NPC와 지형

MM4220 게임서버 프로그래밍

정내훈

내용

- NPC
- 지형구현
- 길찾기
- NPC-AI

NPC

- NPC
  - Non Playing Character
  - 예)
    - Monster
    - 상점 주인
    - 퀘스트 의뢰인
  - 서버 컴퓨터가 조작
    - 인공지능 필요

NPC(2023-월화)

- 인공 지능
  - 재미있는 행동/반응
  - 재미
    - Cheating사용은 재미를 떨어 뜨림
    - 너무 잘해도 곤란 : monster의 존재 목적?
  - 복잡한 AI : Script로 구현
  - 가장 기본적인 인공지능 : 충돌 방지 이동, 반격
  - 부하가 큰 기본 인공 지능
    - 길 찾기, 어그로 몬스터의 적 인식

NPC

- Script
  - 뒤에 더 자세히
  - 사용 목적 : 게임 제작 파이프라인 단축
    - 프로그래머의 개입 없이 기획자가 직접 작성/Test/수정
    - 서버 컴파일/리부팅 없이 수정/Test.
  - 서버프로그래머는 스크립트 언어 연동 시스템을 구현하고 샘플 스크립트를 작성해 주어야 한다.
  - XML, LUA 같은 언어를 많이 사용
    - 독자적인 언어를 쓰는 곳도 있음

NPC

- 적 자동 인식 (어그로 몬스터)
  - 몇 되지 않는 능동적 AI
  - 능동적 AI
    - 플레이어에 의하지 않는 자발적 동작
    - 서버에 막대한 부하. 절대 피해야 함
      - NPC 개수 10만
  - 실제 구현은 꺼구로
    - 플레이어의 이동을 근처 NPC에게 broadcast
      - 플레이어 생성, NPC생성 시에도 broadcast 필요
        - 움직이지 않으면 인식 못하는 경우가 생길 수 있음.

NPC

- NPC 구현
  - NPC 서버를 따로 구현 하는가?
  - NPC 서버 구현의 장점
    - 안정성 : NPC 모듈이 죽어도 서버 정상 작동
    - 부하 분산 : 메모리 & CPU
  - NPC 서버 구현의 단점
    - 통신 overhead,
      - 공유메모리 참조로 끝날 일이 패킷통신으로 악화.
      - 서버 입장에서는 NPC도 플레이어와 비슷한 부하

NPC

- NPC의 이동
  - NPC의 이동은 서버에서 관할한다.
    - 해킹 방지
  - 장애물 인식 필요
    - 서버에 지형과 장애물 정보가 있어야 한다.
- 지형 구현과 장애물
  - 2D, 3D?
  - Tile, Polygon?

내용

- NPC
- 지형구현
- 길찾기
- NPC-AI

지형 구현

- 클라이언트 만의 문제가 아니다.
- 비쥬얼로 끝나는 것이 아니다.
- 서버가 지형을 인식해야 한다.
  - NPC 이동의 구현을 위해
  - 플레이어 해킹 방지 (벽 뚫기)
- 클라이언트가 가지고 있는 모든 데이터를 서버에 복사하는 것은 과부하
  - NPC의 이동에 필요한 정보만 필요.
    - 클라이언트도 같은 정보 필요 => 아바타 이동에 사용
  - 충돌, 높이

지형 구현

- 서버에서만 지형충돌을 검사하면 되는가?
  - 클라이언트 에서도 Avatar 이동시 지형 충돌 필요
    - 서버에서의 검사 결과는 네트워크 딜레이가 있다
    - Avatar의 이동은 즉시 이루어 져야 한다. (UI 반응속도문제)
      - 클라이언트에서 지형 충돌 검사와 이동을 Delay없이 처리해야 한다.
    - 해킹 시도(벽 뚫기) 차단은 서버에서 사후 검증을 하면 된다.
    - 이렇게 하면 서버의 랙이 심한 경우에도 이동만은 랙 없이 할 수 있다.
    - 서버에서 Avatar 클라이언트로 보내는 이동 패킷을 생략할 수 있다.
    - 잘못된 이동 시
      - 해킹 : 강제 로그아웃, 계정 영구 정지
      - 오동작 : (문 닫힘 delay, 지형 버그) 클라이언트에 Roll-Back 패킷 전송
  - 이동 뿐만 아니라 아이템 집기나 화살 겨냥에도 장애물 검사 필요
    - 서버에서 거부하기 전 일차적으로 패킷 낭비 없이 오동작 원천 차단
    - UI 반응 속도 개선

지형 구현

- 2D와 3D게임의 난이도 차이가 크다.
- 메모리 용량과 검색 속도 둘다 중요
  - 메모리 용량
    - 지형의 정밀도를 결정
    - 지형의 밀도가 균등하지 않음 (특히 3D)
  - 검색 속도
    - N = 지형 데이터의 크기 = 면적 X 복잡도
    - 검색 속도 = O(1), O(N), O(logN), O(N^2)
    - 서버 부하 = O(검색 속도 X 동접)
  - 메모리 용량 vs 정밀도 vs 검색 속도는 서로 Trade Off

지형 구현

- 2D 지형
  - Tile방식
  - 2D 배열로 지형 표현
  - 이동 가능 불가능 flag이 cell마다 존재
  - 서버 안에서의 모든 Object의 좌표는 정수
  - 자로 잰듯한 줄서기(만!) 가능
  - 2D 이미지를 토대로 사람이 tile을 작성
    - 주로 레벨디자이너

지형 구현


천상카이져: 나 처럼
비류연v: 아녀 ''
쓰러버린다: ㅎㅎ
엘리오엔이: 형아 나 힘
헤헤히힛: 노노
POLISgirl: ——
왕사장: 호칭을 바꿔 줘세요
RKDTKDTJQM: 누가 나한테 흑단을
HP:323/361
MP:428/428
Normal zone
LEV:47 0%
-16
32497
엘리오엔이: 형아 나 힘
비류연v: 아녀 ''
왕사장: 백유황님
POLISgirl: ——
왕사장: 호칭을 바꿔 줘세요
헤헤히힛: 노노
RKDTKDTJQM: 누가 나한테 흑단을
지하1층 타로스의 지하내성
Normal zone

지형 구현

- 2D 지형
  - 2차원 배열로 표현 가능
  - `bool can_move[WORLD_WIDTH][WORLD_HEIGT]`
  - WORLD_WIDTH는 전체 맵의 크기 및 장애물 표현 정밀도로 결정됨
    - WORLD_WIDTH = 맵의 가로 크기 / 장애물 최소 크기
    - 예) 20 km / 50 cm = 40000
    - can_move의 크기 = 1600MByte
    - bool대신 bit를 사용하면 => 200MByte

지형 구현

- 3D 지형
  - 다층 지형을 위해 필요
    - 건물의 2층, 복잡한 던젼, 다리
  - 이동 시 높이 검사 필요
    - 이동 가능 경사
    - 머리 부딪힘 검사
  - 2가지 방식이 있음
    - 확장 타일
    - Polygon
  - 충돌 검사용 데이터 자동 생성 필수

지형 구현 (2024-화수)

- 3D 지형 확장 타일 방식
  - Tile방식
  - 3D 배열로 지형 표현
    - 배열 구현 시 메모리 낭비가 심해서 Sparse Matrix로 표현
      - 2D 지형을 기본으로 일부분만 3D로 표현
    - 여러 개의 2D 타일로 다층 구조 표현
      - 이동 가능 정보 이외에 높이 정보도 포함
      - 복잡한 입체 구조 표현 어려움 : 예) 창문
  - 서버 안에서의 모든 Object의 좌표는 실수
    - Float or Double?
    - 3D게임에서의 정수 좌표는 비쥬얼 적으로 error

지형 구현

- 3D 지형 Polygon 방식
  - 클라이언트의 visual data를 그대로 사용
    - 1차 가공을 통한 단순화 필요
      - 삭제 : 노말 벡터, uv값, 텍스쳐…
      - 삭제 : 통과 가능한 Object (풀, 안개, 커튼…)
      - 평면 폴리곤들의 병합
    - 어차피 최신 클라이언트 들은 Collision Polygon따로 요구
    - 이외로 메모리 사용량은 확장타일과 큰 차이가 없음
  - 클라이언트와 비슷한 방법으로 이동 가능 검사
    - 물리 엔진 필요
    - 지형 표현 정밀도 증가
  - 정수 좌표 불가능
  - Tile방식에 비해 속도는 떨어지지만 확장성 증가

지형 구현

- 장단점
  - 2d 타일
    - 압도적인 속도, 메모리 절약
    - 3차원 지형 불가
  - 2d 타일 확장
    - 빠른 속도
    - 복잡한 지형 표현 불가
  - 폴리곤
    - 느린 속도
    - 클라이언트와 똑같은 충돌 판단
      - 물리엔진 사용 필요
      - 3D 게임엔진의 Dedicated Server모드 사용가능 => 서버 부하 대폭 증가

지형 구현

- 지형 구현의 목적 : 충돌 & 길찾기
- 길찾기 자료구조 필요
  - 그래프 필요
    - Path Node
    - Path Mesh (Navigation Mesh)
  - 문의 구현
    - 특수 지형 속성?
    - NPC?
  - 자체 이동 지형
    - 엘리베이터, 배, 이동 발판

지형 구현

- 길찾기 정보

Path node

입력 지형

Path mesh

지형 구현

- Path Mesh 생성 from 폴리곤
  - 객체가 올라설 수 있는 폴리곤 필터링
    - 경사각, 주위 장애물 여부 검사
  - 연결된 폴리곤 합성
    - 최대한 큰 볼록 다각형으로 합성
    - 약간의 경사 차이 무시
    - 최적화를 위함 (메모리, 검색속도)
  - 다각형을 Node로하고, 연결 여부를 Edge로 한 그래프 생성
    - 높이 차와 경사를 고려한 일방통행 그래프 필요
    - Jump로 이동 가능하면 연결로 취급

내용

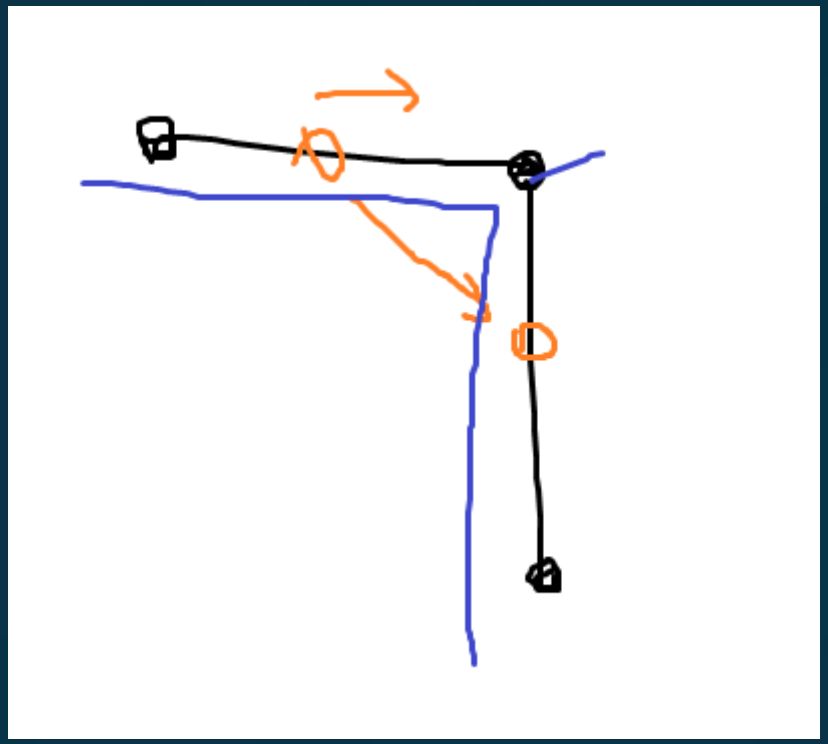
- NPC
- 지형구현
- 길찾기
- NPC-AI

길 찾기 (2023 화목)

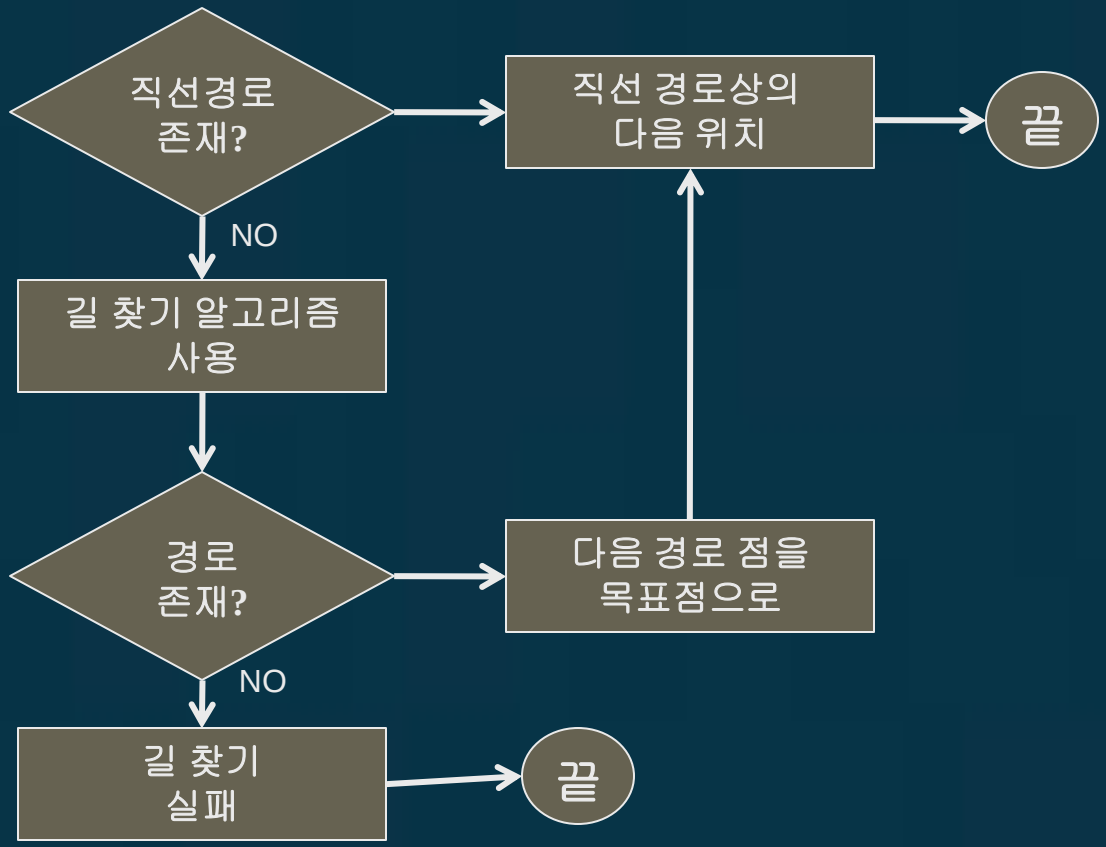
- 기본 길 찾기 방식
  - **다음 Step의 위치 정하기**
    - 단위 시간에 갈 수 있는 직선상의 위치
    - 다음 Tile : 단위 시간이 가변
    - 방향 전환 점
  - Step이 필요한 이유
    - 패킷 개수 절약, 계산 시간 절약 => Timer를 통한 이동
  - 매 step 마다 다시 길 찾기 필요
    - 목표이동, 지형변화, 장애물 변화
  - 길 찾기의 단위 (길 찾기 알고리즘의 단위)
    - Tile
    - Node

길 찾기

- NODE에서 뭠춰야 하는 이유



길 찾기


직선경로
존재?
직선 경로상의
다음 위치
끝
NO
길 찾기 알고리즘
사용
경로
존재?
다음 경로 점을
목표점으로
NO
길 찾기
실패
끝

길찾기

- 길 찾기 알고리즘
  - 단위 : 길 찾기 알고리즘은 기본적으로 그래프 최단 경로 검색
    - 그래프의 노드가 무엇인지 정의 필요
    - Tile혹은 미리 찍어놓은 좌표들
    - Weight를 줄 수도 있음
      - 경사, 이동 속도를 느리게 하는 장애물

길찾기

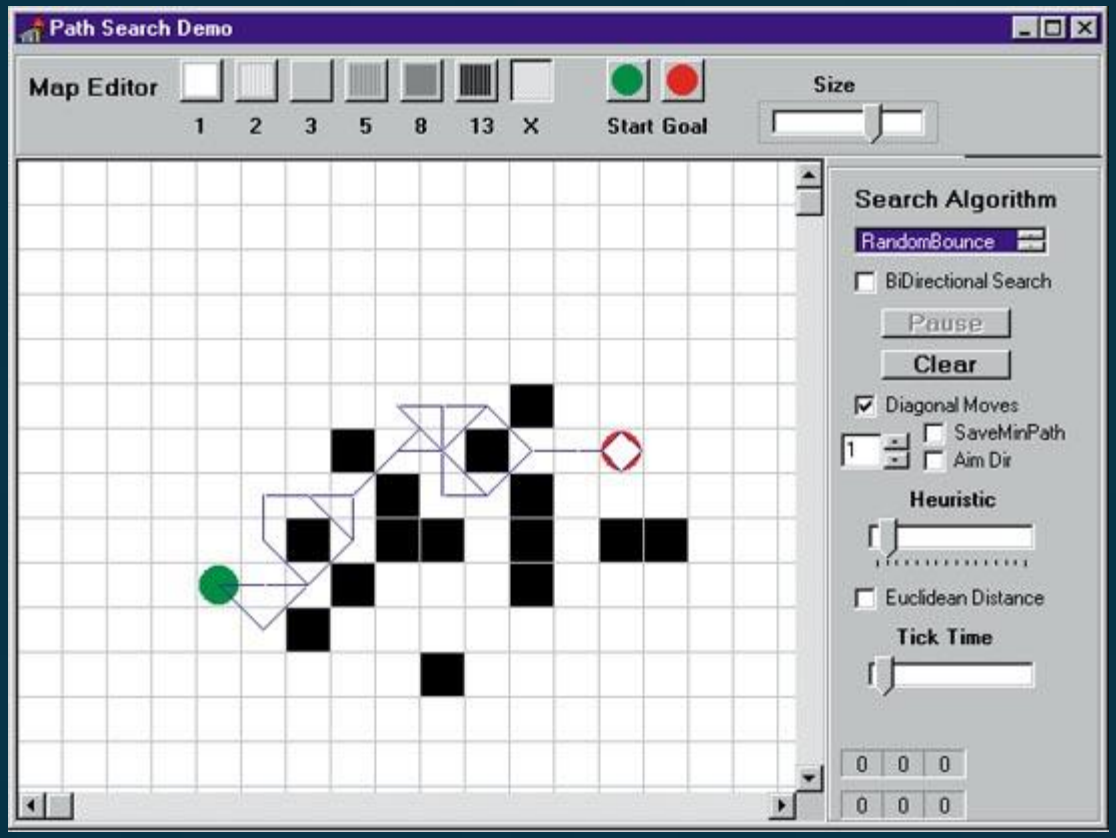
- 길 찾기 알고리즘의 종류
  - 가면서 찾기 (Path-finding on move)
  - 미리 찾기
    - Dijkstra
    - A*
  - http://www.gamasutra.com/view/feature/131505/toward_more_realistic_pathfinding.php

길찾기

- 가면서 찾기
  - 지형 전체를 알 수 없는 경우
    - 마이크로 마우스 미로 찾기
    - 멍청한 NPC
    - 빠른 계산
    - 장애물이 거의 없는 경우
  - 다음 경로점
    - 랜덤
    - 장애물 따라 돌기
    - 직선 찾아 돌기

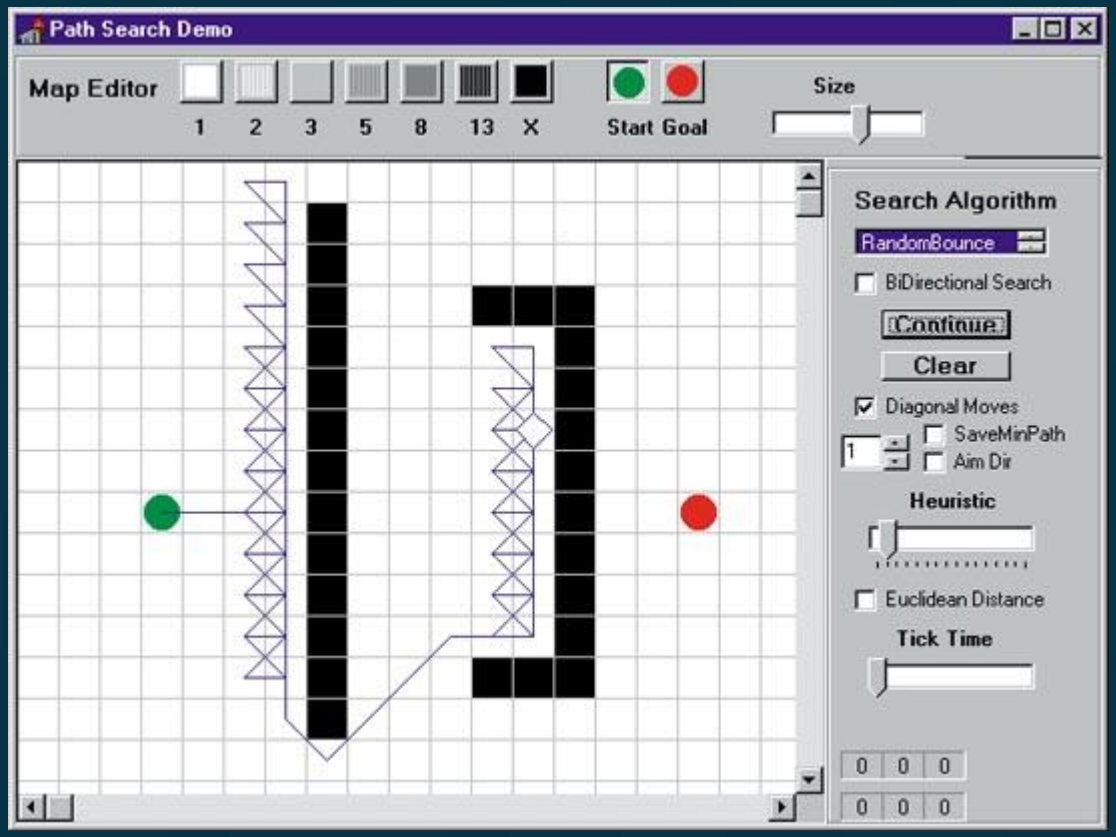
길 찾기

- 가면서 길찾기 : 랜덤



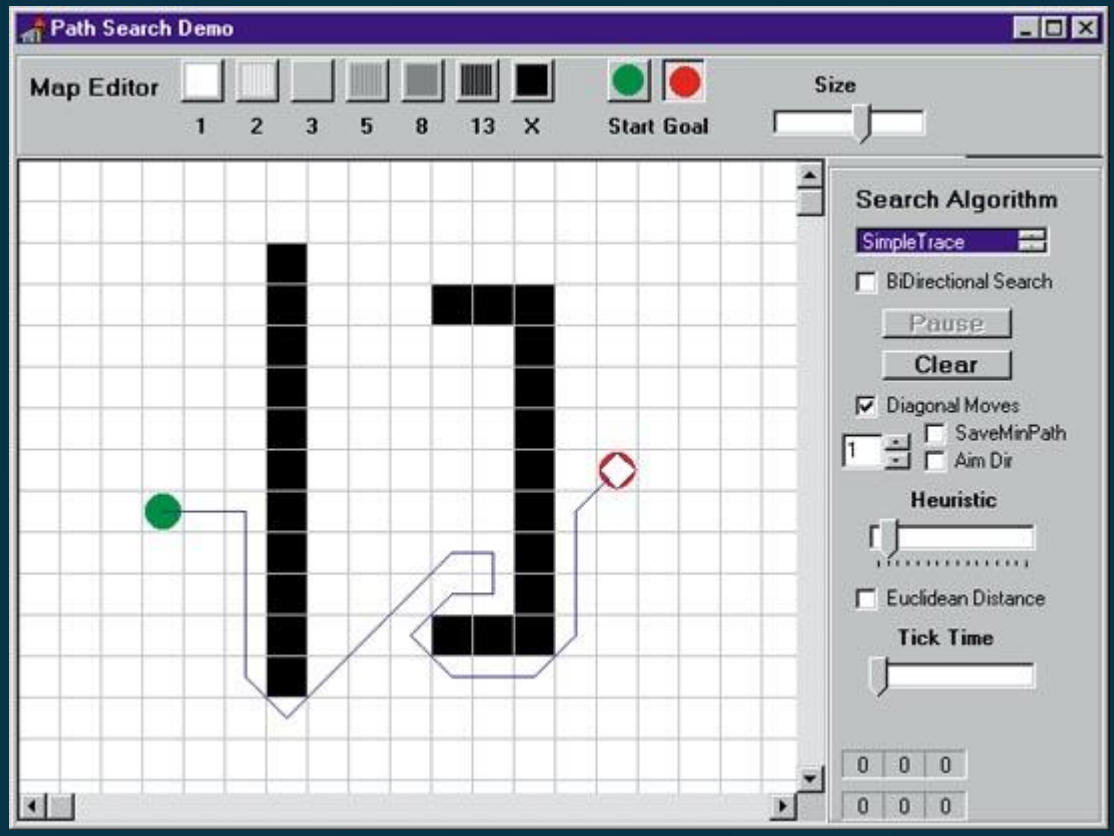
길 찾기

- 가면서 길찾기 : 랜덤



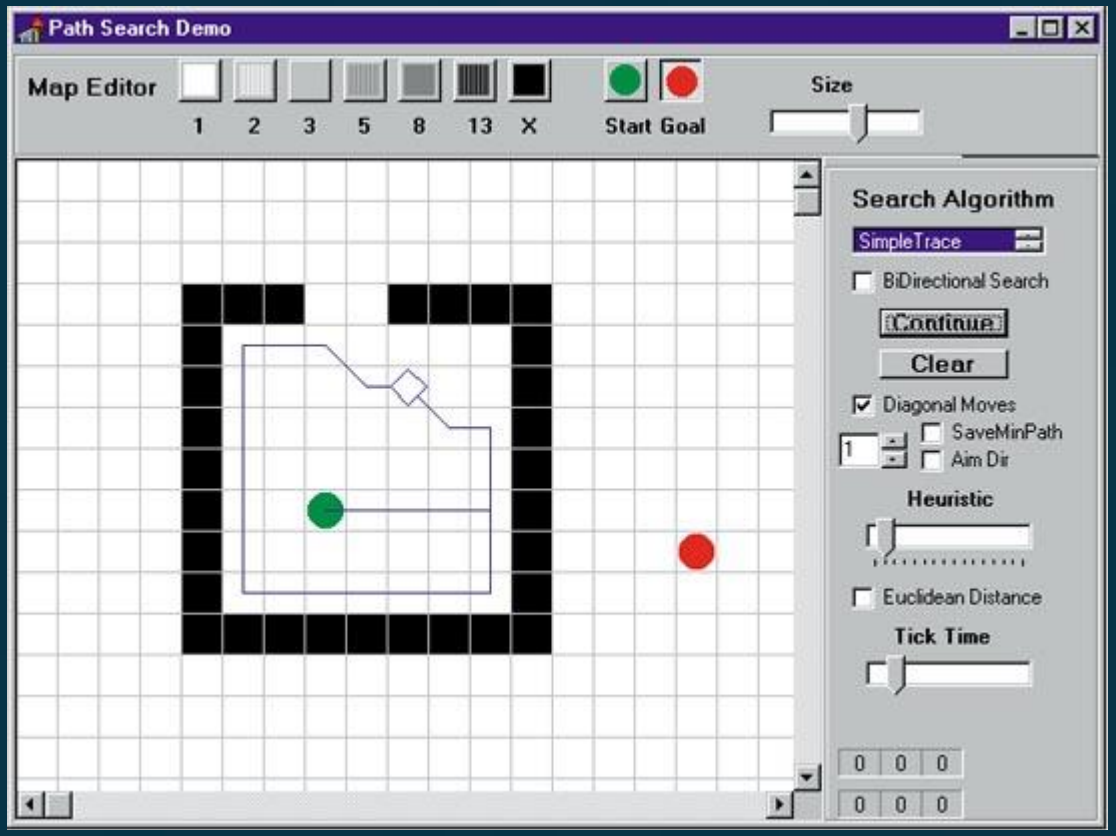
길 찾기

- 가면서 길찾기 : 장애물 따라 돌기



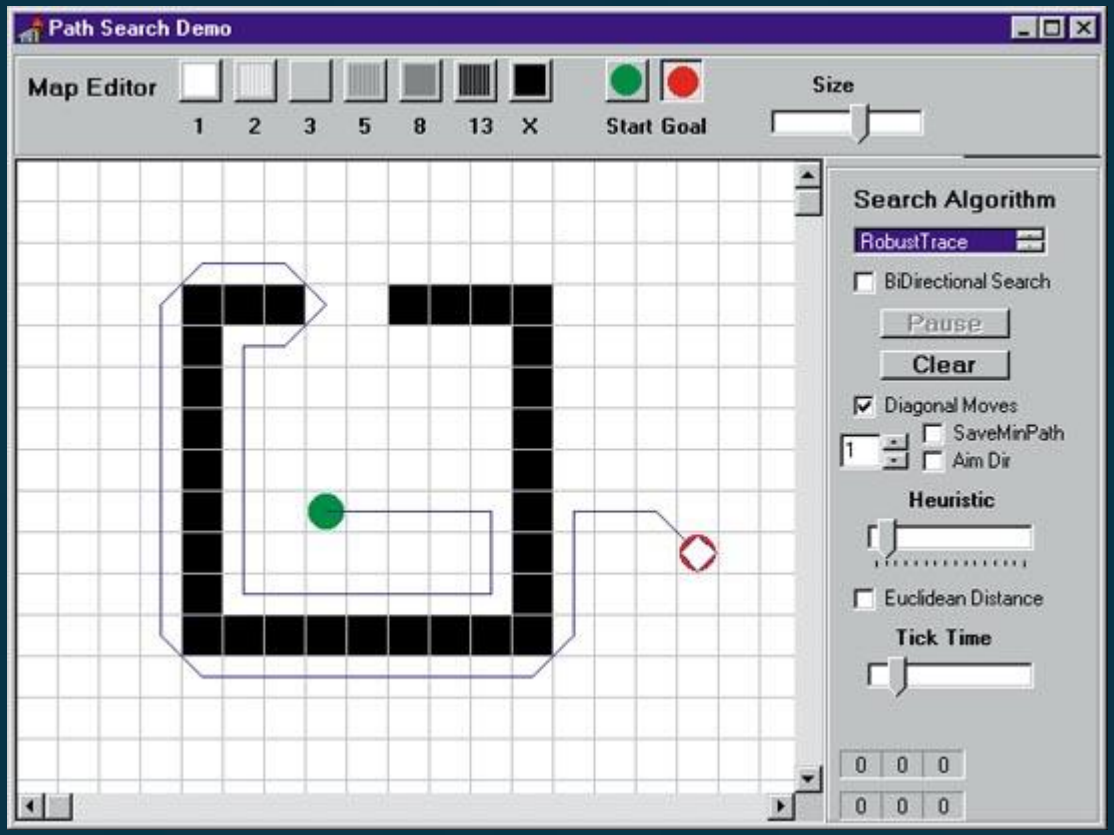
길 찾기

- 가면서 길찾기 : 장애물 따라 돌기



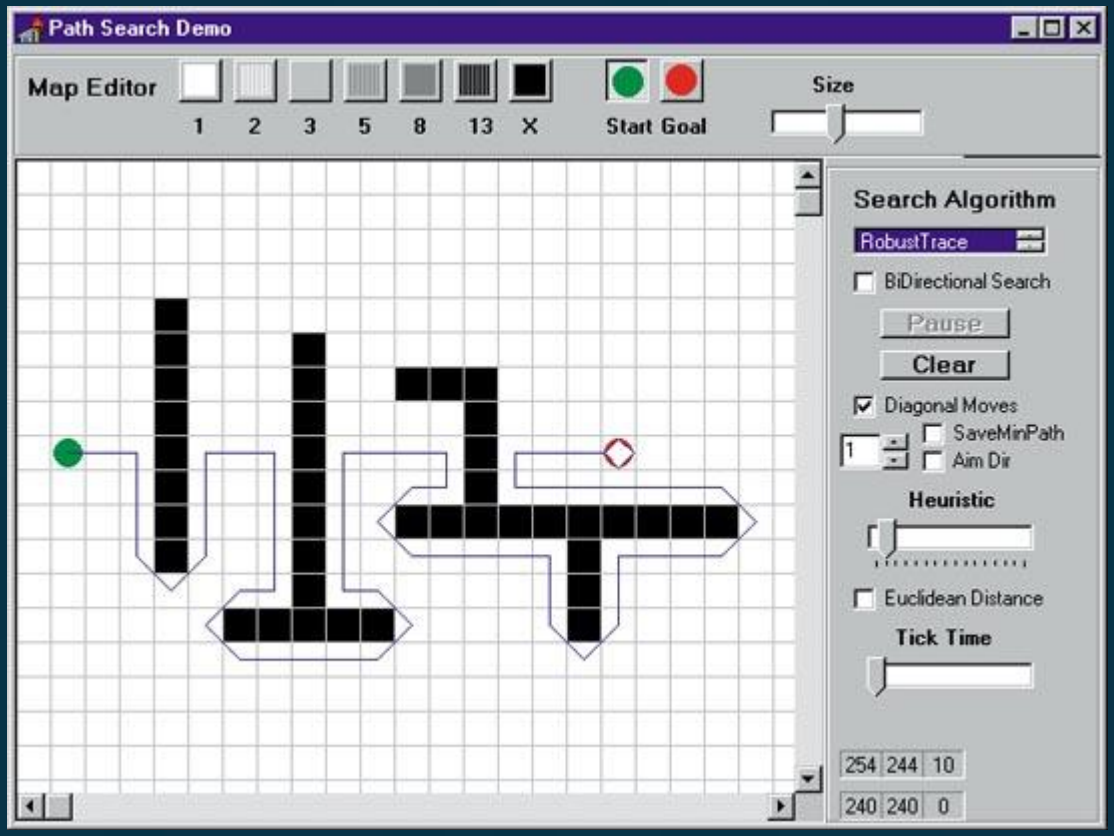
길 찾기

- 가면서 길찾기 : 직선 찾아 돌기



길 찾기

- 가면서 길찾기 : 직선 찾아 돌기



길찾기

- 미리 찾기
  - 똑똑한 NPC를 위해서는 필수
- Depth-first search
  - IDDF (Iterative-deepening depth-first search)
- Breadth-first search
  - Bidirectional breadth-first search
  - Dijkstra's algorithm
  - Best-first search
  - A* search

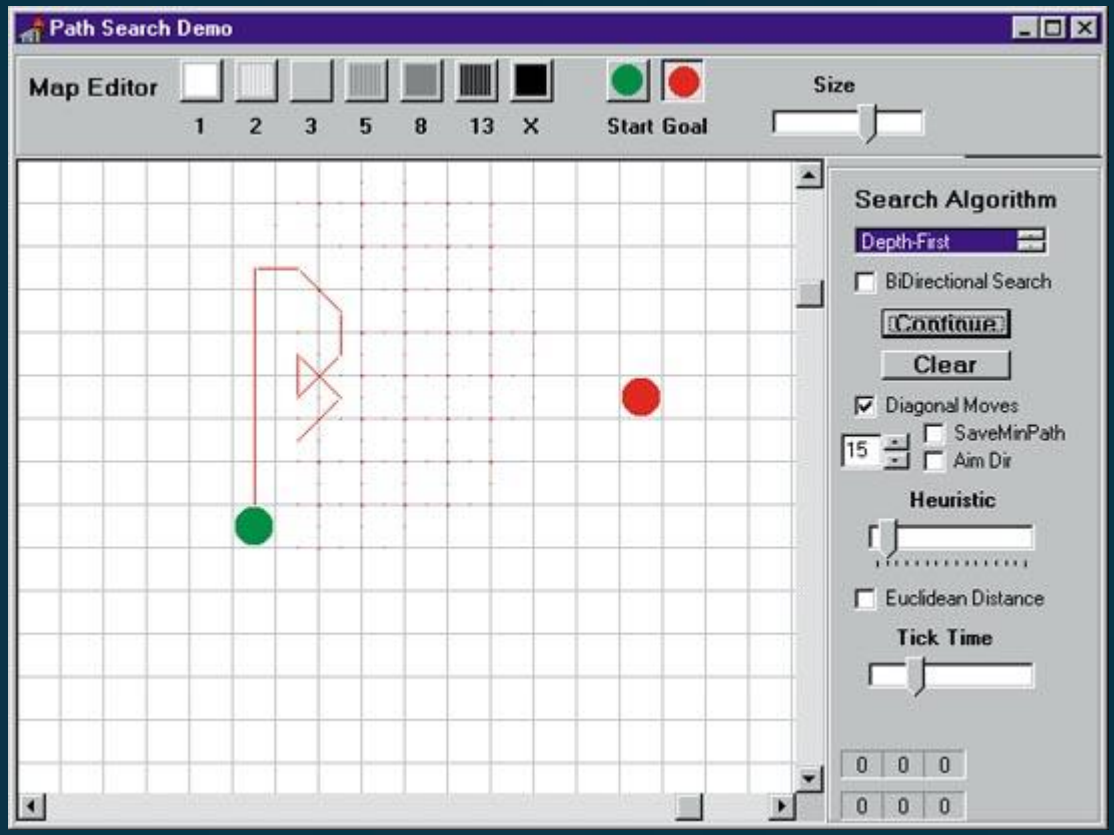
길찾기

- Depth-first search
  - 탐색 길이 제한 필요
  - IDDF

```
DepthFirstSearch( node n )
        node n'
        if n is a goal node
                return success
        for each successor n' of n
                if DepthFirstSearch( n' ) is success
                        n'.parent = n
                        return success
        return failure          // if no path found
```

길 찾기

- Depth-first search



Visit Marking
Save Min Path
Aim Dir

길찾기

- Breadth-first search
  - 모든 경로를 점진적으로 검색
  - 모든 방향 검색

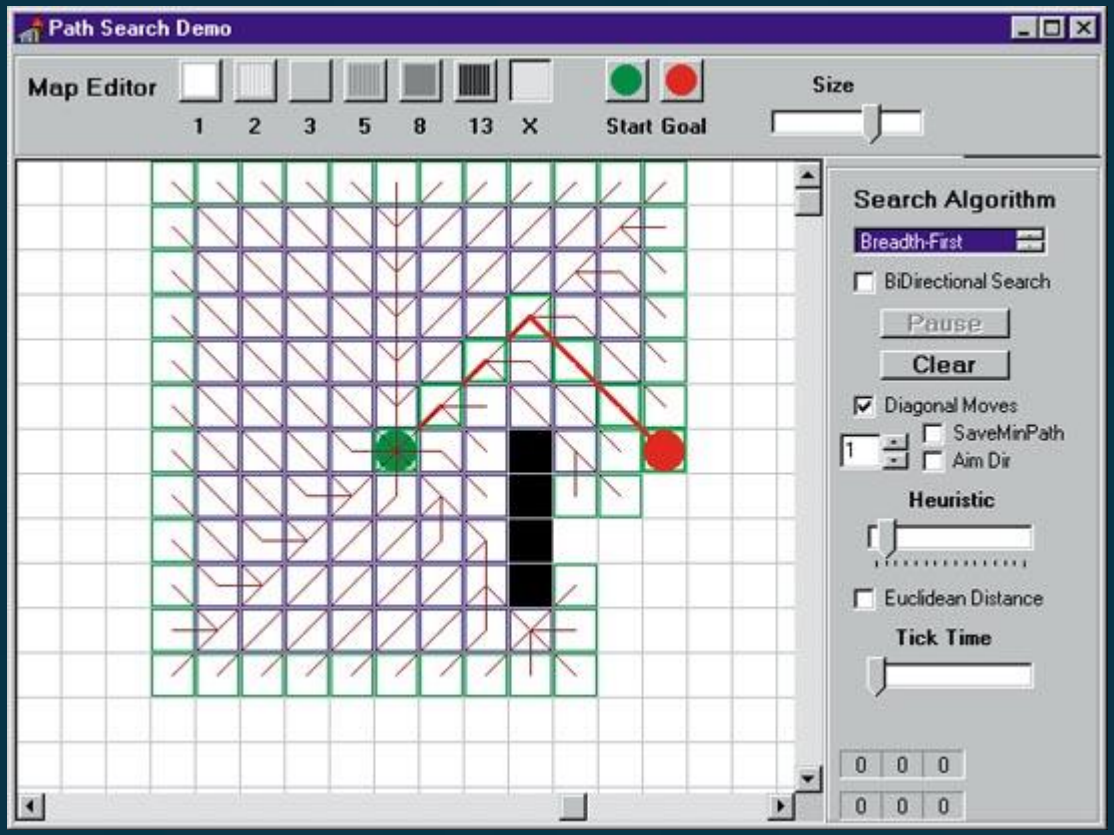
길찾기

- Breadth-first search

```
queue       Open

BreadthFirstSearch
            node n, n', s
            s.parent = null          // s is a node for the start
            push s on Open
            while Open is not empty
                        pop node n from Open
                        if n is a goal node
                                    construct path
                                    return success
                        for each successor n' of n
                                    if n' is in Open
                                                continue
                                    n'.parent = n
                                    push n' on Open
            return failure           // if no path found
```

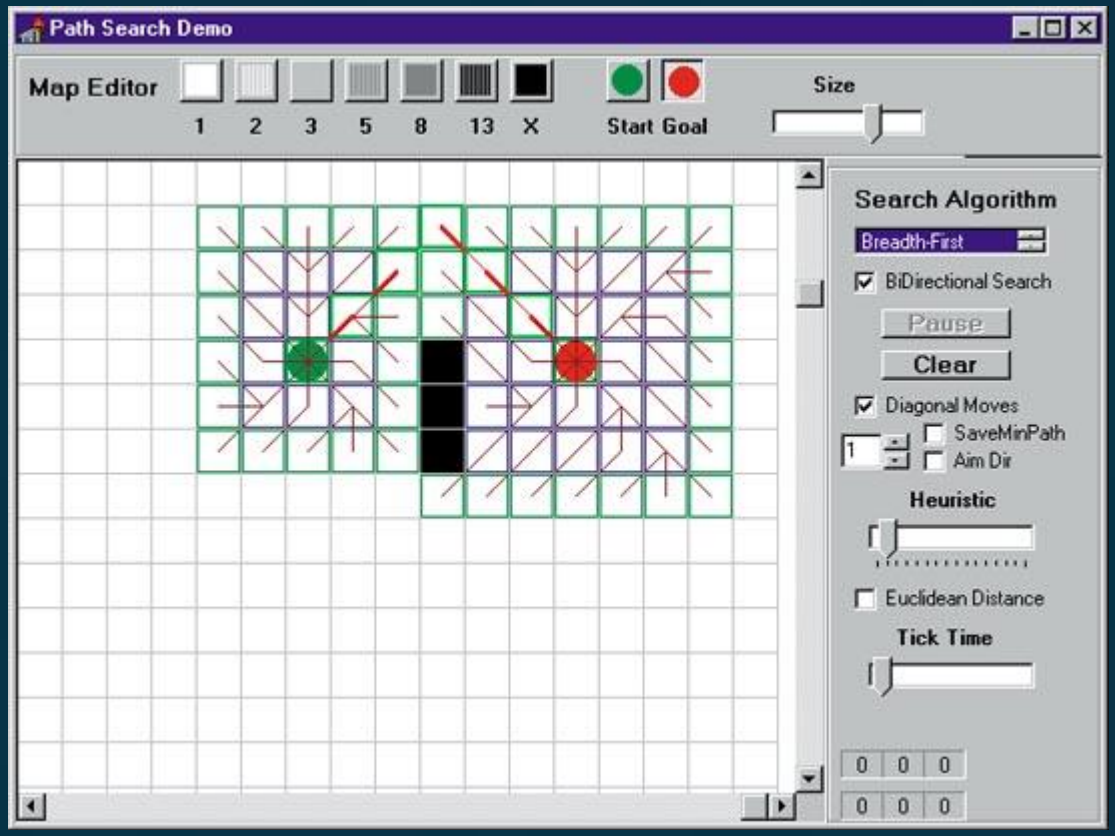
길 찾기

- Breadth-first search



길 찾기

- Breadth-first search :bidrectional



길찾기

- Dijkstra's 알고리즘

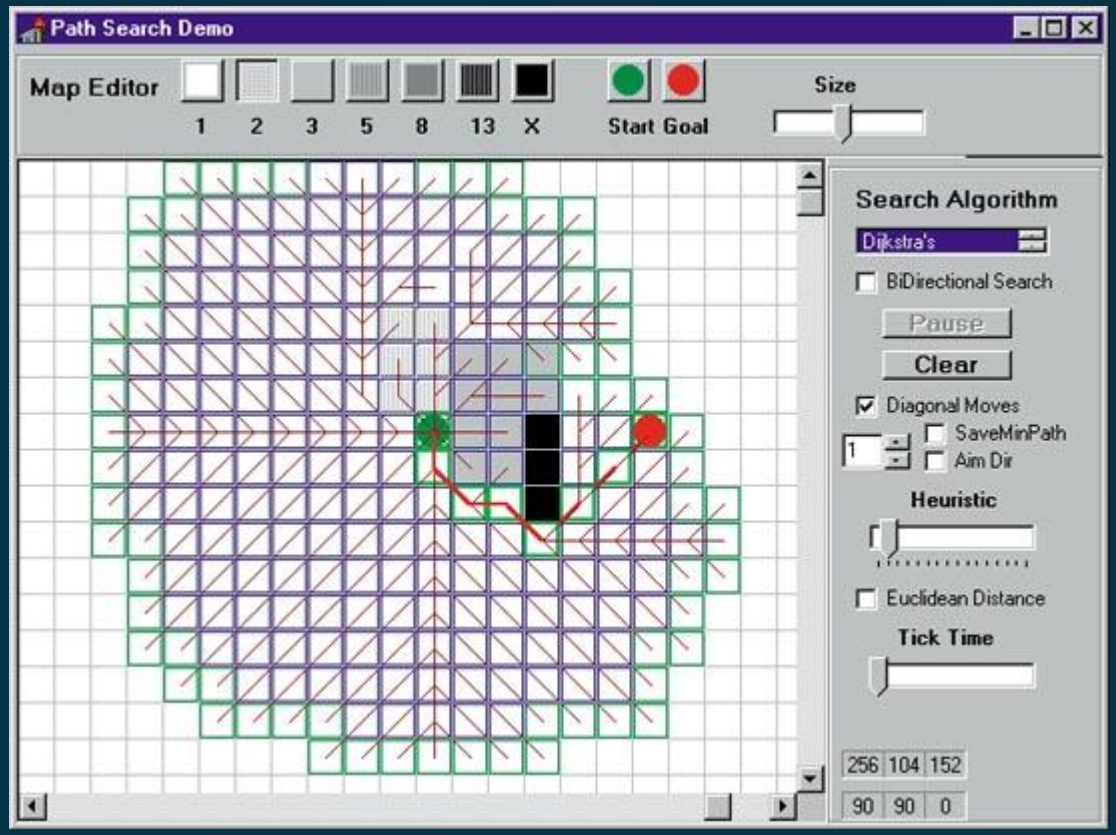
```
priority queue                  Open

DijkstraSearch
                node n, n', s
                s.cost = 0
                s.parent = null                 // s is a node for the start
                push s on Open
                while Open is not empty
                                pop node n from Open        // n has lowest cost in Open
                                if n is a goal node
                                                construct path
                                                return success
                                for each successor n' of n
                                                newcost = n.cost + cost(n,n')
                                                if n' is in Open and n'.cost <=newcost
                                                                continue
                                                n'.cost = newcost
                                                n'.parent = n
                                                push n' on Open
                return failure  // if no path found
```

길찾기

- Dijkstra's 알고리즘
  - 노드에 cost를 줄 수 있다.
  - Tile구조가 아닌 Graph구조에 유용
  - 항상 최적의 답

길 찾기

- Dijkstra's algorithm

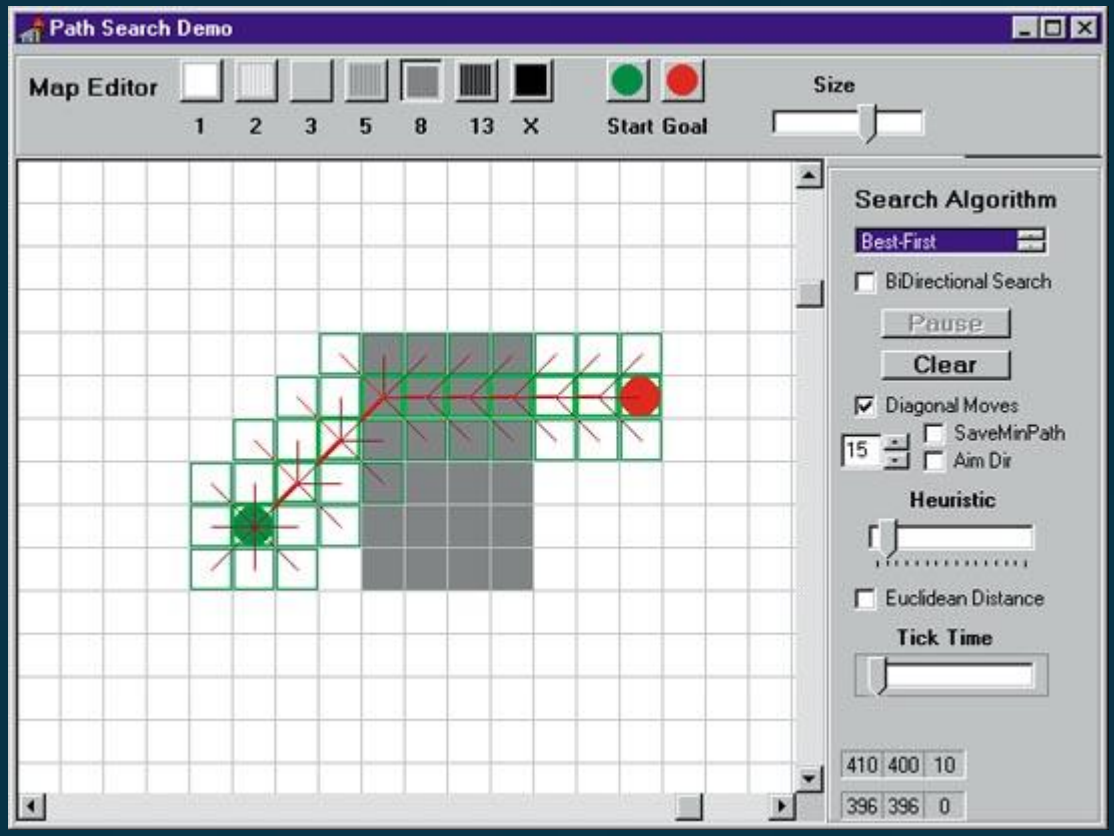


길찾기

- Best-first search
  - 휴리스틱
  - 항상 최적의 해를 내놓지는 않음
  - Dijkstra에서 비용을 도착지까지의 예상 비용으로 대체

길 찾기

- Best-first search



길찾기

- A*
  - 가장 많이 쓰이는 알고리즘
  - Guided Dijkstra
  - Cost function
    - F(n) = G(n) + H(n)
      - F(n) : 노드 n의 비용
      - G(n) : 시작점에서 n까지의 최소 비용
      - H(n) : 도착점까지의 근사 비용
    - A* = Dijkstra's + Best-first search

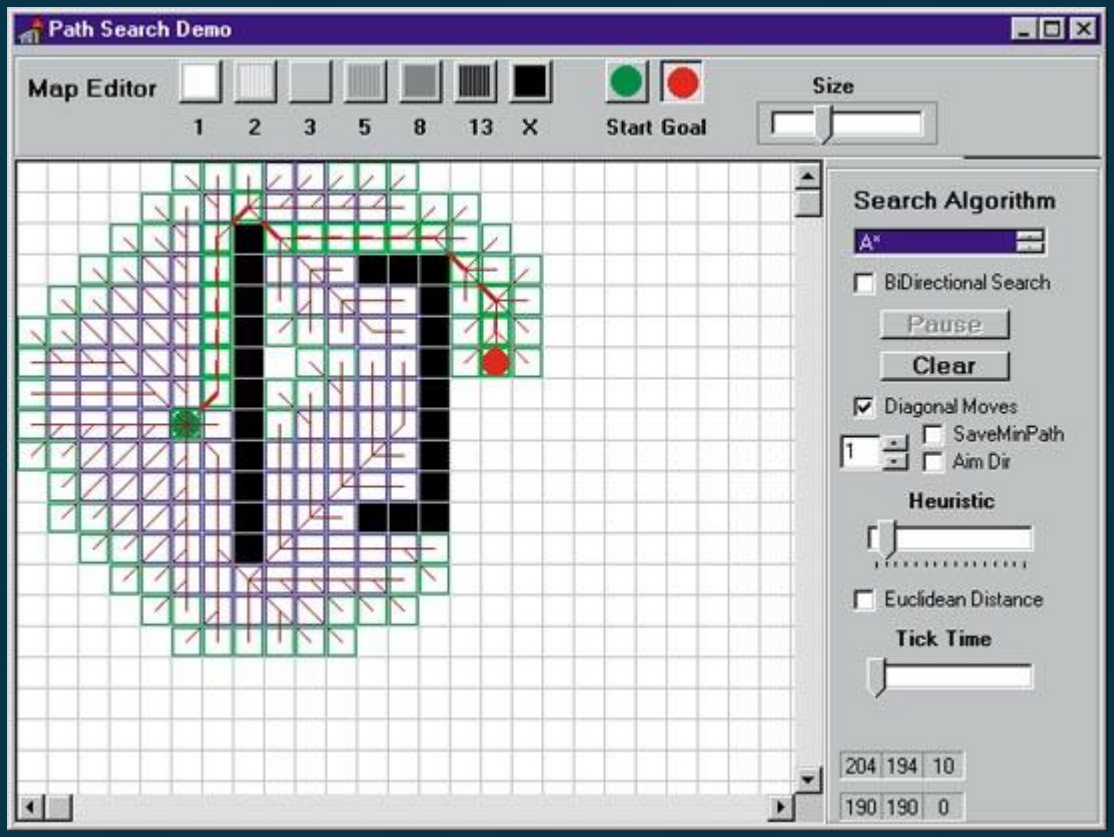
길찾기

- A*

```
priority queue  Open
List            Closed

AStarSearch
                s.g = 0
                s.h = GoalDistEstimate( s )
                s.f = s.g + s_h
                s.parent = null
                push s on Open
                while Open is not empty
                                pop node n from Open          // n has lowest cost  in Open
                                if n is a goal node
                                                construct path
                                                return success
                                for each successor n' of n
                                                newg = n.g + cost(n,n')
                                                if n' is in Open or Closed, and n'.g<=newg
                                                                continue
                                                n'.g = newg
                                                n'.h = GoalDistEstimate( n' )
                                                n'.f = n'.g + n'.h
                                                n'.parent = n
                                                if n' is in Closed
                                                                remove it from Closed
                                                if n' is not in Open
                                                                push n' on Open
                                push n onto Closed
                return failure  // if no path found
```

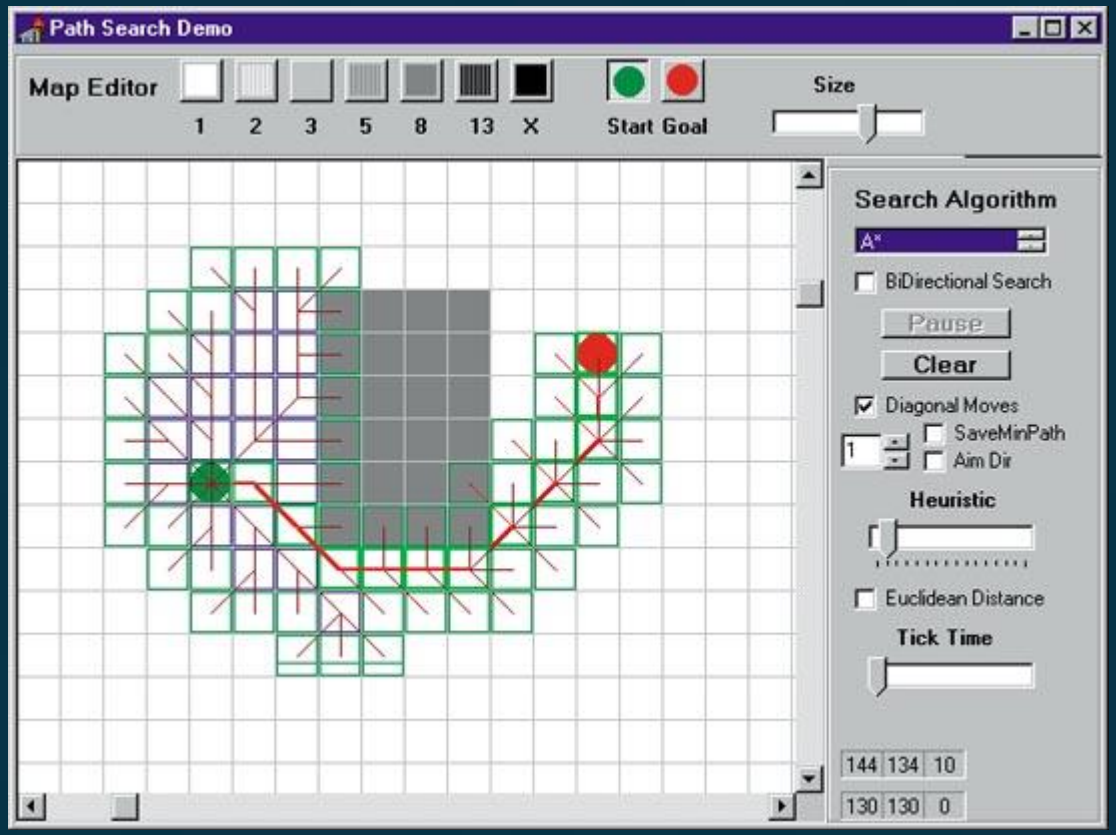
길 찾기

- A*


Path Search Demo
Map Editor
1
2
3
5
8
13
X
Start Goal
Size
Search Algorithm
A*
BiDirectional Search
Pause
Clear
Diagonal Moves
SaveMinPath
Aim Dir
Heuristic
Euclidean Distance
Tick Time
204 194 10
190 190 0

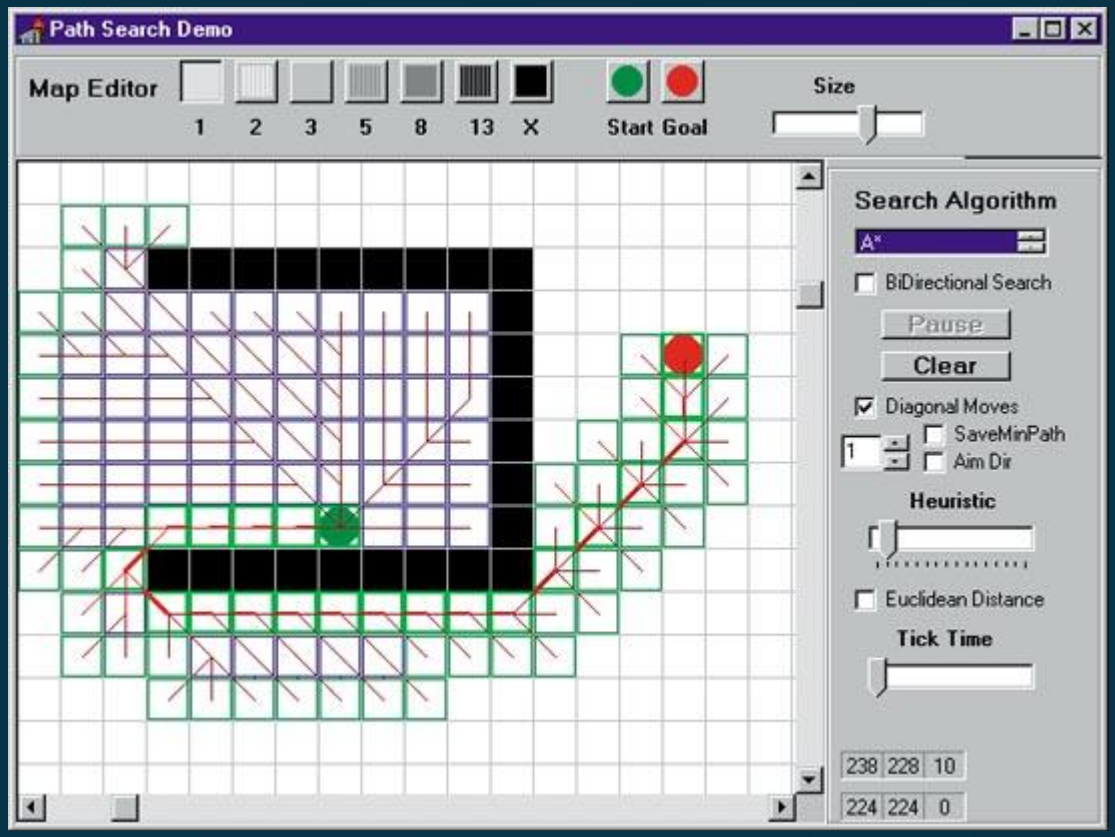
길 찾기

- A*


Path Search Demo
Map Editor
1 2 3 5 8 13 X
Start Goal
Size
Search Algorithm
A*
BiDirectional Search
Pause
Clear
Diagonal Moves
SaveMinPath
Aim Dir
Heuristic
Euclidean Distance
Tick Time
144 134 10
130 130 0

길 찾기

- A*


Path Search Demo
Map Editor
1
2
3
5
8
13
X
Start Goal
Size
Search Algorithm
A*
BiDirectional Search
Pause
Clear
Diagonal Moves
SaveMinPath
Aim Dir
Heuristic
Euclidean Distance
Tick Time
238 228 10
224 224 0

내용

- NPC
- 지형구현
- 길찾기
- NPC-AI

NPC-AI(2023)

- 컨텐츠의 핵심
  - MO와의 차이
    - Overwatch, 배틀그라운드
    - LOL?
  - Timing에 맞춘 동작들 구현
    - 이동, 마법 시전, HP회복…
    - 캐스팅 타임, 쿨타임
  - NPC AI
    - Timer 기반의 Finite State Machine
    - 주기적으로 상황 파악. (그러나 현실은…)

NPC-AI

- 구현
  - AI Code를 어디서 실행하는가?
  - main()? worker_thread()? AI_thread()?
  - 어떻게 구현하는 것이 효율적인가?
    - 하지 않아도 되는 코드 실행 없애기.
- 예제
  - DO-A -> 1초 delay -> DO-B
  - 1초마다 Heal
  - 1초에 1번 랜덤 무브

NPC-AI

```
DO (A);
Sleep(1000);
DO (B);
```

일반 프로그램

```
Loop(true) {
  if (false == A_done) {
       DO (A); A_done = true;
       B_time = current_time() + 1000; }
  if (B_time <= current_time()) {
    DO(B);
    B_time = MAX_INT;
   }
```

게임 클라이언트 프로그램

NPC-AI

- 앞의 동작의 문제점
  - 10만개의 NPC가 출동하면?
    - 매 루프 마다 10만번의 if 문이 필요.
      - 실제로는 NPC마다 여러 개의 if가 필요 (move, heal, attack..)
      - Busy Waiting
    - 캐시 문제, pipeline stall
- 해결책?
  - NPC Class에 heart_beat() 함수를 두고 일정 시간 간격마다 호출 되게 한다.
    - 모든 NPC AI가 일정 시간 간격마다 실행

NPC-AI

```
Loop(true) {
    if (objA.next_heal_time < current_time) {
    objA.m_hp += HEAL_AMOUNT;
    objA.next_heal_time += HEAL_INTERVAL;
   }
```

서버 메인루프에서 검사 : busy wait

```
Cobj::heart_beat()
 {
   m_hp += HEAL_AMOUNT;
 }
```

1초마다 호출 : heart_beat

Timer

- Heart_beat 함수.
  - 자율적으로 움직이는 모든 NPC를 살아있도록 하는 함수.
  - 외부의 요청이 없어도 독자적으로 AI를 실행
  - 구현
    - Heart_beat_thread

```
while(true) {
  curr_heart_beat = current_time();
  for (int i =0 ; i < MAX_NPC; ++i)
       NPC[i].heart_beat();
  delay = DURATION - (current_time() - curr_heart_beat);
  delay = MAX(0, delay);
  Sleep(delay);
}
```

Timer

- Heart_beat 함수의 문제
  - 10만개의 NPC라면?
    - busy waiting은 없지만 아무일도 하지 않는 heart_beat이 시간을 잡아 먹는다.

```
heart_beat()
 {
   my_hp += HEAL_AMOUNT;
 }
```

이론

실제

```
heart_beat()
 {
   if (my_hp < my_max_hp)
      my_hp += HEAL_AMOUNT;
 }
```

Timer

- Heart_beat 함수의 문제 해결 - 1
  - 필요한 경우만 heart_beat이 불리도록 한다.
    - 복잡한 NPC의 경우 프로그래밍이 어려워 진다.
      - Heart_beat함수안의 수많은 if
      - 불리지 않는 경우를 판단하기가 힘들다.
        - 판단하는 것 자체도 오버헤드
- Heart_beat 함수의 문제 해결 – 2
  - heart_beat함수를 없앤다.
  - 각 모듈에서 timer를 직접 사용한다.

Timer

```
heart_beat()
 {
   if (my_hp < my_max_hp)
      my_hp += HEAL_AMOUNT;
 }
```

```
get_damage(int dam)
 {
   my_hp -= dam;
   add_timer(my_heal_event, 1000);
 }

my_heal_event()
  {
    my_hp += HEAL_AMOUNT;
    if (my_hp < my_max_hp)
       add_timer(my_heal_event, 1000);
  }
```

Timer

- Timer thread의 구현

```
Priority_queue <Event> timer_queue

TimerThread()
        do  {
                sleep(1)
                do  {
                        event k = peek (timer_queue)
                        if  k.starttime > current_time()
                                break
                        pop (timer_queue)
                        process_event(k)
                } while true;
        } while true;
```

Timer

- Timer Thread와 Worker Thread의 연동
  - timer thread에서 할일
    - 모든 AI
    - 이동, 길찾기 등
  - timer thread의 과부하 => 서버 랙
  - 실제 작업은 worker thread에 넘겨야 한다.

Timer

- Timer Thread와 Worker Thread의 연동

```
NPC_Create()
  foreach NPC
   add_timer(my_id, MOVE_EVENT, 1000)

Timer_thread()
  …
  overlap_ex.command = MOVE
  PostQueuedCompletionStatus(port, 1, id, overlapex)
  …

Worker_thread()
  …
  if (overlap_ex.command == MOVE) move_npc(id);
  …
```

Timer

- 이벤트 큐
  - 저장 정보
    - 어떤 오브젝트가 언제 무엇을 누구에게 해야 하는가.
    - 타이머 쓰레드가 큐에서 이벤트를 꺼내서 활성화

```
struct event_type {
    int obj_id;
    high_resolution_clock::time_point wakeup_time;
    int event_id;
    int target_id;
};
```

Timer

- 이벤트 큐
  - 시간 순서대로 정렬된 우선순위 큐가 필요하다.

```
struct event_type {
   int obj_id;
   high_resolution_clock::time_point wakeup_time;
   int event_id;
   int target_id;

   constexpr bool operator < (const event_type& _Left) const
   {
      return (wakeup_time > _Left.wakeup_time);
   }
};

priority_queue<event_type> timer_queue;
mutex timer_lock;
```

NPC

- NPC
  - Timer Queue와 Worker Thread 만으로는 부족
  - 대부분의 NPC가 timer queue로 동작한다면 timer thread의 과부하
  - 플레이어가 관찰할 수 있는 NPC만 움직여야 한다.
    - 플레이어가 깨웠을 때에만 NPC AI 작동
    - 플레이어가 근처에 없으면 NPC AI 비활성화
    - is_active 변수를 통해 제어
      - 중복 activate 방지.
      - Mutex 대신 CAS를 사용하는 것이 부하가 적다.

NPC

- NPC : 타이머를 사용한 이동

```
Event_queue timer_queue

NPC_Create()
        foreach NPC
                push (timer_queue, id, MOVE_EVENT, 1)

NPC_CALLBACK(id, event)
    if (event == MOVE_EVENT)
        id -> move_npc()
        push (timer_queue, id, MOVE_EVENT, 1)

MOVE_NPC()
        overlap_ex.command = MOVE
        PostQueuedCompletionStatus(port, 0, &NPC_INFO, overlapex)
```

NPC

- NPC : 타이머를 사용한 이동

```
NPC_Create()

MOVE_PLAYER() {
        foreach_monster_in_range(&monster_id)
            if (!NPC[monster_id].m_is_active) {
               if (CAS(&NPC[monster_id].m_is_active, false, true))
                   add_timer(monster_id, MOVE_EVENT,1000);
            }
}
worker_thread()
      …..
      if (event == MOVE_EVENT)
         id -> move_npc()
         if (near_player_exist(m_id)) add_timer(id, MOVE_EVENT, 1000)
          else NPC->m_is_active = false;

move_npc()
         좌표 = 길찾기();
                m_x = 좌표.x; m_y = 좌표.y;
                broadcast_move(this->m_id);
```

NPC

- NPC : 실습
  - is_active 상태 구현
  - 플레이어나 NPC 이동/생성 시 active 여부 검사

NPC

- NPC 구현
  - 기존 : 서버에 객체가 Player밖에 없음
  - NPC객체와 Player객체는 서로 공통점이 있고 차이점 있다.
    - 공통점 : x, y, hp, id
    - 차이점 : Session 기능 유무
  - 문제 : 어떠한 컨테이너에 담아야 하는가?
    - 같은 컨테이너에 담아야 하는가?
      - NPC가 Session정보를 갖는 낭비가 발생
    - 서로 다른 컨테이너에 담아야 하는가?
      - ID를 별도로 관리하는 오버헤드 발생.

NPC (2023-화목)

- 차이
  - 같은 컨테이너
    - 장점
    - 단점
  - 다른 컨테이너
    - 장점
    - 단점
  - 대안 : 상속 사용

NPC

- 최적화 문제
  - 1만개의 NPC가 있어도 ai_thread과부하
  - 해결책
    - 주위에 플에이어가 있을 경우에만 NPC AI 실행

```
void do_ai()
{
  while (true) {
  auto start_t = chrono::system_clock::now();
  for (auto& npc : clients) {
     if (false == is_npc(npc._id)) continue;
     << if (false == player_exists(npc._0d)) continue; >>
     do_npc_move(npc._id);
}
```

    - 문제
      - 모든 NPC 모든 플레이어 검색 오버헤드
      - AI_THREAD 과부하

NPC

- AI thread 과부하 해결
  - Worker Thread에게 떠넘긴다.

  => Worker Thread 과부하

  - Timer를 사용해서 검색을 없앤다.

숙제 (#6) (2023-화목)

- NPC AI 작성
  - 내용
    - 숙제 (#5)의 프로그램의 확장, 실습 시간 제작 프로그램 완성
      - 월드 크기를 2000 X 2000으로 확장
    - 200,000마리의 몬스터의 랜덤 무브
    - Stress Test Client를 사용해서 성능을 측정할 것.
      - 목표 : 고사양 PC에서 동접 2000이상
    - 수업시간에 작성한 코드 참조
  - 목적
    - NPC_AI 및 Timer 개념 사용 (PQCS와 GQCS를 사용)
    - 플레이어 주위의 NPC만 이동하도록 최적화
  - 제약
    - Windows에서 Visual Studio로 작성 할 것

다음 시간

- SCRIPT
- DB