NPC와 지형

MM4220 게임서버 프로그래밍

정내훈

내용

- NPC
- 길찾기
- TIMER
- 숙제

NPC

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

NPC

- 인공 지능
  - 재미있는 행동/반응
  - 재미
    - Cheating사용은 재미를 떨어 뜨림
    - 너무 잘해도 곤란 : monster의 존재 목적?
  - Script로 구현
  - 가장 기본적인 인공지능
    - 길 찾기
    - 적 인식

NPC

- Script
  - 뒤에 더 자세히
  - 사용 목적 : 게임 제작 파이프라인 단축
    - 프로그래머의 개입 없이 기획자가 직접 작성/Test
    - 서버 리부팅도 필요 없게 할 것.
  - 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 수동적
    - 플레이어의 이동을 근처 NPC에만 broadcast
      - 플레이어 생성, NPC생성시 검사 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

길 찾기

- Avatar, NPC 이동의 기본
  - Avatar이동?
    - WASD이동 시 불필요
    - 마우스 클릭 이동 시 필요 => 꼭 서버에서 할 필요는 없음
  - NPC
    - 서버구현 필수
      - 서버가 지형과 장애물을 알아야 한다.
- 지형 구현과 밀접한 관련
  - 2D, 3D?
  - Tile, Polygon?

지형 구현(2019-수목)

- 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

- 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

- 지형 구현
  - 3D 지형 확장 타일 방식
    - Tile방식
    - 3D 배열로 지형 표현
      - 배열 구현 시 메모리 낭비가 심해서 Sparse Matrix로 표현
      - 이동 가능 정보 이외에 높이 정보도 포함
      - 많은 부분이 단층지형 이므로 2D배열과 혼용
    - 이동 가능 불가능 flag이 cell마다 존재
    - 서버 안에서의 모든 Object의 좌표는 실수
      - Float or Double?
      - 3D게임에서의 정수 좌표는 비쥬얼 적으로 error

지형 구현

- 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
      - 최신 클라이언트 들은 Collision Polygon따로 요구
    - 클라이언트와 비슷한 방법으로 이동 가능 검사
      - 물리 엔진 필요
      - 지형 표현 정밀도 증가
    - 정수 좌표 불가능
    - Tile방식에 비해 속도는 떨어지지만 확장성 증가

지형 구현 (2019-화목)

- 장단점
  - 2d 타일
  - 2d 타일 확장
  - 폴리곤

지형 구현

- 지형 구현의 Issue
  - 충돌 정보 가공
    - Path Node
    - Path Mesh (Navigation Mesh)
  - 문의 구현
    - 특수 지형 속성?
    - NPC?
  - 이동 가능 지형
    - 엘리베이터, 배, 이동 발판 (2018 가을)

지형 구현

- 충돌 정보

Path node

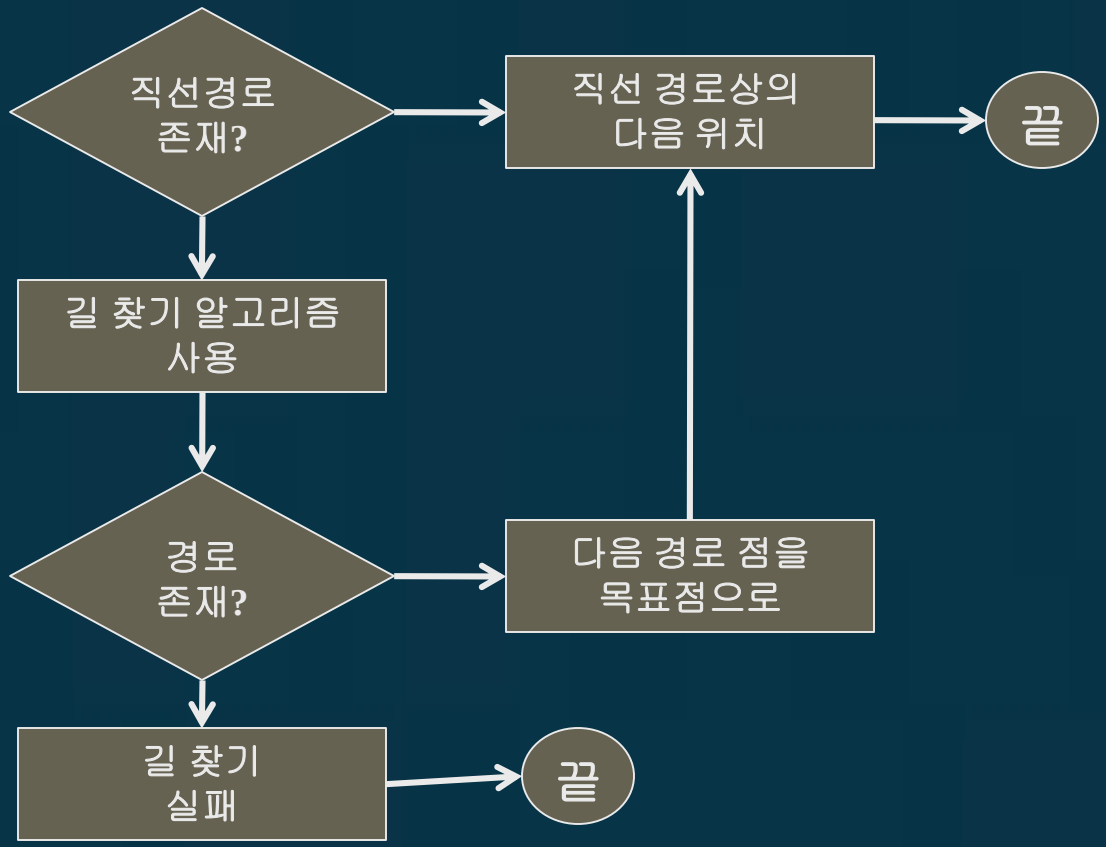
입력 지형

Path mesh

길 찾기

- 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



길찾기(2019-수목)

- 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

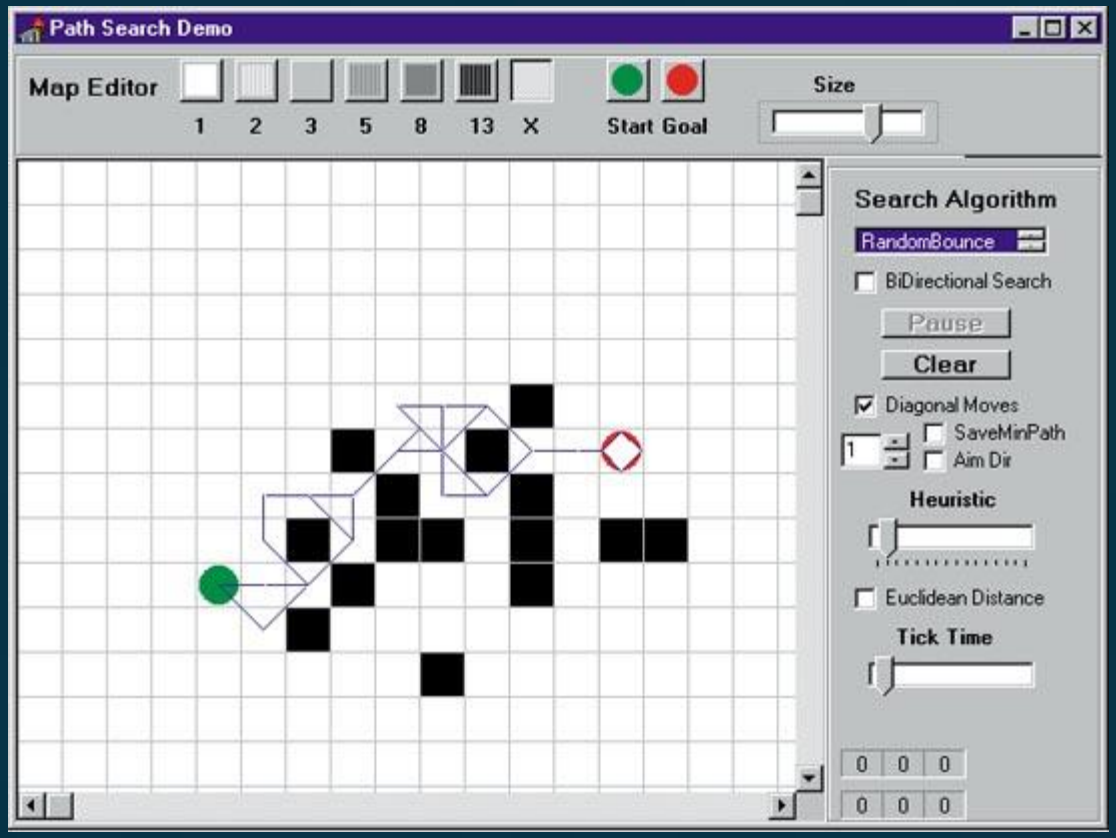
- 길 찾기 알고리즘의 종류
  - 가면서 찾기 (Path-finding on move)
  - 미리 찾기
    - Dijkstra
    - A*
  - http://www.gamasutra.com/view/feature/131505/toward_more_realistic_pathfinding.php

길찾기

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

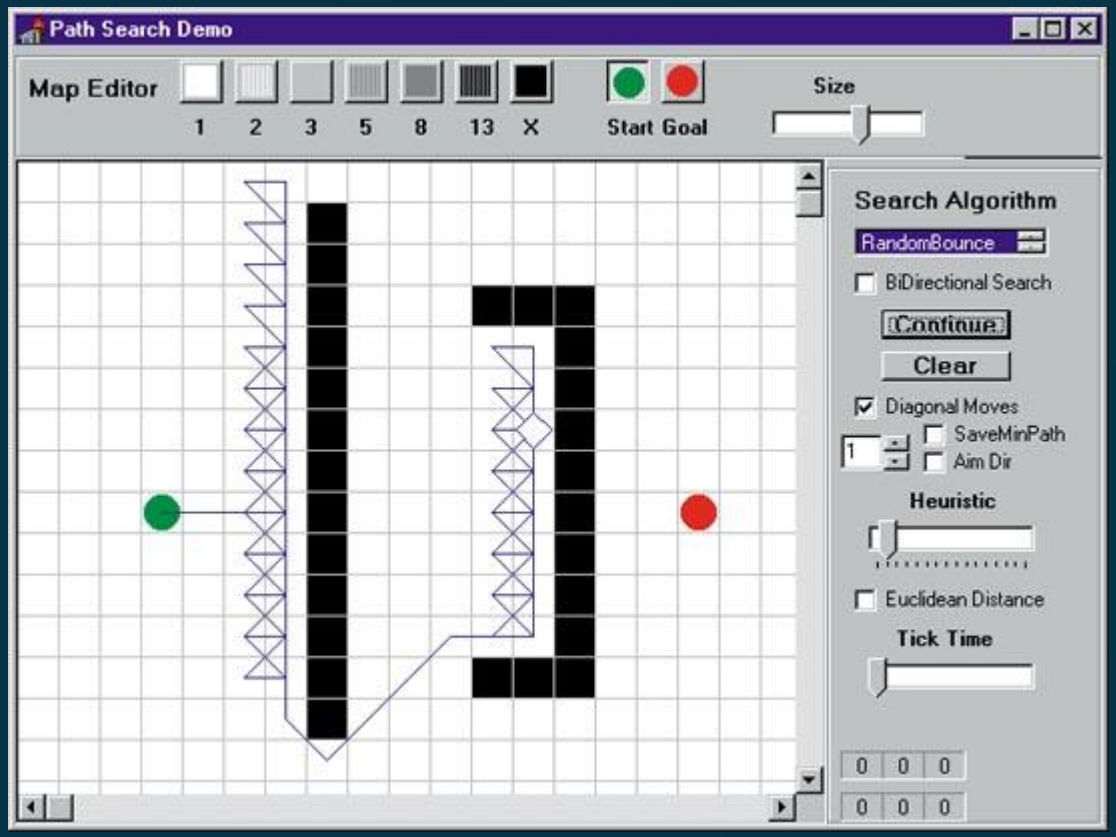
길 찾기

- 가면서 길찾기 : 랜덤



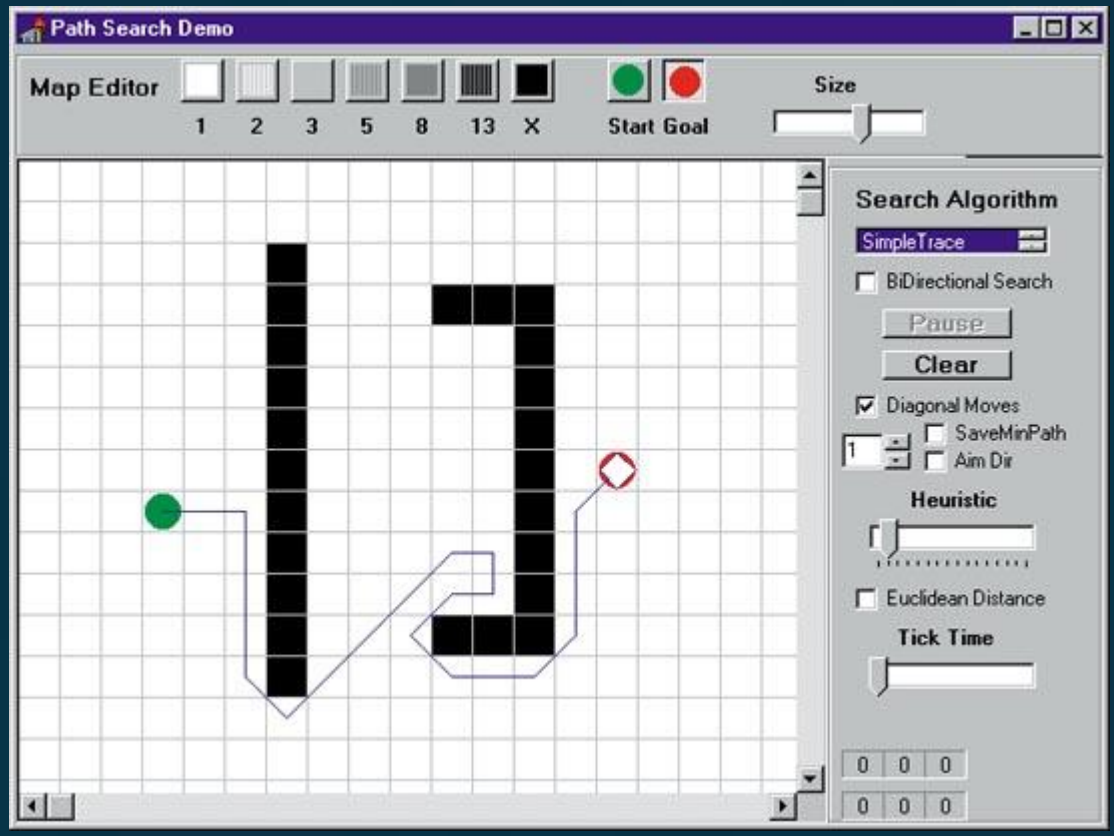
길 찾기

- 가면서 길찾기 : 랜덤



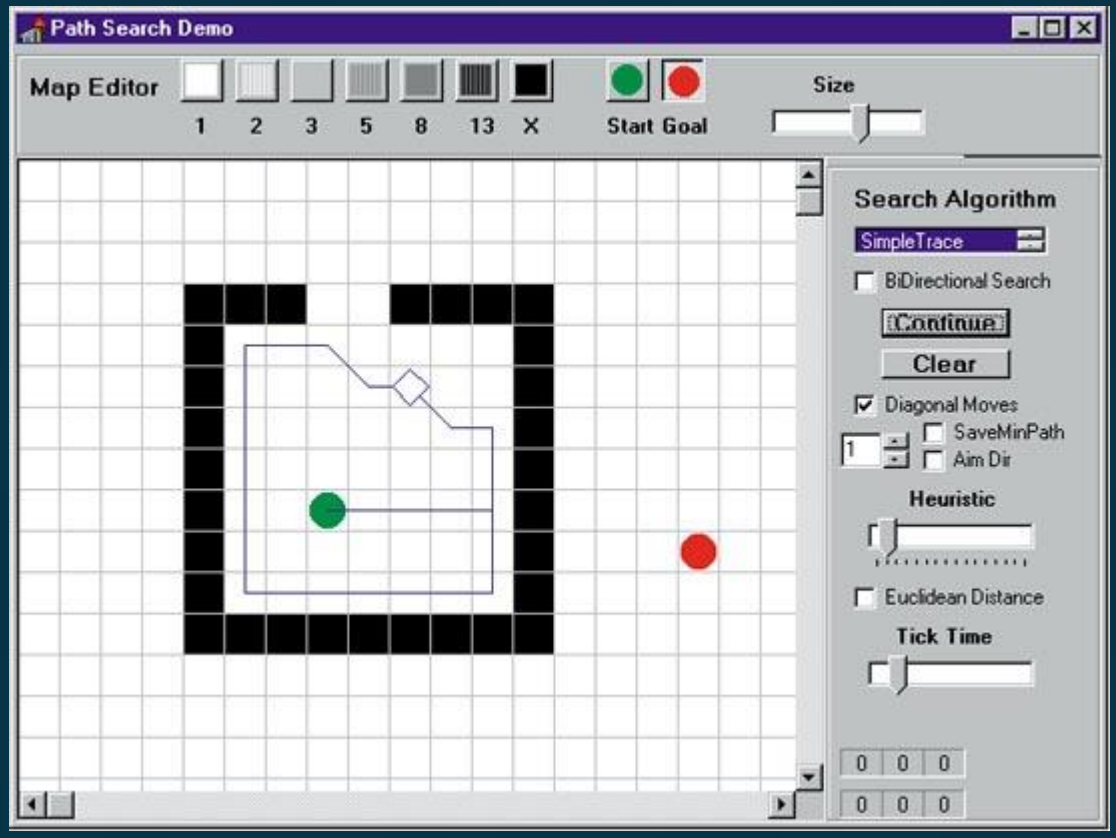
길 찾기

- 가면서 길찾기 : 장애물 따라 돌기



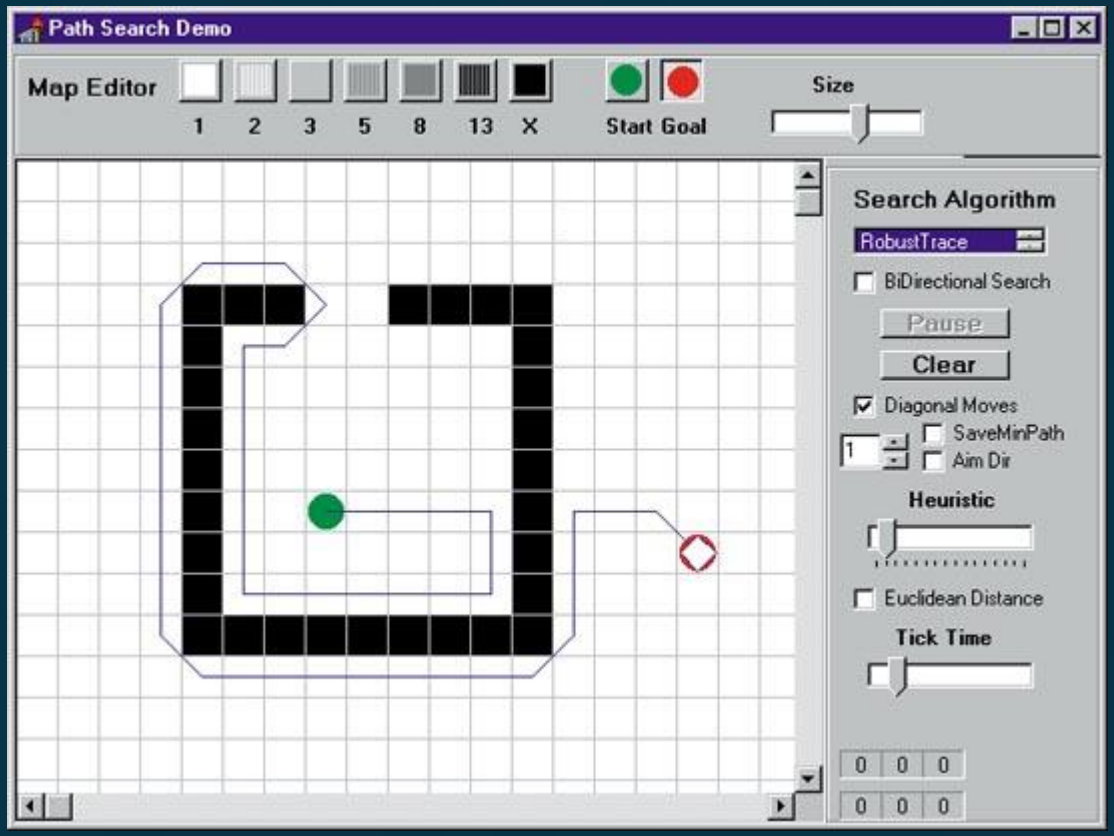
길 찾기

- 가면서 길찾기 : 장애물 따라 돌기



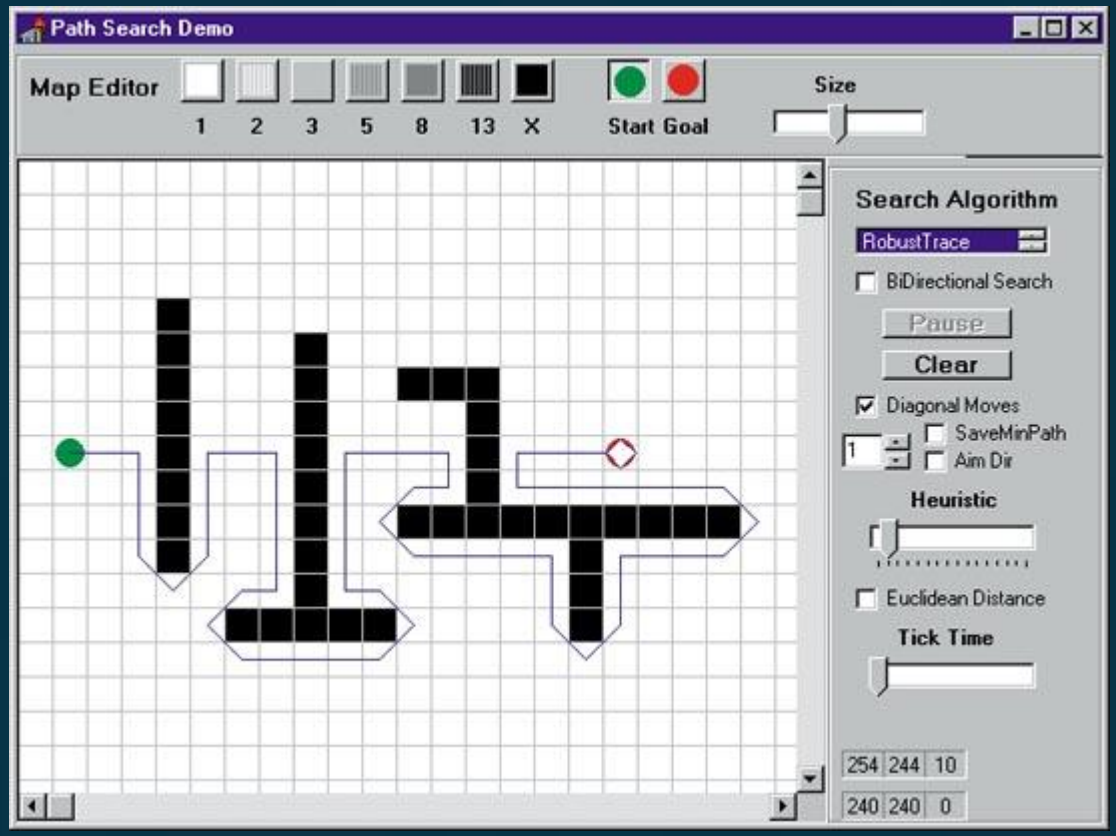
길 찾기

- 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

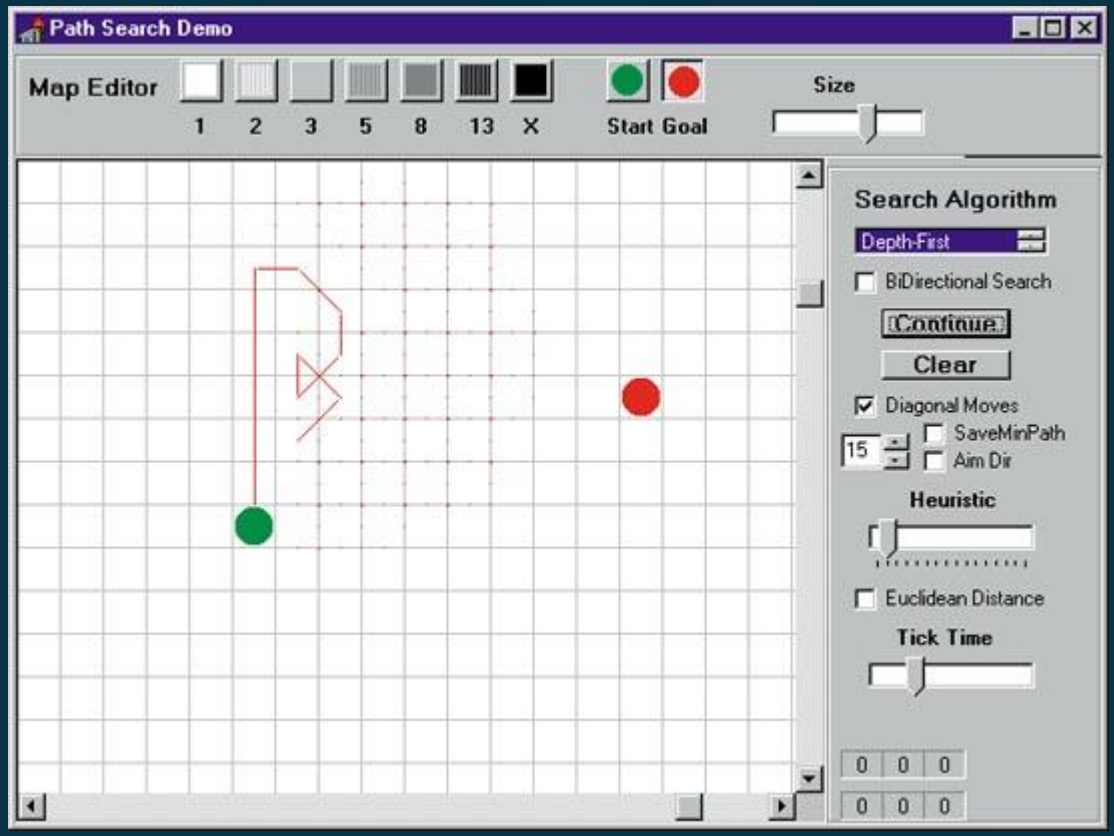
길찾기

- Depth-first search (2018-수목반)
  - 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

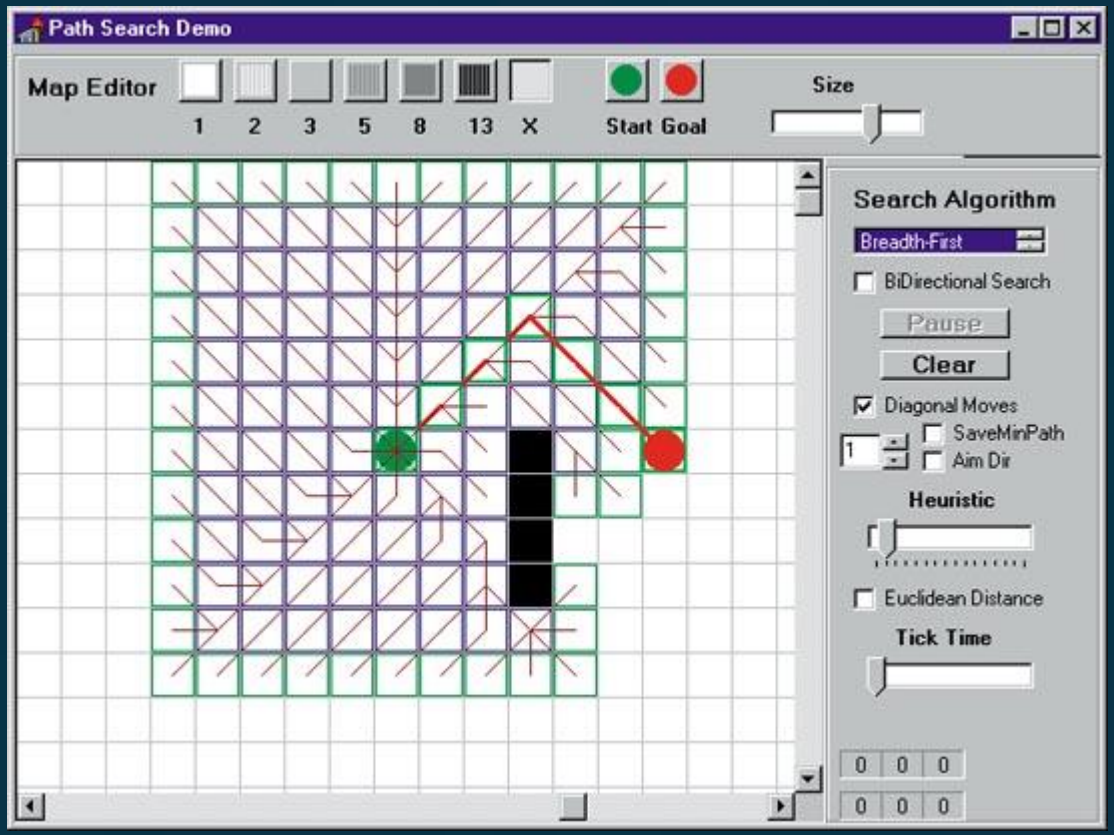
길 찾기

- 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
```

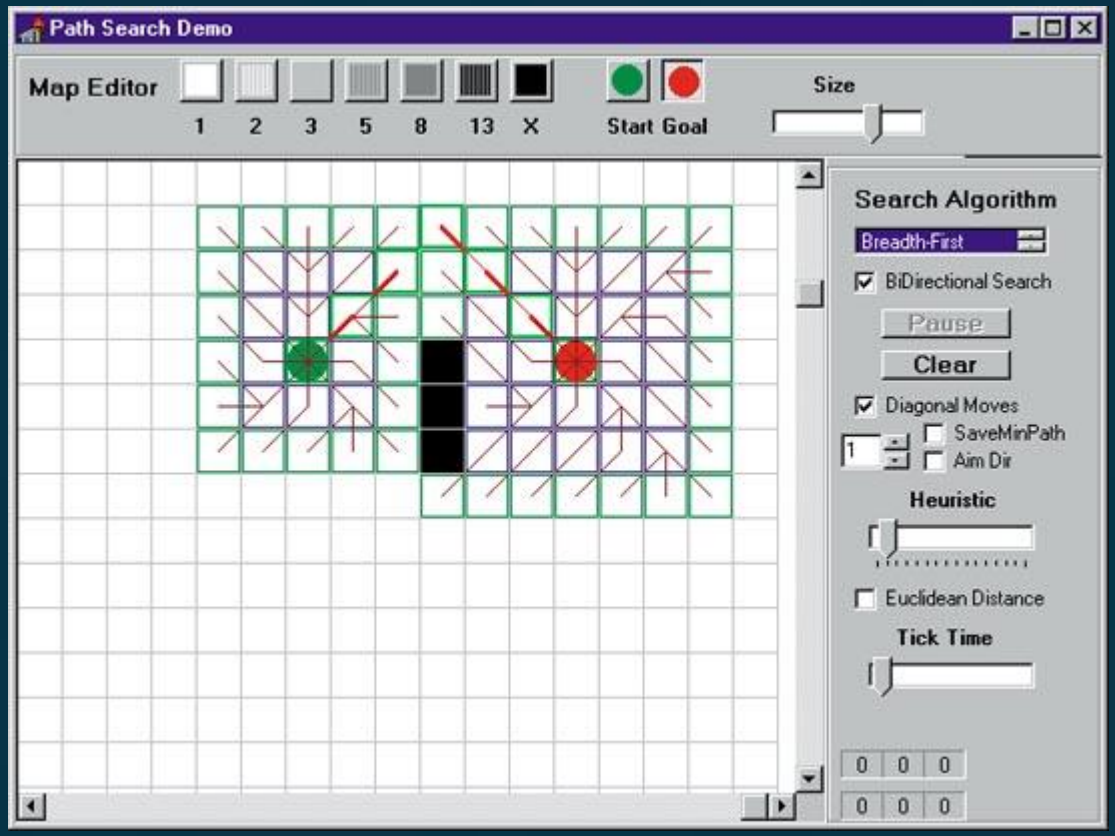
길 찾기

- Breadth-first search



길 찾기

- Breadth-first search :bidrectional



길찾기

- Dijkstra's 알고리즘

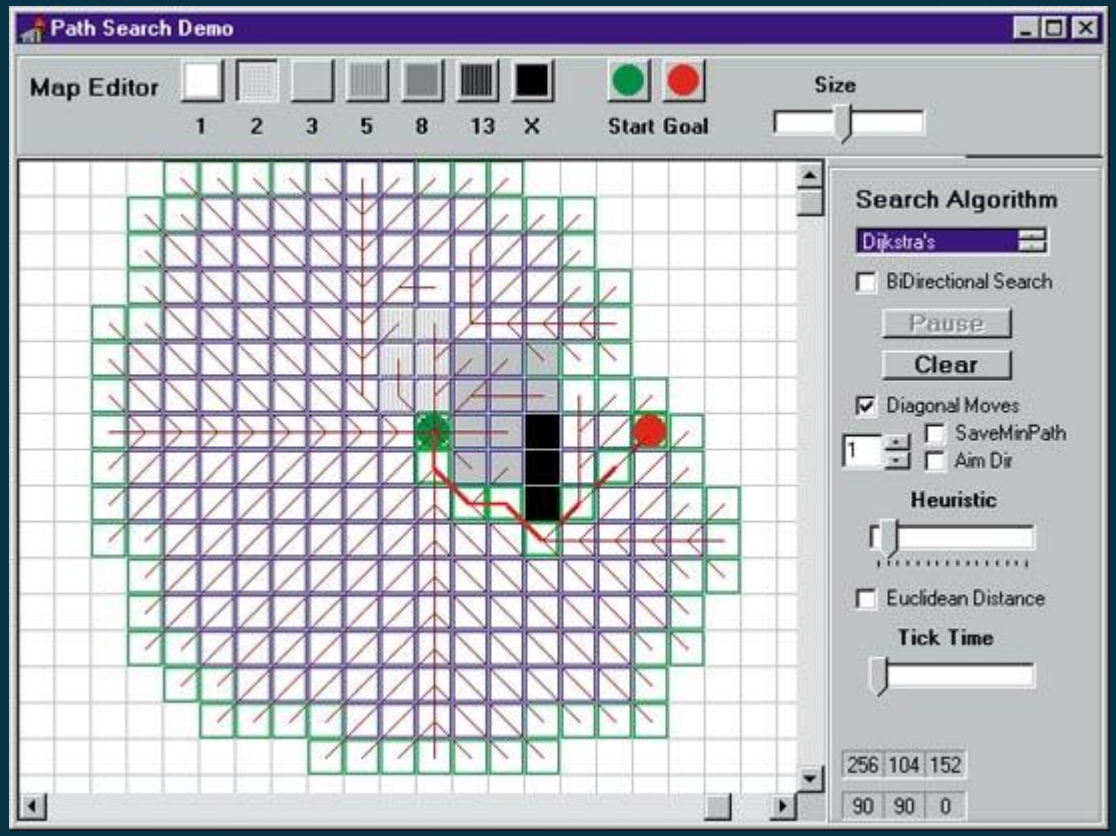
```
priority queue                Open

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

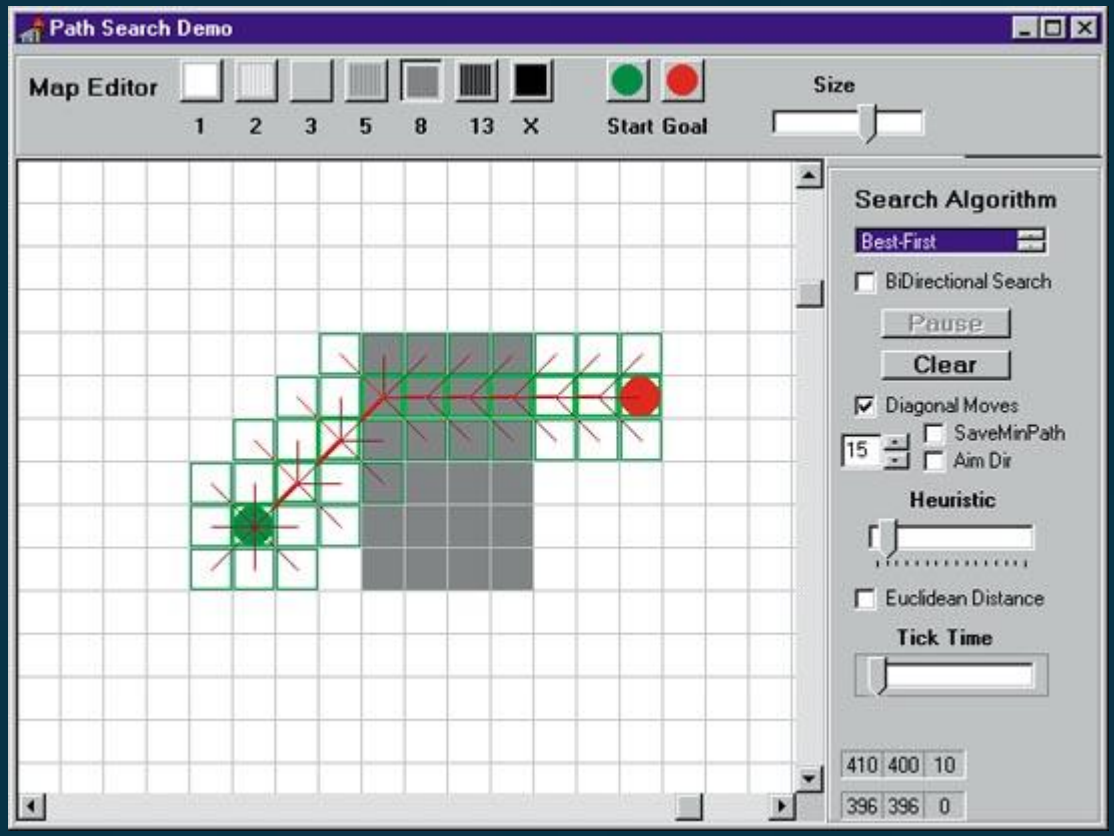


길찾기

- 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

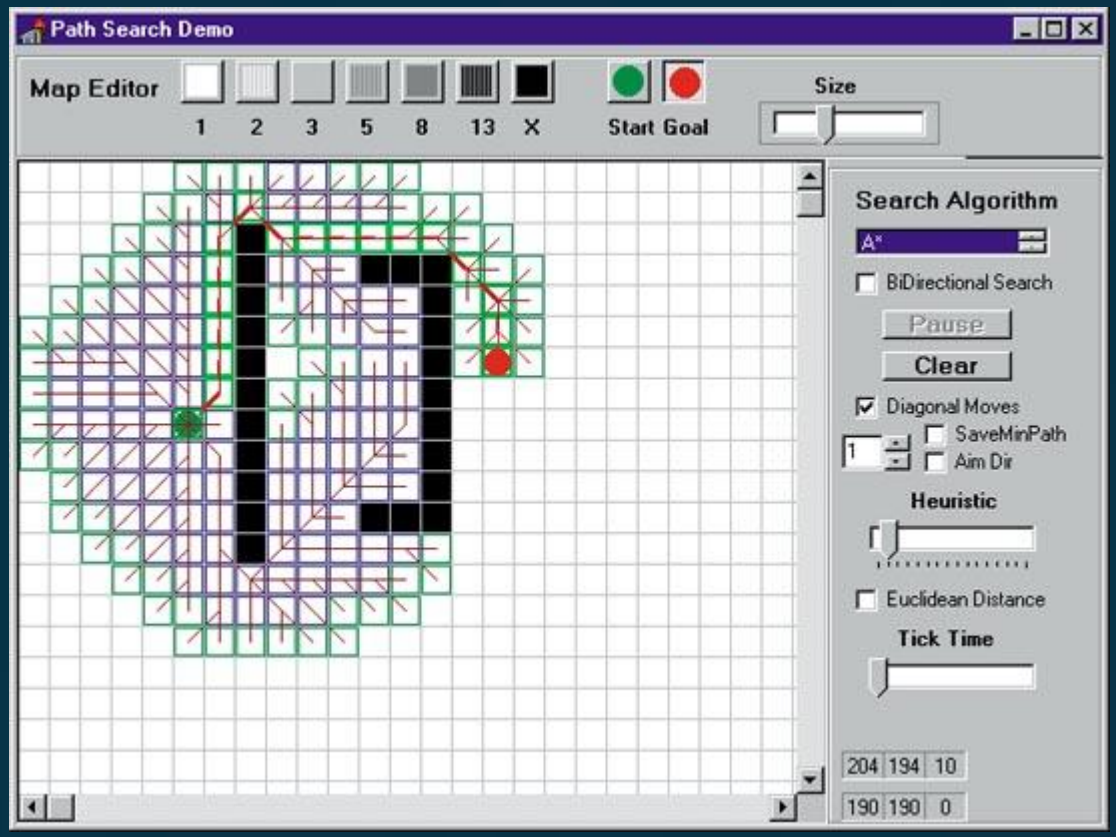
길찾기

- 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
```

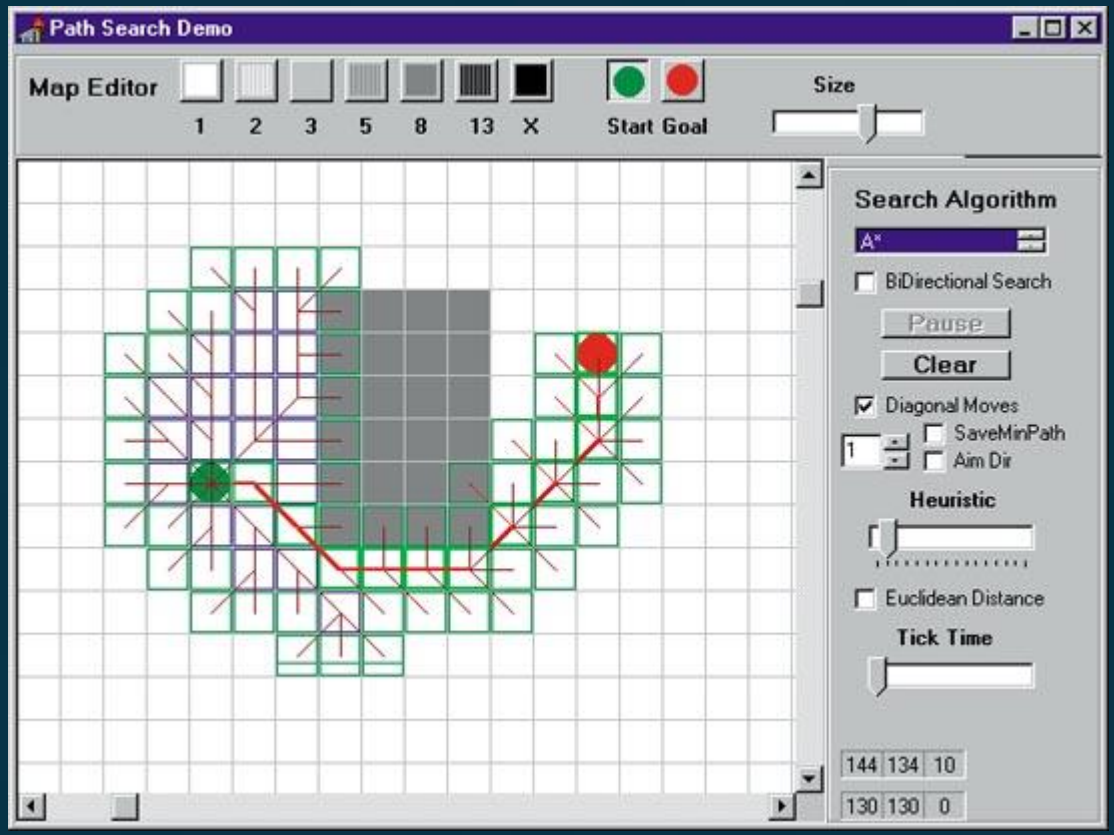
길 찾기

- 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

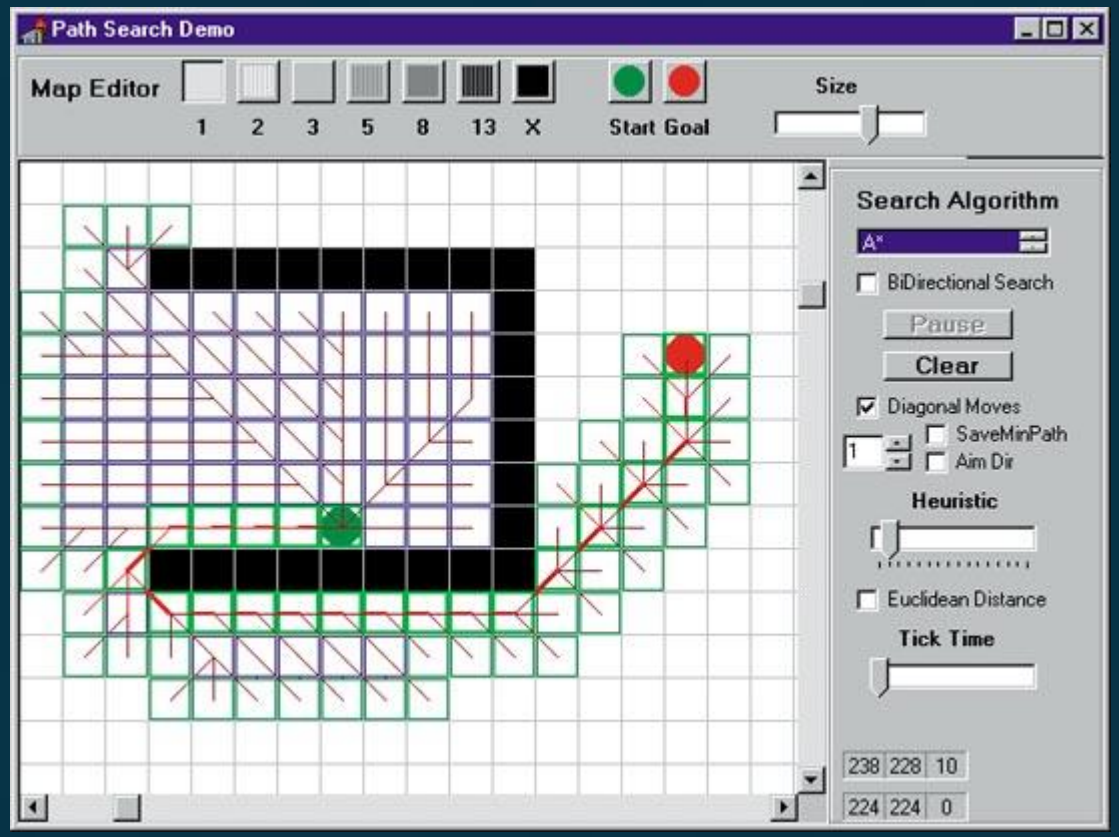
길 찾기

- 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

Timer (2019 화목)

- 컨텐츠 구현의 뼈대
  - 각종 이벤트 구현
  - Timing에 맞춘 동작들 구현
    - 캐스팅 타임, 쿨타임
    - 이동, 마법 시전, HP회복…
  - NPC AI
    - Timer 기반의 Finite State Machine
    - 주기적으로 상황 파악. (그러나 현실은…)

Timer(2018 화수)

```
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;
   }
```

클라이언트 프로그램

Timer

- 앞의 동작의 문제점
  - 10만개의 NPC가 출동하면?
    - 매 루프 마다 10만번의 if 문이 필요.
      - 서버는 GPU bound가 아니다.
      - Busy Waiting
    - 메모리 낭비, 캐시 문제, pipeline stall
- 해결책?
  - NPC Class에 heart_beat() 함수를 두고 일정 시간 간격마다 호출 되게 한다.

Timer

```
Loop(true) {
    if (objA.next_heal_time < current_time) {
    objA.m_hp = objA.m_hp + 1;
    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

실재

```
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.
- 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

```
Event_queue 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.
    - concurrency control도 겸한다.

Timer

- Timer Thread와 Worker Thread의 연동

```
NPC_Create()
  foreach NPC
   add_timer(my_id, MOVE_EVENT, 100)

Timer_thread()
  …
  overlap_ex.command = MOVE
  PostQueuedCompletionStatus(port, 0, id, overlapex)
  …

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

숙제 (#5)

- 게임 서버/클라이언트 프로그램 작성
  - 내용
    - 숙제 (#4)의 프로그램의 확장
    - 프로그램 수정
      - 전체 지도는 800 x 800
      - 클라이언트는 자기 말 주위 20x20 표시, 시야는 15x15
      - 200000개의 NPC : 1초 마다 한 칸 씩 이동
  - 목적
    - NPC 및 Timer 개념 사용 (PQCS와 GQCS를 사용)
  - 제약
    - Windows에서 Visual Studio로 작성 할 것
  - 제출
    - 제목을 “2019 게임서버[화목] 학번, 이름 숙제5”로 할 것
    - 5월 5일 오후 1시까지 제출 (1일 당 10% 감점)
    - Zip으로 소스를 묶어서 e-mail로 제출
      - 소스만(sdf, obj, log, manifest 같은거 제외!)
    - E-mail주소는 nhjung@kpu.ac.kr

다음 시간

- 실습
  - NPC
- 부하테스트
- 중간 발표
- DB
- SCRIPT