길찾기 솔루션

pathfinding solution

A star & jump point search

2020년 12월

노 영 래

목 차

제 1 장 서론

전략 시뮬레이션 게임, mmorpg 혹은 다양한 장르의 게임에서 길찾기 알고리즘을 필요로 한다. 여기서는 대표적인 길찾기 알고리즘인 a star와 이를 개선한 알고리즘인 jump point search를 구현했다. 그리고 2.5d 게임이나 3d게임에서 이를 그대로 사용할 경우 어색한 경우가 생기므로 브레즈햄 알고리즘을 사용하여 보정했다.

제 2 장 A star

2.1 테스트 환경

개발과정에서 직관적인 확인을 위해 윈도우 어플리케이션을 만들어 각각의 과정 사이사이에 랜더링 과정을 넣었다. 또 로직 사이사이에 랜더링 과정이 들어간 버전(개발용) 과 랜더를 떼어내어 로직만 있는 버전(실제 사용)이 찾은 길이 다를 수 있기 때문에 각각의 찾은 길이 일치하는지 확인하는 코드를 넣어 테스트했다.

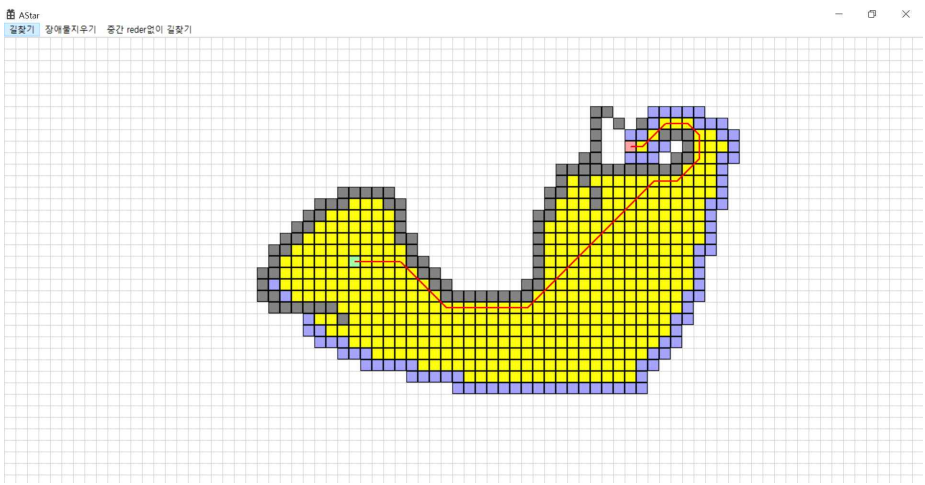

그림 1 A star by window application

2.2 개념&구현

A star 알고리즘은 최적의 길을 찾기 위한 게임들에 사용된다. (주로 전략 시뮬레이션 게임에서 이용된다.) 다음은 주요 구현함수들에 대한 설명이다.

```
bool CAstar::FindRoute(Point start, Point end, Point* route, int *num, ITile* iTile)
```

1. start 노드를 생성하여 openList에 삽입
2. openList (갈 곳, 파란색 노드)에서 노드를 꺼낸다. (더 이상 꺼낼 노드가 없다면 찾지 못한 것 false return. 꺼낸 노드가 end 위치와 같다면 out parameter에 node의 부모 참조해가며 값 셋팅 후 true return)
3. 위 조건에 부합하지 않는다면 closeList에 넣고(간 곳, 노란색 노드) openList에는 팔방향에 대해 MakeNode 함수를 호출한다. (노드를 만들지 말지 결정 후 openList에 넣어주는 함수)
4. openList를 안의 노드들의 f (g는 가중치, h는 목표점과의 거리를 나타낸다. f 값은 g와 h를 mix하여 만든다.) 를 기준으로 작은 순으로 정렬한다. (그래야 2번 노드를 꺼낼때 가장 최적의 노드를 선택한다.)
5. 2,3,4 과정을 2번에서 return 될 때까지 반복

```
void CAstar:: MakeNode(int tile_x,int tile_y,st_Node*pParent,float ginc)
```

1. 해당 위치가 유효한지 확인 (영역 안 인지, 장애물이 아닌지) 유효하지 않다면 return
2. closeList를 확인하여 인자로 받은 위치를 간 적이 있는지 확인 후 있다면

return

3. openList를 확인하여 인자로 받은 위치에 해당하는 노드가 있다면 현재 셋팅할 f값과 비교(h는 목적지와의 거리기 때문에 변하지 않지만 g의 경우 부모의 g값에 일정량을 더하고 있기 때문에 부모가 달라지면 다른 값이 나온다.) 후 openList에 존재하는 f 값이 더 작다면 return 아니라면 노드 생성

4. 2,3 어디에도 속하지 않는다면 노드 생성

5. openList에 삽입

2.3 구현시 주의

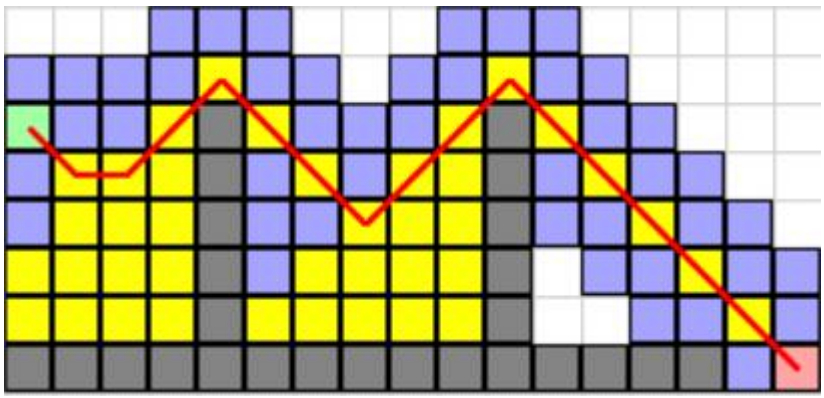

그림 2 A star 잘못된 예

부모에 대한 가중치를(g) 직선과 대각선을 똑같이 주면 다음과 같은 상황이 발생할 수 있다.(대각선은 더 높게 주어야 한다) 또 목적지에 가까운 노드들을 우선적으로 뽑게 하기 위해 f값을 생성할 때 h값의 비중을 높게 하여 믹스 할 경우에도 같은 상황이 나올 수 있다. 두가지 다 고려를 해 주어야 한다. 우리가 원하는 경로는 다음과 같다.

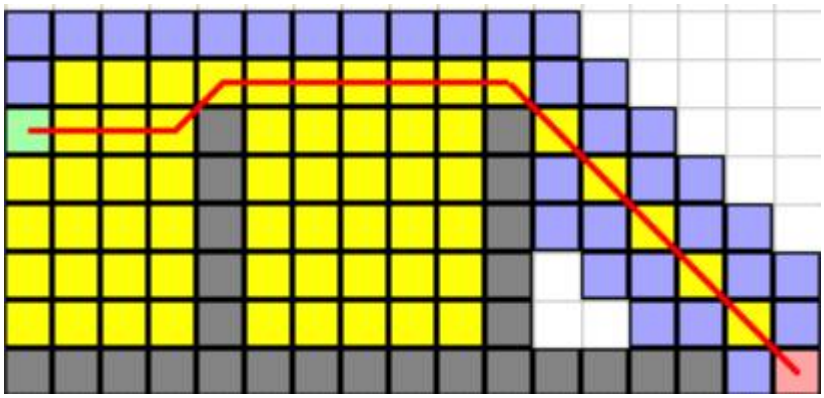

그림 3 A star 잘된 예

2.4 단점

노드가 너무 많이 생성된다는 점이다. 성능상 이슈가 있을 수 있다. 또 2d게임이 아닌 2.5d 게임이나 3d 게임의 경우 다음과 같이 한차례 보정을 해 주어야 하는데(파란선)

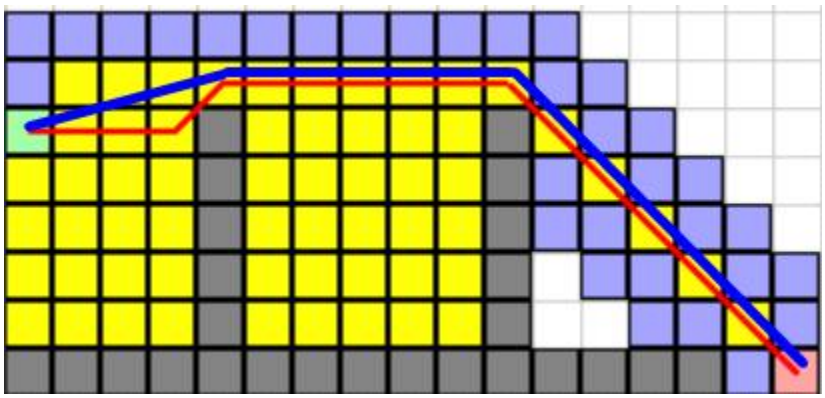

그림 4 A star의 보정

어디가 꺽이는 부분인지 찾기 위해 노드를 일일이 검사해야 하는데 이것또한 노드가 많기 때문에 많은 비용이 든다.

제 3 장 jump point search

3.1 테스트 환경

A star와 같다.

3.2 개념&구현

A star는 너무 많은 노드를 생성하고 있기 때문에 성능상 이슈가 생길 수 있다. jump point search는 이를 보완하기 위해 만들어졌다. 이를 위해 무분별하게 8방향으로 노드를 만들어가는 것이 아닌 grid에 노드를 만들지 말지를 확인하는 작업이 추가로 들어갔다.(파란색과 노란색 이외의 색깔)

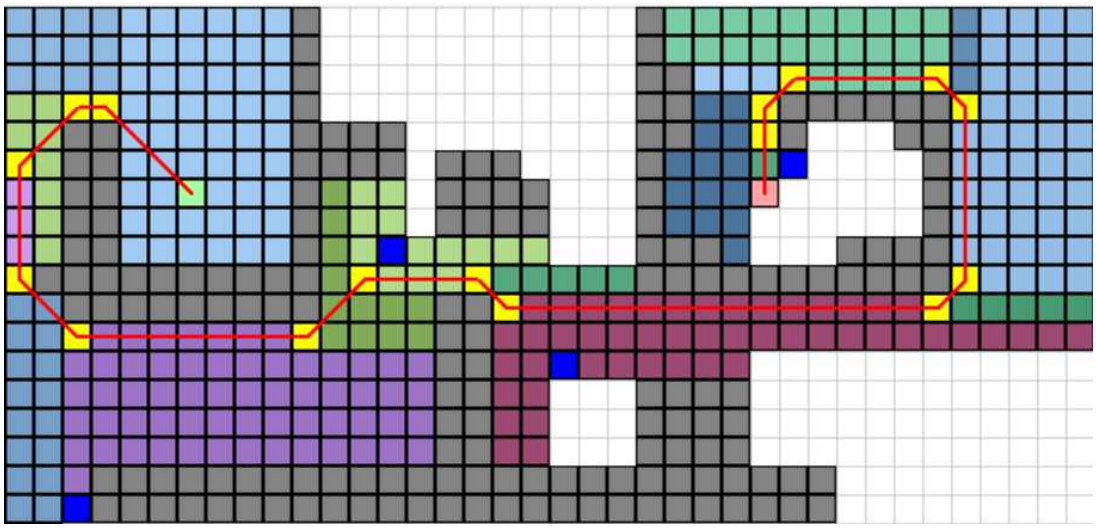

그림 5 jump point search by window application

구현의 매커니즘은 A star와 비슷하다. 후보 노드들을 openList에 넣어 놓고 그 후보중 f 가 작은 놈을 우선적으로 뽑아 closeList에 넣는 방식이다. 단, 이전에는 8방향으로 뻗어가며 무분별하게 노드를 생성했다면 jump point search에서는 openList로부터 노드를 뽑아 노드의 방향성에 따라 선택적으로 탐색하고(최초에

만 8방향 탐색) 또 무조건 노드를 만드는 것이 아니라 선택적으로 만든다.

- 직선방향 탐색

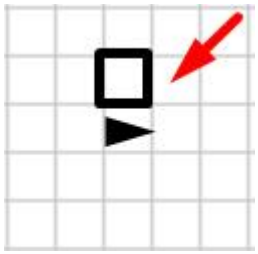

그림 6 직선 코너

코너를 만나면 노드를 생성한다. 직선방향에서 코너의 기준은 현재 위치에서 진행방향을 앞으로 봤을 때 진행방향 옆에 장애물이 있고 그 앞이(화살표) 비어 있으면 그 곳이 코너다. 현재 위치에 노드를 만들고 openList에 넣는다. 이 노드는 직선 방향성(여기서는 RR)을 갖게 되고 나중에 꺼내어 코너가 어느방향인지 체크한 후에 가던 방향과(RR) 코너 방향을(RU) 탐색하게 된다.(코너가 최대 두 방향이 생길 수 있기 때문에 직선포함 2~3개의 방향을 탐색하게 된다.)

- 대각선방향 탐색

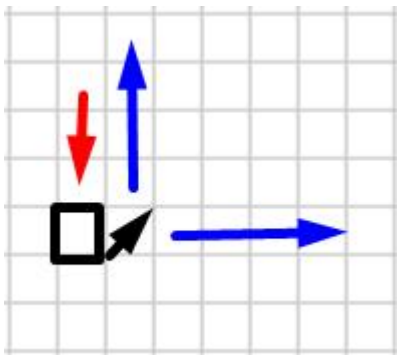

그림 7 대각선 코너

대각선은 노드를 생성하는 기준이 직선과 달리 두 개다.

1. 코너를 만났을때(대각선 기준) 노드 생성. 진행방향 기준 옆이 비어 있고 옆 뒤에 장애물이 있다면 코너다.
2. 진행 방향기준 양 45도 각도의 직선 방향을 탐색했을 때 코너가 있다면 노드를 생성.

이렇게 생성되어 openList에 넣는다. 그리고 이렇게 대각선일 경우 노드를 꺼냈을 때 가던 방향(1), 양 45도 각도 방향(2) 그리고 현재 위치에 양 90도 위치에 대각선 코너가 있다면 코너 방향(0~2) 총 3~5개의 방향을 탐색하게 된다.

3.3 보정

바로 갈 수 있음에도 다른 노드를 거쳐 가는 경로들이 생긴다. 2d 게임이라면 그렇게 어색함이 없지만 3d 게임에서는 어색할 수 있다. 그래서 jump point search 알고리즘을 통해 나온 경로를 가지고 한번 더 보정을 해 주어야 한다. 이를 위해선 한 노드에서 다른 노드로 뻗은 직선에 해당하는 타일에 장애물이 있는지 판단할 수 있어야 한다. 이를 위해서는 직선을 타일 좌표로 얻어내는 방법이 필요하고 여기선 브레즌햄 알고리즘을 이용했다.

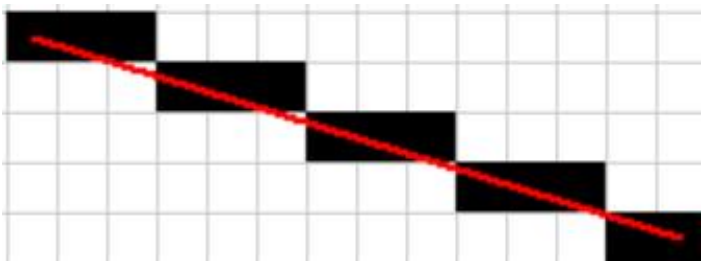

그림 8 브레즌햄 알고리즘

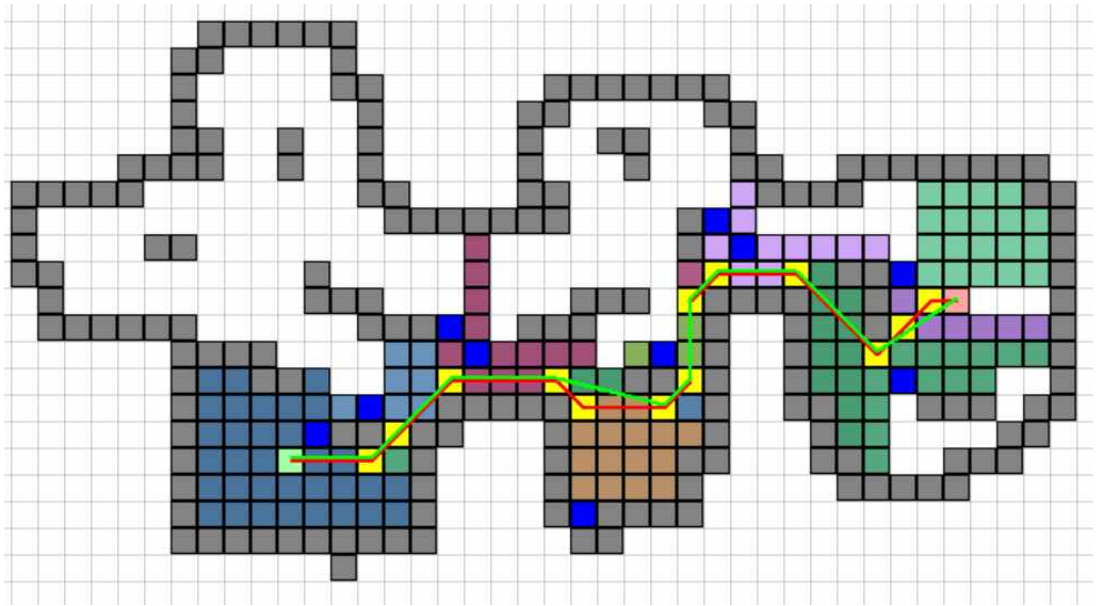

그림 9 보정 결과(before : 빨 after : 연두)

제 4 장 결론

2d 전략 시뮬레이션 게임에 이용되는 A star를 구현해봤다. 그대로 쓸 때도 노드 생성량 때문에 성능의 이슈가 있었지만, 3d 게임에서 사용하기 위해서는 보정을 해 주어야 하는데 노드가 너무 많아 보정시 성능에도 이슈가 있었다. 그래서 jump point search를 구현하고 브레즈햄 알고리즘을 이용하여 보정했다. 단순 2d에서 A star와 jump point search를 성능 비교 했을 때 jump point search는 노드를 적게 만들지만 코너를 만나기 전까지는 계속 탐색하기 때문에 장애물이 많이 없는 뻥 뚤린 극단적인 맵의 경우는 A star의 성능이 좋을 수도 있다. 하지만 그 외 일반적인 경우에는 jump point search의 성능이 앞섰다.