포인터를 덮어씁니다.	map과 달리 키 정렬을 유지하지 않습니다. 이름으로 해시 조회하며 평균 복잡도는 O(1)입니다.	조회 방식보다 중복 로드 처리가 우선입니다. 기존 자원 재사용과 해제 후 교체를 구분했으면 더 명확했을 것 같습니다.