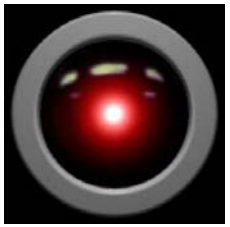
Game AI

305900
2007년 가을학기
11/29/2007
박경신

HAL


HAL (Heuristic ALgorithmic) in Standley Kubrick's film "2001: A Space Odyssey" (1968)

David




Monica "turns on" David's "love" function in which he virtually morphs from a robot to a boy

The *mecha* David in Steven Spielberg's film "A.I.: Artificial Intelligence" (2001)

들어가면서..

- 컴퓨터는 전체 가상세계에 대해 완벽하게 알고 있음 – 즉, 이것은 격투게임에서 모든 펀치를 회피하고, 슈팅게임에서 모든 총알을 교묘히 비킬 수 있음을 의미함
- 우리의 목표는 게임플레이어가 자신이 상대하고 있는 에이전트 (agent)가 지능적이라고 믿게 만들 수 있도록 AI를 설계하는 것임
  - 때로는 전혀 수고할 필요가 없는 경우도 있음 – Halo 인공지능 설계자들은 에이전트를 죽이는데 요구되는 타격 점수를 단순히 증가시킴으로써 더 지능적이라고 생각하도록 속일 수 있다는 것을 사용자테스트에서 발견함
  - 그러나, 실제 이는 매우 힘든 것이며 많은 AI 시스템이 특정 상황에서는 매우 "바보"같아지는 경우도 있음 – 예를 들어, Jedi Academy 게임에서 NPCs가 종종 죽고 마는 경우
  - 이렇게 하기 위해서, AI 시스템은 (사용자가 알아채지 못할 정도로) 약간의 속임수를 사용하기도 함
- NPCs는 아직도 FSM이나 규칙기반 시스템 (rule-based system)에 기반을 둔 제한된 규칙들을 따르고, 기존에 기대하는 이상 뭔가를 할 수는 없음 – 특히, RPG에서 사용자의 입력에 극히 일부에만 반응할 뿐이지 사용자와 오랫동안 일반적인 대화를 하기란 어려움

인공지능 (Artificial Intelligence)

- 인공지능 (Artificial Intelligence)
  - 사람이 수행했을 때 지능을 필요로 하는 일을 기계에게 시키고자 하는 학문/기술 즉, 생각하는 기계를 만드는 연구
  - 인간이나 지성을 갖춘 존재 혹은 시스템에 의해 만들어진 인공적인 지능
- 인공지능 발달사
  - 컴퓨터 발명이후 50 여년간 부단히 계속되는 신기술의 출현과 퇴조

– 논리학
– 최적화 이론
– 확률적 모형
– 탐색 이론
– 규칙기반 시스템
– 전문가시스템
– 퍼지 논리
– 신경회로망
– 유전자 알고리즘
– 카오스 이론
– 인공생명
– .....

게임 인공지능 (Game AI)

- **게임 인공지능 (Game AI)**
  - **게임 내에서 플레이어의 상대적 역할을 수행하면서 마치 플레이어가 다른 사람과 게임을 하고 있다는 착각을 유도하는 기술**
- **게임 인공지능의 역사**
  - 1970년대 - 아주 간단한 FSM 규칙만 사용 (Pong, Pac-Man, Space Invaders)
  - 1980년대 ~ 1990대 초반 - 게임 AI보다는 그래픽 기술과 하드웨어 요소에 초점을 둠
  - 1990년 중반 이후 - 하드웨어 성능이 향상되어 게임에 AI기술을 적용할 수 있는 환경이 조성

게임 인공지능 기술

- Finite State Machine (유한상태기계)
- Trigger System (트리거시스템)
- Production System (생성시스템)
- Search (탐색)
- Planning System (계획시스템)
- Multi-agent System (멀티-에이전트시스템)
- Robot (로봇)
- Artificial Life (인공생명)
- Evolutionary Algorithm (진화알고리즘)
- Flocking (무리짓기)
- Neural Network (신경망)
- Fuzzy Logic (퍼지 논리)
- Path Finding (경로탐색): A* Algorithm (A* 알고리즘)
- Scripting (스크립팅)

Mainstream AI techniques that have been applied to Games

- Finite State Machines (유한상태기계)
  - 주어지는 모든 시간에서 처해 있을 수 있는 유한 개의 상태를 가지고 주어지는 입력에 따라 어떤 상태에서 다른 상태로 전화시키거나 출력이나 액션이 일어나게 하는 장치 또는 그런 장치를 나타낸 모델
  - 즉, 시스템을 상태 (states), 입력 (inputs), 전이 (transitions)로 표현
  - 거의 모든 게임에 실제로 사용되고 있음
  - FSM 예
    - Pac-Man에서 모든 ghost에 동일한, 하나의 도피하기 상태가 있고, 각 ghost들은 각각에 대한 액션이 다르게 구현된, 자신만의 추적하기 상태가 있음. 플레이어가 입력한 힘의 알약들 중 하나를 먹게 되면 추적하기에서 도피하기로 전환하기 위한 조건이며, 시간이 줄어드는 타이머 입력은 도피하기에서 추적하기로 전화하는 조건임.
    - Quake 유형의 bot들은 무기찾기 (FindArmor), 건강약찾기 (FindHealth), 엄호물찾기 (SeekCover), 도망가기 (RunAway) 등과 같은 상태를 가짐. 또한 무기들도 로켓이 움직임(Move), 물체를 건드림 (TouchObject), 죽음 (Die) 등의 상태를 가짐.

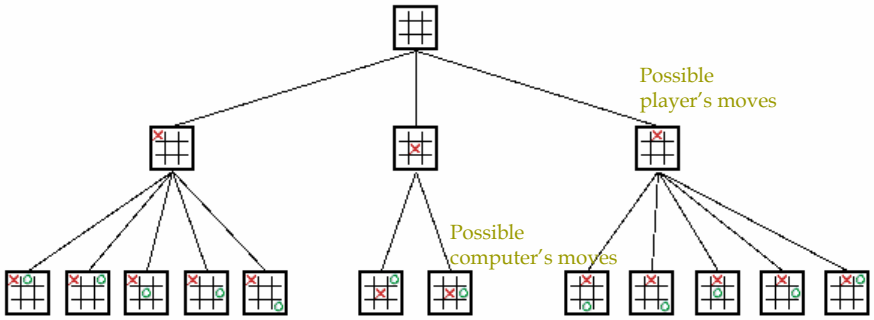
Mainstream AI techniques that have been applied to Games

- Trigger System (트리거 시스템)
  - 트리거(trigger)는 조건(condition)을 정의하는 객체
  - 트리거 시스템은 조건을 평가하고 반응을 수행 (action)하는 목적을 가진 중앙 집중화된 시스템
  - 조건의 예
    - 플레이어가 지점(x, y, z)의 반경 (radius) 이내에 존재, 적이 일정 거리 (distance) 이상 근접, 생명이 X개 이하, 적이 공격함
  - 반응의 예
    - 임의의 음향을 재생, 공격한 적으로부터 도망감, NPC X를 죽임, 위치 또는 플레이어/적에 특수효과를 적용
  - 평가
    - 여러 조건이 boolean 논리로 연결될 수 있음
    - 예를 들어, 적이 10미터 이내에 AND 적의 HP가 50%이하면 공격
    - 이벤트 메시지로 처리하거나 주기적 점검으로 평가
  - 트리거 편집기 예로 던전시스2 에디터가 있음

Mainstream AI techniques that have been applied to Games

- Production Systems (생성시스템)
  - 기본적으로 if-then-else 규칙을 사용해서 입력을 찾고, 그에 따른 새로운 입력을 생성해서, 그것이 다른 규칙을 유인해 내는 것으로 시스템이 추론을 실행할 수 있도록 함.
  - 어드벤쳐 게임에서 입력 (inputs)과 상태 (states)의 조합이 어떤 상황에 대한 무엇인가를 추론할 때 유용함
- Search (탐색)
  - 결과를 예측하기 위해 게임 보드의 모든 가능한 상태를 치환하는 것
  - 체스나 Pathfinding (경로탐색) 에서 사용함

E.g. Search : Tic-Tac-Toe Search Tree (with symmetric moves removed)



플레이가 움직일 수 있는 모든 경우와 컴퓨터가 움직일 수 있는 모든 경우를 번갈아 치환함. 그래서 각 단계별로 "fitness (적합한지)"를 예측하여, 컴퓨터가 "lookahead (예지능력)"을 가지도록 "the best move (최적의 움직임)"를 결정할 수 있게 함.

Mainstream AI techniques that have been applied to Games

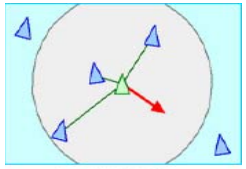
- Planning Systems (계획 시스템)
  - 시작 상태에서부터 끝 상태까지 가는데 필요한 일련의 task를 결정하는데 사용
  - 통합 목표를 위한 군대 중대를 조직하는데 유용함
- Mult-agent Systems (멀티-에이전트 시스템)
  - 제한된 지식을 가진 협조적인 에이전트와 상호작용하는 가운데 어떻게 불시의 행동이 발생하는지 연구
  - 에이전트는 본질적으로 센서를 가진 개별적인 FSMs
  - NPCs는 본질적으로 에이전트로 모델링함
- Robotics (로봇)
  - 게임 AI의 문제와 가장 비슷하나 훨씬 더 어려움 – 로봇은 실 세계에서 들어온 입력을 사용함
  - 게임과 비슷하게, 로봇에 원하는 기능을 구현할 수 있도록 우리가 고려해야 할 가장 적은 량의 세계 정보가 무엇인지 이해하는 것이 중요함

Mainstream AI techniques that have been applied to Games

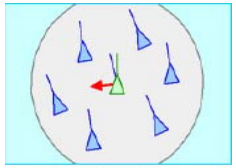
- Artificial Life/A-Life (인공생명)
  - 살아있는 시스템을 AI 에이전트로 사용하는 멀티-에이전트 시스템
  - SimCity와 SimAnt 게임에서 사용됨
- Evolutionary Algorithm (진화 알고리즘)
  - 진화와 자연 선택에 근거를 둔 여러 최적화 및 검색 알고리즘들을 통틀어서 일컫는 용어로, Genetic Algorithm (유전자 알고리즘), Evolutionary Computing 등이 이에 속함
  - 진화 알고리즘은 크고 복잡한, 또는 제대로 이해되지 못한 검색 공간과 비선형 문제들에 대한 최적의 해를 찾음
  - RTS 게임에서 사람 플레이어의 약점을 대상으로 하는 AI 전략 또는 NPC의 다변성으로 사용
  - 유전자처럼 시스템의 특성을 부호화하고 보다 완벽한 시스템으로 진화하기 위해 적정성 기능 (fitness function)와 함께 돌연변이(mutation)와 교배(crossover)를 사용
  - 보다 지능적인 인공생명체 (A-life creature)로의 진화에 사용됨

Mainstream AI techniques that have been applied to Games

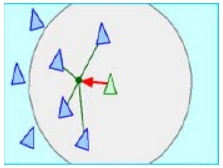
- Flocking (무리짓기)
  - 새 떼나 물고기 떼 - 또한, 군사 시뮬레이션에서 전쟁의 군대 편대가 어떻게 형성되고 움직이는지 -와 같은 무리들의 집단 행동을 자연스럽고 유기적인 방식으로 흉내 내어 묘사한 에이전트 기법
  - Unreal Tournament, Half-Life 등 게임에서 캐릭터 무리의 행동을 자연스럽게 표현하기 위하여 사용
  - 무리짓기 규칙
    - 분리 (separation) 주변 다른 개체들과 너무 가까워지지 않도록 함
    - 정렬 (alignment) 주변 다른 개체들과 평균적으로 같은 방향으로 이동함
    - 응집/결합 (cohesion) 주변 다른 개체들과 평균적인 거리의 위치로 이동함
    - 회피 (avoidance) 장애물이나 다른 개체들에 부딪히지 않도록 방향을 조정함



[그림] 분리 규칙



[그림] 정렬 규칙



[그림] 응집 규칙

Mainstream AI techniques that have been applied to Games

- Neural Network (신경망)
  - 인간의 두뇌 자체에서 영감을 얻어 모델화한 정보처리 시스템. 외부로부터 받아들이는 입력에 대하여 동적 반응을 일으킴으로써 필요한 출력을 생성함
- Fuzzy Logic (퍼지 논리)
  - 참/거짓, 온/오프, 예/아니오, 부정/긍정같은 기존의 이분법적인 개념이 아닌 "어느 정도"나 "얼마나"같은 애매함을 기반으로 함
  - 퍼지 집합 (Fuzzy Set)
    - 전통적인 집합이론에서 주어진 개체는 그 집합에 속하거나 속하지 않았지만 퍼지 집합 이론에서는 주어진 개체가 퍼지 집합에 속하는 정도가 연속적임 – 예를 들어, 한 생물이 아주 약간 배가 고프다고 할 때, 그 생물은 소속도 0.1로 배고픈 집합에 속하고 반면 배가 많이 고프다면 소속도 0.9로 배고픈 집합에 속함
  - 게임 NPC의 감정이라던가, 무리짓기 알고리즘, 퍼지기반 FSM에 사용

Pathfinding

- Pathfinding (경로찾기)
  - 문제: NPCs는 3차원 지형에서 A 지점으로부터 B 지점까지 어떻게 이동(navigate) 해야 할 지 실시간으로 찾아내야 함
  - 무엇이 가장 효과적인 방법인가?

Pathfinding: A* Algorithm

- 두 지점들 사이의 경로를 찾는데 일반적으로 가장 빠른 목표 지향적인 (directed) 알고리즘
- 맵 (Map), 그래프 (Graph)
  - A* 가 두 지점 사이의 경로를 찾고자 할 때 사용하는 공간
- 노드 (Node)
  - 맵 상의 위치를 표현하는 자료 구조
- 거리 (Distance), 휴리스틱 (Heuristic)
  - 탐색되는 노드의 적합성 (Fitness)를 평가하는데 사용
- 비용 (Cost)
  - 어떤 경로를 택할 것인가에 대한 것인지에 대한 비용 – 예를 들어, 시간, 에너지 등

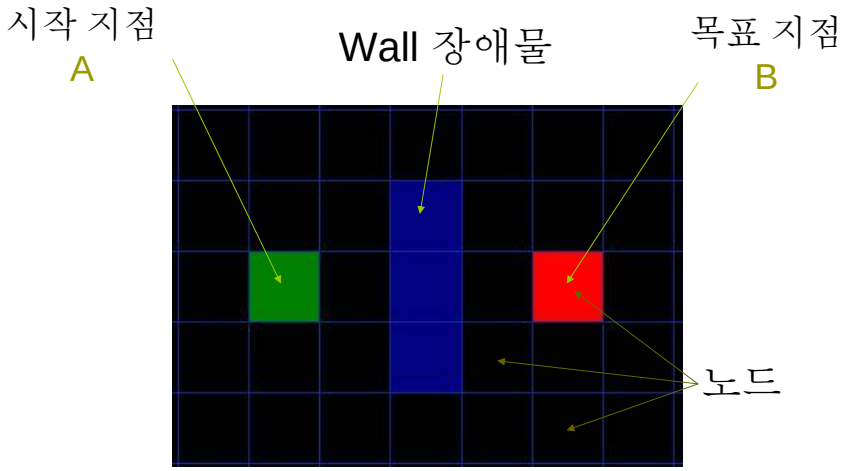
Pathfinding: A* Algorithm

- G (Goal)
  - 시작 노드로 부터 이 노드까지 오는데 드는 비용
- H (Heuristic)
  - 이 노드에서 목표까지 가는데 드는 '추정된' 비용
- F (Fitness)
  - 이 노드를 거쳐가는 이 경로의 비용에 대한 최선의 추측
  - F = G + H
- 열린 목록 (Open List 또는 TODO List)
  - 아직 탐색하지 않은 노드들의 집합
- 닫힌 목록 (Closed List 또는 Done List)
  - 이미 탐색한 노드들로 구성

Pathfinding: A* Algorithm

1. 시작 지점을 노드 A로 둔다
2. A에 F, G, H 값들을 배정한다
3. A를 열린 목록에 추가한다 (이 시점에서 열린 목록에는 A밖에 없다)
4. 열린 목록의 노드들 중 최선의 노드 (즉, F 값이 가장 작은 노드)를 S로 선택한다
   a. S가 목표 노드이면, 경로를 찾은 것이므로 알고리즘을 끝낸다
   b. 열린 목록이 비어있다면, 경로를 찾을 수 없는 것이므로 역시 알고리즘을 끝낸다
5. S에 연결된 유효한 노드를 C로 선택한다
   a. C에 F, G, H 값들을 배정한다
   b. C가 열린 목록에 들어있는 지 점검한다
      a. 만일 들어있다면, 새 경로가 더 효율적인지 (즉, F 값이 더 적은지) 점검하고 만약 그렇다면 경로를 갱신한다
      b. 들어있지 않다면, C를 열린 목록에 추가한다
   c. 5단계를 S에 연결된 모든 유효한 자식 노드들에 대해 반복한다
6. 4단계부터 다시 반복한다

E.g. Given a terrain map as follows:

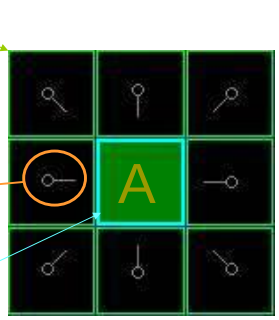


격자 구조의 2차원 맵

A Well Known Algorithm in AI called A*

Main idea: 시작 지점 A에서부터 근접한 노드를 점검하여 가장 짧은 경로를 찾는다 – 일반적으로 우리가 찾고자 하는 목표지점의 바깥쪽을 탐색한다

1. 시작 노드 A를 열린 목록 (**TODO List**)에 추가한다 - **TODO List는 점검해야 할 노드들의 목록**
2. 시작 지점에 근접한 (wall, water, illegal terrain을 제외한) 가능한 노드들를 찾아서 열린 목록에 추가한다. 가장자리에 있는 모든 자식 노드들이 중앙에 있는 부모 노드 (parent square)를 가리키고 있다.
3. 시작 노드 A를 열린 목록에서 제외하고 닫힌 목록 (Done List)에 추가한다 - **DONE List는 이미 점검한 노드들의 목록**



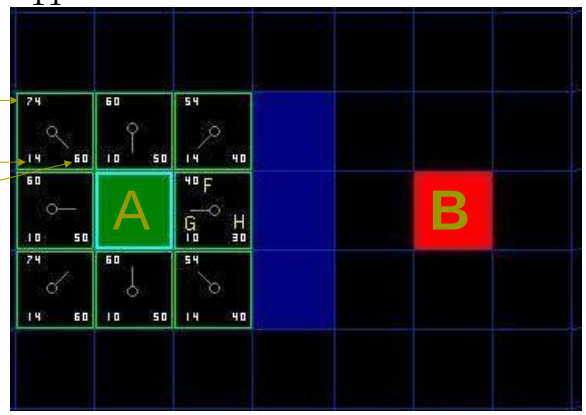
Now choose one of the TODO list items to explore next- but which one?

- Path Scoring (경로 점수 계산)
  - A에 근접한 모든 노드들한테 가는데 드는 비용 (Cost)을 계산
  - F = G + H
  - F (Fitness) 이 노드를 거쳐가는 이 경로의 비용에 대한 최선의 추측
  - G (Goal) A로부터 현재 노드까지 오는데 드는 비용
  - H (Heuristic) 이 노드에서 목표지점 (B)까지 가는데 드는 '추정된 (Estimated)' 비용
  - 이 예의 경우 Hueristic은 "Manhattan Distance" - 즉, 도시의 거리를 걷듯이 목표 지점까지 수직과 수평으로 걸어간 거리

Calculate F,G,H at each of the TODO List items

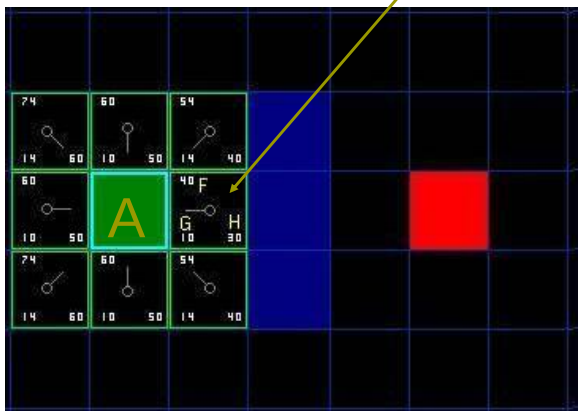
이 예의 경우 아래와 같이 가정함
수직/수평으로 움직인 비용 = 10
대각선으로 움직인 비용 = 14

F
G
H



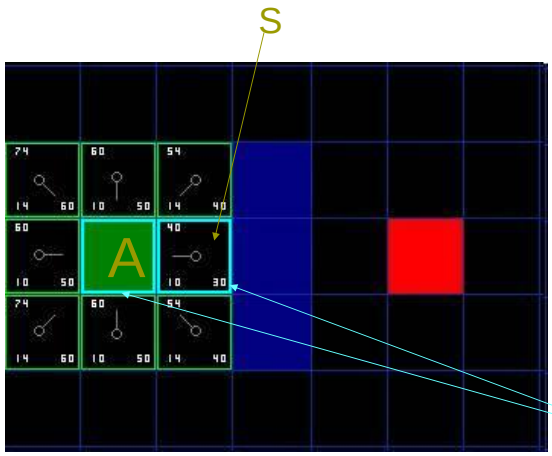
Select the node (S) with the smallest F from the TODO List

이 노드가 가장 작은 F=40를 가짐. 이것을 선택된 노드 S라 부름.

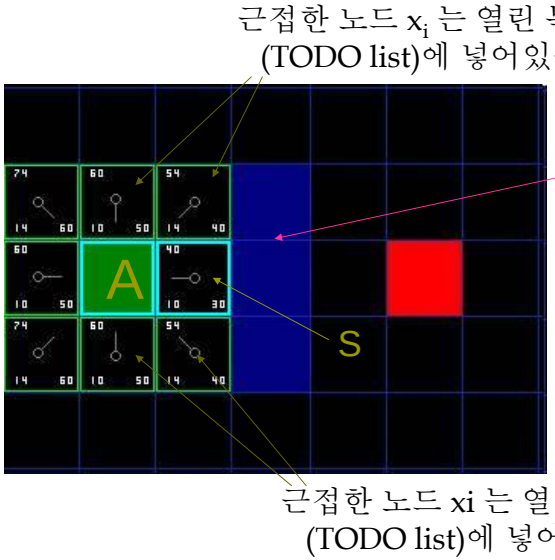


초기 단계에서 A* 알고리즘을 한 번 수행하고 난 후의 모습
- S는 목표에 가장 가까운 가능성이 높아서 선택된 노드

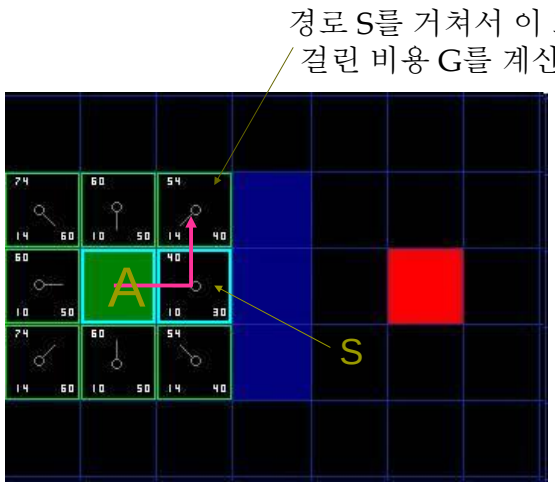
Select S, Remove it from TODO List, Add it to DONE List



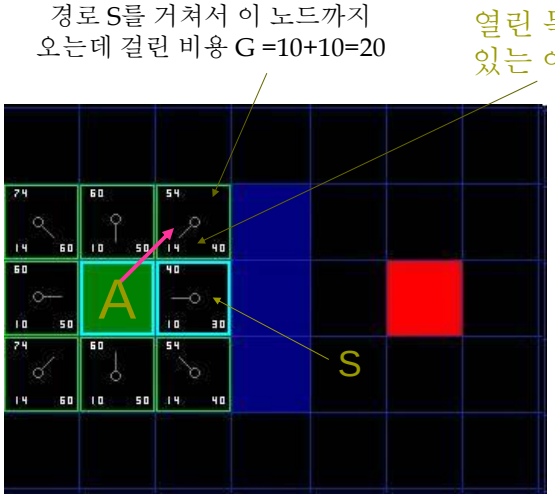
Check adjacent squares from S, and add to TODO List unless they are already on the list.



So look at each of the Xi that are already on the TODO list…

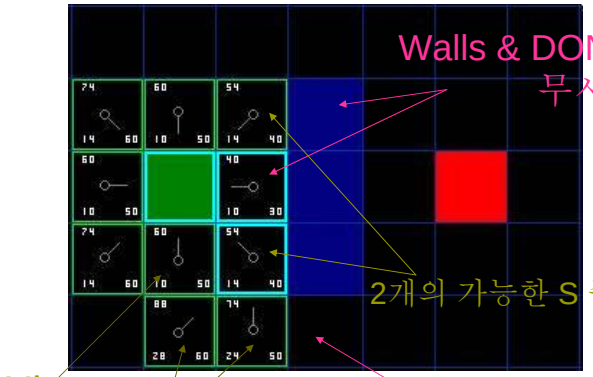


…and see if the current path (A→S→Xi) is better than the previous ones (A→Xi)



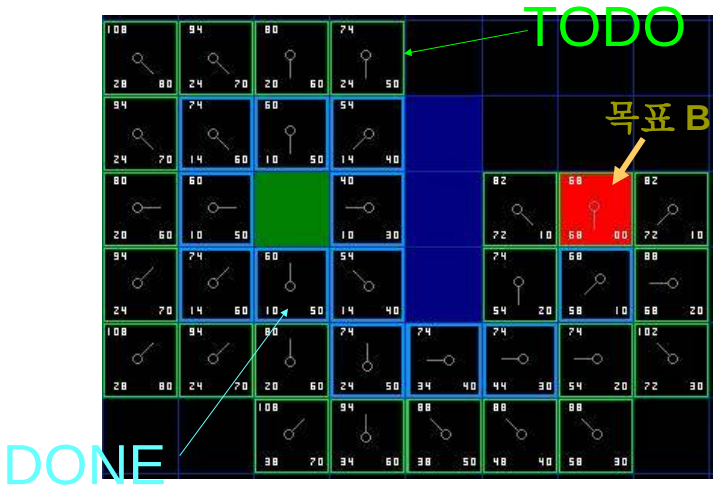
이 경우에 답은
A→S→Xi 가 A→Xi 보다
비용이 더 들으므로 그냥
열린 목록을 그대로 놔둔다 –
즉, A→S→Xi 경로는
무시하고 A→Xi 경로만 열린
목록에 두고 계속해서 경로
찾기를 수행한다

Now repeat the algorithm: Select node (S) with the smallest F in the TODO List, add to DONE List, and examine its adjacent squares (Xi). Add new Xi if not on TODO list.

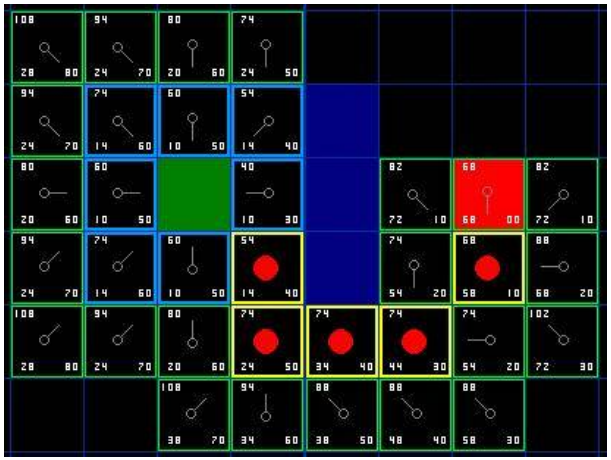


A* 알고리즘을 한 번 더 수행하고 난 후의 모습

Repeat until Destination B is reached.



Compute the final path by following the parents from B.



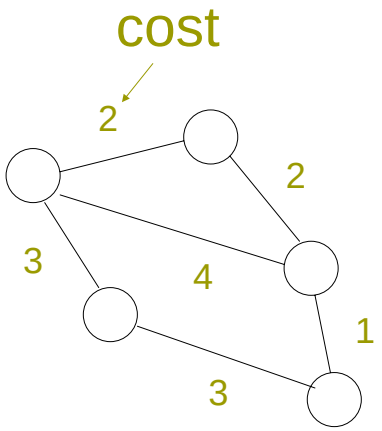
시작 노드에서 목표 노드로 길 찾기를 수행한 모습

The Whole A* Algorithm Review at your Leisure

1. Add the starting square to the TODO list.
2. Repeat the following:
   a. Look for the lowest F cost square on the TODO list. We refer to this as the current square.
   b. Switch it to the DONE list.
   c. For each of the 8 squares adjacent to this current square …
      - If it is not walkable or if it is on the DONE list, ignore it.Otherwise do the following.
      - If it isn't on the TODO list, add it to the TODO list. Make the current square the parent of this square. Record the F, G, and H costs of the square.
      - If it is on the TODO list already, check to see if this path to that square is better,using G cost as the measure. A lower G cost means that this is a better path. If so, change the parent of the square to the current square, and recalculate the G and F scores of the square. If you are keeping your TODO list sorted by F score,you may need to resort the list to account for the change.
   d. Stop when you:
      - Add the target square to the TODO list, in which case the path has been found, or
      - Fail to find the target square, and the TODO list is empty. In this case, there is no path.
3. Save the path. Working backwards from the target square, go from each square to its parent square until you reach the starting square. That is your path.

Note: Maps need not necessarily be Gridded.

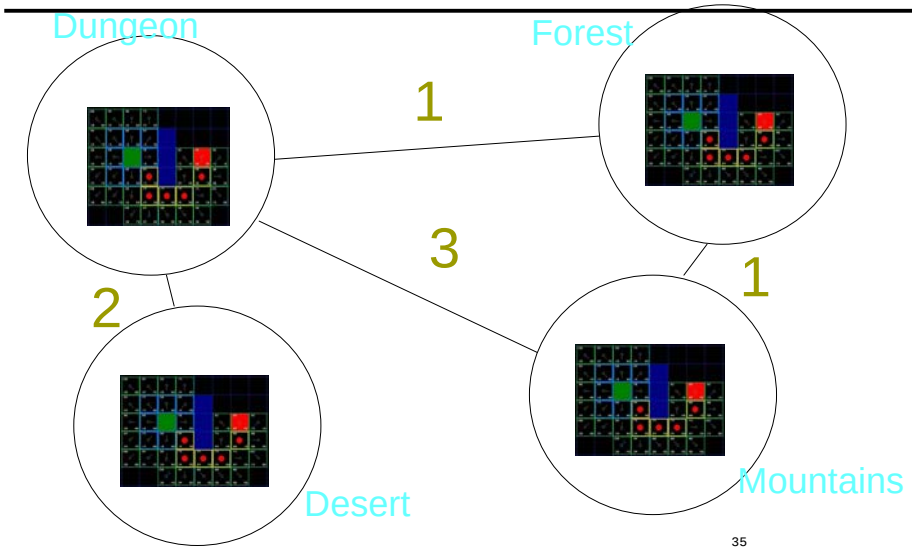
- Maps은 (예를 들어, 거대한 던젼에서 방들간의) waypoints
- Navigational Maps은 볼록한 다각형들 (convex polygons)의 지형
- 각 다각형은 A* 알고리즘의 노드로 고려함



A* Mods

- 열린 목록을 정렬해서 유지한다
  - 만약 찾아야 할 공간 (search space)이 커진다고 생각된다면, 가장 작은 F값을 찾기 위해 열린 목록을 전체적으로 자세히 살펴보는 (traverse) 것을 줄이기 위해 정렬 (sort)를 하고 있는 것이 좋음
  - 삽입과 삭제가 쉬운 링크드 리스트를 사용하는 것이 좋음
- 만약 모바일 유니트 (mobile units)가 같이 움직인다면?
  - 모바일 유니트의 움직임에 다른 코드를 사용하고 최종 목표 지점까지의 경로를 계산할 것
- 가변적인 지형에 대한 비용 (Variable Terrain Costs)
  - G는 늪이나 AI가 많은 군사를 잃게 되는 지역을 지나가는 비용을 고려해서 계산해야 함. AI는 군사를 잃게 되는 지역에 대한 정보인 **Influence Map**을 놓치지 않고 가지고 있어서, G 계산에 드는 비용을 증가시켜 전체적으로 자세히 살펴보는 (traverse)를 피하도록 함.
- 거대한 지형 핸들링
  - Hierarchies of maps – Tiered A* Pathfinding
  - Pre-computed paths

Tiered A* Pathfinding



Pre-Computed Paths

	Dungeon1	Dungeon2	Dungeon3
Dungeon1	x	D1→D2 가는데 가장 적은 비용의 경로	D1→D3 가는데 가장 적은 비용의 경로
Dungeon2	반대의 경로는 대칭적이거나 아닐 수도 있음	x	D2→D3 가는데 가장 적은 비용의 경로
Dungeon3			x

Complementary Navigational Strategies

- 시간에 따라 쇠퇴하는 디지털 자취 (Digital scent)
  - 사용자가 공간에서 걷고 있을 때, 그들의 자취를 3차원 벡터 경로로 남기며, 그 벡터 경로의 꼬리 부분은 시간에 따라 사라지게 함
  - NPCs는 만약 그들이 따라가고자 하는 경로의 꼬리 노드와 충분히 가까운지를 알 수 있음
  - 만약 사용자가 멀리 도망치는데 A*를 거치지 않고 NPC가 플레이어를 찾기 위한 방법이 필요할 때 좋음
- NPCs가 죽지 않도록 방지함
  - NPCs는 궁극적으로 에이전트임 – 에이전트는 (wall, floors 등과 같은) 자기 주변 환경을 감지하기 위해 센서를 가지고 있음
  - 또는, 환경에 에이전트를 끌어들이거나 물리치기 위해 "attractors" 와 repulsors" 를 넣을 수도 있음 – 예를 들어, 용암에 위해 죽는 것을 방지하기 위해 사용할 수 있음
- 미리 지정된 정찰 루트 (Pre-defined patrol routes)
  - 에이전트가 idle 모드일 때는 미리 지정된 정찰 루트를 따라갈 수 있게 함

Possible Agent Navigation AI

- IDLE 상태일 때, 미리 지정된 N개 정찰 루트 중 한 개를 수행
- 만약 너가 AI로 부터 일정한 거리 밖에 있다면AI의 근접 센서 (proximity sensors)를 끔
- With proximity sensors:
  - 만약 너 또는 적을 감지했으면, "attack (공격) " 상태로 바뀌고 너한테 옴
  - 만약 너가 총을 쏘면, "evade (피하기)" 상태로 바뀌고, 에이전트는 너의 무기의 총 쏜 방향의 수직으로 움직임
  - 만약 너가 일정 영역 밖에 있다면 (그래서 움직임이 좀 어려워지는), 사용자의 자취를 따라감
  - 만약 에이전트가 자취를 잃어버렸는데 찾고자 한다면, 너의 위치로부터 A* 알고리즘을 계산하여 속임수를 씀
  - 그러지 않다면 지정 루트를 다시 수행 – 그런데 문제는, 어떻게 너가 다시 예전의 정찰 루트로 돌아갈 수 있는지? 그래서, 에이전트는 본인의 자취를 유지관리하고 있어서 만일의 경우에 예전 정찰 루트로 돌아갈 수 있게 함
- 이 모든 경우에 환경 센서가 항상 작동하고 있어서, 에이전트가 실수로 용암으로 걸어가거나 심해로 빠지거나 다른 에이전트로부터 도망가는 일을 없도록 함
- 속임수 모드 (cheat mode)에서 게임을 플레이해서 어떻게 에이전트들이 실제 게임에서 행동하는지를 관찰해 볼 것

Teaching the Computer to Aim

- Dead reckoning (추측 항법)
  - 현재의 시간에서 Entity (예, 게임 플레이어)의 현 위치 (position)와 속도(velocity)와 가속도(acceleration)를 기반으로 다음 위치를 예측
- 이 추측 항법을 사용하여, AI가 히트할 확률 (odds of a hit)을 높이게 어디로 총을 쏠 수 있을지를 결정할 수 있음
- 3D dog fight games같은 FPS (First Person Shooter)에서 매우 중요함
- 그러나, 또한 조준을 살짝 무작위로 바꿔줘서 컴퓨터가 완벽한 저격병으로 느껴지지 않도록 함

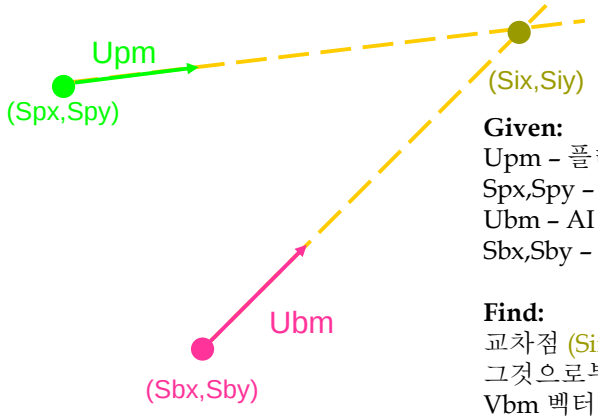
Calculating Dead Reckoning

- 물리에서 잘 알려진 운동 (motion) 공식:
  - $v = v_0 + at$
  - $\mathbf{x}(t) = \mathbf{x}_0 + vt$

$$\mathbf{x}(t) = \int_0^t (v_0 + at)dt = \mathbf{x}_0 + v_0 t + \frac{1}{2} a\, t^2$$

- 추측항법 (dead reckoning)을 계산하기 위한 방법:
  - The precise method
  - The approximate method

Precise Method (Assume a 2D case)



Given:
Upm – 플레이어의 속력 (speed)
Spx,Spy – 플레이어의 위치 (position)
Ubm – AI 총알의 속력 (speed)
Sbx,Sby – AI 총알의 위치 (position)

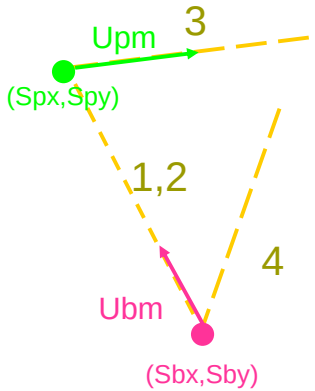
Find:
교차점 (Six,Siy) &
그것으로부터, 총알을 쏘기 위해
Vbm 벡터를 결정함

Note: 사용자와 총알은 가속도를 가질 수 있으나, 간단하게 하기 위해서 가속도는 0이라 가정함

- Ubm (즉, 총알의 속력)은 벡터의 길이 (magnitude)임
- 총알의 속도 (velocity)는 벡터이므로 x, y 부분을 가지며, 이것이 총알이 나가는 방향 (direction)을 결정하게 되는 것임
- Ubm = sqrt(Vbx^2+Vby^2)
- 교차점 (Six,Siy)에서 만나는 시간 (the time of the intersection) t를 계산해야 함
- 플레이어는 종종 실수하는 경향이 있기 때문에, 실제로는 이 방법이 사용되거나 필요로 하진 않음. 그래서 대신 approximation 방법이 사용됨

Approximate Method

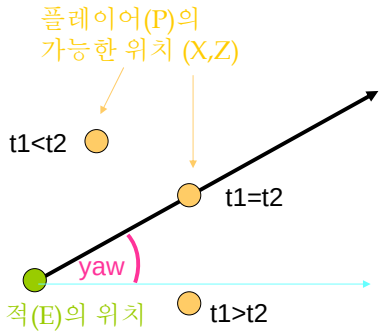
Given:
Upm – 플레이어의 속력 (speed)
Spx,Spy – 플레이어의 위치 (position)
Ubm – AI 총알의 속력 (speed)
Sbx,Sby – AI 총알의 위치 (position)



1. 플레이어와 AI 총알 사이간의 거리 (distance)를 계산함
2. $S(t)=ut+(at^2)/2$ 공식을 사용하여 시간 (t)을 찾음 – 이는, 주어진 총알의 속력에서 총알이 플레이어에게로 날아가는 것을 고려한 것임
3. Upx, Upy, Spx, Spy, t와 위의 공식을 사용해서, t 시간에 플레이어가 어디에 위치할 지를 계산함
4. 그 장소로 총알을 쏨

Normally you need the enemy to turn to face the player before it is allowed to shoot.

- 2차원 Hemi-plane 테스트를 사용하여 플레이어가 어떤 쪽에 있는지를 테스트함
- 선의 공식 (parametric equation of line):
  - P.x= position of E.x + cos(E.yaw) * t
  - P.z= position of E.z + sin(E.yaw) * t
- t1과 t2를 계산:
  - t1= (P.x - position of E.x) / cos(E.yaw)
  - t2= (P.z - position of E.z) / sin(E.yaw)
- 만약 t1=t2, 플레이어는 적에게 죽을 것임
- 만약 t1 > t2, 플레이어는 line의 한 편에 있을 것임
- 만약 t1 < t2, 플레이어는 line의 다른 한편에 있을 것임



But if you really want the exact ANGLE you can calculate it as follows

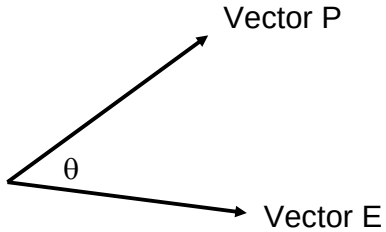
Magnitude

Dot product

$$E \cdot P = \|E\| \|P\| \cos\theta$$

$$\theta = \text{acos}\left(\frac{E \cdot P}{\|E\| \|P\|}\right)$$

$$\theta = \text{acos}(E \cdot P), \textit{where } E, P \textit{ are unit vectors}$$



NOTE: 이 공식은 적 (Enemy)이 플레이어의 왼쪽인지 오른쪽에 있는지를 알려주진 않으므로, 이전 슬라이드의 공식을 사용해서 어느 편에 있는지를 알아낼 것

Scripting

- 게임의 자료나 로직을 게임의 원본 코드 외부에서 지정하는 기법으로 과거에는 스크립팅 언어를 개발팀이 직접 만들 때가 많았지만, 요즘은 Python이나 Lua 같은 기존 언어를 사용하는 경우가 많아지고 있음
- 스크립트 언어는 초기화 파일에서 변수와 게임 데이터를 읽기 위한 빠르고 쉬운 방법으로 사용
- 스크립트 언어 예
  - Lua: C++ 코드와 접착이 가능하며 문법이 간단하고 속도가 빨라 근래 가장 인기있는 스크립트 언어
  - Python: C++ 코드와 접착이 가능하며 문법이 간단하고 많이 라이브러리가 지원되고 있는 스크립트 언어
  - XML: 확장 가능한 마크업 언어
  - 기타: Unreal Script, GameMonkey, AngelScript

Scripting

- 스크립팅의 장점
  - 코드를 다시 컴파일하지 않고도 게임 로직을 변경하거나 테스트 할 수 있음
  - 프로그래머 자원을 낭비하지 않으면서 디자이너에게 게임 변경 능력을 부여할 수 있음
  - 플레이어에게 스크립트 기능을 제공함으로써 직접 게임을 확장할 수 있게 함
- 스크립팅의 단점
  - 디버깅이 어려움
  - 비프로그래머들이 프로그래밍을 해야함
  - 스크립팅 언어의 구현이나 도입, 그리고 보조 디버깅 도구들의 개발에 시간을 들여야 함

References

- **A* Pathfinding for Beginners -** Patrick Lester http://www.policyalmanac.org/games/aStarTutorial.htm
- AI Game Programming Wisdom – Steve Rabin, Charles River Media.
- Programming Game AI by Example – Mat Buckland
- http://www.aiguru.com/pathfinding.htm
- http://www.gameai.com/pathfinding.html
- http://www.gamedev.net/reference/list.asp?categoryid=18#94
- http://www.netcomuk.co.uk/~jenolive/vect14.html